

ALCHA

Architectural Level Computational Hardware Abstraction:
A New Programming Language for FPGA Projects



Presented by:
John-Philip Taylor

Prepared for:
Dr Simon Winberg
Dept. of Electrical and Electronic Engineering
University of Cape Town

Submitted to the Department of Electrical Engineering at the University of Cape Town in
partial fulfilment of the academic requirements for a Doctor of Philosophy degree in
Electrical and Computer Engineering.

August 2022

Key words:
FPGA, HDL, Firmware, Programming Language

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

I know the meaning of plagiarism and declare that all of the work in the dissertation, save for that which is properly acknowledged, is my own.

Signed by candidate

.....
John-Philip Taylor

ALCHA

Architectural Level Computational Hardware Abstraction: A New Programming Language for FPGA Projects

John-Philip Taylor

August 2022

Abstract

Recent years have seen vast improvements to the capability of programmable processing platforms, especially field programmable gate arrays, or FPGAs. Modern software languages have been developed, adding features such as duck-typing, dynamic interpretation, built-in high level data structures, etc. Yet, FPGA development is still mostly using traditional hardware description languages such as VHDL and Verilog, and the industry is resorting to third party tools and scripting-based automation in order to increase developer efficiency.

This dissertation presents ALCHA: a new object-oriented language aimed at low-level FPGA development. Main language objectives include increasing the architectural abstraction capabilities, introducing structured programming to FPGA development, automating fixed-point related design, integrating design constraints and increasing the generalisation capability. In short, the ALCHA language is designed to allow the user to increase abstraction and reduce maintenance effort.

After ensuring that the language grammar is parsable, the resulting language design is evaluated by means of a radar-based case study. Language complexity measurement is based on the number of lines of code, and language power is based on the cost of maintenance. ALCHA is shown to support code that is about half as complex and twice as powerful as traditional HDL-based design, based on these metrics.

In future, ALCHA could evolve into a hardware description language in its own right, allowing developers to leverage the strengths of FPGAs.

Contents

List of Figures and Listings	xv
List of Tables	xvii
Nomenclature	xix
Acronyms	xix
Terminology	xxii
1 Introduction	1
1.1 Research Questions	2
1.2 Problem Statement	3
1.3 Objectives	4
1.4 Scope and Limitations	4
1.5 Dissemination Plan	5
1.5.1 Public Domain Hosting	5
1.5.2 Conference Contributions	6
1.5.3 Journal Publications	6
1.6 Document Outline	7
1.7 Summary	8
2 Review of Firmware Development	9
2.1 Abstraction	9
2.1.1 Physical Abstraction	10
2.1.2 Architectural Abstraction	11
2.2 High Level Synthesis	12

2.3	Graphical Design Entry	13
2.3.1	Scalability	13
2.3.2	Finite State Machines	14
2.4	Library Based Solutions	15
2.5	Related Work	15
2.5.1	Python-based Solutions	15
2.5.2	Cx	16
2.5.3	VHDPlus	18
2.5.4	Alchitry Lucid	20
2.5.5	Scala-based Solutions	24
2.6	Summary	26
3	Research Methodology	29
3.1	Research Method	29
3.1.1	Scientific Method	30
3.1.2	Engineering Method	30
3.1.3	Comparing Methods	31
3.1.4	The Chosen Method	31
3.2	Research Process	32
3.2.1	Base Language	32
3.2.2	New Language vs. Markup	32
3.2.3	Guidelines	33
3.2.4	Development Phases	33
3.3	Test and Evaluation	34
3.3.1	Practicality	34
3.3.2	Language Power	34
3.3.3	Counting Lines of Code	36
3.4	Summary	37

4	User Requirements and Language Objectives	39
4.1	Language Properties	39
4.2	Language Objectives	40
4.2.1	Architectural Abstraction	40
4.2.2	Control Flow Abstraction	47
4.2.3	Fixed-point Abstraction	48
4.2.4	Design Constraints	49
4.2.5	Generalisation	51
4.3	Summary	55
5	Design	57
5.1	Architectural Abstraction	57
5.1.1	Moving Pins into Abstraction	58
5.1.2	Modular Design	60
5.1.3	Dependency Considerations	64
5.1.4	Combinational Assignments	66
5.1.5	Registers	69
5.1.6	Abstraction Summary	71
5.2	Control Flow Abstraction	72
5.2.1	Finite State Machines	72
5.2.2	Function Basics	77
5.2.3	FSM Functions	79
5.2.4	RTL Functions	81
5.2.5	Other Considerations	82
5.2.6	Control Flow Summary	85
5.3	Fixed-point Abstraction	85
5.3.1	Extending the Net Type	85
5.3.2	Multiplication	87
5.3.3	Division	87
5.3.4	Addition	88
5.3.5	Assignment and Casting	89

5.3.6	Raw Operations	89
5.3.7	Fixed-point Summary	90
5.4	Design Constraints	90
5.4.1	Attributes	90
5.4.2	Location Constraints	94
5.4.3	Timing Constraints	94
5.4.4	Compiler Behaviour	99
5.4.5	Design Constraints Summary	100
5.5	Generalisation	101
5.5.1	Choosing the Model	101
5.5.2	Scripting Variables	102
5.5.3	Conditional Compilation	104
5.5.4	Compiler Implications	105
5.5.5	Generalisation Summary	106
5.6	Summary	106
6	Compiler Considerations	109
6.1	Translation	109
6.1.1	The Build Chain	110
6.1.2	Parsability	110
6.1.3	Compiler Output	111
6.1.4	Translation Stages	111
6.2	Core Language Features	112
6.2.1	Fixed-Point	112
6.2.2	Maintaining Precision	112
6.2.3	Branching and Loops	113
6.2.4	Register Transfer Logic	113
6.2.5	Finite State Machines	114
6.2.6	Function Calls	114
6.2.7	Attributes and Scripting	114
6.3	Summary	115

7	Evaluation	117
7.1	Project Overview	117
7.1.1	Platform Description	118
7.1.2	Master Trigger and System Timing	118
7.1.3	System Controller	119
7.1.4	Registers Decoder	119
7.1.5	Hardware Control	119
7.1.6	Waveform Generator	120
7.1.7	Receiver Abstraction	120
7.1.8	Radar Processor	120
7.2	Implementation in ALCHA	120
7.2.1	Architectural Abstraction	121
7.2.2	Control Flow Abstraction	124
7.2.3	Fixed-point Abstraction	127
7.2.4	Design Constraints	127
7.2.5	Generalisation	127
7.3	Language Power	129
7.3.1	Code Complexity	129
7.3.2	Cost of Maintenance	133
7.4	Summary	137
7.4.1	Language Objectives	137
7.4.2	New Language Features	138
7.4.3	Potential Language Shortcomings	138
7.4.4	Language Power	139
8	Conclusion	141
8.1	HDL Developer Challenges	141
8.2	Automation	142
8.3	Development Efficiency	143
8.4	Compiler Practicality	143
8.5	Novel Contributions	144

8.6	Practical Applications	144
8.7	Future Work	145
Bibliography		157
A Evaluation Metrics		159
A.1	Code Complexity	159
A.2	Code Maintainability	160
A.3	Language Productivity	160
B ALCHA Design		161
B.1	Changes to the Baseline	161
B.1.1	Blocks	161
B.1.2	Concatenation	162
B.1.3	Array Builders	162
B.1.4	Array Slicing	162
B.1.5	Vectorised Expressions	162
B.1.6	Complex Arithmetic	163
B.1.7	Stringification	163
B.1.8	Loops	163
B.1.9	Comments	163
B.1.10	Unicode Names	164
B.1.11	Extended Character Names	164
B.2	Guidelines	164
B.2.1	Platform Independence	164
B.2.2	Target Independence	165
B.2.3	Expandability	165
B.3	Miscellaneous Features	166
B.3.1	Groups and Classes	166
B.3.2	Import Search Paths	166
B.3.3	Existing Code Integration	167

C SPI Example	169
D Case Study Implementation in ALCHA	173
D.1 Main Module	173
D.2 Platform Description	173
D.2.1 Pin Mapping and Naming	175
D.2.2 Unused Pins	175
D.2.3 Clocking and Reset Sequencing	176
D.3 System Controller	176
D.3.1 Importing the SoC System HDL	177
D.3.2 HDL Source Files	179
D.3.3 Bidirectional Ports	179
D.3.4 Avalon Master Interface	180
D.4 Registers Decoder	183
D.5 Master Trigger	185
D.6 Hardware Control	185
D.6.1 Inheritance	187
D.6.2 Finite State Machine Construct	189
D.6.3 Mutual Exclusion	191
D.7 Waveform Generator	191
D.7.1 Helper Functions	194
D.7.2 Pins	194
D.7.3 SPI State Machine	194
D.7.4 Container Module	194
D.8 Receiver Abstraction	196
D.8.1 ADC Abstraction	197
D.8.2 Packetiser	199
D.8.3 Debug Stream	200
D.9 Radar Processor	201
D.9.1 Programming Model	201
D.9.2 FIFO Queue	202
D.9.3 Window	204

D.9.4	FFT	205
D.9.5	Transpose	207
D.9.6	Alpha Filter	207
D.10	Version Autogen	207
E	ALCHA Grammar	209
E.1	Scanner	209
E.2	Declarations	211
E.3	Expressions	213
E.4	Statements	215
E.5	Verification	216
F	ALCHA Specification	217
F.1	Introduction	217
F.2	Lexical	218
F.2.1	Literals	218
F.2.2	Identifiers	219
F.2.3	Predefined Constants	219
F.2.4	Special Keywords	220
F.2.5	Syntax Highlighting Scripts	220
F.3	Grammar	221
F.4	Modules	221
F.4.1	Target Independence	221
F.4.2	Import and Name-space	221
F.5	Declarations	223
F.5.1	Nets	223
F.5.2	Pins	225
F.5.3	Groups	227
F.5.4	Structures	228
F.5.5	Enumerations	228
F.5.6	Scripting Data Types	230
F.5.7	Built-in Members	230

F.6	Expressions	230
F.6.1	Operators	232
F.6.2	Global Attribute Definition and Access	234
F.6.3	Net use Before Assignment	234
F.7	Statements	236
F.7.1	Control Flow Structures	236
F.7.2	Alias	239
F.8	Arrays	240
F.8.1	Defining Arrays	240
F.8.2	Slices	241
F.8.3	Array Literals	241
F.8.4	Concatenation	242
F.8.5	Vectorised Statements	242
F.9	Functions	243
F.9.1	Combinational Functions	243
F.9.2	Clocked Functions	245
F.9.3	RTL Functions	246
F.9.4	Example	247
F.9.5	Calling Clocked Functions from Combinational Circuits	252
F.10	Synchronous Circuits	253
F.10.1	Clock Networks	253
F.10.2	Register Transfer Level	255
F.10.3	Finite State Machines	256
F.11	Classes	258
F.11.1	Constructor	258
F.11.2	Parameters	259
F.11.3	Member Functions	259
F.11.4	Finally Member Functions	259
F.11.5	Reference to Self	260
F.11.6	Access Specifiers	260
F.11.7	Instantiation	261

F.11.8 Abstract Signal Types	262
F.11.9 Inheritance	262
F.11.10 Polymorphism	263
F.11.11 Attributes	264
F.12 Operator Overloading	264
F.13 Scripting Features	264
F.13.1 Algorithmic Code Generation	264
F.13.2 Mathematical Functions	266
F.13.3 I/O Functions	267
F.13.4 Conversion Functions	268
F.13.5 Dynamic Arrays	269
F.13.6 Shell Commands	269
F.14 Advanced Attributes	270
F.14.1 Logic Regions	270
F.14.2 Partition Assignments	271
F.14.3 Compiler Behaviour	271
F.15 High-level Structures	273
F.15.1 Megafunctions and External HDL Modules	273
F.16 Simulation and Verification	275
F.16.1 Simple Test Benches	275
F.16.2 Model Based Verification	277
F.16.3 Assertion Based Verification	279
F.16.4 Formal Verification	282
F.16.5 Hardware in the Loop Verification	282
F.16.6 Coverage	283
F.16.7 Verification Frameworks	285
F.16.8 Finite State Machines	285

List of Figures and Listings

1	Introduction	1
Fig. 1.1	Abstraction level placement of ALCHA	5
2	Review of Firmware Development	9
Fig. 2.1	Abstraction triangle (adapted from [1] pg. 6)	10
Fig. 2.2	Spaghetti code in LabVIEW, after [2]	14
Lst. 2.1	Implementing a blinking LED in Migen, adapted from [3]	16
Lst. 2.2	Example showing two tasks communicating in Cx, adapted from [4] .	17
Lst. 2.3	Simple VHDLPlus example showing a blinking LED, adapted from [5] .	19
Lst. 2.4	Generated VHDL from listing 2.3, adapted from [5]	19
Lst. 2.5	Example used to showcase Lucid features (part 1), adapted from [6] .	21
Lst. 2.6	Example used to showcase Lucid features (part 2), adapted from [6] .	21
Lst. 2.7	Example used to showcase Lucid features (part 3), adapted from [6] .	21
Lst. 2.8	Lucid DFF declatation forms [7]	22
Lst. 2.9	Example showing SpinalHDL register declaraion, adapted from [8] . .	24
Lst. 2.10	Example showcasing SpinalHDL routing features, adapted from [8] . .	25
Lst. 2.11	Example of a SpinalHDL FSM, adapted from [8]	26
3	Research Methodology	29
Lst. 3.1	Example code used to illustrate counting lines of code in ALCHA . .	36

4	User Requirements and Language Objectives	39
Fig. 4.1	Two options of implementing OSI layers up to the ‘Transport’ level. . .	42
Fig. 4.2	Timing diagram showing the concept of a data stream.	42
Fig. 4.3	Example of a radar top-level block diagram	44
Fig. 4.4	Conceptual overview of a radar	44
Fig. 4.5	A typical use-case for the hexagonal architecture, adapted from [9] . .	47
Lst. 4.1	Example showing how a FIR filter might be generalised in Verilog. . .	52
Fig. 4.6	Pyhon-based auto-generation build chain	54
5	Design	57
Fig. 5.1	ALCHA pin structure	58
Lst. 5.1	Example of a simple counter circuit in ALCHA	59
Lst. 5.2	SystemVerilog equivalent of listing 5.1	60
Lst. 5.3	Example of using <code>import</code> to switch platforms	61
Lst. 5.4	An overview of the ALCHA class syntax	63
Lst. 5.5	I ² C bus attachment	66
Lst. 5.6	Example used to illustrate use-before-assign semantics	67
Fig. 5.2	Equivalent circuits of listings 5.6(a) to 5.6(c)	67
Fig. 5.3	Possible equivalent circuits of listing 5.6(d)	68
Lst. 5.7	Circular combinational circuit example	68
Lst. 5.8	Verilog RTL boiler plate, showing a synchronous reset scheme	70
Lst. 5.9	ALCHA RTL construct	70
Lst. 5.10	RTL example	71
Fig. 5.4	Equivalent circuit of listing 5.10	71
Fig. 5.5	Traditional FSM architecture, adapted from [1]	73
Lst. 5.11	SPI interface example in Verilog	74
Fig. 5.6	SPI waveforms of the ADXL345 magnetometer	74
Lst. 5.12	ALCHA equivalent of the SPI example in listing 5.11	76
Lst. 5.13	Experimental syntax showing an explicit-fences option	77

Lst. 5.14	Example function showing duck-typing	77
Lst. 5.15	Possible Verilog boilerplate for FSM function calls.	80
Fig. 5.7	The go-busy handshake, as implemented by listing 5.15	81
Lst. 5.16	Example showing an RTL version of the ‘Transaction’ function	83
Lst. 5.17	An event handler implemented with an anonymous function	85
Lst. 5.18	Examples of fixed-point declarations	86
Lst. 5.19	ALCHA implementation of an NCO, with and without using raw bits	89
Lst. 5.20	Example showing ALCHA attribute syntax	92
Lst. 5.21	Examples of attribute groups	93
Lst. 5.22	Example of a global attribute	94
Lst. 5.23	Example showing how to derive clocks from a base with multiple sources	96
Lst. 5.24	Example showing external timing requirements	98
Lst. 5.25	Example showing various numerical scripting variables	103
Lst. 5.26	Example showing the target specifier block	104
6	Compiler Considerations	109
Fig. 6.1	Concept block diagram	110
Lst. 6.1	Example showing a simple fixed-point operation	112
Lst. 6.2	Naive implementation of the Verilog equivalent of listing 6.1	112
Lst. 6.3	Correct implementation of the Verilog equivalent of listing 6.1	113
7	Evaluation	117
Fig. 7.1	Overview block diagram of the case study project	118
Fig. 7.2	Simplified timing diagram of the case study project	119
Fig. 7.3	Functional block diagram of the case study digital processing chain . .	120
Lst. 7.1	Main state machine implemetation of the AD9915 module	126
Lst. 7.2	Refactored version of the AD9915 <code>UpdateRegisters</code> function	126
Lst. 7.3	Helper function implemented to aid in adding debug pins	137

C	SPI Example	169
Lst. C.1	SPI interface example in Verilog (part 1)	169
Lst. C.2	SPI interface example in Verilog (part 2)	169
Lst. C.3	SPI interface example in Verilog (part 3)	170
Lst. C.4	SPI interface example in Verilog (part 4)	170
Lst. C.5	SPI interface example in Verilog (part 5)	171
Lst. C.6	SPI interface example in Verilog (part 6)	171
D	Case Study Implementation in ALCHA	173
Lst. D.1	The complete <code>main.alc</code> , showing the top-level code	174
Lst. D.2	Excerpt from <code>DE0-Nano-SoC.alc</code> showing the Arduino pin mapping .	175
Lst. D.3	Excerpt from <code>Platform.alc</code> showing some project-specific pin mapping	175
Lst. D.4	Clock and reset generation	176
Fig. D.1	Platform Designer view of the System Controller subsystem	177
Lst. D.5	Basic form of the System Controller module	178
Lst. D.6	Example showing how the <code>hdl_map</code> function is used	179
Lst. D.7	Implementation of the <code>AvalonInterface</code> base class	180
Lst. D.8	Implementation of the <code>AvalonMaster</code> class	181
Lst. D.9	General form of encapsulating the light weight bus	181
Lst. D.10	General form of encapsulating the SDRAM interface	182
Lst. D.11	General form of encapsulating the stream buffer interface	182
Lst. D.12	Implementation of the <code>RegistersDecoder</code> class (part 1)	183
Lst. D.13	Implementation of the <code>RegistersDecoder</code> class (part 2)	184
Lst. D.14	Implementation of the Master Trigger module	185
Lst. D.15	Implementation of the TriggerGen module	185
Lst. D.16	Source listing of the <code>HardwareControl</code> module	186
Lst. D.17	Source summary of the I2C module	187
Lst. D.18	Source summary of the LTC2991 module	188
Lst. D.19	Source listing of the I2C FSM	189

Lst. D.20	Verilog equivalent of the I2C start sequence	190
Lst. D.21	Source listing of the I2C FSM	190
Lst. D.22	Implementation of the MutEx module	192
Lst. D.23	Class overview of the ADF4519 module	193
Lst. D.24	Code listing of the SPI function of the ADF4519 module	195
Lst. D.25	Implementation of the WaveformGenerator module	196
Lst. D.26	Implementation of the ADS7056 module	197
Lst. D.27	Implementation of the ADS7056 module, continued	198
Lst. D.28	Declaration of the ADC pins in the Platform module	198
Lst. D.29	Implementation of the ReceiverAbstraction module	199
Lst. D.30	Implementation of the DebugStreamer module	200
Lst. D.31	Implementation of the RadarProcessor module	201
Lst. D.32	Implementation of the RadarProcessor module	202
Lst. D.33	Implementation of the FIFO module	203
Lst. D.34	Implementation of the Window module	204
Lst. D.35	Implementation overview of the FFT module	206
Lst. D.36	Implementation of the FirmwareVersion module	208

List of Tables

7	Evaluation	117
TABLE 7.1	Lines of Code: Top-level Project	130
TABLE 7.2	Lines of Code: Library Modules	131
TABLE 7.3	Lines of Code: DSP Modules	132
TABLE 7.4	Lines of Code: Other Subsystems	133
TABLE 7.5	Cost of Maintenance: Registers	134
TABLE 7.6	Cost of Maintenance: New Synthesiser	135
TABLE 7.7	Cost of Maintenance: Test Points	136
D	Case Study Implementation in ALCHA	173
TABLE D.1	Radar Parameters	174

Nomenclature

Acronyms

ABI	Application Binary Interface
ADC	Analogue to Digital Converter
API	Application Programmer's Interface
ARM	Advanced RISC Machine
ASIC	Application Specific Integrated Circuit
AXI	Advanced Extensible Interface
BCD	Binary-Coded Decimal
Bd	Baud, in symbols per second
CAD	Computer Aided Design
CMOS	Complimentary Metal-Oxide Semiconductor
CPLD	Complex Programmable Logic Device
DDR	Double Data Rate
DDS	Direct Digital Synthesis
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processor (or processing)
EDA	Electronic Design Automation
EBNF	Extended Backus–Naur Form
FIFO	First-in, First-out (queue)
FIR	Finite Impulse Response
FLOP	Floating Point Operation

FLOPS	Floating Point Operation per Second
FMCW	Frequency Modulated Continuous Wave
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GUI	Graphical User Interface
HDL	Hardware Description Language
HLS	High Level Synthesis
HPS	Hard Processor System
HSTL	High Speed Transfer Logic
I/O	Inputs/Outputs
I ² C	Inter-IC
IC	Integrated Circuit
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
LE	Logic Element
LHS	Left Hand Side
LO	Local Oscillator
LSb	Least Significant Bit
LSB	Least Significant Byte
LUT	Look-Up Table
LVC MOS	Low Voltage Complementary Metal Oxide Semiconductor
LVDS	Low Voltage Differential Signalling
LVPECL	Low Voltage Positive Emitter Coupled Logic
LVTTL	Low Voltage Transistor-Transistor Logic
MISO	Master Input / Slave Output
MOSI	Master Output / Slave Input
MSb	Most Significant Bit
MSB	Most Significant Byte
NCO	Numerically Controlled Oscillator
NVMe	Non-Volatile Memory Express

OSI.....	Open Systems Interconnection
PC	Personal Computer
PCB.....	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PLL	Phase Locked Loop
PRF	Pulse Repetition Frequency
PRI	Pulse Repetition Interval
RADAR.....	Radio-Assisted Direction and Ranging
REST	Representational State Transfer
RF	Radio Frequency
RHS.....	Right Hand Side
RISC.....	Reduced Instruction Set Computer
SDC.....	Synopsys Design Constraint
SI	Système International d'Unités
SoC	System On Chip
SPI.....	Serial Peripheral Interface
SRAM.....	Static Random Access Memory
SSTL.....	Stub Series Terminated Logic
TCP.....	Transmission Control Protocol
TTL.....	Transistor-Transistor Logic
UART.....	Universal Asynchronous Receiver Transmitter
UDP.....	User Datagram Protocol
V	Voltage
VHDL.....	VLSI HDL
VLSI.....	Very Large-Scale Integration
XML	eXtensible Markup Language

Terminology

Developer	FPGA firmware developer, using any firmware development tool.
Device	The specific target FPGA.
Entity	Akin to a VHDL entity or Verilog module; i.e. unit of digital circuit that has ports to the outside world and can exist at any level of the design hierarchy.
Megafunction	A target-specific module, typically generated from within the vendor IDE, after the Altera nomenclature.
Module	An ALCHA source file that can be pulled into the design by means of an import statement, optionally into its own namespace.
Object file	An intermediary file used in the ALCHA compilation process. It is the result of compiling a single translation unit and describes a collection of objects.
Peripheral	Any device, external to the FPGA, that interfaces directly with the FPGA.
Platform	The platform that the ALCHA compiler runs on, including operating system and computer hardware.
Target	The platform ALCHA is compiling to, including FPGA, PCB, peripherals and vendor IDE.
Translation unit	Akin to a C language translation unit, i.e. a collection of source files and headers that translate to a single object file
User	The developer who is using ALCHA to develop FPGA firmware.
Vendor	The FPGA manufacturer.

Chapter 1

Introduction

Recent years have seen vast improvements to the capability of programmable processing platforms. As processing capability increases, so does the complexity of the software and firmware used to develop applications for these platforms.

During the past decade or two, the serial processing industry has enjoyed the development of a variety of new programming languages, including C++¹, C#, D, F#, Go, Java, JavaScript, Julia, Python, R, Ruby, Rust, Swift and others [10].

Many of these languages provide useful additions to the programming model, including object orientation, garbage collection, duck-typing, templates, built-in high-level data structures, etc. All these features are aimed at increasing the programmer's efficiency by hiding software complexity behind abstraction, thereby enabling the developer to write code that is easier to read and maintain, faster to write and more reusable.

Field programmable gate arrays (FPGAs) are highly parallel and flexible processors with an attractive power budget [11]. The mid-range Arria 10 from Intel², for example, boasts a peak single-precision floating point performance of 1.5 TFLOP per second at 40 GFLOPS per Watt [12]. The high-end Stratix 10 NX is optimised for neural network loads, boasting a power budget of 1 TFLOPS per Watt at 143 TFLOPS for half-precision floating point [13].

In addition to the processing capabilities, modern FPGAs offer a larger level of system integration. Xilinx, for instance, integrate the FPGA fabric with a multi-core ARM processor, ARM-based vector processor, hardened interface controllers³ and high-speed data conversion⁴,

¹And subsequent versions, but C++11 was the major step forward.

²Formally known as Altera

³Such as DDR4, SATA 3.0, PCIe and Ethernet

⁴8 channels of 4 GSps, 12-bit ADCs and 8 channels of 6.4 GSps, 14-bit DACs

all in one ball grid array package [14]. Intel is following a similar route, albeit with a more data-centre focus, integrating the FPGA fabric with a multi-core processor (ARM or XEON), high-bandwidth memory⁵, hardened interface controllers and many⁶ 56 Gbps transceivers [15].

This high level of integration makes modern FPGAs an attractive option to implement both processing and control functionalities on the same device, thereby obtaining a physically smaller product that is easy to update remotely.

The most popular languages for firmware⁷ development, however, is traditional VHDL and Verilog [16]. These hardware description languages (HDLs) are designed to be used at the register transfer level (RTL) of abstraction [17, 18]. The developer would use relatively low-level operators⁸ and constructs⁹ to describe the combinational logic portion of a synchronous circuit. HDLs do support functions and procedures to aid abstraction, but those are semantically limited to a relatively small set of use-cases.

1.1 Research Questions

Firmware development is notoriously difficult [19]. Developers often resort to higher level abstractions, such as high level synthesis or third-party scripts and libraries, instead of developing using the hardware description language (HDL) directly. Indeed, marketing for firmware tools and libraries often emphasise that the user is not required to write HDL code.

This dissertation attempts to make low level FPGA firmware more accessible by answering the following research questions:

- What aspects of current firmware development makes the process difficult to reason about, and therefore challenging for developers?
- What common and tedious development tasks exist, and can they be automated?
- Can low level FPGA development be made more efficient?

⁵Up to 8 GB with a bandwidth of 512 GB per second [15]

⁶Up to 144 in the largest Stratix 10 devices

⁷“Firmware”, in this dissertation, implies code that is used to produce FPGA firmware. Code aimed at embedded processors are classified as “software”. “Hardware” implies the physical printed circuit board and associated integrated circuits.

⁸Add, subtract, multiply and bit-wise logical

⁹Such as `if` and `case` statements

1.2 Problem Statement

This study found that one of the fundamental problems with existing HDL-based development is insufficient abstraction, as well as a shortage in language support for the developer to implement abstraction. A similar conclusion was drawn by Cochlovius *et al.* in 1993 [20], but little has changed in HDL based development since then. The aspects of this problem are discussed in chapter 2 and summarised below.

- The existing HDL¹⁰ programming model makes it near impossible to implement clean layered systems, because the low level detail cannot be hidden in abstraction. The two most popular solutions are: route the low level interfaces through all the intermediate layers in order to talk to the outside world, or instantiate all the architectural layers on the same level of the design hierarchy. Both options break the concepts of abstraction and information-hiding.
- Finite state machines implemented in HDL are verbose and challenging to follow. There is no support for abstracting the control flow beyond what is essentially at the same level as CPU assembly.
- Existing HDL has no direct support for fixed-point arithmetic. It is left to the developer to manually implement correct shifting, clipping and truncation operations, which is an error-prone process.
- Design constraints are specified separately from the objects that they apply to, thereby making maintenance more challenging and labour-intensive.
- Timing constraints of external pins are specified such that a single constraint takes into account both external IC parameters and the PCB trace model. This coupling makes it difficult to reuse existing peripheral abstraction modules without an in-depth understanding related to the external device being controlled.
- There is limited support for module generalisation, thereby limiting opportunities for code reuse.

¹⁰HDLs, for the purposes of this dissertation, include traditional VHDL [17], as well as Verilog [21] and, by extension, SystemVerilog [22].

1.3 Objectives

This dissertation proposes a new firmware programming language: Architectural Level Computational Hardware Abstraction (ALCHA). ALCHA aims to solve the problems summarised in section 1.2 by achieving the objectives derived in chapter 4 and summarised below. It seeks to improve the current firmware development process by exploring a new language aimed at increasing the efficiency of the developer.

- Enable implementation of layered systems by allowing the user to hide all lower level detail and interfaces in abstraction.
- Reduce finite state machine boilerplate and provide mechanisms to abstract the control flow of finite state machines.
- Include fixed-point arithmetic as a core language feature.
- Increase code maintainability by consolidating distributed constraints into a single declaration, or at least the same source file as the object that the constraints apply to.
- Promote code reuse through greater generalisation capabilities, including algorithmic circuit generation.

1.4 Scope and Limitations

This dissertation is limited to the language specification of ALCHA. Implementation details are explored and presented in chapter 6, but due to overall effort required and time-constraints of the project, large components of the implementation is left to future work.

This work does not contribute to the field of high level synthesis (HLS). Future work considered for the ALCHA language include an “`hls{...}`” language construct, but for the purposes of this dissertation, ALCHA is only concerned with streamlining HDL-related language constructs and programming models.

Academic research is limited to synthesisable code. Impacts on simulation are briefly considered, and simulation-related language features are presented in appendix F.16, but simulation-related research are left to future work. It is assumed, for the purposes of this work, that the user will simulate the HDL output of ALCHA, not the ALCHA source code directly.

Furthermore, little consideration is given to ASIC or VLSI design. This work is specifically limited to FPGA development¹¹, and leveraging the flexibility and short development cycle associated with FPGA development.

These limitations place ALCHA within the mid-level layers of abstraction, as shown in figure 1.1. The long-term goal is to see ALCHA as an HDL in it's own right, independent of traditional HDL. It would be compiled, by the vendor tools, directly to a netlist. In the interim, however, ALCHA is compiled to Verilog, which in turn is compiled by the vendor tools.

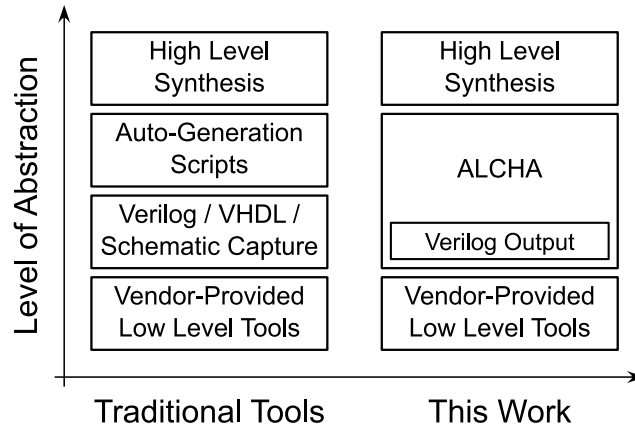


Fig. 1.1. Abstraction level placement of ALCHA

1.5 Dissemination Plan

Two avenues of dissemination were followed: public domain hosting and conference contributions. A journal publication is planned as an additional product of this work, after submission of this dissertation.

1.5.1 Public Domain Hosting

The ALCHA project is hosted on SourceForge [23] and GitHub [24]. The SourceForge page contains a Wiki that describes the language, and both hosts a Git repository with the source code of the prototype compiler.

¹¹FPGAs are limited devices compared to ASICs or general VLSI design. FPGAs do not, for instance, have level-sensitive latches, and the ALCHA language takes advantage of this.

The GitHub further features a getting-started guide and some test cases. Planned refinements are included on the GitHub Projects tab, and the issue tracker can be used by future users and experimenters to provide feedback on the project.

At the time of writing this dissertation, the compiler is in pre-alpha / planning phase, with minimal functionality. It supports the fixed-point engine and can compile very simple combinational circuits to a Quartus project.

1.5.2 Conference Contributions

The preliminary concept was introduced at the IEEE International Conference on Industrial Technology, Taiwan 2016 [25]. More detail relating to fixed-point and procedural programming capabilities was presented at the IEEE Workshop on Signal Processing Systems, Cape Town 2018 [26]. Finally, the programming model was presented at the IEEE Radar Conference, Boston 2019 [27].

1.5.3 Journal Publications

The conference contributions provide a description of the language and some of the more prevalent features, but no real test of the objectives of the project. The planned journal article is therefore a summary of chapters 6 and 7 of this dissertation. The working title is “Analysing ALCHA Features Through a Radar-Related Case Study”.

1.6 Document Outline

This dissertation is structured as follows:

- Chapter 1 provides a broad overview of the project, including research questions, the problem statement and project scope.
- Chapter 2 discusses some necessary background on, and limitation of, firmware abstraction. It includes a discussion on abstraction as applied to this project, explores high level solutions such as high level synthesis, graphical design entry and library based solutions, and concludes with related work.
- Chapter 3 presents the design phases and methodology, based on the findings of chapter 2. In addition, it motivates the choice of research method and presents the test and evaluation aspects.
- Chapter 4 derives the user requirements and language objectives from the findings presented in chapter 2. It presents the five fundamental objectives carried throughout this dissertation.
- Chapter 5 discusses the detail design of the ALCHA language, including arguments relating to design decisions. The chapter is organised around the five fundamental objectives derived in chapter 4.
- Chapter 6 provides thoughts on how a compiler could be implemented. Each of the core language features are presented in turn.
- Chapter 7 investigates a non-trivial case study in which the ALCHA language is evaluated. It centres around the development of the firmware required for a radar system, and evaluates the language power based on the implementation of said firmware.
- Chapter 8 rounds off the dissertation with a summary of the contributions of this research, as well as a brief outlook on potential future work.

1.7 Summary

This dissertation makes the following contributions to firmware development research within the field of electronic engineering:

- A software-like and object-oriented programming model that can be applied to FPGA firmware development, without losing low-level control over the circuit description.
- A means by which fixed-point arithmetic can be automated as an integral component of the hardware description language.
- A significant reduction in boilerplate and complexity when implementing finite state machines, including the ability to apply structured and procedural programming principles and paradigms.

Chapter 2

Review of Firmware Development

“The purpose of abstraction is *not* to be vague, but to create a new semantic level in which one can be absolutely precise.”

Edsger W Dijkstra [28]
The Humble Programmer

This chapter explores the literature related to FPGA development and firmware abstraction strategies. It considers the problem statement as summarised in section 1.2 and investigates how the literature and existing tools are attempting to achieve the objectives highlighted in section 1.3.

A large part of language expressibility relates to the ability of the developer to implement clean abstraction [28]. The concept of abstraction is discussed in section 2.1. Sections 2.2 to 2.4 present common solutions and industry practises aimed at increasing the level of abstraction in firmware code. Section 2.5 concludes the chapter with work more closely related to the ALCHA project.

2.1 Abstraction

The concept of abstraction, in relation to software and, by extension, firmware, has various different definitions, depending on context. Some forms of abstraction are supported directly by the language used in development, and others are coding styles used by the developer to hide certain detail of the design.

Shaw discusses two aspects of abstraction: higher-level programming languages, and abstract data types [29]. The former is a core feature of the development language, whereas the latter

is typically implemented by the developer¹. The concept of data abstraction through object orientation is discussed by Liskov [31].

A related concept is “architecture”, which can be defined as an abstraction of the run-time behaviour of a system, rather than a property of the static source code [32]. The term “architecture” can also be applied to the concept of defining a system in terms of components and of interactions among those components [33].

Shaw describes five popular software architectures: object-orientation, pipes and filters, layered systems, rule-based systems and blackboard systems [29]. These forms of abstraction have one thing in common: information hiding [34]. Information hiding is the concept of hiding the low level detail so that the developer can concentrate on the larger system. In the words of Edsger Dijkstra:

“There is also an abstraction involved in naming an operation and using it on account of ‘what it does’ while completely disregarding ‘how it works’.”

Edsger W Dijkstra [35]
Structured Programming

2.1.1 Physical Abstraction

For the purposes of this dissertation, physical abstraction is defined as the abstraction of the physical detail of the circuit being developed. Figure 2.1 shows the various levels of physical abstraction related to FPGA firmware design in general.

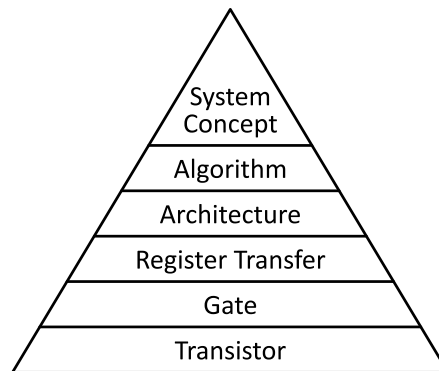


Fig. 2.1. Abstraction triangle (adapted from [1] pg. 6)

FPGA firmware developers do not need to concern themselves with the gate and transistor levels, as this is the domain of the FPGA manufacturer, or vendor. The vendor provides tools to translate a digital circuit description, typically in a hardware description language (HDL),

¹Data abstraction can also be a core feature of the language, as is the case of Python [30]

to an FPGA bitstream [36, 37]. This bitstream is then programmed into the FPGA SRAM cells in order to realise the circuit.

2.1.2 Architectural Abstraction

Architectural abstraction relates to information hiding, as discussed by Colburn *et al.* [34]. The means by which information hiding is achieved leads to the architectural style used by the developer. In the words of Michael Feathers:

“Encapsulation is important, but the reason why it is important is *more* important. Encapsulation helps us reason about our code. In well-encapsulated code, there are fewer paths to follow as you try to understand it. Encapsulation isn’t an end in itself; it is a tool for understanding.”

Michael Feathers [38]

Working Effectively with Legacy Code

Not all applications benefit from the same architectural style. It is up to the developer to choose the architecture best suited to the project under development. The goal is to make use of an architecture that is scalable, simple, modifiable, visible, portable and reliable, as defined by Fielding. In addition, Fielding subdivides modifiability into evolvability, extensibility, customisability, configurability and reusability. [32]

A programming language is most useful if it allows the user to express the solution in as large a variety of architectural styles as possible. The choice of architecture should not be dictated by the language.

HDLs, such as Verilog [21], VHDL [17] or MyHDL [39], typically target the register transfer level (RTL). Basic architectural level physical abstraction is provided by means of entities², functions and procedures. The various entities in a design can be connected in hierarchical fashion in order to build a larger circuit from smaller components [1].

More sophisticated architectural level abstraction is supported by means of higher level tools, such as Vivado IP Integrator [40] and Quartus Platform Designer³ [41]. These tools provide graphical design-entry for top-level modules and are especially well-suited for defining bus architectures, such as AXI [42] and Avalon [43]. The tool abstracts the underlying low-level bus detail, such as arbitration, hand-shaking and clock crossing. The user can connect modules to the bus without worrying about such low-level details.

²Or “modules”, in Verilog nomenclature.

³Formally known as QSys, and SOPC Builder before that.

Various aspects of abstraction are discussed further, from a user requirement perspective, in section 4.2.1.

2.2 High Level Synthesis

High level synthesis (HLS) generally covers the algorithm level of abstraction. HLS tools generate firmware from traditionally sequential languages, such as C++ or Scala [44, 45, 46, 47].

HLS tools such as Matlab HDL Coder [48], SystemC [45], Handel-C [49], OpenCL [50] and many others⁴ raises the level of abstraction by taking care of resource management, pipe-lining, circuit timing, fixed-point types and other lower level details.

HLS attempts to synthesise a circuit based on a behavioural model subjected to user-provided boundary conditions [44, 56]. These conditions include, among others, maximum latency, maximum area and minimum throughput requirements. The HLS tool then optimises the cycle boundaries, resource sharing strategies and other microarchitecture features required to satisfy those conditions.

It is therefore no surprise that HLS research tends to focus on optimisation strategies, including custom operator identification [57], unrolling nested loops [58], partitioning a design over multiple FPGAs [59] and optimising multi-cycle paths [60].

Memory abstraction is also a common topic of research. Liu *et al.* proposes a means by which data memory organisation schemes can be optimised for C-like languages [61]. Gallo *et al.* describes a means by which a high level synthesis tool can automatically optimise internal memory partitioning [62].

The automation of these optimisations hides the timing-related detail from the user [63]. There are many use cases where this is undesired, where the user wants to retain low level control over the timing-related behaviour of the circuit. Examples include real-time DSP applications (e.g. radar) and peripheral abstractions with timing-accurate external interfaces. HLS is therefore not the correct tool for the job in these cases. The topic of control-flow abstraction, from a user requirement perspective, is discussed further in section 4.2.2.

⁴Notable further examples of HLS tools include: PyHDL [51], PHDL [52], SR [53], THDL++ [54] Intel HLS [55] and Vivado HLS [36]

HLS tools also have the capability to synthesise fixed-point circuits based on floating-point designs. Although useful, HLS tools remove low-level control from the user, making it impractical to fine-tune the final implementation [64].

Fixed-point arithmetic has different rounding behaviour to floating-point arithmetic. It is generally left up to the developer to analyse the required signal to quantisation noise ratio and then choose the appropriate fixed-point format. This process can be automated, however [65]. The topic of fixed-point processing and related abstraction is discussed further, from a user requirement perspective, in section 4.2.3.

2.3 Graphical Design Entry

Three prominent examples of tools that support graphical design entry for FPGA firmware are Matlab Simulink with HDL Coder, LabVIEW and Aldec Active HDL [66, 48, 67, 68]. These tools generally focus on two aspects of firmware design: the high level block diagram, which has scalability issues; and the internal flow of finite state machines.

2.3.1 Scalability

Graphical programming tends to work well for small projects, but does not scale well. Consider figure 2.2, taken from a blog post on LabVIEW coding style.

In the post, figure 2.2 is used to illustrate bad coding practices, so in all likelihood this is not truly representative of “real” LabVIEW code. This said, it is easy to imagine that the majority of large projects turn into this form, unless careful attention is given to hierarchical design principles.

Graphical representations of programs are difficult to maintain. Any addition of functionality implies that the schematic must be rearranged in order to retain an ordered appearance.

Connections are not simply lists of assignments that can be moved around at will, as is the case with text-based source code. Any modification to the diagram interferes with existing wires (or connections), which must therefore be moved out of the way, which implies that the modules connected to those wires must also be moved. The effect quickly snowballs into a large undertaking for even the most trivial updates.

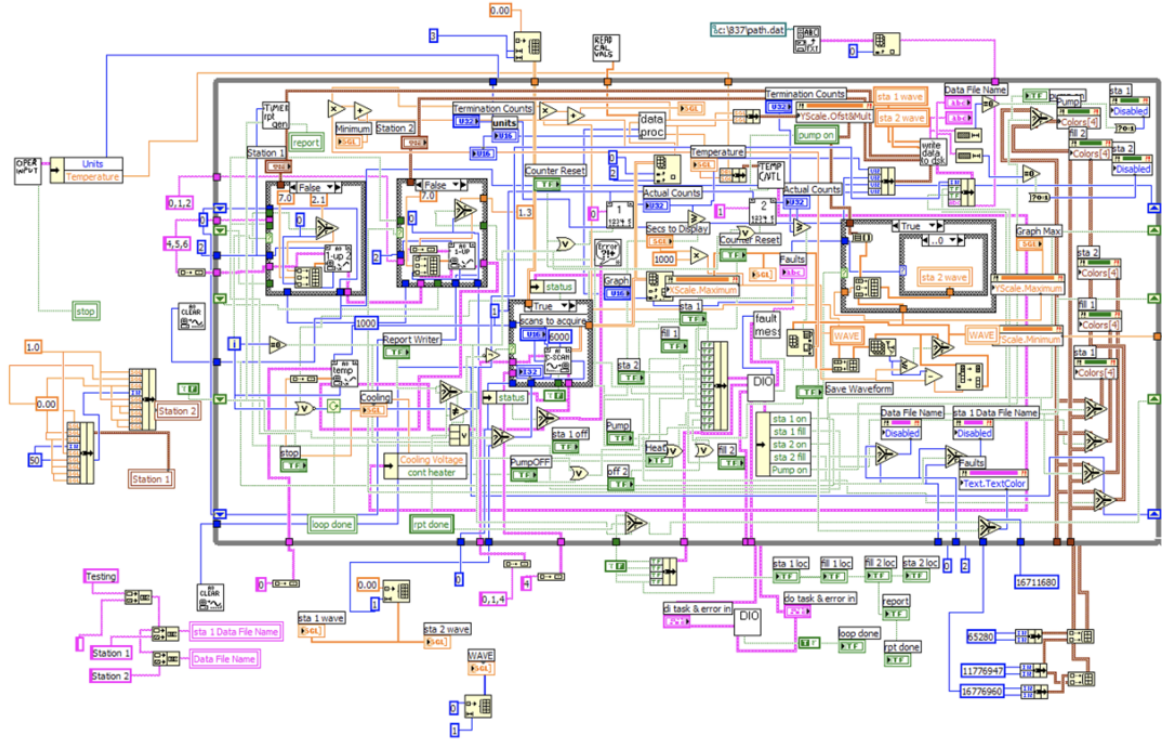


Fig. 2.2. Spaghetti code in LabVIEW, after [2]

2.3.2 Finite State Machines

The same is true for graphical finite state machine (FSM) design entry, as provided by tools such as LabVIEW and Aldec Active HDL [67, 68]. The technique works well for small (i.e. trivial) state machines, but quickly becomes unmanageable as the functionality increases.

The fundamental purpose of engineering diagrams, such as block diagrams and state charts, is abstraction beyond a synthesisable form. The target audience wants to concentrate on the concept (or some specific aspect) of the system, not all the details at the same time.

Graphical programming requires that there is sufficient detail included in the diagram that a compiler can synthesise the program (or circuit) from the description. This additional detail adds sufficient clutter to render the diagram useless from a conceptual standpoint.

In addition, graphical FSM design does not alleviate the problem of control flow, as described in section 4.2.2. A developer unfamiliar with the design must decode a confusing mass of connections in order to understand the control flow well enough to add new functionality.

2.4 Library Based Solutions

Code re-usability is one of the most powerful time-saving techniques in modern software development [69]. Intel and Xilinx both provide a large array of highly parametrisable IP (intellectual property) modules. These modules include anything from a simple unsigned multiplier to a fully-featured multi-gigabit Ethernet MAC and physical. The user can also develop their own modules and add them to the list, or even put them up for sale.

The higher-level modules typically make use of standardised input and output interfaces in order to facilitate interoperability. Modules targeting Intel devices typically make use of Avalon [43] interfaces, whereas those targeting Xilinx devices make use of AXI [42]. In order to ease the interconnection further, the FPGA vendors provide tools to facilitate the connection of these modules [41, 40].

Quartus Platform Designer and Vivado IP Integrator are graphical interfaces where the user can set the various parameters of the modules and connect them together by means of point-and-click. The tool will then synthesise all necessary connections, memory-mapped buses, phase locked loops, delay locked loops, clock-domain crossings, arbitration, etc. that is required in order to realise those connections.

Other library-based solutions also exist, such as PYNQ [70], which targets a Xilinx SoC-based development board with a Python interface. It is domain-specific to streaming processors, utilising a library of so-called ‘overlays’, which are essentially processing kernels.

2.5 Related Work

This section explores a selection of existing and up-coming solutions more closely related to ALCHA, i.e. frameworks and languages with a higher level of abstraction, but without losing the low-level control provided by HDLs.

2.5.1 Python-based Solutions

There are a variety of Python-based HDLs such as MyHDL, Migen and YAHDL [39, 3, 71]. These are essentially Python scripts and data structures aimed at generalisation and auto-generation of firmware source-code.

Although useful, the syntax resulting from building a language within the Python framework is often quite verbose, as illustrated in listing 2.1. The concept involves wrapping the resulting

HDL constructs in Python classes and data structures, which implies that only a limited subset of operations can be implemented by native Python operators. The rest must be implemented by means of methods and functions.

```

from migen.fhdl.std import *
from migen.fhdl import verilog

class Blinker(Module):
    def __init__(self, led, maxperiod):
        counter = Signal(max=maxperiod+1)
        period = Signal(max=maxperiod+1)
        self.comb += period.eq(maxperiod)
        self.sync += If(counter == 0,
                        led.eq(~led),
                        counter.eq(period)
                    ).Else(
                        counter.eq(counter - 1)
                    )

led = Signal()
my_blinker = Blinker(led, 30000000)
print(verilog.convert(my_blinker, ios={led}))

```

Listing 2.1. Implementing a blinking LED in Migen, adapted from [3]

The advantage, however, is that all the capabilities of the Python language can be directly applied to the firmware source. The user therefore has access to powerful parametrisation and algorithmic circuit generation capabilities.

2.5.2 Cx

Synflow Cx is a language that models FPGA circuits as Kahn process networks [72, 73]. Cx addresses many of the issues relating to control flow, as highlighted in section 4.2.2.

2.5.2.1 Tasks and Queues

The main principle behind Cx is illustrated by the example presented in listing 2.2. All ports are treated as streams (i.e. `counter` in this case). Streams are accessed through their `read` and `write` methods.

The example in listing 2.2 uses non-blocking read. In other words, the `print` function will output something every clock-cycle of simulation. If `counter` was defined using the `sync` keyword, however, the `read` accessor is turned into a blocking call that will only allow further execution when there is something to be read. A future version of the language will also support back-pressure (i.e. making the `write` function block when there is no active `read`).

```
network N {
  t1 = new task {
    out uint counter;
    uint count;
    void loop() {
      count++;
      fence;
      counter.write(count);
    }
  };

  t2 = new task {
    void loop() {
      print("count = ", t1.counter.read);
    }
  };
}
```

Listing 2.2. Example showing two tasks communicating in Cx, adapted from [4]

Networks can declare ports themselves (i.e. outside the task), so that networks can import other networks in a hierarchical structure. Networks can also contain **properties** structures, so that new instances can be parametrised.

2.5.2.2 Cycle-accurate Modelling

A task in Cx is essentially a finite state machine with implicit states. The statements within a task are executed concurrently up to the next cycle break. There are two types of cycle breaks: implicit and explicit.

An implicit cycle break occurs: [4]

- when a port is read that has already been read in this cycle, or
- when a port is written that has already been written in this cycle, or
- before each iteration of a sequential loop.

An explicit cycle break can be introduced with two instructions: [4]

- **fence** ends the current cycle and introduces a new one.
- **idle(n)** ends the current cycle and waits for **n** cycles.

2.5.2.3 Structural Programming

The syntax of Cx tasks allows for relatively easy expansion to structural programming. Loops can span multiple clock cycles, and the developer can refactor any portion of a task into a named function. The cycle-related semantics summarised in section 2.5.2.2 are used to ensure correct execution flow.

2.5.2.4 Cx Summary

The Cx programming model provides an elegant solution to the issues of finite state machine abstraction discussed in section 4.2.2. The typical `case` statement used to describe FSMs in HDL is completely hidden in abstraction, thereby reducing verbosity by a significant factor.

The use of Kahn process networks as the foundation of the programming model is somewhat limiting though. The use of Cx is therefore mainly restricted to data-flow applications, such as streaming processors. In addition, Cx suffers from the same architectural abstraction issues discussed in section 4.2.1, especially layered systems that must connect to external devices.

2.5.3 VHDLPlus

VHDLPlus is a language that significantly reduces boilerplate in VHDL code [5]. It mainly addresses readability and maintainability of relatively simple processes and state machines.

The target user is akin to that of the Arduino environment and libraries. One can do quite powerful things with very little code, at the cost of inefficient underlying mechanisms.

2.5.3.1 Structure

A VHDLPlus module is equivalent to a VHDL module, and follow largely the same syntax and type system. One can even embed VHDL code within a VHDLPlus module. A simple example is presented in listing 2.3, with the equivalent VHDL in listing 2.4. The interested reader is encouraged to visit the VHDLPlus website [5] for more comprehensive examples and documentation.

A VHDLPlus process is equivalent to a VHDL process with an implicit clock. The same variable vs. signal semantic rules apply (i.e. blocking vs. non-blocking). The language makes use of FPGA register boot initialisation for reset purposes, without an implicit reset mechanism in the process block.

```

Main (
  LED : OUT STD_LOGIC; -- Declare output
){
  Process() {           -- For parallel and procedural programming
    Thread {             -- Procedural programming
      LED <= '1';        -- LED on
      Wait(1000ms);      -- Wait
      LED <= '0';        -- LED off
      Wait(1000ms);      -- Wait
    }
  }
}

```

Listing 2.3. Simple VHDLPlus example showing a blinking LED, adapted from [5]

```

library IEEE;
  use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

entity Blink is port(
  Clk : in std_logic;
  LED : out std_logic -- Declare output
); end Blink;

architecture Behaviour of Blink is begin
  process(Clk)
    variable Thread0 : natural range 0 to 24000004 := 0;
  begin
    if(rising_edge(Clk)) then
      case(Thread0) is           -- Generated state machine
        when 0 =>
          LED <= '1';           -- LED on
          Thread0 := 1;
        when 1 to 12000001 =>
          Thread0 := Thread0 + 1; -- Wait
        when 12000002 =>
          LED <= '0';           -- LED off
          Thread0 := 12000003;
        when 12000003 to 24000003 =>
          if (Thread0 < 24000003) then
            Thread0 := Thread0 + 1; -- Wait
          else
            Thread0 := 0;
          end if;
        when others => Thread0 := 0;
      end case;
    end if;
  end process;
end Behaviour;

```

Listing 2.4. Generated VHDL from listing 2.3, adapted from [5]

Signals and variables can be declared on-the-fly within the process block, thereby increasing readability by moving the declaration to where the object is used. The compiler will move those declarations to the appropriate declaration blocks in the output VHDL.

A VHDLPlus process can further contain one or more thread blocks, each executing in parallel. These are compiled to counter-and-case structures, i.e. the statements within a thread are executed one statement at a time by means of gating the assignment with the value of a counter.

The thread block is quite versatile, supporting control flow structures such as `if`, `case`, `for`, `while` and `wait` constructs. It is difficult, however, to force the compiler to execute two consecutive statements in the same clock cycle.

The VHDLPlus language also supports `Function` and `SeqFunction` structures, which are equivalent to VHDL `function` and `procedure` structures, respectively, but with the same reduction in verbosity facilitated by the rest of the language.

2.5.3.2 Clocking and Timing

In VHDLPlus, the clock is implicit and does not require definition or routing. There is only one clock in the system. The language also supports “wait” statements, which support both clock-cycle and absolute time forms, the latter requiring a 12 MHz clock [5].

2.5.3.3 VHDLPlus Summary

VHDLPlus does not offer a new programming model. Rather, it provides an abstraction layer over the existing VHDL programming model in addition to a significant reduction in verbosity. The language is aimed at beginner and hobbyist developers and, as a consequence, does not provide the developer with the full functionality possible with FPGAs.

2.5.4 Alchitry Lucid

Another language of interest is Alchitry Lucid [6]. It is essentially SystemVerilog with reduced verbosity and a cleaner syntax. Consider the example in listings 2.5 to 2.7.

```

module hello_world_rom(
    input  address[4],
    output letter [8]
){
    const TEXT = "\n\r!dlroW olleH"; // text is reversed to make 'H' address [0]

    always{
        letter = TEXT[address];
    }
}

```

Listing 2.5. Example used to showcase Lucid features (part 1), adapted from [6]

```

module greeter(
    input  clk,
    input  rst,
    input  new_rx,
    input  rx_data[8],
    output new_tx,
    output tx_data[8],
    input  tx_busy
){
    const NUM_LETTERS = 14;

    // Declare local registers
    .clk(clk){
        .rst(rst){
            fsm state = {IDLE, GREET};
        }
        dff count[$clog2(NUM_LETTERS)];
    }

    // Create an instance of the "hello_world_rom" submodule
    hello_world_rom rom;
}

```

Listing 2.6. Example used to showcase Lucid features (part 2), adapted from [6]

```

always{
    // Hook up the rom ports
    rom.address = count.q;
    tx_data     = rom.letter;

    new_tx = 0;

    // Run the state machine
    case(state.q){
        state.IDLE:
            count.d = 0;
            if (new_rx && rx_data == "h")
                state.d = state.GREET;

        state.GREET:
            if(!tx_busy){
                count.d = count.q + 1;
                new_tx  = 1;
                if (count.q == NUM_LETTERS - 1)
                    state.d = state.IDLE;
            }
    }
}
}

```

Listing 2.7. Example used to showcase Lucid features (part 3), adapted from [6]

2.5.4.1 Explicit Registers

An `always` block in Lucid behaves differently to one in Verilog. In Lucid, everything outside an `always` block must be a declaration, and everything inside an `always` block are combinational assignments.

Lucid registers are implemented by creating instances of the `dff` sub-circuit. This circuit takes `clk` and `rst` port parameters at declaration.

It also exposes two other ports: `d`, which must be assigned a signal or expression, and `q`, which is the output port. This completely eliminates the common confusion regarding blocking vs. non-blocking statements in Verilog, or signal vs. variable semantics in VHDL.

Various forms of declaration are presented in listing 2.8. The different forms are:

- Form (1): A 32-bit register that resets to 15 (decimal).
- Form (2): A signed 8-bit register with clock and reset lines (it resets to 0).
- Form (3): A register set for the structure `color`, which is defined elsewhere.
- Form (4): An enumerated register, typically used as a finite state machine variable, hence the `fsm` keyword.

```
dff ctr[32] (#INIT(32d15), .clk(clk), .rst(rst)); // (1)
signed dff value[8]      (.clk(clk), .rst(rst)); // (2)
dff<color> rgbValue      (.clk(clk), .rst(rst)); // (3)
fsm state                (.clk(clk), .rst(rst)) = // (4)
{ IDLE, START, WAIT_CMD, READ, WRITE, STOP, WAIT };
```

Listing 2.8. Lucid DFF declatation forms [7]

This explicit register sub-circuit concept has another interesting side-effect. It moves the reset out of the traditional `always` block and makes it part of the register declaration, similar to a C-style initialiser.

2.5.4.2 Assignment Blocks

It is quite verbose to manually port-map every register in a design, so the user can make use of assignment blocks instead, as shown in the lower half of listing 2.6.

The port mapping is taken out of the declaration and turned into an environment within which all declarations automatically effect that mapping. This works for any port mapping, not just clock and reset.

The example in listing 2.6 only has one assignment per assignment block, but the language also supports multiple assignments in a single block (i.e. multiple ports can be mapped without nesting assignment blocks).

2.5.4.3 Direct Port Access

The `hello_world_rom` module (listing 2.5) is purely combinational. It is therefore instantiated external to the clock and reset assignment blocks (listing 2.6). It does have two ports, however, seemingly unmapped.

The ports are assigned values in the first two lines of the `always` block (listing 2.7), showcasing the feature of direct port access. There is no need to define wires in order to connect sub-circuit ports to the local state machine, thereby removing quite a large contributor to verbosity in HDL.

A convenient coding style that makes use of this feature could be to assign only the input ports of the sub-circuit immediately below its instance declaration. The output ports will be connected where they are used.

This scheme significantly reduces code clutter prevalent in top-level entities. Traditionally, each mapping has three parts: the wire declaration, the input port mapping and the corresponding output port mapping. In the worst case scenario, each output port fans out to only one input port, which is common in DSP chains.

By using the scheme described, the top-level plumbing can be reduced to only the input mapping, which is a 66% reduction in lines-of-code (assuming unity fan-out and negligible boilerplate).

2.5.4.4 Alchitry Lucid Summary

By means of a few simple language constructs, Alchitry Lucid has removed a large portion of verbosity from the Verilog on which it is based, in addition to making the language less confusing and more accessible to beginner firmware developers.

It still follows the traditional hierarchy-of-modules architecture though, and therefore suffers from the same shortcomings described in section 4.2.1.

2.5.5 Scala-based Solutions

Chisel, SpinalHDL and DFiant are three Scala-based languages offering object-orientation features [74, 75, 76]. This object orientation aspect facilitates user-defined abstraction. The language includes inheritance (with polymorphism) as well as templating features.

DFiant follows a data-flow programming model, and therefore falls more into the category of HLS. Chisel and SpinalHDL are very similar in feature set (even though they differ in the details), so only SpinalHDL is considered in this subsection.

2.5.5.1 Explicit Registers

SpinalHDL offers a similar programming model to Alchitry Lucid (as described in section 2.5.4) in terms of registers. Similar explicit register declarations are used (even though the `d` and `q` ports are hidden in abstraction), thereby offering the same advantages. Examples of register declaration is shown in listing 2.9.

```
// UInt register of 4 bits
val reg1 = Reg(UInt(4 bit))

// Register that samples reg1 each cycle
val reg2 = RegNext(reg1 + 1)

// UInt register of 4 bits initialized with 0 when the reset occurs
val reg3 = RegInit(U"0000")
reg3 := reg2
when(reg2 === 5){
  reg3 := 0xF
}
// Register that samples reg3 when cond is True
val reg4 = RegNextWhen(reg3, cond)
```

Listing 2.9. Example showing SpinalHDL register declaration, adapted from [8]

2.5.5.2 Clocking Areas

SpinalHDL abstracts clock and reset signals into so-called clocking areas. Any register declared within a clocking area (including any part of the sub-circuit hierarchy) uses the current clocking area. In other words, the sub-circuit inherits the clocking area from the parent circuit.

This abstraction is quite convenient, because it removes a significant part of routing overhead. If, however, the user wants to instantiate a sub-circuit with multiple clock domains, those clock domains can be passed as parameters of the class constructor.

2.5.5.3 Routing and Interface Abstraction

SpinalHDL supports a convenient construct for abstracting complex routing of buses, as shown in listing 2.10. The bus connection logic is abstracted into a class member function, in the form of an overloaded `<<` operator. During use, the entire bus connection (both directions) is reduced to a single line of code.

```
class MyBus(payloadWidth: Int) extends Bundle with IMasterSlave{
  val valid  = Bool
  val ready  = Bool
  val payload = Bits(payloadWidth bits)

  // Define the direction of the data in a master mode
  override def asMaster(): Unit = {
    out(valid, payload)
    in(ready)
  }
  // Connect that to this
  def <<(that: MyBus): Unit = {
    this.valid  := that.valid
    that.ready  := this.ready
    this.payload := that.payload
  }
  // Connect this to the FIFO input, return the fifo output
  def queue(size: Int): MyBus = {
    val fifo = new MyBusFifo(payloadWidth, size)
    fifo.io.push << this
    return fifo.io.pop
  }
}

class MyBusFifo(payloadWidth: Int, depth: Int) extends Component{
  val io = new Bundle{
    val push = slave (MyBus(payloadWidth))
    val pop  = master(MyBus(payloadWidth))
  }
  val mem = Mem(Bits(payloadWidth bits), depth)
  // ...
}
```

Listing 2.10. Example showcasing SpinalHDL routing features, adapted from [8]

This example goes even further, showing how the bus interface can be turned into a FIFO queue by means of a single function call.

2.5.5.4 Finite State Machines

Finite state machines are implemented with a more friendly syntax than traditional HDL, including helper-constructs such as `onEntry`, `whenIsActive` and `onExit`. Similar levels of detail and verbosity are used, however, as illustrated in listing 2.11. There is little, if any, control flow abstraction.

```

val fsm = new StateMachine{
  io.result := False
  val counter = Reg(UInt(8 bits)) init (0)

  val stateA : State = new State with EntryPoint{
    whenIsActive (goto(stateB))
  }
  val stateB : State = new State{
    onEntry(counter := 0)
    whenIsActive {
      counter := counter + 1
      when(counter === 4){
        goto(stateC)
      }
    }
    onExit(io.result := True)
  }
  val stateC : State = new State{
    whenIsActive (goto(stateA))
  }
}

```

Listing 2.11. Example of a SpinalHDL FSM, adapted from [8]

2.5.5.5 Scala Based Summary

Chisel and SpinalHDL both offer vast improvements to routing verbosity in the top-level entity, as well as user-defined abstraction by means of object oriented features. By abstracting the clock and reset into inheritable clocking areas, most of the associated clocking and reset routing verbosity is also eliminated.

There is little improvement relating to other aspects of architectural and control flow abstraction, however. Function calls are purely combinational, just like traditional HDL, and the same module-and-port structure is followed, resulting in the same issues discussed in section 4.2.1.

2.6 Summary

This chapter presented the background for this dissertation. Various different firmware development concepts, paradigms and methods have been considered, including a brief definition of abstraction, high level synthesis, graphical design entry and library based solutions.

Related work has been investigated, most of which provide significant improvements in one area of FPGA development or another. This said, there is no “all-inclusive” solution to the various issues highlighted in this chapter.

The next series of chapters present how the concepts introduced by other languages can be unified into a single language, in addition to solving the other abstraction-related issues of existing HDL.

Chapter 3

Research Methodology

Chapter 2 discussed aspects of modern firmware development, including related work aimed at improving firmware development efficiency. This chapter presents concepts and methods used in this study to build upon existing concepts and further improve the firmware development process.

Section 3.1 motivates the choice of research method. Section 3.2.4 presents the development phases and design methodology. Section 3.3 presents the means by which the resulting ALCHA language is tested and evaluated.

3.1 Research Method

Generally, research follows one of two paradigms: the scientific method and the engineering method [77]. This section explores the two methods as applied to the development of the ALCHA language, and concludes with the choice of paradigm followed for the rest of this study.

3.1.1 Scientific Method

The scientific method is generally used to develop mathematical models or theories that describe the behaviour of nature. The process generally involves the following steps [78]:

1. Observe some specific aspect of nature.
2. Develop (or imagine) a theory that describes the observation. Formulate it in such a way that it can be used to make predictions.
3. Make a prediction by using the new theory.
4. Test the prediction against an observable truth (i.e. attempt to falsify the theory).
5. Analyse the results of the test.
6. Refine the theory with the new insights gained from the analysis.
7. Repeat from step 3.

3.1.2 Engineering Method

The engineering method, by contrast, is generally used to develop a new method by which to implement something or solve a problem. This ‘something’ can be physical or abstract. The process generally involves the following steps (adapted from [77]):

1. Acknowledge that end-users are dissatisfied with the status quo, and that the problem should be addressed.
2. Assess the needs of the market or end-user, in relation to the existing design or implementation.
3. Analyse needs of the market and formulate a set of user requirements.
4. Decompose the problem, and related requirements, into smaller components:
 - (a) Derive and refine subcomponent requirements based on the more general requirements formulated in step 3.
 - (b) Design each component individually.
 - (c) Integrate the system from the smaller components (this step is often virtual and conceptual during the first few iterations).
 - (d) Test the integrated concept against the user requirements.
 - (e) Repeat from step 4a until the design adequately satisfies the user requirements.

5. Evaluate the system more thoroughly against the needs of the market or end-user.
6. Repeat from step 2 until the design satisfies the user needs.

3.1.3 Comparing Methods

Braha *et al.* draws parallels between the scientific and engineering methods [77]. The two methods are similar, but there are some key differences.

The scientific method makes the fundamental assumption that the product of the research can be tested against a concrete observable truth. A model, or theory, is only considered “good” if it can be falsified, i.e. if it can be proven that the prediction deviates from the observable truth [79].

The engineering method makes the assumption that the product of the research can only be tested against qualities of the old implementation, elements of practicality and user requirements, i.e. there is no observable absolute truth.

3.1.4 The Chosen Method

The development of ALCHA follows the engineering method. There is a dissatisfaction with the existing firmware design tools, as evident in the number of new languages being developed (refer to chapter 2).

There is no observable truth that ALCHA can be tested against, but there is a set of user needs that can be derived from the features introduced in new languages, as well as industry feedback.

3.2 Research Process

This section summarises the research process followed. It describes the baseline, guiding principles and development phases of the project.

3.2.1 Base Language

Instead of designing a completely novel language from scratch, it is deemed more efficient to start with an exiting “base” language and build ALCHA from there. The choice of base language is based on what users are already familiar with. The aim is to make use of syntax that is easy to learn, because it does not contradict the user’s existing knowledge to a large extent.

Verilog and VHDL have approximately equal market share [80, 16], so verbosity and flexibility are used as selection criteria instead. Verilog is less verbose and more flexible, and is therefore the base language of choice.

Another available option is to choose one of the software languages, such as C, Python, Scala or Haskell. The argument in this case relates to existing features. One of the key user requirements is to have access to all the existing low-level control provided by HDL (i.e. increase the language power without influencing its efficiency).

Using a software language as a baseline would require significant modification to create a functional HDL, which has a fundamentally different target architecture. Such undesirable modification is evident in the syntax of Python-based HDLs such as MyHDL, Migen and YAHDL [39, 3, 71], which adds justification to the decision to use Verilog as the base language.

This Verilog baseline is then incrementally expanded and modified, as described in chapter 5 and appendix B, in order to accommodate the user requirements discussed in chapter 4.

3.2.2 New Language vs. Markup

One of the original ideas when starting out on this project was to use comment-based markup to existing SystemVerilog HDL – similar to Doxygen [81]. It was quickly realised, however, that the markup will be overly verbose and complex if all the required features were to be implemented.

The second option considered was to extend Verilog with new features, like C++ related to C. The problem with this was syntax clashing, especially with fixed-point types and control flow abstraction.

It was therefore decided to develop a new language instead, even though it is heavily influenced by existing Verilog HDL.

3.2.3 Guidelines

It is important to keep certain design guidelines in mind, as summarised below. These guidelines keep the project in line with the goals and objectives presented in section 4.1.

- **Platform Independence** – Different developers make use of different platforms, such as Windows, Linux or MacOS. Additionally, the file and folder structure in use could be in any of a number of languages. ALCHA source-code should be able to seamlessly switch between platforms, which is especially important when considering an international collaboration.
- **Target Independence** – In order to be useful, library components must be as vendor- and device-independent as possible. Even the same device on a different PCB can sometimes prove problematic. The ALCHA language should therefore be designed so that the user can make use of a new device with minimal changes to the source code, and no changes to the core language.
- **Expandability** – ALCHA is a young language, and still under development. It should therefore be relatively easy to add additional features not yet considered.

3.2.4 Development Phases

The engineering method, as summarised in section 3.1.2, can readily be adapted for use in developing the ALCHA language. Chapter 2 established that the user needs are not adequately satisfied with existing tools. Chapter 4 presents the user requirements of low-level firmware development tools and presents the related language objectives.

Chapter 5 proceeds to design the solution, based on the various language objectives derived in chapter 4. Implementation practicalities are considered in chapter 6, and the ALCHA language is evaluated by means of a radar-based¹ case study in chapter 7.

¹A radar-based study was chosen because it is the use-case the author is most familiar with, and radar-firmware typically incorporate a wide variety of circuit types, including a centralised controller, control registers, peripheral abstractions and digital signal processing chains.

In the interest of brevity, the inner development loop is incorporated into the case study presented in chapter 7. Various findings throughout the case study prompted the introduction of new language features, which are then presented *in situ*.

Chapter 8 concludes this dissertation by considering the next iteration of the outer development loop.

3.3 Test and Evaluation

This section describes the methods used to test the ALCHA language for practicality and language power. Aspects of language power are considered, and a verification strategy is presented.

3.3.1 Practicality

A compiler for the ALCHA language must be practical to implement. To this end, the language grammar will be formalised into an extended Backus-Naur form (EBNF), after which a parser will be developed that translates the language into a usable abstract syntax tree (AST), while at the same time rejecting invalid input.

Core language features must also be practical. This will be argued from the viewpoint of similar features that exist in software languages, and is therefore, by extension, practical to implement for firmware as well. In addition, the fixed-point engine is implemented to the point of producing a synthesisable netlist.

The language practicality is tested and evaluated in chapter 6.

3.3.2 Language Power

Various software-related evaluation metrics are presented and discussed in appendix A, including complexity, maintainability and productivity. Productivity can be decomposed into power and efficiency components [82], the latter of which are equivalent between ALCHA and HDL, which implies that the productivity of ALCHA is equivalent to its power.

3.3.2.1 Evaluation

Language power is difficult to measure objectively: a large pool of expert programmers are required to implement the same problem in both languages being compared, which is impractical. This dissertation proposes an estimate of language power based on the number of lines of code in combination with the cost of maintenance. Evaluation therefore progresses as follows:

1. Measure lines of code to provide an estimate of relative language complexity.
2. Measure the cost of maintenance by comparing the number of deletions and insertions² that require modification during typical maintenance tasks³.
3. The resulting measurements are compared and discussed in isolation, rather than combined into a single metric.

Another aspect to consider is developer productivity, which is a complex subject [84]. For the purposes of this dissertation, developer productivity is considered to be directly correlated to the maintenance cost, and therefore not directly measured.

3.3.2.2 Code Reusability

The level of code reusability is not included in the proposed evaluation. The rationale behind this is as follows:

- An increase in code re-usability implies a drop in library size, not a decrease in derived project development time. HDL libraries tend to have multiple implementations of a particular circuit⁴, which the higher level project can readily make use of. The effect on project development time is therefore negligible.
- Libraries are projects themselves, which must be developed and maintained. By including the library module source as part of the metric ensures that this aspect is incorporated. If the library is difficult to maintain, so too will the project.

²Deletions and insertions are measured by means of a Git diff [83].

³Typical tasks include adding a control register to a previously hard-coded parameter, accommodating a hardware change and adding debugging test-points

⁴A firmware abstraction of a particular DDS IC, for example, might have two implementations: one aimed at using it as an LO that rarely changes frequency, and another aimed at using it as an FMCW waveform generator. The two implementations are different because they optimise different usage of the DDS registers, and will most likely have different interfaces to the higher level module as well.

3.3.3 Counting Lines of Code

The number of lines of code (LoC) is used as one of the parameters of language complexity. A consistent, style-agnostic and cross-language means by which to count lines of code is therefore required. Subsequent subsections present the LoC strategy for languages used to evaluate the ALCHA language power.

The line counter source code can be obtained from the [Python/LoC Counter](#) folder in the case study Git repository [85].

3.3.3.1 ALCHA

The algorithm used to count the lines of code in the ALCHA implementation involve the following rules:

- Ignore comments
- Ignore anything between matching double-quotes
- Ignore anything between matching square brackets
- Ignore concatenation sequences
- Count the remaining semicolons, commas and opening braces

By ignoring code between square brackets, the count is biased to not count large constant arrays, such as pin location assignments. Listing 3.1, for example, will count 3 LoC: one line for the declaration of `Data`, and another two for the two attributes of `DataStrobe`.

```
pin(32) <location = [ "AA27", "Y27" , "T24", "R24",
                    "W26" , "AA28", "R25", "R26",
                    "V27" , "R27" , "N27", "N26",
                    "U28" , "T28" , "N25", "N24",
                    "N28" , "M28" , "M26", "M27",
                    "J28" , "J27" , "L25", "K25",
                    "F28" , "G27" , "K26", "J26",
                    "D27" , "E28" , "J24", "J25" ]> Data;
pin(4) <location = [ "U19-T20", "T19-T18", "R19-R18", "R17-R16" ],
      standard = "Differential 1.5-V SSTL Class I"> DataStrobe;
```

Listing 3.1. Example code used to illustrate counting lines of code in ALCHA

Code between round brackets are counted, which implies that function parameters are counted as individual lines. This is equivalent to counting the port mapping in Verilog code, so it is deemed a fair comparison.

3.3.3.2 Verliog

Counting lines of code in the Verilog implementation involve the following rules:

- Ignore comments
- Ignore anything between matching double-quotes
- Ignore anything in matched braces, but only when the opening brace is not preceded by the “struct” keyword
- Count the remaining semicolons, commas, “begin” and “case” keywords

By ignoring code between matching braces, the counter ignores concatenation sequences. Counting commas will result in counting a function call with many parameters as many lines, but inspection of the code-base shows that this is a rare case that will not significantly influence the result.

Port-mapping is counted as independent lines, because code in round brackets are not ignored. In ALCHA, port mapping is mostly done by means of assignment statements, so this bias was deemed to be a fair comparison between the two languages.

3.3.3.3 Scripting

A significant portion of the Verilog implementation include Python and TCL scripting used to auto-generate HDL modules and specify design constraints. Both of these languages are strongly based on newline characters, which makes counting lines of code relatively straightforward. Scripting lines are counted by the following rules:

- Ignore comments and empty lines
- Ignore anything between matching single or double quotes
- Ignore escaped newline characters
- Count the remaining semicolon and newline characters

3.4 Summary

This work is based on the engineering method. Specifically, it is based on the spiral model as presented by [77]. As such, this dissertation presents a design based on user requirements derived from features of related work, as well as the author’s industry experience.

The resulting design is evaluated against these user requirements, as well as language practicality and power. Language practicality is based on the ability to implement a prototype compiler, and language power is based on the level of maintainability of an implementation of a case study.

Chapter 4

User Requirements and Language Objectives

This chapter presents the user requirements, as derived from the literature studies presented in chapter 2, as well as the author’s industry experience¹. In particular, key language properties are derived by inspecting the features introduced by the new languages presented in section 2.5, working under the assumption that new languages are being developed mainly because there exists dissatisfaction with the status quo. These properties are then used to derive language objectives.

4.1 Language Properties

Section 2.5 presented related efforts in firmware language development. These efforts highlight common desirable language properties, which can be summarised as follows:

Verbosity – Most evident in the efforts of the solutions presented in section 2.5 is a reduction in verbosity and boilerplate. Synflow Cx reduces finite state machine (FSM) verbosity, whereas the Scala-based solutions focus on reducing routing verbosity. VHDPlus and Alchitry Lucid are essentially VHDL and SystemVerilog, respectively, with reduced verbosity and a cleaner, easier to understand syntax.

¹The author started work as lead firmware engineer at Reutech Radar Systems in March 2016, involved specifically in the FPGA based subsystems. Prior to that, the author was working as firmware engineer at NECSA, Pelindaba for five years.

Control Flow – Another common trait is data stream and processing flow abstraction, especially applying structured and procedural programming models to firmware design. Synflow Cx abstracts processing state machines and pipelines by means of Kahn process networks, VHDPlus abstracts finite state machines into “threads”, which allows the user to specify sequential statements.

Architectural Freedom – Both Python-based and Scala-based solutions provide user-defined abstraction by means of an object orientated programming model, thereby facilitating various architectural patterns [86].

Language Adoption – A reduction in verbosity and increase in abstraction inevitably leads to less expensive maintenance, as well as easier language adoption [87]. All four languages presented in section 2.5 aim to reduce beginner confusion, making them easier to learn than traditional hardware description languages (HDLs).

Generalisation – The Python-based solutions target scripting and generalisation, thereby providing algorithmic circuit generation. This capability improves code reuse, in addition to providing powerful abstraction mechanisms. The user can, for instance, auto-generate a register controller circuit based purely on a JSON-based description of the registers; or auto-generate a FIR filter based on the desired filter response.

Low Level Control – Although recent years have seen significant effort from FPGA vendors, as well as academia in general, towards improving high level synthesis, the associated loss in low level control is often undesired. When implementing firmware, it is advantageous to maintain the low level control provided by RTL. All the solutions presented in section 2.5 have this in common.

4.2 Language Objectives

This section presents five language objectives, as well as an argument for each. These objectives are based on the properties highlighted in section 4.1, as well as the author’s industry experience.

4.2.1 Architectural Abstraction

Architectural abstraction satisfies the user requirements of reduced boilerplate and verbosity, as well as reducing the cost of maintenance. A clean abstraction-based architecture is easier to maintain, so the language should support the implementation of such architectures.

This does not imply that the language itself must be at a higher level of abstraction (such as many HLS solutions), but rather that the user should be provided the freedom to create clean abstraction layers.

4.2.1.1 Layered Systems

A popular form of architectural abstraction is layering [29]. One prominent example is the Open System Interconnection (OSI) layers used in communication systems [88]. Layered systems are implemented such that each layer can be designed in isolation. Only the interfaces to the immediate neighbouring layers are important.

Any layer can be exchanged with a different implementation without affecting the other layers. A TCP-based communication system, for example, might run over ArcNET, or Wi-Fi, or Ethernet, or even RS-232 without affecting the implementation of the TCP layer itself [89].

The HDL programming model is strongly based on physical hardware models. Each firmware module exposes ‘ports’ to the containing module. This is akin to pins on an IC package, which provide the interface to the printed circuit board (PCB), or a PCB connector used to interface to the back-plane.

This programming model essentially limits the implementation of a layered system to two options. Either each lower layer becomes a sub-circuit of the layer above it, or all the layers are implemented in entities instantiated at the same level of the design hierarchy. These two options are presented in figure 4.1

The first option (figure 4.1(a) – nested) is often more verbose, because all interfaces to the outside world has to be connected at the top level of the circuit hierarchy. This implies that the lower level interfaces has to be routed through the higher level entities, thereby breaking the isolated architecture (i.e. the high level entities have to have awareness of the low level interfaces).

The second option (figure 4.1(b) – flat) is more popular, because it retains the property of isolated development and makes it easier to swap out interface abstractions for different protocols. The issue here is that the architecture loses the property of abstraction. The developer has to create an instance of each layer at the same level of the circuit hierarchy.

Both options are difficult to maintain. The nested option requires modification to the higher level entities when changing the low level protocols or interfaces. The flat option has significant code clutter and name-space pollution in the top level entity (i.e. lack of encapsulation),

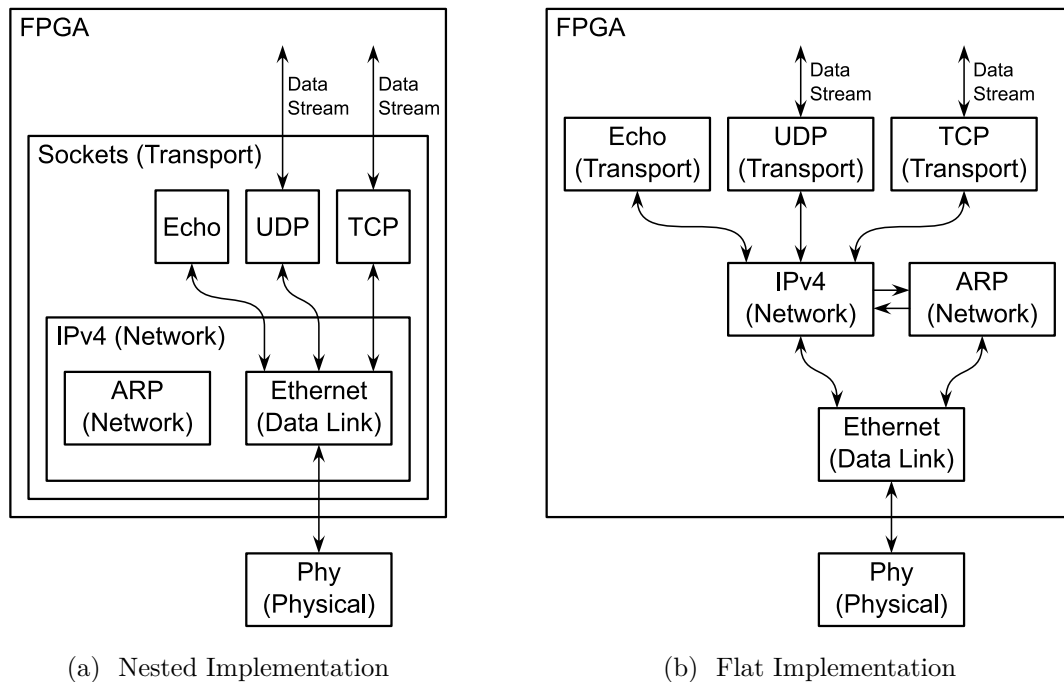


Fig. 4.1. Two options of implementing OSI layers up to the ‘Transport’ level.

making it difficult to read and follow the layout. For another example, Heath *et al.* illustrates this problem for a relatively simple processor design [90].

4.2.1.2 Data Streams

Another form of architectural abstraction is data streams (“pipes and filters” in [29]). The concept is most widely used in digital signal processing applications and packet-switched networks.

In HDL, a stream can be abstracted by means of a structure (or ‘record’ in VHDL) that contains metadata, data, a ‘valid’ strobe, as well as start- and end-of-packet markers. A typical stream timing diagram is presented in figure 4.2.

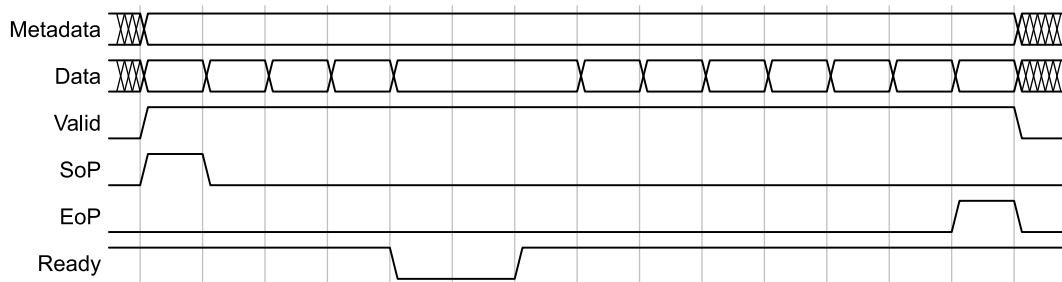


Fig. 4.2. Timing diagram showing the concept of a data stream.

One property of HDL structures is that all members of a structure must pass through a port in the same direction. In other words, all members must be either ‘input’ or ‘output’ relative to the sub-circuit. This is problematic when adding back-pressure to a streaming interface (i.e. the ‘Ready’ signal in figure 4.2), which always run against the data flow direction. This back-pressure signal cannot be defined in the same structure as the rest of the interface, thereby breaking the abstraction.

SystemVerilog introduced the ‘interface’ construct, which can be used to mitigate this issue, but is quite verbose to use. This said, SystemVerilog interfaces are very powerful constructs. The developer can, for instance, define a FIFO interface, including ‘read’ and ‘write’ functions, as part of the interface definition. [91]

4.2.1.3 Project Development

Architectural abstraction provides the means by which systems can be broken into independent modules, or components. Parnas describes the process in relation to design decisions that are likely to change. Volatile decisions should be packaged into modules in such a way that they remain hidden from other modules. [92]

This modular breakdown strategy relates to a popular project development methodology: “top-down design, bottom-up implementation” [93]. This is only feasible if the tools in use support architectural abstraction.

During the early design phases, the project architecture is sketched out in terms of a high level overview. In the ideal modular firmware design, this high level overview is the contents of the top level entity, but this is not possible with existing HDL. In both options presented in figure 4.1, the top level entity is cluttered with low level detail (such as the physical signals routed to the external IC, or instances of the lower level modules themselves).

To illustrate the concept, consider the radar system design in figure 4.3. This diagram contains too much detail. The system overview should not concern itself with volatile design decisions such as the type of storage used for the processor (i.e. flash, SD-card or NVMe), or even the fact that the radar waveform is generated by means of a DAC instead of a DDS or PLL-based synthesiser.

It would be more useful, and more modular, to hide those details in abstraction, as presented in figure 4.4. It is not important at this stage of development whether the data is being logged to SD-Card, or NVMe, or SATA. Nor is it important what type of system controller is employed (i.e. ARM-based SoC, embedded Nios II or basic FSM). The design is even immune to changes in waveform generation strategy.

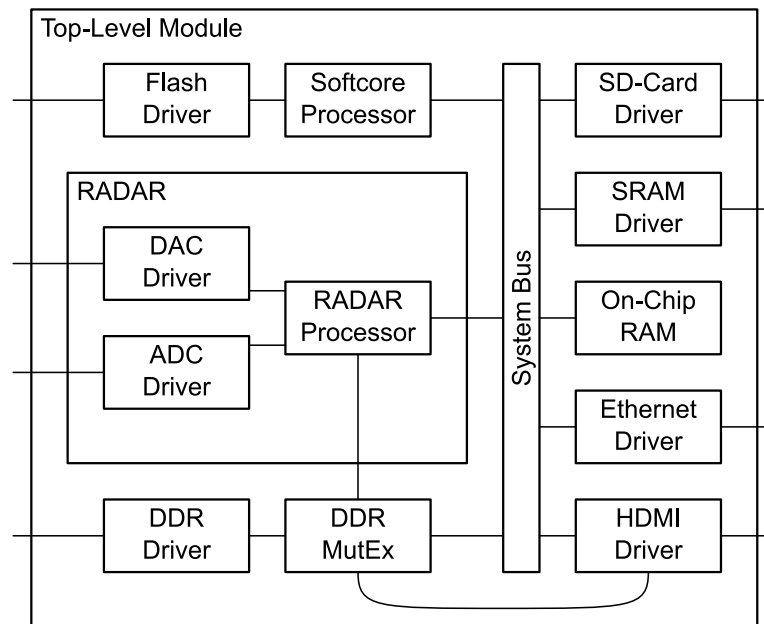


Fig. 4.3. Example of a radar top-level block diagram

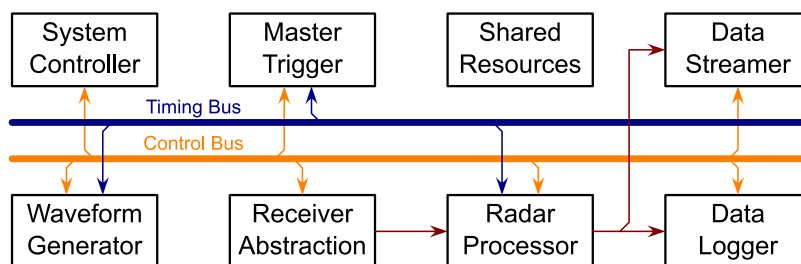


Fig. 4.4. Conceptual overview of a radar

After finalising the broad-strokes system overview, the firmware architect can implement the entire top-level entity by means of black-box modules (i.e. entities that have their interfaces defined, but without a body, or implementation). Each block in the design is a layered system, and the data paths are data stream abstractions.

Each entity's interfaces and behaviour can then be specified independently, after which a test-driven design methodology can be followed on each individual component [94].

Each module can be developed and tested in isolation, even on real hardware. This saves compilation time, because only the module under development (along with its test environment) is included in the design. It is also relatively easy at this point to wrap modules from a library in the black boxes, or even replace the black-box modules with library implementations.

After the development of the individual modules, integration is relatively straightforward. The top-level entity exists, and all the individual modules follow the interfaces specified therein. Integration is a simple case of replacing the black box entities with the newly developed ones and recompiling the project. There is no need for further pin-mapping or design constraints, because those properties are already encapsulated within the modules.

4.2.1.4 Maintenance

The concept of architectural abstraction is also useful for mature projects. If any low-level detail of an existing project must change (due to e.g. part obsolescence or design modernisation), it is relatively easy to exchange the firmware module in-place without affecting any other part of the system.

This remains true as long as the module being replaced is fully abstracted, i.e. no part of it (including its interfaces to the outside world) may be visible to the rest of the system. If, for example, an obsolete DAC is replaced by a DDS in the waveform generator block of figure 4.4, the top level routing and all other modules in the system can remain completely unchanged, even though the internals of the waveform generator module changes significantly.

In addition, the new waveform generator module can be unit-tested in isolation before integration into the larger system, thereby saving a significant amount of compilation time during development.

4.2.1.5 Shared Resources and Interface Abstraction

Figure 4.4 contains an unconnected block “Shared Resources”. This module encapsulates a collections of resources, such as external DDR memory or an internal cache that is shared between the CPU and radar processor.

Traditional HDL requires the developer to connect interfacing signals, in the top level entity, between all the shared resources and all the modules that make use of them. This creates a large amount of routing clutter in large projects.

It would be more convenient and maintainable if the shared resources reside in the global namespace, even though the modules that describe them are defined within the “Shared Resources” module.

One can take this concept even further: the shared resource can expose a globally accessible interface that allows another module to connect itself to the resource. That interface is then responsible to sort out the addressing (if any), arbitration, etc.

This concept is fairly common in software. A simple example is Berkeley sockets, where the process (or function) that wants to make use of a socket (which is an abstraction of the globally accessible TCP/IP stack) simply makes use of the Berkeley socket interface API [95]. No other function in the system needs to know about that connection. A similar argument can be made for any other modern computer shared resource.

HDL does not support this, and neither does any other firmware development tool that adheres to the strict module-and-port hierarchical model.

4.2.1.6 Hexagonal Architectures

In software, especially web development, a common architecture is the so-called “hexagonal architecture”, also known as “ports and adaptors” [96]. A typical use-case is presented in figure 4.5.

The idea is to place the application, or business logic, on a lower (or inner) layer, and all the external interfaces (including the user and the database) on upper (or outer) layers.

At first glance, this concept is very similar to the traditional HDL modular architecture. All the external interfaces are at the top level, whereas the application (i.e. the core processing modules) are hidden inside.

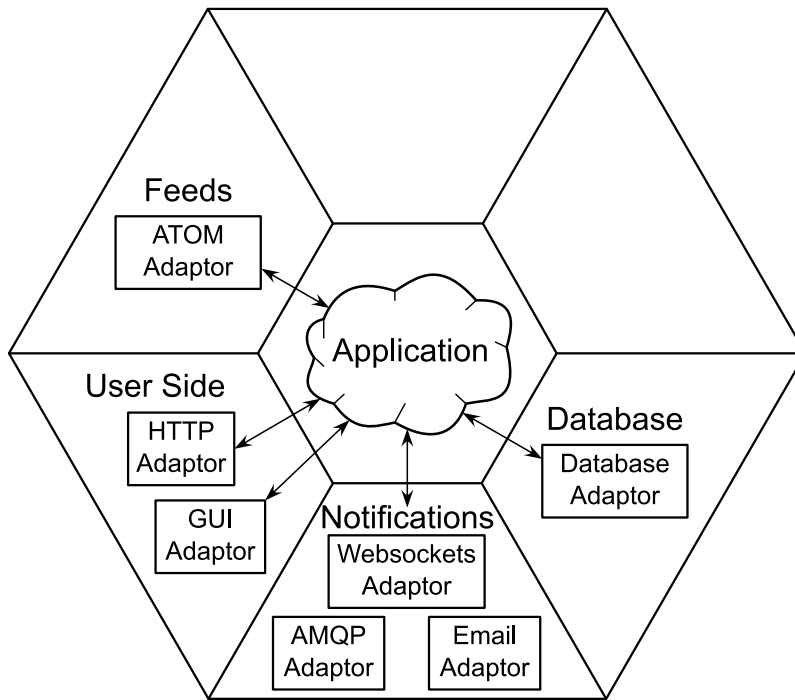


Fig. 4.5. A typical use-case for the hexagonal architecture, adapted from [9]

This analogy quickly deteriorates once one realises that this is only a high-level overview of the system. The exact details of the database (i.e. SQL or flat file or otherwise) is hidden by the ‘Database Adaptor’. Similarly, the low-level socket and email APIs used in the ‘Notifications’ sector is not part of this view of the architecture. The same argument can be made for the other sectors and adaptors.

There is a close correlation between the architecture presented in figure 4.5, and the conceptual overview of a radar presented in figure 4.4. The inner layer is shared between the ‘System Controller’ and ‘Radar Processor’ modules. The other modules take the role of adaptors (including the ‘Shared Resources’ module). The role of ‘port’ is fulfilled by the ‘Control Bus’.

By adding clean architectural abstraction to the language feature-set, the developer is also capable of implementing this form of high-level layering.

4.2.2 Control Flow Abstraction

The user can only make use of structured and procedural programming patterns and paradigms if the language supports control-flow abstraction. In addition, control flow abstraction provides a gentler entry-point for beginners, thereby contributing to flattening the learning curve.

HDL provides little support for describing the circuit in terms of its algorithm, or function. An algorithm must be broken down into a form that describes what should happen during each clock-edge event. [1] pg. 115.

This typically takes the form of a finite state machine, a pipeline, or a combination thereof. Even function and procedure constructs provide only combinational circuit information-hiding (i.e. physical abstraction), and cannot be used for abstracting a sequence of instructions that span multiple clock cycles.

HDL code used to describe finite state machines have considerable boilerplate and are therefore challenging to follow. There is no direct language support for abstracting the control flow, i.e. the user must manually implement state transitions, which is at a similar abstraction level to CPU assembly.

It is possible to refactor a portion of a larger state machine out into a smaller state machine, which is then ‘called’ by the larger one. In firmware, the process involves dedicated control signals between the parent and child state machines, which must be manually managed by the developer. The process is analogous to calling a function on a CPU by manually managing the stack frames and return data paths.

Just like a CPU high level language hides the stack management, jump statements and other assembly-level detail in abstraction, so too a high level firmware language should be able to hide the control signals and other state-machine level detail in abstraction. The user should be able to concentrate on his / her algorithm without worrying about the low level details of implementation. In other words, it should be possible to make use of procedural and structural programming principles when implementing finite state machines [35].

This said, the user should not loose low level control in the process. Exact timing should be implemented by the user, not auto-generated by the compiler. It should also be relatively simple to infer the resulting circuit from the code provided, just like it is relatively simple in the case of a high level computer language to infer the stack-frame management performed by the compiler, even though it is not explicitly present in the high level source.

4.2.3 Fixed-point Abstraction

Many firmware projects have some digital signal processing (DSP) component. Although floating point DSP is becoming more popular as FPGAs grow in capability, it is still advantageous to make use of fixed-point operations instead. Fixed-point operations are low power, uses little resources and have low latency. [97]

All signals in traditional HDL, however, are treated as integers. There is no direct abstraction for fixed-point, i.e. the user has to manually shift, clip and truncate correctly, which is an error-prone process best left to a capable compiler.

A language with built-in fixed-point abstraction will further flatten the learning curve, as well as contributing towards easier maintenance.

The complexity of fixed-point implementations makes optimisation challenging. Any small change somewhere in the design has a ripple-effect throughout the rest of the design. Iterative optimisation techniques are therefore expensive and often neglected.

There are HLS solutions, as described by, for example, Coussy *et al.* [98] and Menard *et al.* [65]. Such high level approaches typically do not provide the low-level control required for cutting-edge applications, however.

4.2.4 Design Constraints

Generally, HDLs are not designed to specify all aspects of firmware design. Hardware constraints, for example, are often delegated to the electronic design automation (EDA) tools used to generate the circuit, which introduces some interesting side-effects discussed in subsequent subsections.

4.2.4.1 Distributed Constraints

Design constraints (pin mapping and timing constraints in particular) are not encapsulated, and therefore breaks the principles of abstraction [34]. The user must maintain more than one place, i.e. the HDL that defines the signal that is connected to the pin, the vendor-provided settings file that maps a physical pin to the signal and assigns various other physical properties², and the Synopsys design constraints (SDC) file that specifies the timing constraints.

This disjointed design entry pattern makes it challenging to maintain the system. Every update must be effected in all three distinct places. It would be more convenient if all the pin parameters can be specified as part of the pin declaration, for instance.

Integrated design constraints would contribute to easier maintenance, as well as increased code reuse. All relevant design constraints can be integrated as part of a library module,

²This includes, but is not limited to: I/O standard, drive current, pull up / down resistors, matching resistors, high speed channel compensation parameters and PCB trace model parameters.

therefore putting all maintenance of that module in one place, as well as making it simpler to reuse in various different projects.

It should be noted that modern vendor-provided tools such as Quartus and Vivado implement integrated design constraints by means of signal attributes included in the VHDL or Verilog source. This solution is vendor-specific and not all functionality is supported by both vendors³.

4.2.4.2 Constraint Coupling

A related problem is timing constraint coupling. External timing constraints have two independent parts: timing related to the PCB traces, and timing related to the external device that communicates with the FPGA.

Traditionally, these two aspects are combined into a single timing constraint. This makes it challenging to reuse existing peripheral abstraction entities, because the timing constraints of the existing implementation must first be decomposed into the two aspects before being modified for application to new hardware.

For this reason, many designs make use of TCL variables to split the external timing requirements into separate parts [99], but this falls back to the distributed constraints problem highlighted in section 4.2.4.1.

The problem is exacerbated by third-party library developers who might want to provide closed-sourced solutions. The existing model necessitates the library vendor to include detailed instructions on how to set up the timing requirements for the external device the user wants to control.

It would be more convenient if the constraints relating to the PCB (i.e. pin mapping, PCB trace timing requirements, etc.) are specified as part of the pin declarations, and the timing requirements that relate to the external device are specified as part of the peripheral abstraction module source code. This way, the library includes all the requirements except the PCB trace model, which the user can then specify as part of the pin declaration.

³Intel Quartus supports pin assignment by means of HDL attributes, whereas Xilinx Vivado does not. Additionally, Quartus allows the user to apply any property that can be applied by means of TCL-based assignments in the settings file by means of a general `altera_attribute` attribute. Vivado has no equivalent in the 2021 version.

4.2.4.3 Multicycle Paths

Many designs contain schemes to allow certain combinational calculations to span multiple clock-cycles [60]. The compiler, on the other hand, always assumes that all results must be valid on the next clock edge, resulting in more stringent timing requirements than necessary [100]. It is up to the user to specify otherwise.

These multicycle path constraints are often neglected, resulting in more work for the fitter⁴. The result is longer compilation times and a less-than-optimal circuit. If automatic multi-cycle path detection can be implemented, the design will meet timing requirements easier, resulting in faster compilation times and less resource usage.

4.2.4.4 Placement Constraints

On larger designs, it is common to constrain various subsystems to specific placement on the FPGA device. This limits the domain of the fitter, thereby speeding up the process. It also has the added benefit of using the developer's cognitive insight and pattern-matching abilities as part of the compile process, thereby resulting in a superior layout.

A clean layered abstraction is advantageous for specifying these constraints. The placement constraint could, for instance, be specified as part of the module instantiation. In the case of a well-defined layered system, each placement constraint is only specified in one place for each subsystem.

4.2.5 Generalisation

The more general a module can be made, the more reusable it is. Traditional HDL modules are challenging to generalise. Module parametrisation is based on instantiation constants, with limited support for algorithmic circuit generation.

This model makes sense when developing code for ASIC devices. Updating a module within an ASIC device is expensive, so each unit cell (or module) is vigorously tested and verified in isolation [101]. The ASIC device is then developed similarly to a large physical hardware system, where each subsystem interface is strictly defined.

⁴The fitter is a stage in typical firmware build chains. The analyser parses the source code into a convenient internal data structure. This is followed by the synthesiser, which translates that data structure into a netlist. The fitter then performs a place-and-route operation for the target device, after which the assembler converts the resulting circuit to a bit-stream that can be programmed onto the FPGA.

FPGA firmware, by contrast, is inexpensive to modify and much more flexible, and therefore require a more software-like programming model in order to leverage this flexibility.

4.2.5.1 Parametrisation

Traditional HDL supports the parametrisation of modules, but this is generally limited to signal widths, simple constants and `generate` blocks.

Consider the case of generalising a FIR filter, for example. In traditional HDL, the developer is able to generalise the length of the filter by means of an instantiation constant, as shown in listing 4.1. The problem comes in when one considers the contents of the coefficients.

```

module FIR_Filter #(
    parameter N      = 8,
    parameter MIF    = "FIR8.mif",
    parameter Vendor = "Altera"
)(
    input Clk,
    input Reset,

    input  [17:0] Input,
    output reg [17:0] Output
);

parameter AddressWidth = $clog2(N);

reg [AddressWidth-1:0] CoefAddress;
wire [17:0] Coefficient;

ROM #(.Vendor(Vendor), .N(N), .MIF(.MIF)) Coefficients(
    .Clk      (Clk),
    .Reset    (Reset),
    .Address  (CoefAddress),
    .Output   (Coefficient)
);

// The rest of the FIR filter implementation.

endmodule;

```

Listing 4.1. Example showing how a FIR filter might be generalised in Verilog.

Typically, a ROM module is instantiated by using the vendor-provided wizard (or code template) rather than depending on compiler inference⁵. By using the vendor-provided mechanism, the developer gains access to advanced features of the block not available through pure HDL.

⁵The ROM module in listing 4.1 is a vendor-neutral wrapper. Inside this wrapper is a `generate case` block that switches between code templates depending on the vendor specified.

This ROM module is then initialised by means of a memory initialisation file⁶ (MIF). The MIF must correspond to the parameters of the ROM module, so by changing the N parameter in listing 4.1 also necessitates a modification to the MIF.

It is therefore impractical to have more than one instantiation of the FIR filter in the same design, unless one also passes the filename of the desired MIF as a parameter, as illustrated in listing 4.1.

The typical design involves the auto-generation of the MIF by means of a scripting tool (such as Python or Matlab). The filter is designed according to desired filter parameters. the scripting tool then provides a plot of the filter response in relation to the ideal filter parameters, as well as the auto-generated MIF.

It would be convenient, however, if this auto-generation step can be part of the firmware code-base instead of a separate process. It is not possible to do this with existing HDL tools, which leads to the concept of algorithmic circuit generation.

4.2.5.2 Algorithmic Circuit Generation

One of the more powerful techniques used to generalise a firmware design is through auto-generation scripts [102]. As an example, consider control registers. A large design can have a register-count in the high hundreds, but the general architecture of registers is always the same for each case.

In addition, these firmware registers must be mapped by the software that controls them. There is therefore a necessity to maintain a C++ header file (or Python module, etc.) that performs the mapping.

Finally, the firmware must be documented. Part of the documentation generally includes register tables (i.e. which register resides at which address).

If the basic properties of the registers (such as direction, width, functional description, etc.) is specified by means of a JSON5 [103] file, that file can be provided as source to a Python script that automatically generates the HDL source files, C++ header files and L^AT_EX documentation files, as shown in figure 4.6.

The large advantage in doing this is to put all the maintenance in one place. Whenever a register must be added, removed or modified, the developer can simply change the JSON5 description and re-run the build chains (all three of them: firmware, software and

⁶Xilinx Vivado calls these ‘coe’ files.

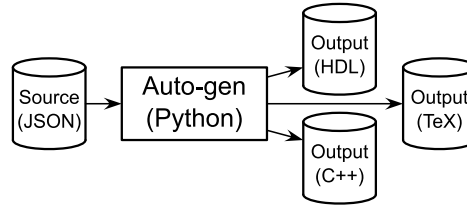


Fig. 4.6. Python-based auto-generation build chain

documentation). The same concept can be applied to other circuits, such as FIR filters, other DSP components and timing generators.

One disadvantage of this approach is that each project has its own register JSON5 document, even though many of the modules being controlled is reused from a library. The problem is mitigated somewhat by structuring the registers into a hierarchy of banks (one for each module), but this still requires the developer to modify at least two places (more if the library module is used in more than one project) during maintenance.

It would be more convenient if every module can specify its own registers, and the auto-generation script then builds the complete project-wide register map and generates the various source files appropriately.

4.2.5.3 Platform and Target Independence

Another aspect of generalisation is platform and target independence. The platform relates to the system that the compiler runs on, while the target refers to the FPGA device and attached hardware. It should be possible to switch platforms without any modification to the general code-base. It should also be possible to switch target with minimal effort. This sense of platform and target independence is the drive behind many software development systems, e.g. C, C++ and Python [104, 105, 30].

Target independence is made challenging by the presence of existing vendor-specific (and sometimes device-specific) libraries. A change in device implies a change in library. It should therefore be possible to generalise an existing library by means of something akin to a global constant or compiler switch. This could be something similar to, but more expressive than, the existing `generate` statements supported by Verilog and VHDL.

4.2.5.4 Automated Design Constraints

Another use case for auto-generation is pin-mapping. The PCB developer provides the firmware developer with a netlist exported from his / her CAD system. The firmware developer then feeds this netlist to an auto-generation script that produces the TCL code required by the FPGA vendor tool.

Such automated design constraints can be extended to timing parameters. Most PCB CAD tools can export a matrix of trace lengths, which can be used to generate the external timing parameters.

4.2.5.5 Integrated Scripting

It would be quite convenient to include the auto-generation and generalisation capabilities described above as part of the core feature set of the firmware language. This is the motivation behind Python-based HDLs such as MyHDL, Migen and YAHDL [39, 3, 71].

These Python-based tools model the firmware as an internal data structure within the Python script. This implies that all the data-structure manipulation abilities available in Python is accessible to the firmware developer, which can be used to implement superior parametrisation and algorithmic circuit generation.

As presented in section 2.5.1, these Python-based solutions can be quite verbose at the HDL level. It would therefore be convenient if the same scripting features are implemented as part of the core language feature set of an HDL.

4.3 Summary

After considering the user requirements as derived from related language design efforts, this chapter presented arguments for five language objectives, namely: architectural abstraction, control flow abstraction, fixed-point abstraction, integrated design constraints and generalisation features. These objectives are used in the remainder of this dissertation to structure the design and evaluation.

Architectural abstraction was introduced by means of two popular patterns, namely layered systems, data streams. The advantages of a language that supports many architectural patterns was further highlighted in terms of typical project development flows and issues surrounding maintenance. Flexible architectural options allow the developer to implement

clean abstraction, thereby isolating functionality that is more likely to change, which in turn reduces the cost of maintenance.

Control flow abstraction was presented in relation to finite state machines. Existing HDLs have few mechanisms to assist in refactoring large state machines into smaller blocks, and existing solutions are often verbose and cumbersome to follow.

Most FPGA based digital signal processing require some form of fixed-point processing. The ALCHA language should feature some form of automation to make it easier to implement the scaling and low-level shifting required.

In existing FPGA development, design constraints are typically separate from the source code, thereby increasing the number of places requiring modification during code maintenance. This workflow is well-suited to ASIC development, which is typically more fixed, but ill-suited to more flexible and reprogrammable FPGAs.

And finally, there is significant effort spent in automating the design flow and increasing implementation generality by means of third-party scripting. It would therefore be advantageous if the main programming language used for development supports similar automation and generalisation features.

Chapter 5

Design

Chapter 2 presented background information, and chapter 4 specified the user requirements, language objectives and motivations for the ALCHA language. This chapter continues the journey by presenting the design of ALCHA, structured according to the five language objectives formulated in section 4.2.

In order to keep this chapter focused on the more novel contributions of ALCHA, much of the detail have been moved to the appendices. The interested reader is referred to appendix B for design decisions, appendix E for the language grammar (EBNF) and appendix F for the abbreviated language specification.

5.1 Architectural Abstraction

Section 4.2.1 established the need for cleaner abstraction. The language should support layered systems, bus encapsulation and the hiding of globally shared resources.

In particular, low level detail does not belong in high level modules, and vice versa. Internal details of layered systems should not influence the external interface, the multitude of signals present in bus architectures should be abstracted to a single object and globally accessible low-level resources and services should not clutter higher level components.

This section discusses the basic programming model of ALCHA, which aims to satisfy the architectural abstraction objective. Parts of this section was presented at the IEEE Radar Conference in 2019 [27].

5.1.1 Moving Pins into Abstraction

The major property of traditional HDL that limits architectural abstraction is that pins have to be defined as the ports of the top-level entity. The obvious first step, therefore, is to make it possible to define pins anywhere in the code-base.

This would allow the user to implement clean layered systems, follow a top-down design and bottom-up implementation strategy in project development and easily perform maintenance on leaf modules without influencing higher-level modules.

5.1.1.1 Modelling Pins

It could be argued that pins are simply nets¹ with a pin location associated. It is not quite this simple, however. Pins should be modelled as a structure of three nets, as depicted in figure 5.1. The `enable`, `driver` and `pad` are `net` types.

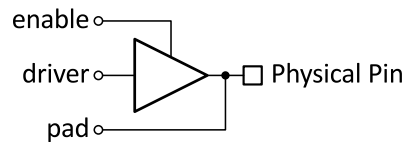


Fig. 5.1. ALCHA pin structure

Traditional HDL abstracts this concept by using multi-valued logic², which is convenient in simulation but confusing when synthesising hardware. Code that models the pin components separately and explicitly with two-valued logic has the following advantages:

- By inspecting the values assigned to the various components, the compiler can infer pin direction [26]. This is useful for pins of structured type (like a GMII interface) where the structure members do not all have the same direction.
- A two-valued model is closer to the reality of the hardware, thereby making the code more representative of the synthesised circuit.
- All compiler rules that apply to nets can be applied directly to the pin components, thereby removing many edge-cases from the implementation of the compiler.
- The code can be written more explicitly, thereby avoiding misunderstanding between the user and the compiler.

¹ALCHA uses ‘net’ to mean the same thing as a Verilog ‘wire’, ‘reg’ or ‘logic’; or a VHDL ‘signal’ or ‘variable’ of synthesisable type.

²Verilog is 4 valued (high, low, high-impedance and don’t-care) [21], whereas VHDL is 9 valued (uninitialised, strong-unknown, strong-high, strong-low, high-impedance, weak-unknown, weak-high, weak-low and don’t-care) [17].

This said, it is tedious to always refer to the pin members when using them in expressions and assignments. For this reason, ALCHA implicitly uses the `driver` member when the pin is used on the left-hand side (LHS) of an assignment, and the `pad` member when the pin is used in an expression. This is consistent with traditional HDL semantics.

5.1.1.2 Virtual Pins

A pin with no location assigned to it is a useful concept. Both Quartus and Vivado model this as a ‘virtual pin’, which causes the compiler to not optimise away that node, even though it is not associated with a physical pin. This is useful for development purposes:

- The developer can obtain a good estimate of resource usage without connecting the upper layers of the system.
- The firmware under development can be synthesised and analysed by means of an embedded logic analyser (e.g. ChipScope and SignalTap) [36, 37].

Vivado and Quartus requires the user to explicitly assign a design constraint in order to make the pin virtual, otherwise the compiler automatically assigns an unused pin. This is dangerous: if the user forgot to assign a pin to a specific location, the compiler-chosen location could cause hardware damage³. For this reason, ALCHA automatically makes all unassigned pins virtual.

5.1.1.3 Syntax

Pins must be distinguished from nets, so a dedicated `pin` keyword is used, as illustrated in the example in listing 5.1. Pin declarations are legal anywhere a net declaration can be used, which implies that the user can hide all pin declarations in any form of abstraction he / she desires.

```
pin    Clk;      // 50 MHz
pin( 8) LED;     // Defines an 8-bit pin
net(26) Counter; // Defines a 26-bit net

rtl(Clk) Counter++; // Increments "Counter" once every clock edge
LED = Counter >> 18; // Assigns the top 8 bits of Counter to LED
```

Listing 5.1. Example of a simple counter circuit in ALCHA

³The compiler could assign an output pin to a location routed to the output of an external IC. The two output pins fight, causing large currents and possible pin damage.

5.1.2 Modular Design

ALCHA does not require the user to package circuits into an entity or module. The example in listing 5.1 is complete and equivalent to the Verilog in listing 5.2.

```
module main(  
    input    Clk,  
    output [7:0] LED  
);  
    reg [25:0] Counter;  
  
    always @(posedge Clk) Counter++;  
  
    assign LED = Counter[25:18];  
  
endmodule
```

Listing 5.2. SystemVerilog equivalent of listing 5.1

In order to support architectural abstraction, it must be possible to package the code-base as a hierarchy of components. It is customary in most (if not all) languages to package these components in separate files.

5.1.2.1 Translation Units

C (and C++) has the concept of translation units. Each source file is translated into an object, independently of the other source files (apart from the declaration headers). After all the objects have been compiled, they are linked together in order to form the executable program. [104]

This model proved problematic when the generalisation features (see section 5.5) were considered. It is not possible to fully compile a source file prior to knowing how it is parametrised. In essence, the scripting part of the source has to be dynamically interpreted in order to function as desired, so it is impractical to compile each source file independently and then link them afterwards.

5.1.2.2 Modules

The second option considered was to use something loosely based on Python modules [106]. The “import “file”” statement in ALCHA is essentially equivalent to saying “from file import *” in Python⁴. Similarly, the “import “file” as name” in ALCHA is equivalent to saying “import file as name” in Python.

⁴The semantics of this is equivalent to simply parsing the imported file as if it was written in place of the import statement. This was a deliberate design decision, which is substantiated later.

The Python interpreter ensures that there is only one instance of a module, no matter how many times it has been imported by various other modules. This essentially implies that Python implicitly follows a singleton pattern as far as modules are concerned. [106]

This model works well in most cases, but falls short when considering changing a low-level aspect of the design – especially when that low level feature is being used in many different places.

5.1.2.3 Global Resources

Consider the target hardware import example in listing 5.3. The main purpose here is to allow an easy switch between development and deployment target platforms. The concept is equivalent to operating system headers in C / C++. In this case, `DevKit.alc` and `PCB_8473_v3.0.alc` are two files that define the same pin names, but with different design constraints.

```
!! Choose a target:
import "DevKit"
// import "PCB_8473_v3.0"

net(26) Counter;
rtl(Clk) Counter++;
LED = Counter >> 18;
```

Listing 5.3. Example of using `import` to switch platforms

Every module that requires access to pins must import either the `DevKit` or `PCB_8473_v3.0` module. If the user wants to switch to a different hardware platform, many modules must be modified.

The simple solution to this problem is to introduce another level of indirection. One form of indirection that could be used is an intermediate ‘target’ module that contains the choice of target description module. Another form is a configuration module that contains a constant designating the intended target.

At first glance this is a good solution, but quickly becomes cumbersome in large projects – choice of target is only one example, there are also various other global resource modules that might require similar indirection.

5.1.2.4 Global Namespace

Chisel and SpinalHDL have solved the problem of global clock resources quite neatly. Every instance of a register or sub-circuit inherits the clock domain of the parent module in which it

has been instantiated [107, 8]. Cx has a similar mechanism for clock inheritance throughout the module hierarchy [4].

ALCHA generalises these concepts to include all declarations, not only clocks. Everything declared within the parent module is directly accessible by all subsequently imported modules, without any of the imported modules requiring further import statements.

This is not a new idea – $\text{\LaTeX}$ makes use of the same programming model. The `\documentclass`, `\usepackage` and `input` statements in $\text{\LaTeX}$ are all equivalent to the `import` statement in ALCHA.

This $\text{\LaTeX}$ -like model has consequences relating to compile times. The model does not allow the compilation process to make use of “translation units”, which it then links together later. All the code has to be interpreted and compiled, in full, every time.

The advantage in flexibility, however, makes it worth the price. In addition, the ALCHA compile-time is still expected to be much faster than the rest of the FPGA build process, especially the fitter.

5.1.2.5 Namespace Model

The global namespace concept lends itself to namespace clutter, which is undesired. The namespace-specific form of the `import` statement solves this issue. The parent module must refer to the objects defined within the imported module through the assigned namespace. Within the imported module, a reference to an object will cause the compiler to search through the namespace stack. The local module is searched first, after which the parent namespace is searched, and so forth.

One effect of this is that a local declaration hides a higher declaration of the same name. This is not considered a problem, because it can easily be solved by means of an appropriate naming convention.

5.1.2.6 Object Orientation

The mechanism of importing modules into namespaces provides a convenient way to build a circuit hierarchy, but it is very limited. It does not, for instance, support abstraction of streams and buses, and while parametrisation is possible, it is not ideal⁵.

⁵Import parametrisation requires the user to set up the environment by means of specific scripting constants and aliases, which has obvious drawbacks from a code readability point of view.

In order to solve these drawbacks, object orientation support was added to the ALCHA feature set [27]. The general syntax is presented in listing 5.4.

```
class MyClass(net Clk, net Reset): BaseClass1(Clk, Reset), BaseClass2(){
    net(8) A; // A public member (no specifier)
    private net(8) B; // A private member
    protected net(8) Y; // A protected member

    private{
        !! Private declarations
    }
    protected{
        !! Protected declarations
    }
    public{
        !! Public declarations
    }
    // All other declarations are public by default

    // Statements form part of the class constructor
    rtl(Clk, Reset) Y = A + B;

    // Member functions have direct access to local members
    void AssignB(net(8) N){
        if(N > 123) B = 0;
        else      B = N;
    }
}
```

Listing 5.4. An overview of the ALCHA class syntax

In principle, the class declaration itself serves as the constructor. The class parameters are specified during class instantiation, whereas members defined within the constructor body are specified and / or used after instantiation.

There is no concept of a destructor in ALCHA, because a circuit instance cannot be destroyed. One can argue that a purely scripting class (i.e. a class that does not contain any synthesisable components) can be destroyed, but that is a use-case that is sufficiently rare that it was not considered useful. A pure scripting class that falls out of scope is simply removed from memory, without automatically calling a destructor.

The class parameters are semantically equivalent to private class members. Any class member function can make use of those parameters directly, but they remain hidden outside the class unless assigned to a public member.

Functions are presented in section 5.2. For now it is enough to note that the `AssignB` example in listing 5.4 is purely combinational, and that `N` is passed by reference. Whatever net is given to the function is assigned to `B` in a combinational hardware sense, i.e. continuously.

5.1.2.7 Access Specifiers

By default, any public class member is bidirectional, i.e. the member object can be read and / or modified by any external statement. This is often undesired, and the compiler should be able to protect the class from this kind of access.

The user could define the net as private and then make use of ‘getter’ and / or ‘setter’ functions in order to enforce access rules, but it is simpler to make use of the `input` and `output` access specifiers instead. These specifiers are used similarly to `private`, `protected` and `public`. A member that has been defined as `output` is read-only (i.e. no external statement can modify it), and a member that has been defined as `input` is write-only (i.e. no internal class statement can modify it).

5.1.3 Dependency Considerations

The proposed L^AT_EX-like programming model has various issues related to dependencies. A particular lower level module might become dependent on the parent having imported some other module, which removes the dependency control from the low-level module.

The trade-off in this case is between flexibility and explicitness. Languages like C++ and Python require the user to explicitly import all dependencies, which is the explicit option. The dependencies are fixed, and the module can therefore only be used where those specific dependencies exist.

5.1.3.1 Hexagonal Architectures

There are various use cases where the choice of lower level dependency is a system-level design decision based on the target hardware. One example is the choice of controller: e.g. ARM SoC vs. NIOS II embedded processor (or similar) vs. a simple FMS-based controller.

In this example, all three options provide the same (or equivalent) interface to control all the other modules, and all other modules are dependent on those interfaces existing.

It could be argued that all modules being controlled by the controller should actually be dependencies of the controller, not the other way around. Assume for the moment that the developer is following the hexagonal architecture principles as described in section 4.2.1.6.

One could route all the interface connections in the higher level module, but that would put low level detail (i.e. the control bus routing) in the higher level, thereby failing to satisfy the objective of information hiding.

5.1.3.2 Interface Pattern

The software world solves this issue by means of the interface-based architectural pattern (also known as the adaptor pattern) [108]. Each module imports a dependency that defines an abstract interface base class. The implementation, which is a derived class, is typically implemented in the controller. In order to obtain an instance of the class, there are two options.

The first option is to create instances of the derived classes in the higher-level module, but this puts the routing at the the wrong level again. The solution (and the second option) is to make use of the factory pattern [108].

The concept behind the factory pattern is that the interface base class includes a virtual function that, when called, returns a new instance of the correct derived class. This quite neatly hides the interface routing.

5.1.3.3 Global Namespace Model

The interfaces and factory patterns have the general feel of a workaround. It makes use of quite a convoluted scheme to implement what is essentially a simple concept. It is used to allow a module to be dependent on a parallel or higher level interface without having direct access to the implementation of that interface.

This workaround is required because of the strict downwards dependency model typically enforced by software languages⁶. By using the global namespace model of ALCHA, there is no need for all this indirection. The interface simply exists, because the controller (in the example) is expected to implement it. If it does not implement the interface, the compilation will fail because the controlled module expects it to be there.

⁶The downward dependency rule applies well to software and is a very useful rule to enforce. The problem is that firmware requires a different approach simply by the nature of the target hardware architecture.

5.1.4 Combinational Assignments

Section 4.2.1.5 presented a scenario where it could be useful to be able to modify a global net from various places. Furthermore, it should be possible to distribute combinational assignments across the design. Such improvements would allow the user to implement architectural patterns that better facilitate data streams and the ports-and-adaptors paradigm. This subsection explores how the combinational programming model can be adapted in order to achieve this.

5.1.4.1 Distributed Assignments

Traditional HDL groups algorithmic combinational statements (i.e. combinational circuits built by means of for-loops, if-statements, etc.) into a single process block (known as an ‘always’ block in Verilog). These blocks model an event-driven architecture, which is useful for simulation purposes [1].

ALCHA is first and foremost a synthesis-oriented language. As shown in previous examples, ALCHA combinational assignments can occur anywhere in the code, which allows the user to group the assignments by whatever means the current project requires.

One use case for this freedom of grouping relates to multiple modules connected to an I²C bus. The signals of the bus can be defined as globally-accessible signals, and each module can attach itself to the bus by means of the code in listing 5.5.

```
// Global definition and initialisation:
group I2C{ // A group is sort of like a C "struct"
  pin<pullup = true> Clk = 0; // The attribute adds an external pull-up resistor
  pin<pullup = true> Data = 0;
  Clk .enable          = 0; // Default to high-impedance, i.e. "high"
  Data.enable          = 0;
}

// Local signals in each module:
net Clk, DataIn, DataOut;

// The module can now attach itself to the bus:
I2C.Clk .enable |= ~Clk;
I2C.Data.enable |= ~DataOut;
DataIn = I2C.Data.pad;
```

Listing 5.5. I²C bus attachment

The statements that connect the peripheral abstraction to the bus is grouped with the peripheral abstraction, instead of some higher-level module. If, for example, a future design removes some devices from the bus, and adds some others, the new instances sort out the bus connections without user intervention.

5.1.4.2 Semantics

ALCHA net objects are implemented as references to netlist data structures, which take the form of expression trees. In the simplest case, any assignment replaces the previous assigned expression with the new expression, but things are not always that simple. Consider, for example, the case where a net is used in an expression before a source is assigned to it. Such a scenario is common in HDL design, especially when there are feedback paths. A simple example is presented in listing 5.6 and figure 5.2.

<pre> net(8) A; net(8) B = 3; net(8) C = A + B; </pre>	<pre> A = 5; </pre>	<pre> net(8) D; A += D; </pre>	<pre> net(4) E, F; A = E * F; </pre>
(a)	(b)	(c)	(d)

Listing 5.6. Example used to illustrate use-before-assign semantics

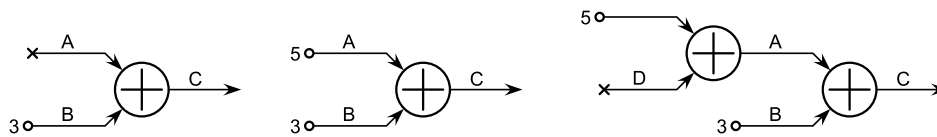


Fig. 5.2. Equivalent circuits of listings 5.6(a) to 5.6(c)

The code in listings 5.6(a) and 5.6(b) is relatively straightforward. Listing 5.6(c) presents a problem: should *C* keep the previous circuit assigned to it, or should it make use of the new *A* instead?

ALCHA is a hardware description language and, as such, is used to describe the connections of physical hardware. A net in ALCHA is simply an abstract model of a physical piece of wire. When viewed in this light, the solution is obvious: replace the expression that drives the net and leave all other connections unchanged, as shown in figure 5.2.

This rule implies that listing 5.6(d) results in a dangling circuit, as shown in figure 5.3(a), even though the user might have meant figure 5.3(b). This is not really a problem, it simply means that ALCHA does not support the reuse of temporary wires. The user must declare a new wire for each case.

5.1.4.3 Removing Circular Circuits

The semantics presented in section 5.1.4.2 presented the rules behind normal assignments, as well as modify-and-assign statements. There is a case that has not been covered: circular dependency.

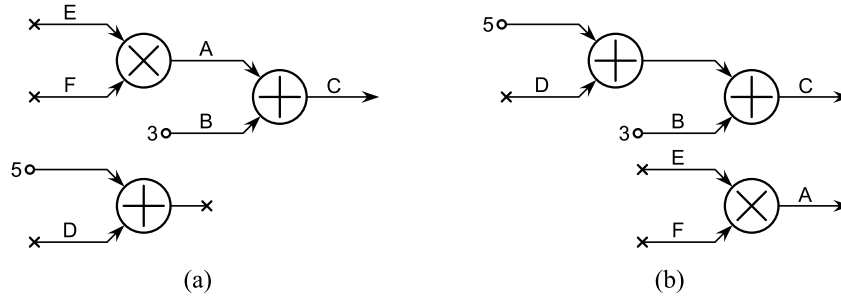


Fig. 5.3. Possible equivalent circuits of listing 5.6(d)

The semantic rules of section 5.1.4.2 can be adapted into the following algorithm:

1. Evaluate the right-hand-side (RHS) expression to produce an expression tree.
2. Inspect the left-hand-side (LHS) and the expression tree to determine whether or not the statement is circular.
3. If it is circular, update the RHS expression tree to use the current expression assigned to the LHS net, instead of a reference to the net itself.
4. Assign the modified expression tree to the LHS net.

This algorithm resolves circular combinational circuits only when the circular reference is within one statement. If the circular reference spans multiple statements, as presented in listing 5.7, the compiler will not remove it, which is something to watch out for.

```

net(8) A;
net(8) B = A;
A = B + 1;

```

Listing 5.7. Circular combinational circuit example

The major unknown at this point is user intention. The user could, for instance, use net B inside an 'rtl' block, at which point A becomes the reset value of B. Because the reset is synchronous, the combinational loop is broken.

5.1.4.4 Inferred Latches

FPGAs are primarily register-based. There is no direct hardware support for latches, although they can be emulated. The FPGA programmer must therefore ensure to write code that does not infer latches. FPGA code that do infer latches are more often than not unintended. [109]

Traditional HDL semantics dictate that the target signal should keep its previous value unless otherwise assigned. In other words, any combinational circuit that contains a decision tree

that is not fully populated (i.e. there exists a path that does not assign something to the target signal) causes the compiler to infer a latch.

ALCHA, by contrast, treats all unpopulated paths of a combinational decision tree as “don’t care”, thereby allowing the compiler to optimise the circuit [110].

5.1.4.5 Typecasting

It is sometimes required to typecast an object to another type. A literal must be cast to a net of a specific width in a bit-wise concatenation, for example. This is achieved by means of the at symbol (`@`), followed by the target type.

Even though classes are generally automatically cast based on their inheritance graph, a class can also be explicitly cast, as in `Myclass@some_other_type`. In order to cast something to a net of specific width, the user could use the full net type, as in `123@net(8)`, or simply the width, as in `123@8` (the former is easier to read, whereas the latter is more convenient for the experienced user).

5.1.5 Registers

Registers in ALCHA is based on ideas from Alchitry Lucid, Chisel and SpinalHDL, as introduced in section 2.5. The associated reduction in verbosity would reduce maintenance cost, because the resulting code is less cluttered and easier to read.

5.1.5.1 Syntax and Boiler Plate Reduction

Traditional HDL contains a relatively large portion of boilerplate code when it comes to register transfer logic (RTL), as shown in listing 5.8. In the example, the reset is local⁷ and synchronous, as recommended by [111].

The HDL description in listing 5.8 is not quite correct – the `Enable` is supposed to take priority over the synchronous reset [112]. This said, the circuit presented is preferred: it ensures that the reset signal is caught regardless of the state of `Enable` (as long as it is asserted for longer than one clock cycle).

The overhead is minimal. The compiler synthesises a combinational circuit between `LocalReset` and `Enable`, which is then fanned out to the various clock-enable ports of

⁷Localised resets reduce the reset fanout, thereby making the circuit easier to route.

```

reg [7:0]A;
reg LocalReset;

always @(posedge Clk) begin
    LocalReset <= Reset;

    if(LocalReset) begin
        A <= A_ResetValue;

    end else if(Enable) begin
        // Put the RTL description here
    end
end
end

```

Listing 5.8. Verilog RTL boiler plate, showing a synchronous reset scheme

the A registers. The `LocalReset` signal is fanned out to the dedicated synchronous clear ports of the registers directly.⁸

Alcha hides this boilerplate in abstraction, as presented in listing 5.9. The `Reset` and `Enable` signals are optional.

```

net(8) A = A_ResetValue;

rtl(Clk, Reset, Enable){ // Reset and Enable are optional
    !! Put the RTL description here
}

```

Listing 5.9. ALCHA RTL construct

5.1.5.2 Semantics

The register A in listing 5.9 is reset to whatever expression has been assigned to it at the entry to the `rtl` construct. If the net is not initialised, the ALCHA compiler will synthesise a register without a reset (i.e. it will reset to “don’t care”)⁹.

All statements within an ALCHA `rtl` construct are evaluated with non-blocking semantics, as recommended by [115]. In principle, all the RHS expressions are evaluated and then the assignments happen, all at the same time, on the rising edge of the clock signal. If the application requires blocking semantics (such as VHDL variables), a combinational circuit can be used instead. If there is no assignment specified for the current circuit state, the register keeps its previous value, which is consistent with Verilog and VHDL RTL semantics.

⁸This has been confirmed by means of a test compile on Quartus (Cyclone V target). The resulting circuit has been analysed by means of the Quartus RTL Viewer and Chip Planner tools.

⁹Reset-less registers are advantageous during circuit synthesis, because it reduces the fan-out of the reset net [111]. It is also required when making use of registers that do not have dedicated reset capabilities, such as the HyperFlex registers in Altera Stratix 10 devices [113], [114].

As far as the synthesised netlist goes, an ALCHA `rtl` construct inserts a register at the head of the expression tree of the target net. External to the `rtl` construct, the same rules apply as for combinational circuits, as described in section 5.1.4.

Listing 5.10 shows an example use case and figure 5.4 its equivalent circuit. The first assignment creates a combinational circuit, which becomes the value of `A` upon reset. The second creates a register and assigns the output to `A`. The third creates a combinational circuit and assigns the output to `A`.

```

net(8) B, C;
net(8) A = B + C;

rtl(Clk, Reset) A++;

A += B;

```

Listing 5.10. RTL example

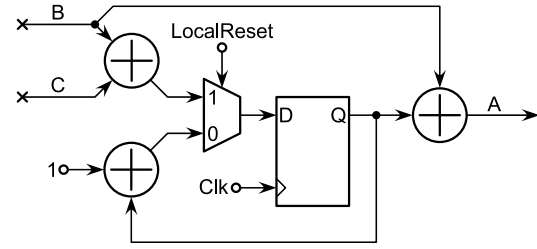


Fig. 5.4. Equivalent circuit of listing 5.10

5.1.6 Abstraction Summary

The various forms of abstraction provided by the ALCHA language provides the user with the ability to implement a variety of architectural patterns, while at the same time ensuring that the source code is easy to relate to the resulting circuit.

Pins are moved into abstraction, thereby allowing the user to move low-level connections to the leaf modules, resulting in a clean separation of concerns across abstraction boundaries. This allows the user to implement clean layered system, follow a bottom-up implementation strategy and easily perform maintenance on leaf modules without influencing higher-level modules.

Global resources, such as clocks and bus structures, can be instantiated in higher-level modules, thereby making them visible to all lower level modules. This reduces routing of globally accessible resources. The namespace model can be used to reduce global namespace clutter. Dependency-related issues was considered, and the $\text{\LaTeX}$ -like programming model was considered appropriate for use in developing firmware.

Object orientation was introduced as another mechanism of abstraction, thereby providing the user with additional mechanisms by which code-reuse can be improved.

Combinational statements in ALCHA can be more freely distributed across the code-base than with traditional HDL, thereby facilitating design patterns where lower-level modules attach themselves to globally-accessible resources. This allows the user more architectural

freedom to implement various forms of abstraction, especially useful in data streams and hexagonal architectures.

And finally, the verbosity generally associated with registered circuits was significantly reduced by means of the `rtl` construct. This allows for reduced maintenance cost, because the code is less cluttered and therefore easier to read.

5.2 Control Flow Abstraction

Section 4.2.2 established the need for control flow abstraction in the form of increased language expressiveness, reduced verbosity and structured programming.

The assembly-like nature of HDL-based finite state machines (FSMs) should be abstracted by means of software-like control flow structures such as if-statements, loops, procedures and functions that can cross clock-cycle boundaries. In the process, FSM-related boilerplate should be reduced as much as possible.

In addition, the resulting code should be structured [35]. In other words, it should be possible to refactor any block of FSM code into functions and procedures.

This section builds upon the features presented in section 5.1 in order to obtain the objective of control flow abstraction. Parts of this section was presented at SiPS 2018 [26].

5.2.1 Finite State Machines

Section 5.1.5 described how ALCHA reduces the boiler-plate typical of RTL descriptions. Finite state machines (FSMs) are a subset of RTL, so the same principles can be reused for FSM descriptions.

5.2.1.1 Modelling FSMs

Traditionally, FSMs are modelled from first principles, typically as either Mealy or Moore machines, or sometimes a combination thereof [1] pg. 195. The traditional FSM architecture is presented in figure 5.5. Moore machines are a subset of Mealy machines – the output only depends on the current state, not the input as well.

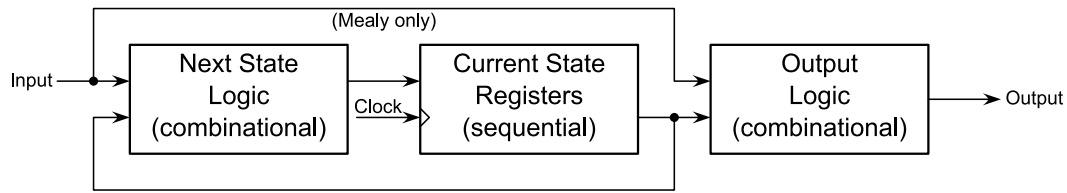


Fig. 5.5. Traditional FSM architecture, adapted from [1]

Many traditional implementations of FSMs code the three components presented in figure 5.5 as separate processes (or ‘always’ blocks), but this is not necessary. It is generally easier to understand the code if all the statements are in one place.

5.2.1.2 RTL Modelling

Many state machines are easier to implement directly in RTL. One example is the SPI interface presented in listing 5.11 (for the unabbreviated version, see appendix C). It abstracts the SPI interface of the ADXL345 accelerometer, which is shown in figure 5.6.

The outer state machine follows the traditional explicit “State” variable and case statement approach, but the inner “Transaction” state machine does not. Instead, it makes use of the SPI signals to represent the inner state directly, which is easier to relate to the interface waveforms presented in figure 5.6.

5.2.1.3 Sequential States

The SPI example presented in listing 5.11 has two parts. The outer part is sequential: it runs `Setup` once, and then `ReadX`, `ReadY` and `ReadZ` in a loop. The inner `Transaction` part is not coded as if it is sequential, but it is running sequential states: first drop the `nCS` line, then clock out `BitCount` bits of `Data` while also reading the return data into `Data`; then lift the `nCS` line and return to the outer state machine.

This sequential nature of state machines leads to the idea that the behind-the-scenes explicit state can be hidden in abstraction, as done by Cx [72].

As presented in section 2.5.2, Cx is limited to circuits that can be modelled as Kahn process networks. Although possible, it is challenging to use that model to implement the equivalent of listing 5.11. ALCHA builds on the Cx concept by introducing different rules for determining state boundaries, or fences.

```

module ADXL345 #(...)(
    input Clk, Reset,
    ...
    output reg nCS, SClk, SDI,
    input      SDO
);
...
reg [ 4:0]BitCount;
reg [15:0]Data; // R/W, MB, Address, Byte

typedef enum { Setup, ReadX, ReadY, ReadZ, Transaction } STATE;
STATE State, RetState;

always @(posedge Clk) begin
    Local_Reset <= Reset;
    if(Local_Reset) begin
        nCS <= 1'b1; SClk <= 1'b1; SDI <= 1'bX; State <= Setup;
        ...
    end else if(Clock_Enable) begin
        case(State)
            Setup: ... // These "outer" states call "Transaction"
            ReadX: ... // as if it is a function. The input parameters
            ReadY: ... // are Data and BitCount, and the
            ReadZ: ... // data read is returned in Data.

            Transaction: begin
                if(nCS) begin
                    nCS <= 1'b0;

                    end else if(SClk) begin
                        if(BitCount == 0) begin
                            nCS <= 1'b1;
                            State <= RetState;
                        end else begin
                            SClk <= 1'b0;
                        end
                        {SDI, Data} <= {Data, SDO};
                        BitCount <= BitCount - 1'b1;
                    end else begin
                        SClk <= 1'b1;
                    end end endcase end end
endmodule

```

Listing 5.11. SPI interface example in Verilog

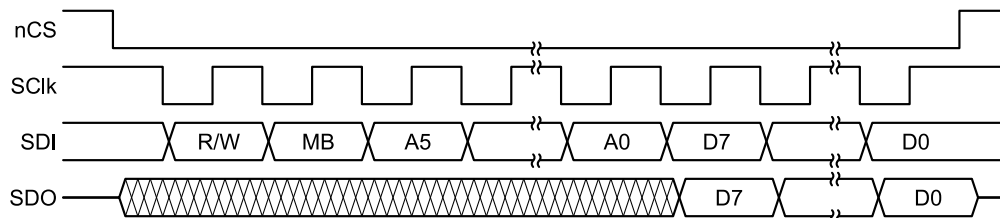


Fig. 5.6. SPI waveforms of the ADXL345 magnetometer

Listing 5.12 presents the Alcha equivalent of listing 5.11. It includes various ALCHA features that are presented later in this chapter – for now it is sufficient to concentrate on the `fsm` construct near the end.

The `fsm` construct in ALCHA removes the same boilerplate as the `rtl` construct, including the reset and clock enable semantics. The difference is that the body of an `fsm` construct is executed by means of sequential states.

The rule for determining a fence is based on whether a statement is terminated with a comma or semicolon. A comma continues the next statement in the same clock cycle, whereas a semicolon puts the next statement in the next clock cycle. As with the `rtl` construct, statements are executed with non-blocking semantics.

The largest difference between the ALCHA `fsm` construct and the Cx programming model is in the way loops are handled. Cx always inserts a fence before entering the next iteration of the loop [4], while ALCHA only inserts a fence when the user explicitly uses a semicolon. This implies that ALCHA supports loop unrolling, whereas Cx does not.

5.2.1.4 Syntax Considerations

A discussion at the SiPS 2018 conference [26] highlighted a potential problem with using comma and semicolon symbols to denote fences: they are too close to each other, visually. It is difficult to quickly scan through the code and see where all the fences are. In listing 5.12, for instance, it is not immediately clear that `Sclk` going low and the data shift happens in the same clock cycle, and then `Sclk` high the next.

There are two aspects to consider here: the algorithm and the timing. From the algorithm point of view, the almost-hidden timing information regarding fences is a good thing, because there is little to distract the reader from the function of the circuit.

From the timing perspective, something more explicit would be preferable. The syntax could be altered with something like listing 5.13, for instance. The result is rather cluttered in option (a), but the stylistic changes in (b) hides the fence so well that it can just as well not be there, even with syntax highlighting. It was decided that the extra clarity provided by the `fence` keyword is not worth the clutter.

Another option is to group all the statements that should be in a specific state by means of a blocked construct. This has serious limitations to control flow abstraction, however. There would be no way to have more fences in one branch of an if-statement than another, for example.

```

class ADXL345(net Clk, net Reset, num BAUD){
    pin nCS = 1; // Public so that the parent module (or a "pin-mapper" module
    pin SClk = 1; // that is imported from the parent) can assign pin locations.
    pin SDI, SDO;

    net(16) X = 0, Y = 0, Z = 0;
    //-----

    private{
        num Clk_Div = round(Clk'frequency / BAUD / 2.0);
        num Clk_Div_N = ceil(log2(Clk_Div));

        net(Clk_Div_N) Clk_Count;
        net Clk_Enable = ~|Clk_Count;

        rtl(Clk){
            if(Clk_Enable) Clk_Count = Clk_Div - 1;
            else Clk_Count = Clk_Count - 1;
        }
        //-----

        net(16) Transaction(net R_nW, net(6) Address, net(8) Data, num Count){
            Transaction = :(R_nW, Count > 16, Address, Data);

            fsm{ // The clock reset and clock enable is inferred from the caller
                nCS = 0;
                loop(Count){
                    SClk = 0,
                    :(SDI, Transaction) = :(Transaction, SDO);
                    SClk = 1;
                }
                nCS = 1;
            } }
            //-----

            net(16) Data;

            fsm(Clk, Reset, Clk_Enable){
                Transaction(0, 31, 0b0000_1001, 16);

                loop{
                    Z = :(Data[7..0], Data[15..8]),
                    Data = Transaction(1, 32, 0, 24);

                    X = :(Data[7..0], Data[15..8]),
                    Data = Transaction(1, 34, 0, 24);

                    Y = :(Data[7..0], Data[15..8]),
                    Data = Transaction(1, 36, 0, 24);
                } } } }

```

Listing 5.12. ALCHA equivalent of the SPI example in listing 5.11

```

fsm{
  nCS = 0;
  fence;

  loop(Count){
    SClk = 0;
    :(SDI, Data) = :(Data, SD0);
    fence;

    SClk = 1;
    fence;
  }
  nCS = 1;
  fence;
}

```

(a)

```

fsm{
  nCS = 0; fence;

  loop(Count){
    SClk = 0;
    :(SDI, Data) = :(Data, SD0); fence;
    SClk = 1; fence;
  }
  nCS = 1; fence;
}

```

(b)

Listing 5.13. Experimental syntax showing an explicit-fences option

The optimal solution is unclear at this point, and therefore left to future work. Perhaps it is as simple as using an appropriate coding style, such as placing states in paragraph-type structures, where appropriate. A linter can then be used to produce warnings, based on the surrounding coding style.

5.2.2 Function Basics

A function declared in ALCHA has a similar form to a function in C or C++. One difference, however, is that the type specifiers are optional, as shown in listing 5.14.

```

auto Add(A, B){
  result = A + B;
}

```

Listing 5.14. Example function showing duck-typing

Purely combinational functions such as this can be called from anywhere, including `rtl` and `fsm` constructs.

5.2.2.1 Pass by Reference

A key difference in semantics between C and ALCHA functions is that everything is passed by reference. The object that receives the result from the function is referenced by the `result` keyword, and the parameters of the function become aliases for the objects passed to the function. In other words, the `Add` function presented in listing 5.14 is a combinational circuit that is evaluated continuously.

Function parameters can optionally be defined as input or output, which function similarly to class member access specifiers. A parameter that is defined as input is strictly an input, i.e. the function body may not modify it. A parameter that is defined as output is treated normally (i.e. as if the `output` specifier was not given), and is intended to act as a hint to the user that the function will write to the object bound to that parameter.

5.2.2.2 Parameter Type Casting

There are three scenarios where the function call creates temporary objects. The first is where a function is called using literals instead of objects. The second is when the result from a function is being used by an expression, rather than directly assigned to an object. These first two cases are relatively straightforward to handle, because the data-flow direction is known at the point of calling the function.

The third case is more tricky, and happens when the function parameter is of a different type, but the compiler is able to perform an implicit cast. The question is: in which direction should the cast be applied?

A naive solution involves direction inference. If the parameter is only ever read by the function body, data-flow is from external to internal. If the function modifies the parameter, data-flow is from internal to external. Both these cases allow the compiler to know in which direction to perform the cast by inspection of the function body.

This solution becomes complicated when considering attributes (see section 5.4) and modify-and-assign operations. These scenarios imply bidirectional behaviour. Attributes should be passed by reference, and any modification to the parameter attributes should propagate back to the original object used when calling the function. Similarly, bidirectional use, such as modify-and-assign operations, should infer a cast in both directions. The compiler should therefore create both ‘input’ and ‘output’ temporary objects.

5.2.2.3 Duck Typing

If the function parameter or return types are not specified, the function is essentially duck-typed [116, 117], which makes the function more general-purpose. If the types are specified, as in listing 5.12, the compiler type-checks the function call, thereby making the code less error-prone.

5.2.3 FSM Functions

Functions can also contain embedded `fsm` constructs. The semantics of such functions depend on the context from which they are called.

5.2.3.1 Combinational Context

If the function is called from a combinational context, the solution is simple: synthesise the contents of the function as if it was written in place of the call. The same by-reference semantics apply.

5.2.3.2 FSM Context

If the function is called from within an `fsm` construct, like the ‘Transaction’ function in listing 5.12, the semantics are as follows:

- The clock, reset and clock-enable signals of the function FSM is inherited from the calling FSM¹⁰.
- Any reset logic related to function-local signals happens during the same clock cycle as the calling state.
- The first statement of the function FSM executes in the clock cycle following the calling state.
- The return state (in the parent FSM) executes one clock cycle after the last statement in the function body FSM.
- The function output register keeps its value after execution of the function.

5.2.3.3 Implementation

There are two ways for a compiler to implement FSM-type functions called from FSM context. The first option is as presented in listing 5.11, where the function body is injected into the parent state machine.

¹⁰There is thought to remove this restriction in a future version. If the clock, reset and clock-enable is specified, and not the same as the calling clock-domain, the go-busy handshake can be used to realise clock-crossing.

This is not neatly scalable. Every level of function call would need another `RetState` register (or equivalent) and the various function parameters will cause severe name-clutter that the compiler will then need to solve through some or other name mangling algorithm [118].

An alternative solution (i.e. the second option) is to package the function body into a new module that uses a go-busy handshake to control execution. The boilerplate of such a module could, for instance, look like listing 5.15, which implements the handshake as presented in figure 5.7.

```

module Function(
    input Clk, Reset, Enable,

    input      Go,
    output reg Busy,

    input Param1, Param2, // etc.
    output reg Result
);
//-----

enum { /* ... */ } State;

always @(posedge Clk) begin
    Local_Reset <= Reset;

    if(Local_Reset) begin
        Busy    <= 0;
        Result <= 1'hX;
        State <= Start;
    //-----

    end else if(Enable) begin
        if(Busy) begin
            case(State)
                Start: // Execute the first statement here
                State2: // Execute the second statement here
                // etc...

                Return: begin
                    // Execute the last statement here
                    Busy <= 0;
                end
            endcase

        end else if(Go) begin
            // Execute the "reset" here
            Busy <= 1;
            State <= Start;
        end
    end
end
endmodule

```

Listing 5.15. Possible Verilog boilerplate for FSM function calls.

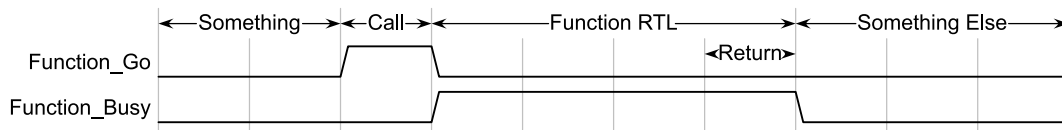


Fig. 5.7. The go-busy handshake, as implemented by listing 5.15

5.2.3.4 Function Call Overhead

A function that is implemented as a go-busy handshake has no timing overhead. The cycle-related execution is identical to the embedded option.

Neither does it introduce significant resource overhead. The `Go` signal is a combinational circuit based on the calling FSM state vector, which is inexpensive on FPGAs¹¹. The `Busy` register replaces the extra state that would be required if the function was embedded into the calling FSM¹².

5.2.3.5 RTL Context

An FSM-based function can also be called from within an `rtl` construct. RTL has no concept of sequencing – it is simply a collection of statements that describe what happens to the input of registers between clock cycles. The go-busy handshake makes no sense in such context.

The solution, therefore, is to translate the function body to RTL, as if it was called from combinational context, and then evaluate it by the rules specified in section 5.2.4.2.

5.2.4 RTL Functions

The final case to consider is functions that contain embedded `rtl` constructs. The semantics of such functions depend on the context from which they are called.

5.2.4.1 Combinational Context

If the function is called from a combinational context, the solution is the same as for FSM functions: synthesise the contents of the function as if it was written in place of the call.

¹¹The LUT-based architecture of FPGAs makes combinational circuits inexpensive.

¹²A large percentage of FSM state vectors are encoded as one-hot, in which case there is no overhead.

5.2.4.2 RTL Context

RTL functions called from RTL context allows the user to refactor portions of RTL out of the main body. The function contents reset synchronously with the calling RTL, and the rest of the function body is essentially evaluated as if it was written in place of the function call.

As with FSM functions, the RTL function's `rtl` construct inherits its clock, reset and clock-enable from the calling construct¹³.

5.2.4.3 FSM Context

When calling an RTL function from FSM context, it is expected to put the calling FSM on hold until the function RTL has completed execution. In addition, function RTL must be kept idle while the FSM is not using it, and signalled when to start execution.

The same go-busy handshake is therefore used, with approximately the same boilerplate as presented in listing 5.15. One key difference, however, is that there is no way for the compiler to know when a piece of RTL code has finished executing.

This is solved by means of a `return` statement. The `return` keyword is essentially replaced by `Busy = 0`, which signals the end of the function. An RTL function that has no `return` statement will block the calling FSM indefinitely, and a `return` statement called from RTL or combinational context has no effect.

The 'Transaction' function can therefore be implemented by using an `rtl` construct instead of the `fsm` construct, as shown in listing 5.16. In similar fashion, the `return` statement can be used to exit an FSM function early.

5.2.5 Other Considerations

Various other topologies and concepts can be implemented by means of functions.

5.2.5.1 Fork and Join Structures

Clocked functions (i.e. functions that are not purely combinational), when called from FSM context, is blocking. In other words, the next state in the calling FSM will only execute once the function returns.

¹³And, as with FSM functions, there is thought to remove this restriction.

```

net(16) Transaction(net R_nW, net(6) Address, net(8) Data, num Count){
    result = :(R_nW, Count > 16, Address, Data);
    net(5) BitCount = Count;

    rtl{ // The clock reset and clock enable is inferred from the caller
        if(nCS){
            nCS = 0;

        }else if(SClk){
            if(BitCount == 0){
                nCS = 1;
                return;
            }else{
                SClk = 0;
            }
            :(SDI, result) = :(result, SDO);
            BitCount = BitCount - 1'b1;

        }else{
            SClk = 1;
        }
    }
}

```

Listing 5.16. Example showing an RTL version of the ‘Transaction’ function

This blocking nature can be expanded. If the function call ends in a semicolon, that function call blocks immediately. If, on the other hand, a function call ends in a comma, the following statements are still execution within the same clock cycle. This implies that multiple functions can be called at the same time.

This process forks multiple threads of execution, which might or might not finish at the same time. The following state in the calling FSM therefore only continues once all the **busy** signals have returned low, thereby performing a join.

5.2.5.2 Mixed Type Functions

It is possible to write a function that contains multiple **rtl** and **fsm** constructs. This complicates the calling convention, because all clocked constructs run in parallel, and may exit at different times.

This case is treated as if the various clocked constructs were put in different functions, and the user called those functions in parallel. In other words, each clocked construct gets its own **Go** and **Busy** handshake signals. The return state waits for all the **Busy** signals to return low before continuing.

5.2.5.3 Operator Overloading

Operators can be modelled as functions, and many languages support operator overloading based on this concept. The semantics of overloaded operators can simply follow the same function semantics presented in previous parts of this section.

This would allow the user to write a function similar to the “Connect that to this” function defined in listing 2.10 in section 2.5.5, as well the DSP pipeline presented in listing D.31 in appendix D.9.1.

In ALCHA any operator can be overloaded as either a binary or unary operator, by means of the `operator` and `unary` keywords, respectively. The syntax and semantics are the same as in C++.

5.2.5.4 State Encoding

Generally speaking, it is best to leave state encoding to the compiler. Both SystemVerilog and VHDL define state variables by means of an enumeration instead of explicit values. The compiler then chooses an encoding, which generally defaults to one-hot unless specified otherwise in the compiler settings.

This behaviour is good. It is very rare to want to change the state machine encoding, but the topic is revisited in section 5.4.4.2

5.2.5.5 Anonymous Functions and Function References

Anonymous functions, or lambda-expressions, are generally used to reduce verbosity and make code easier to read and maintain, when used appropriately. In ALCHA, an anonymous function can be declared by means of the `func` keyword, followed by an optional parameter declaration and a function body. Additionally, any function can be referenced by means of a function reference, also declared by means of the `func` keyword.

Anonymous functions are especially useful to reduce verbosity where the grammar would require an explicit function declaration otherwise. For example, a class might have a function pointer member that is used as a call-back to the parent module, or the user might want to use a small algorithmically-defined combinational circuit inside an `rtl` construct. The former case is illustrated in listing 5.17.

```
class RS232{
    func OnReceive; // Function pointer based event
    !! Implement an RS232 receiver
};
RS232() UART;
UART.OnReceive = func(net(8) Data){
    !! Handle the received character
};
```

Listing 5.17. An event handler implemented with an anonymous function

The same effect can of course be achieved by means of polymorphism and virtual functions. The choice of mechanism is left up to the user.

5.2.6 Control Flow Summary

This section presented the syntax and semantics of the ALCHA `fsm` construct, which provides a means by which the user can make use of structured programming concepts (such as if-statements and loop structures) to implement finite state machines.

The basic concept of sequential statements was expanded to include the use of functions. Function bodies can be purely combinational, RTL-based, FSM-based or a combination thereof. The resulting function calling conventions and semantics were presented for various function styles called from within different contexts.

Finally, it was shown that the mechanism naturally support fork-and-join processing flows.

5.3 Fixed-point Abstraction

The discussion presented in section 4.2.3 established the need for automated fixed-point arithmetic. The user should be able to concentrate on the algorithm, instead of the details of shifting, clipping and truncating correctly, as well as keeping track of the various fixed-point representations throughout the calculation.

This section present means by which the objective of fixed-point abstraction can be met. The contents of this section was presented at SiPS 2018 [26].

5.3.1 Extending the Net Type

So far, all `net` types were defined using a single number, which defines them as unsigned integers. This can easily be extended with an optional second parameter, which specifies the

full-scale real-world value represented by that net [26]. If both the width and full-scale are specified as positive, the net is unsigned. When either the width or full-scale is specified as negative, the net is signed. Listing 5.18 shows various examples of fixed-point declarations.

```
net( 8      ) A; // 8-bit unsigned integer
net( 7, -128) B; // 8-bit signed integer (extra bit for sign)
net(-7      ) C; // As above: integer in range [-128, 128)
net(16,  pi) D; // 16-bit unsigned fixed-point; [0, pi)
```

Listing 5.18. Examples of fixed-point declarations

Fixed-point representations could also be specified by means of the real-world value associated with the least-significant bit, but this creates various limitations. If the user wants to reduce the resolution of a variable, for instance, it is more convenient to keep the full-scale real-world equivalent consistent, which would imply that the LSB representation changes. Furthermore, keeping track of the full-scale equivalent throughout a calculation is easier than keeping track of the LSB representation, especially when clipping and rounding is involved during intermediate steps.

5.3.1.1 Binary Representation

The binary representation of fixed-point nets can be calculated by means of equation 5.1, where A is the real-world value of the fixed-point net, A_b is the binary representation (an integer), A_{fs} is the full-scale of the fixed-point type of A , and A_N is the number of bits used to represent A_b . The same notation is used throughout this section.

$$A_b = \frac{A \cdot 2^{A_N}}{A_{fs}} \quad (5.1)$$

Equation 5.1 is valid for unsigned nets, but it is trivial to extend the concept to signed nets. This form allows the developer to define any arbitrary full-scale value for the net. It does not have to have any relation to powers-of-two.

5.3.1.2 Width of Signed Nets

For signed nets, the specified bit-width does not include the sign bit. The sign bit is extra. The motivation for this design decision is as follows:

- In most digital signal processing applications, the signal-to-noise ratio of the signal is based on the magnitude, not the peak-to-peak range.

- When multiplying signed numbers, the sign bit is a waste of space, because the multiplication operation uses the magnitude of the number, which is only half the range. The result should therefore be the sum of the widths required to store the magnitudes, not the full signed number.
- It is easier to see the relationship between signed and unsigned numbers of the same width, and casting is direct. Signed and unsigned nets of the same width and full-scale representation have the same resolution only if the sign bit is not included in the width.

5.3.2 Multiplication

It is relatively straightforward to extend multiplication to arbitrary fixed-point types. The operation is presented in equation 5.2. The resulting binary representation is the product of the source binary representations, the full-scale of the result is the product of the two source full-scales, and the resulting width is the sum of the source widths.

$$\frac{A_b A_{fs}}{2^{A_N}} = \frac{B_b B_{fs}}{2^{B_N}} \times \frac{C_b C_{fs}}{2^{C_N}} \quad (5.2)$$

In the case where a net is multiplied or divided by a literal, the ALCHA compiler multiplies or divides the source full-scale with the literal value in order to obtain the full-scale value of the result. The binary representation and width remain unchanged.

In the case where the target format is specified (as in the case of an assignment), the multiplication is performed as a two-stage process. First, an intermediary result is calculated, with fixed-point format as described above. Second, the result is transformed (as described in section 5.3.5) to the target format before the assignment is performed.

5.3.3 Division

ALCHA does not directly support the division of two `net` types. If the user wants to use synthesisable division, it must be performed by means of a user-defined function, possibly overloading the division operator.

The reason for this is that division is generally an expensive operation, and there are various means by which it can be implemented: binary long division; look-up tables; binary searching with a multiplier in the loop; etc. These methods can be implemented with short or long

latencies, and various different pipe-lining strategies can be followed. It is up to the developer to choose the appropriate method for the application.

ALCHA does, however, directly support division by a literal. In this case, the ALCHA compiler divides the source full-scale with the literal value in order to obtain the full-scale of the result. The binary representation and width remain unchanged.

5.3.4 Addition

When adding or subtracting two fixed-point numbers, the operands must be scaled such that they all have the same format. The addition operation $A = B + C$ can be arranged into equation 5.3 or 5.4.

$$A_b = \frac{2^{A_N}}{A_{fs}} \frac{B_{fs}}{2^{B_N}} \left(B_b + C_b \frac{C_{fs}}{2^{C_N}} \frac{2^{B_N}}{B_{fs}} \right) \quad (5.3)$$

$$A_b = \frac{2^{A_N}}{A_{fs}} \frac{C_{fs}}{2^{C_N}} \left(B_b \frac{B_{fs}}{2^{B_N}} \frac{2^{C_N}}{C_{fs}} + C_b \right) \quad (5.4)$$

The ALCHA compiler chooses the option that scales the operand with a number that is larger than one. Most often the user will choose full-scale values that are related by a power of two, which then reduces the scaling operation to a bit-shift.

In some rare cases it is convenient to use full-scale values that are not related by a power of two, in which case the compiler will synthesise a multiplier and issue a warning. In order to reduce rounding error, the constant used to scale with is chosen as the natural size of the target FPGA DSP blocks, which is typically 18-bits. The result is bit-shifted accordingly.

In the case where one of the operands is a literal, the literal is transformed to the same format as the other operand, and the result is also of that same format. This behaviour can be overridden by explicitly casting the literal to a fixed-point net type.

If the user assigns the result to a type that is too small to store the carry bit, that bit is dropped and the result overflows. This behaviour is often desired, so no compiler warning is issued.

5.3.5 Assignment and Casting

Assignment and type casting both follow the same rules. When an expression is type-cast, the compiler generates a temporary net of the target type and assigns the expression result to that net. The casting operation $A = B$ can be represented by equation 5.5.

$$A_b = B_b \frac{2^{A_N}}{A_{fs}} \frac{B_{fs}}{2^{B_N}} \quad (5.5)$$

If the scaling factor is a power of two, the appropriate bit-shift and truncation and / or sign extension is applied. If the scaling factor is not a power of two, the compiler will synthesise a multiplier and issue a warning, using the same choice of scaling constant as described in section 5.3.4 above.

When the user assigns a literal to a net, the literal is scaled to the target type and then rounded (not truncated). If the target type is too small to store that value (i.e. the value of the literal is larger than the full-scale of the target net), the compiler issues an error.

5.3.6 Raw Operations

In general, ALCHA performs fixed-point arithmetic calculations based on the real-world value, not the binary representation. It is sometimes useful, however, to perform the calculation (or assignment) using the binary representation, or raw bits. This is done by using the “:” unary operator, bit-vector concatenation (“: (...)”) or bit-vector slicing, as illustrated in line 5 of listing 5.19(a).

In order to perform a raw bit assignment, the “:=” operator must be used. Some operators, such as bit-wise logical operators, always operate on the raw bits, whether the operands have been cast or not. Without the ability to cast to raw bits, the user would have had to manually scale the full-scale of **Frequency**, as shown in figure 5.19(b).

```
net(31, -125e6) Frequency;
net(31, -pi)    Phase;
net(18, -1)     Output;
rtl(Clk_250_MHz){
    Phase := :Phase + :Frequency;
    Output = Sin(Phase);
}
```

(a) With raw bits

```
net(31, -125e6) Frequency;
net(31, -pi)    Phase;
net(18, -1)     Output;
rtl(Clk_250_MHz){
    Phase += Frequency * (pi / 125e6);
    Output = Sin(Phase);
}
```

(b) Without raw bits

Listing 5.19. ALCHA implementation of an NCO, with and without using raw bits

5.3.7 Fixed-point Summary

This section presented the ALCHA fixed-point programming model. The user can define signals to be of a particular fixed-point type, and the compiler automatically synthesises the required scaling, shifting, extending and truncating required to implement the expression.

5.4 Design Constraints

As presented in section 4.2.4, there is a need for integrated design constraints. It should be possible to define design constraints that relate to a pin or signal at an appropriate place in the source code¹⁴.

Some design constraints, such as multi-cycle paths, should be automated. The language should therefore be structured such that it is practical for a compiler to automatically extract multi-cycle paths.

It should also be possible to apply constraints to sub-circuit instances, such as placement constraints. This could be either as part of the module design, or module instantiation.

This section presents means by which the objective of integrated design constraints can be satisfied.

5.4.1 Attributes

Design constraints can be modelled as metadata. The constraints are not important when considering the behaviour of the circuit, but they are important during circuit synthesis in order to guide the lower level details of placement and routing.

5.4.1.1 Attributes in Software

Software has similar problems when considering cross-language interoperability, lower level application binary interfaces (ABIs), etc. These are often solved by means of attributes, as implemented by C#, C, C++, D and others [119, 104, 105, 120].

¹⁴The “appropriate place” depends on the use-case. The user should be able to decide whether to place the constraint where the object is declared, or where it is used, or elsewhere.

Some examples of software attributes include:

- The `extern "C"` attribute common in C headers, used when a module compiled with C++ must link to another module compiled with C.
- The `DllImport` attribute used in C# to import from a lower level C++ based library.
- The `__attribute__((packed))` attribute used in the GCC family of compilers to remove padding from structures.

5.4.1.2 Attributes in Firmware

The concept of adding metadata to objects is a good fit for obtaining the objective of inline design constraints. This concept is not new to firmware: both Verilog and VHDL have standard syntax for adding attributes to objects [21, 17].

Both Quartus and Vivado support a large set of attributes that can be added to HDL code [37, 121]. The problem, however, is that the attributes themselves are not standardised – the vendor tools define their own, thereby making the resulting code non-portable.

5.4.1.3 Attribute Syntax

The original idea for ALCHA attributes was to use XML syntax [122], but this was soon found to be too restrictive. All XML attributes are strings, which must then be interpreted. This creates a discord between the attribute syntax and the rest of the ALCHA language.

The syntax was therefore generalised to something more in line with JSON5 [103]. ALCHA attributes are dictionaries of ALCHA objects, including numeric literals, boolean values, strings, arrays, nested dictionaries and object references. The angular bracket syntax of XML was retained, because the braces used in JSON would interfere with statement blocks. Some examples of ALCHA attributes are presented in listing 5.20.

Attributes are accessed by means of a tick symbol (') in order to distinguish it from structure or class members. Attributes reside in their own namespace in order to reduce namespace clutter. This syntax is consistent with VHDL build-in attributes, which serve a similar purpose.

Physical properties, such as frequency, voltage and current, are specified in SI units, without scaling. Any attribute can be accessed or modified anywhere in the code. The user can use the metadata for any purpose, not only design constraints.

```

// Assign during declaration (pin example)
pin<frequency = 100e6, location = "AA9-AB8", standard = "LVDS", voltage = 2.5> Clk;

// Class instance attributes example
NCO()<partition = "DSP"> LocalOscillator;

// Assigning attributes to array members
pin(4)<location = ["E2", "F8", "D3", "E6"]> DataBus;

// Declare first, then assign an attribute
pin Reset;
Reset'location = "AB7";

// Access an attribute
num BaudCount = Clk'frequency / BaudRate;

```

Listing 5.20. Example showing ALCHA attribute syntax

5.4.1.4 Attribute Arrays

Also shown in listing 5.20 is an example of how to apply different attributes to each bit of a vector. The same format can be followed for arrays of vectors, in which case the attributes are specified as an array of arrays.

The attributes are assigned most significant bit first for vectors, but in order of array index for arrays. This discrepancy is intentional. When writing down the functions of individual bits of a vector, most people in western societies will write them in the same order as they would write a decimal number, i.e. big-endian. As an example, consider writing down the functions of the bits within a single-precision floating point vector: the sign bit is generally written first, followed by the exponent, and finally the mantissa.

By contrast, most people will list the elements of an array from the smallest index to the largest index. ALCHA follows these same conventions.

5.4.1.5 Inheritance

Any object in ALCHA inherits the attributes of its parent, unless overridden by an attribute of the object itself. This allows the user to assign the same attribute (e.g. partition, logic standard, voltage standard, etc.) to a large group of objects with a single assignments.

When the compiler needs to access an attribute, for whatever reason, it first tries the attribute dictionary associated with the object specified. If it cannot find it there, it continues to the object parent, which could be a group name, class instance, class definition or namespace. It continues searching up the hierarchy all the way up to the global namespace.

Inheritance is only used when accessing attributes, not when defining them. In other words, when an attribute is specified on the LHS of an assignment, that attribute applies to the object directly, even if the attribute was defined in some parent namespace. If the attribute is specified on the RHS of an assignment, inheritance applies.

5.4.1.6 Groups

A group structure can be used when assigning multiple attributes to a group of declarations. Groups can be anonymous or named. One use case is to apply an I/O standard to a set of pins, as illustrated in listing 5.21. Groups differ from structures in that they cannot be used as a type, and they are their own instances.

```
group<standard = "CMOS", voltage = 2.5>{
  pin<location = "A6"> LED;
  pin<location = "Y7"> Button;
}
group<standard = "CMOS", voltage = 3.3> SD{
  pin  <location =  "B3"> Clock;
  pin  <location =  "C5"           pullup = true> Command;
  pin(4)<location = ["E2", "F8", "D3", "E6"], pullup = true> Data;
}
```

Listing 5.21. Examples of attribute groups

The LED and Button nets are accessed directly, because they are defined in the parent scope. The SD group members must be accessed through the group name, e.g. “SD.Clock” or “SD.Data[3]”.

5.4.1.7 Function Ports

Generally, function parameters are passed by reference, which implies that functions can access and modify the attributes of the object passed to the function. One edge-case relates to a net that must be type-cast in order to meet the function parameter type definition, which is often the case with fixed-point types.

A similar problem was addressed in section 5.2.2.2. It was decided that the compiler should implement a type-cast circuit that emulates the behaviour of pass-by-reference semantics.

By following the same logic, the attributes can be passed by reference directly, even though the object itself is passed via a temporary.

5.4.1.8 Global Attributes

In some rare cases the user might want to apply an attribute to everything in the current scope from that point onwards. In that case, a global attribute can be used. Listing 5.22 shows an example where every enumeration and finite state machine in the current scope should be encoded using a “one-hot” scheme.

```
'encoding = "one-hot";  
  
!! Rest of the statements in the current scope.
```

Listing 5.22. Example of a global attribute

Global attributes are also used to specify the target device and other platform-specific parameters that the ALCHA compiler should know about.

5.4.2 Location Constraints

Location constraints are relatively straightforward, and described in more detail in appendices F.14.1 and F.14.2. The user is able to constrain pin declarations to physical pin locations, as well as synthesisable circuits to logic regions and partitions, which is useful for decreasing compilation time [123].

5.4.3 Timing Constraints

Timing constraints are more complicated, with many edge-cases. The industry standard for declaring timing constraints is Synopsys Design Constraints (SDC) [124]. The data model therefore closely follows the SDC model, so that translation is relatively straightforward.

This subsection considers the examples in the Intel Quartus Timequest Cookbook [99] in order to derive an appropriate attribute data model for defining timing constraints. Xilinx and other manufacturers follow the same SDC standard, so the derived data model is expected to be applicable to those manufacturers as well.

5.4.3.1 Base Clocks

SDC declares base clocks by means of the `create_clock` TCL command. A clock can be associated with any net, and any net can be associated with multiple clocks.

The majority of clocks used in FPGA designs have a duty-cycle of 50% and is directly received from an external source. In ALCHA, any pin or net that carries the `frequency` attribute is automatically a clock signal, and the ALCHA compiler will generate the appropriate `create_clock` timing constraints for it.

If, however, the clock is more complex, the user can override the automatically generated clock by means of the `create_clock` attribute. This attribute is a dictionary of sub-attributes derived from the TCL command, i.e. `period` and `waveform`. If the object is associated by more than one clock, the user can specify it as an array of dictionaries. If the `frequency` attribute is also present, the `create_clock` attribute overrides it.

5.4.3.2 Automatically Derived Clocks

Both Quartus and Vivado provide the ability for the compiler to automatically derive clocks derived from PLLs. ALCHA automatically issues this command, so that the user only needs to specify the base clock.

The user might still want the nets driven by the PLL to carry the `frequency` attribute so that it can be used for algorithmic circuit generation. In this case, the associated `create_clock` SDC command should not be issued, but there is no way for the ALCHA compiler to know this¹⁵. The user can therefore suppress automatic clock constraints for that net by also applying the `auto_derived_clock = true` attribute.

5.4.3.3 Manually Derived Clocks

In rare cases the user might want to manually declare derived clocks. SDC declares derived clocks by means of the `create_generated_clock` TCL command. This can be handled by means of the `create_generated_clock` attribute, which function similarly to the `create_clock` attribute in that it is heavily based on the original SDC command.

The user must specify the `source` for the derived clock, which is a reference to the clock object (pin or net). If the source net has multiple clocks assigned, the generated clock must specify a master clock. ALCHA handles master clock association by position in the array of dictionaries. This is illustrated in listing 5.23. The `create_generated_clock` attribute supports a variety of sub-attributes (like the SDC command), most of which are optional.

¹⁵ALCHA does not have core language support for generating PLLs. There are too many variants and settings involved to make it practical, so the generation of PLL circuits is best left to vendor tools. The `hdl` construct is then used to link the generated PLL module with the rest of the ALCHA code.

```

// Define the base clock -- the "waveform" attributes are optional
pin<frequency = 100e6, // Used for algorithmic circuit generation
    create_clock = [ // Overrides the frequency attribute, used for SDC
        <period = 10e-9, waveform = [0, 5e-9]>, // 100 MHz clock option
        <period = 20e-9, waveform = [5, 15e-9]>] // 50 MHz clock option
    > BaseClock;

// Define a derived clock with consistent parameters
net<frequency = BaseClock'frequency * 3/2,
    create_generated_clock = // Associates with both 50 and 100 MHz options
        <divide_by = 2, multiply_by = 3, source = BaseClock>
    > DerivedClock_1;

// Define a derived clock with different parameters
net<frequency = 10e6, // Used for algorithmic circuit generation
    create_generated_clock = [
        <divide_by = 10, source = BaseClock>, // Associates with the 100 MHz option
        <divide_by = 5, source = BaseClock>] // Associates with the 50 MHz option
    > DerivedClock_2;

```

Listing 5.23. Example showing how to derive clocks from a base with multiple sources

5.4.3.4 Clock Groups

All clocks defined with SDC are considered in the same clock domain, unless specified otherwise. The developer is expected to use the `set_clock_groups` command to specify related clocks, and drop timing paths between non-related clocks.

ALCHA by contrast, assumes that all base clocks are unrelated. This is quite often the case with FPGA designs: there might be a master clock derived from an on-board crystal oscillator, a PCIe clock generated by the PCIe root hub and an Ethernet clock recovered by the receiver circuit. Even clocks that are derived from the same oscillator might be unrelated if the phase relationship is unknown.

The user can relate two clocks by means of the `clock_group` attribute, which is of string type. Two nets with the same `clock_group` name are considered related and the ALCHA compiler will generate the appropriate SDC commands.

5.4.3.5 External Timing

External timing is generally specified by means of `set_input_delay` and `set_output_delay` SDC commands. The exact parameters used depend on the topology of the external circuit. In addition, each parameter must be specified twice: once for the minimum delay, and again for the maximum.

ALCHA takes the same approach to external timing as with the `create_clock` command, but includes some syntactic sugar in order to reduce the syntax. Any input pin can

carry the `set_input_delay` attribute, which is a dictionary of sub-attributes that mostly follows the SDC command. Similarly, any output pin can carry the `set_output_delay` attribute. The associated clock is specified by means of object reference, similarly to `create_generated_clock` attributes.

A pin that does not contain timing constraints is automatically considered asynchronous and ALCHA will issue the appropriate `set_false_path` command.

The Timequest Cookbook recommends the use of virtual clocks when specifying external timing constraints [99]. ALCHA automatically defines the required virtual clocks based on the attributes of the clock specified.

The first difference to SDC is that the `min` and `max` sub-attributes both carry values, rather than being boolean flags. This means that the rest of the parameters only need to be defined once. In addition, the user can make use of the alternative, but equivalent, `delay` and `uncertainty` sub-attributes. If both forms are specified, the sum is taken (i.e. `min += delay - uncertainty` and `max += delay + uncertainty`).

For double data rate (DDR) signals, the user can simply set the `ddr = true` attribute, rather than specifying the constraint twice, as would be the case with SDC.

5.4.3.6 Split IC and PCB Timing

One of the language objectives is to allow the user to split the declaration of PCB-based timing constraints from external IC constraints. This can easily be achieved by defining only one or the other, and then modifying the constraint later. Listing 5.24 shows an example.

Listing 5.24 also showcases how attributes can be created on-the-fly. The `min` and `max` are not declared prior to the modify-and-assign statements that add the setup and hold requirements. These attributes are implicitly defined, and initialised to zero.

The alternative external IC attributes can be used when the IC has many external pins with the same requirements, which is typically the case. Attributes that do not apply (such as hold timing on an input pin) are ignored.

5.4.3.7 Multicycle Constraints

SDC defines exceptions (such as multicycle constraints) by means of a globally-defined constraints that specifies the “from” and “to” objects, and sometimes also a “through” object.

```

// Pin definitions include only the PCB-based timing parameters
group SPI{
  pin<frequency = 10e6> Clk;
  group<set_output_delay = <clock = Clk, delay = 0, uncertainty = 100e-12>>{
    pin nCS;
    pin MOSI;
  }
  pin<set_input_delay = <clock = Clk, delay = 500e-12, uncertainty = 100e-12>> MISO;
}

// Then, later, the SPI module can include the IC-related parameters
SPI.nCS 'set_output_delay' min -= 5e-9; // Subtract the hold requirement
SPI.nCS 'set_output_delay' max += 10e-9; // Add the setup requirement

SPI.MOSI 'set_output_delay' min -= 5e-9; // Subtract the hold requirement
SPI.MOSI 'set_output_delay' max += 10e-9; // Add the setup requirement

SPI.MISO 'set_input_delay' delay      += 5e-9; // Add the output delay
SPI.MISO 'set_input_delay' uncertainty += 1e-9; // Add the output tolerance

// Alternatively, use attribute inheritance and external IC attributes
SPI.external_setup      = 10e-9; // Only applies to output pins
SPI.external_hold       = 5e-9;  // Only applies to output pins
SPI.external_delay_min  = 4e-9;  // Only applies to input pins
SPI.external_delay_max  = 6e-9;  // Only applies to input pins

```

Listing 5.24. Example showing external timing requirements

A naive solution would therefore be to have a global `set_multicycle_path` attribute that contains an array of constraint dictionaries. This does not work, because attribute inheritance will ensure that all children of the namespace that defines the multicycle constraint carries the same attribute, causing inconsistent semantics and thereby confusing the user.

An alternative is to apply the `set_multicycle_path` attribute to either the source or destination net. If the `to` sub-attribute is present, the net is the source and, similarly, if the `from` sub-attribute is present, the net is the destination. If both `from` and `to` attributes are present, the constraint is applied twice: once with the net as source, and another with the net as destination. The optional `through` sub-attribute functions similarly.

If more than one net should be used for the other side, the user can assign the parent object (e.g. group or class instance), or an array of objects¹⁶.

If the attribute is applied to a clock net (i.e. a net that carries the `frequency` attribute), the constraint applies to the clock, rather than the net that carries the clock. The same rule applies when the object reference attached to the `from` or `to` sub-attributes is a clock net.

¹⁶A future version of the ALCHA language might include a built-in function in order to search the design-space for wildcard-based patterns that return an array of object references. This would work similarly to the `get_pins`, `get_ports`, `get_registers` and `get_fanouts` used for SDC-based constraints.

5.4.3.8 Automated Multicycle

A case for automated multicycle extraction was made in section 4.2.4.3. The ALCHA `rtl` and `fsm` constructs support optional clock-enable nets, which makes it relatively easy to infer multicycle constraints for all assignments within that construct.

In order for this to be feasible, the compiler must know how often the clock-enable signal enables the clock. Because this relates to multicycle constraints, it was decided that the appropriate name for the attribute is `multicycle`. It specifies the minimum number of clock cycles between clock-enables. When the attribute is absent, the compiler does not infer multicycle constraints.

Another means by which to infer multicycle requirements is to analyse the execution graph of the `fsm` construct. This is outside the scope of this dissertation and therefore left to future work.

5.4.3.9 Other Exceptions

In rare cases the user might want to have finer control over the delay of any given path. The `set_max_delay`, `set_min_delay` and `set_false_path` attributes has the same meaning as their SDC counterparts, and follow the same semantics as described in section 5.4.3.7.

5.4.3.10 Compiler Considerations

The vendor compiler prefers timing constraints that make use of wildcards, instead of specifying every path individually. An optimised compiler will therefore generate wildcard-based SDC constraints where possible. It should be possible to do this by means of a simple sorting and pattern-matching algorithm, but the exact details is left to future work.

5.4.4 Compiler Behaviour

The ALCHA compiler reduces boilerplate by employing various default behaviour. The user might want to change the behaviour, such as specifying the rounding and clipping rules for fixed-point calculations or the encoding to be used by a state machine. This section considers some of the more prevalent compiler behaviour modifications a user might want to apply.

5.4.4.1 Reset Behaviour

The ALCHA compiler makes use of synchronous and localised reset signals, but asynchronous resets can be useful [19] pg. 139. The user can apply the `async_reset = true` attribute to a reset net in order to make use of ‘asynchronous assertion, synchronous deassertion’ [19] pg. 144.

5.4.4.2 State Encoding

State-machine encoding influence both performance and area of the synthesised circuit [1] pg. 199. By default, ALCHA lets the vendor compiler decide the most appropriate encoding, but the user can specify the `encoding` attribute instead, which enforces a specific encoding method. The attribute takes a string value, which must be supported by the compiler implementation. Typical encodings include “sequential”, “gray”, “johnson” and “one-hot” [1] pg. 219.

5.4.4.3 Rounding and Clipping Behaviour

The default behaviour of fixed-point arithmetic is to allow overflow. This can be detrimental to fixed-point algorithms such as control systems, which are expected to clip instead. The `overflow = "clip"` attribute can be added the expressions (by means of a `group` structure) in order to change the default behaviour [26].

Similarly, the user might want to change the default truncation behaviour to rounding, which can be achieved by means of the `truncate = "round"` attribute. Other options might include “floor” and “ceil”.

5.4.5 Design Constraints Summary

This section presented ALCHA attributes and the associated syntax and semantics. Attributes can be used to apply design constraints to signals and circuit instances, such as location and timing. Furthermore, attributes can be used to influence compiler behaviour, such as applying rounding and clipping instead of truncation and overflow.

5.5 Generalisation

Various generalisation features are discussed in section 4.2.5. Some key concepts include algorithmic code generation (i.e. auto-generation) and superior parametrisation by means of scripting. Such scripting should be a core feature of the language.

This section investigates the language features that would meet these aims, thereby satisfying the main objective of generalisation.

5.5.1 Choosing the Model

There are a number of means by which superior generalisation can be implemented. This subsection explores some of the more prevalent options.

5.5.1.1 Preprocessing

The original idea for ALCHA was to use something akin to the C-style preprocessor, which is a macro language that is run prior to and independently from the main translation phases of the C language. It is mostly based on token substitution, and as such is most useful for user-defined syntactic sugar and platform independence by means of code selection. Code selection is implemented by simple if-statement structures that use macro constants to determine which part(s) of the code should be compiled, and which parts should be ignored. [104]

Macro-based preprocessing was found to be too restrictive for the goals of the ALCHA language. It is difficult to implement algorithmic circuit generation by using only macro-based substitution. A more flexible solution is required.

5.5.1.2 Templating

The C++ language features powerful means for generalisation, such as templating and automatic type inference. These mechanisms work well for compiled computer languages, especially general-purpose data containers and structures.

The ALCHA build process essentially results in the generation of a circuit, which can be thought of as a data structure, rather than something executable. An algorithmic circuit generation feature would execute during compile-time. The language would therefore offer

the user more flexibility if it provides dynamic interpretation features. Templating is therefore not a feature of ALCHA.

5.5.1.3 Scripting

Dynamically interpreted scripting is a powerful concept. The interpreter has run-time insight to the state of all the objects being used. This facilitates powerful features such as duck-typing, dynamic object definition and manipulation, etc.

Scripting is used in many auto-generation schemes, as well as existing algorithmic circuit generation solutions, as described in section 4.2.5. It was therefore chosen as the method by which to implement ALCHA's generalisation features.

5.5.2 Scripting Variables

The ALCHA scripting model makes use of fairly standard data types compared to existing interpreted languages. There are some considerations unique to firmware, however, which are further explored in this subsection.

5.5.2.1 Numbers

An early specification of the ALCHA language made a distinction between integers and real numbers, similar to most lower-level computer languages, but this distinction proved unnecessary. A single general type is sufficient for numeric variables.

Firmware supports bit vectors of arbitrary length. It is perfectly valid to define a 983-bit number, for instance. It is therefore imperative that the scripting engine can handle constants of any arbitrary size.

In addition, the ALCHA fixed-point model supports signals of arbitrary scaling, not only powers of two. It was therefore decided to make use of arbitrary-precision rational numbers as the basic numeric scripting type [125]. Arbitrary precision rational numbers are exact, and therefore do not introduce rounding errors when the compiler is deciding whether two scaling factors are related by a power-of-two or not.

In rare cases, rational numbers are not sufficient. The user can make use of transcendental functions in order to obtain the various constants employed in the circuit, in which case the numerical types must be able to support irrational numbers. In order to facilitate this,

ALCHA makes use of the GNU MPFR library [126]. The level of precision can be set at the time of definition (similar to the fixed-point format of a synthesisable net), as shown in listing 5.25. The expression follows the precision of the target variable.

```
// All these definitions are arbitrary precision rational numbers
// by default, but have different minimum floating-point precision.

num      A; // Double precision (the default)
num( 53) B; // Also double precision (53-bit significand)
num( 64) C; // Extended precision
num(128) D; // 128-bit significand
```

Listing 5.25. Example showing various numerical scripting variables

In addition to the `num` type being arbitrary-precision rational (or high-precision floating-point), they are also complex (i.e. has a real and imaginary part). This is in order to facilitate operations such as fast Fourier transforms used during filter-design.

5.5.2.2 Bytes and Characters

It is envisioned that the ALCHA scripting engine will be able to read and write files. The user might want to generate pin mapping based on a CAD-generated netlist, or make use of some other text-based input. The scripting engine must also be able to generate text-based output, such as C++ headers, auto-generated Python modules and L^AT_EX documentation.

All text-based functionality in ALCHA is done by means of character arrays. In order to be as platform independent as possible, all characters in the array, from the developer perspective, is UTF-32 encoded. The ALCHA built-in file manipulation functions make sure that the file format is correct, with UTF-8 being the default. The same character arrays are used for console output.

The ALCHA scripting engine might also need to read and write binary files. In this case, the file is modelled by means of byte arrays. Various built-in functions exist to convert between numbers, bytes and characters.

5.5.2.3 Using Attributes

Attributes can be of any type, including numbers, characters, and bytes, and arrays of those. In essence, attributes follow the same dynamic data structure model as JSON [127, 128]. Any object in ALCHA can therefore carry a set of scripting variable based metadata, which can be used for various scripting-related purposes, such as type-state oriented classes [129] and automatic resolution inference [130].

Object attributes can be read, modified or created at any point in the source code, which provides a powerful platform for algorithmic parametrisation. All clock signals, for example, carry a `frequency` attribute, which can be used by communication abstractions to automatically determine the parameters of baud-rate counters.

5.5.2.4 Compiler Insight

Most base types in ALCHA carry implicit attributes that can be used for compiler insight. These are loosely based on the built-in attributes of VHDL, and include attributes such as `width` and `fullscale` for fixed-point types, as well as `identifier` used to stringify the name.

5.5.3 Conditional Compilation

Conditional compilation can be implemented by means of an if-statement (or any other branching statement) that is only dependent on scripting variables. The interpreter will evaluate the scripting portion of the conditional expression and branch the rest of the compilation flow accordingly.

5.5.3.1 Target Independence

The ALCHA compiler makes use of language-defined global attributes in order to implement target independence. Listing 5.26 shows an example of a header in the main module that describes the target device.

```
'target_vendor = "Altera";  
'target_series = "MAX 10";  
'target_device = "10M08DAF484C8GES";  
'target_board = "BeMicro Max 10";
```

Listing 5.26. Example showing the target specifier block

When these attributes are missing, the compiler will generate device-independent SDC and pin-mapping TCL scripts, which the user can then pull into a user-defined project. This is useful when the device is not yet supported by the compiler in use.

5.5.3.2 Vendor-Provided IP

Vendor-provided IP modules can be incorporated into an ALCHA project by means of the `hdl` construct. If multiple devices are supported by the ALCHA module, conditional compilation can be used to pull in the appropriate IP module.

5.5.3.3 Block Scoped Variables

Conditional compilation, as implemented by ALCHA, implies that if-statements cannot contain block scoped variables. The code that is gated by the condition might be a series of declarations (such as pin definitions, for instance), or even an `import` statement. If the language supports block-scoped variables, the declarations will go out of scope the moment the conditional statement is finished, thereby nullifying all the declarations.

Block scoped variables could be implemented by introducing a special keyword for conditional compilation, such as the `generate` keyword used in VHDL and Verilog. This was deemed unnecessary verbosity for little return, so the ALCHA language does not implement block scoped variables.

5.5.4 Compiler Implications

The scripting engine of ALCHA runs as part of the compiler, not a preprocessing step. This means that the user can mix scripting and synthesisable types in function call parameters, for instance. The function is called by the dynamic interpreter, and subsequently converted to the circuit netlist that follows the body of the function.

This is the most powerful and versatile implementation method found. If the scripting engine was used as only a preprocessing step (as was the original idea), the scripting features are crippled and the possible applications much reduced.

As a result, ALCHA classes and functions (which result in HDL modules) are much more customisable than HDL parametrisation supports. This implies that there cannot be a one-to-one relationship between class and / or function declarations and the resulting HDL modules. Each class or function instance must have its own HDL module.

It might be possible to perform parametrisation grouping so that multiple class or function instances can be realised by a single parametrised HDL module, but this would be unnecessary

effort. The intention is that future versions of an ALCHA compiler would compile directly to a netlist, not an intermediary HDL representation.

5.5.5 Generalisation Summary

This section introduced the ALCHA scripting engine, including associated data types. It was deemed a more appropriate solution to the generalisation objective than preprocessing or templating.

Scripting provides the user with a powerful tool with which to implement algorithmic circuit generation, external tool input files, compiler insight, etc.

5.6 Summary

This chapter presented the language design associated with the five main language objectives formulated in section 4.2. Architectural abstraction is achieved by means of object orientation and a more L^AT_EX-like programming model. External device pins are moved to the leaf nodes of the project hierarchy and globally accessible objects can be defined at a higher level without further routing. This results in a cleaner and more maintainable code-base.

Direction inference and duck-typing allows the user more freedom in grouping and structuring. All signals of a bus can be grouped into a single object, regardless of individual signal direction, and different bus types can be used by the same module as long as the interface is consistently abstracted across the various types.

Verbosity is further reduced by means of the `rtl` and `fsm` constructs. The former reduces RTL boilerplate, and the latter allows the user to develop finite state machines with structured programming principles.

Control flow abstraction is further improved by allowing the user to abstract the detail of any part of a circuit or state machine by refactoring it into a function.

The ALCHA language supports fixed-point arithmetic as a core feature. The compiler automatically performs the required shifting, scaling and truncating required by fixed-point calculations.

Design constraints can be integrated with the source by means of attributes, which can be applied at the point of declaration or elsewhere in the code, thereby providing the user with sufficient freedom to choose the appropriate design pattern for the application. Pin location

can be specified at the point of pin declaration, whereas timing constraints can be applied by the peripheral abstraction module, for instance.

And finally, the generalisation objective is satisfied by means of a dynamically-interpreted scripting engine. This engine can be used to implement conditional compilation, algorithmic circuit generations, target independence, etc.

Chapter 6

Compiler Considerations

Even though the details of a compiler implementation is outside the scope of this dissertation, it is nonetheless important to consider the implementability of the ALCHA language features.

This chapter starts by considering the translation strategy. Section 6.1 considers the practicalities of the build process, parsability of the language and envisioned compiler output. The section concludes by considering the translation phases of the compiler.

Section 6.2 presents compiler considerations related to the core language features presented in chapter 5. This section is largely theoretical, because most of the features have not been implemented within the prototype compiler yet.

The experimental (pre-alpha) prototype compiler is available on GitHub [24]. At the time of writing, it is capable of compiling simple combinational statements from ALCHA all the way to a synthesisable Verilog-based Quartus project, including pin mapping and automated fixed-point scaling for addition, subtraction, multiplication and division.

6.1 Translation

This section investigates the practical aspects of translation. It considers the envisioned build chain, the relative ease by which the language grammar can be parsed into an abstract syntax tree and the intended form of compiler output.

6.1.1 The Build Chain

Although ALCHA is designed as a self-standing language, current vendor tools do not support it. It must therefore integrate into the existing infrastructure.

Figure 6.1 shows the conceptual block diagram of how the ALCHA system incorporates into the development chain. The ALCHA compiler produces a vendor-specific project. The user can then use the vendor-provided tools to compile the project to an FPGA bit-stream.

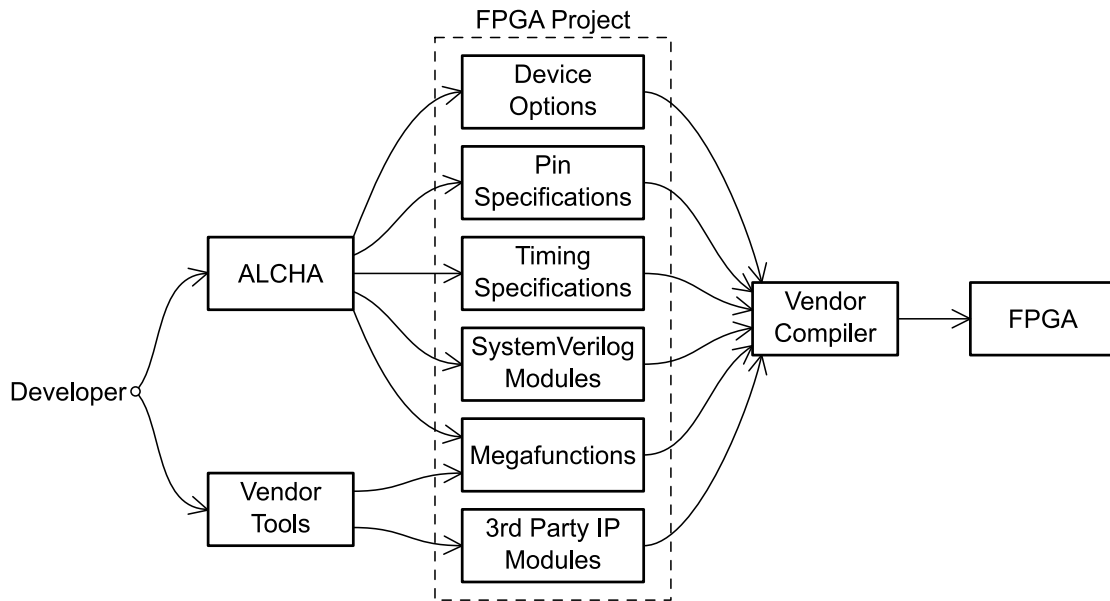


Fig. 6.1. Concept block diagram

6.1.2 Parsability

The extended Backus–Naur form (EBNF) of the ALCHA language is presented in appendix E. The language can be shown as parsable if a compiler can be implemented that translates any valid input to a usable abstract syntax tree (AST), while at the same time rejecting invalid input.

The prototype compiler implements a relatively simple scanner followed by a recursive-descent parser [131, 132]. Each rule in the EBNF is implemented as a separate function. The compiler successfully produces an AST, thereby showing that the ALCHA language is parsable.

The prototype compiler successfully processes the resulting AST by building the symbol tables, interpreting scripting expressions and producing a synthesisable netlist. This shows that the AST can be considered “usable” in the above context.

6.1.3 Compiler Output

The original idea behind the prototype compiler was to output human-readable Verilog, but this proved impractical. Early experiments showed that the Quartus compiler will often truncate critical sections of a large expression, thereby losing precision. The fixed-point casting process therefore requires many intermediary nets in order to fully specify what the circuit should look like, thereby cluttering the resulting output enough to make it impractical to consider “human readable”.

Another complication turned out to be the flexibility of the ALCHA language. Traditional HDL does not have parametrisation that is flexible enough to be able to implement any arbitrary ALCHA function as a single module. Instead, there must be a separate HDL module for every instance of a function.

The ALCHA prototype compiler therefore compiles directly to a netlist. This netlist is then translated to Verilog so that the vendor-provided tools can synthesise (or simulate) the circuit further. It is not meant to be human-readable. This said, the net names are chosen such that they can easily be traced back to the original ALCHA source.

6.1.4 Translation Stages

When first implementing the prototype compiler, the idea was to parse the entire code-base into an AST. The scripting engine was then to evaluate all scripting statements (or sub-trees) in the AST and result in a fully synthesisable tree (simply replacing the scripting variables with synthesisable constants), which can then be synthesised into the netlist.

This proved impractical because, among other reasons: it complicates the symbol table; makes it difficult to handle function calls and class instances that contain both scripting and synthesisable parameters; and complicates the code that implements the AST data-structure.

The superior translation strategy turned out to be handling the scripting and synthesisable parts during the same processing pass. The scripting statements are evaluated as required, and the netlist is built from the synthesisable statements. Sometimes the process is combined, such as when a single expression contains both synthesisable and scripting variables.

Optimisation can then be performed on the resulting netlist, which is a directed graph that is relatively easy to traverse. The exact details of this optimisation is outside the scope of this work, however.

6.2 Core Language Features

This section focuses on some key compiler implementation aspects of the language features introduced in chapter 5.

6.2.1 Fixed-Point

The experimental compiler features an implementation of the arbitrary fixed-point feature for combinational circuits [24]. It has been implemented as described in section 5.3. Unit-testing showed that the concept works as desired for all cases of addition, subtraction, multiplication, division and type-casting.

6.2.2 Maintaining Precision

The fixed-point engine automatically inserts multiply and shift operations in order to convert between fixed-point formats. Say, for instance, the compiler is faced with the ALCHA code presented in listing 6.1. A naive implementation might generate the Verilog in listing 6.2, because the native multiplier size of most modern FPGAs are 18-bit, and the user has not specified otherwise.

```
net(14, 1) a;  
net(14, 1) b = a * 0.1;
```

Listing 6.1. Example showing a simple fixed-point operation

```
wire [13:0]a;  
wire [13:0]b = (a * 18'h33333) >> 21;
```

Listing 6.2. Naive implementation of the Verilog equivalent of listing 6.1

This would be a valid solution, if not for the rules governing Verilog signal widths. The Verilog compiler will evaluate the multiplication with an 18-bit answer, effectively reducing the constant precision to 4-bit instead of the 18-bit specified. It does not keep maximal intermediary precision, which has been verified by using the Quartus RTL viewer tool.

The correct implementation is to specify the constant to have a width of 32-bit, which is the desired width after multiplication, but before the shift. This solution is problematic if the multiplication is done on two signals, instead of one signal and one constant. The solution in this case is to introduce a temporary intermediary signal that is explicitly defined to have the desired width, as presented in listing 6.3.

```
wire [13:0]a;  
wire [31:0]\w..0000000 = a * 18'h33333;  
wire [13:0]b = (\w..0000000 ) >> 21;
```

Listing 6.3. Correct implementation of the Verilog equivalent of listing 6.1

The naming convention of the temporary signals was chosen to minimise name clashes while making it easy to use in TCL scripting to assign timing requirements and location assignments, among other attributes. Escaped identifiers containing special characters are used so that the name cannot clash with any existing ALCHA identifier. The `\w` means “wire” (as opposed to `\m` for “module”, for example). The double-period is used so that it does not clash with structure hierarchy (e.g. `\AD5681R.SCL`), and the number is simply sequentially assigned for as many temporary signals the design might require.

6.2.3 Branching and Loops

Synthesisable branching can be implemented by means of multiplexer circuits, which in turn can be represented by means of HDL branching statements. Loops based on a scripting variables can be implemented by unrolling the loop, whereas loops based on synthesisable variables are only valid within an FSM construct, and can be implemented by means of the control flow of the FSM.

6.2.4 Register Transfer Logic

Statements within an RTL construct can be implemented by inserting a register at the head of the target net’s expression tree. The existing tree becomes the reset value, and the new tree is assigned to the ‘data’ input of the register. Special attributes (internal to the compiler) can be used to detect multiple assignment, thereby causing a compile error. All other aspects of the assignment are similar to those of combinational assignment.

The result of this process, however, is a collection of nets with register-based expression trees assigned to them. They are not grouped into the original code structure. The resulting Verilog will therefore be grouped by net name, not logical circuit flow, which makes it challenging to follow the resulting circuit.

6.2.5 Finite State Machines

This is arguably the most challenging feature to implement, and is mostly left to future work. It is envisioned that the compiler might synthesise an intermediary execution flow graph. This flow graph can then be simulated directly without further synthesis, thereby accelerating simulation. It can also be simplified and optimised by using compiler optimisation algorithms used in modern software.

After synthesising and optimising the control flow graph, the compiler can synthesise the RTL equivalent by generating a state for each node in the graph, after which the netlist extraction is relatively simple.

Another advantage of the flow graph intermediary representation is the ability to extract multi-cycle information, which can then be transferred to the timing requirements. This can potentially result in improvements in maximum clock speed achievable for a given circuit, but such investigation is left to future work.

6.2.6 Function Calls

The details of function calls, as well as the associated practicalities of implementation, have been presented in section 5.2. Function calls applied to combinational or RTL circuits are relatively simple. Every function call is a new instance of the module that implements the function. For functions called from FSM context, however, optimisation can become a bit more complex.

The compiler knows that the function is called from within a specific FSM state. If the same function is then called from within another state, the same instance can be reused instead of creating a new instance. This optimisation strategy is only possible if the function is called with similar parameters (especially scripting parameters). The compiler must therefore remember the conditions of the previous function call so that it can compare the conditions of the current call, thereby deciding whether to create a new instance or reuse the old one. The details of this optimisation is left to future work.

6.2.7 Attributes and Scripting

Attributes are fairly simple to implement. It is strongly based on the dynamic data structures prevalent in modern interpreted languages, and can therefore be implemented using the same methods.

Similarly, the scripting engine itself is essentially a dynamically interpreted language. A typical implementation would translate the source code to an abstract syntax tree, which is then traversed and interpreted on-the-fly in order to produce the netlist.

The arbitrary precision rational numbers can be implemented by means of libraries such as GNU MP and MPFR [125, 126].

6.3 Summary

This section started by testing the practicality of the ALCHA grammar, as well as the fixed-point concept. A prototype compiler was implemented for the purpose, which showed that the language grammar is parsable, and the fixed-point translation feature is practical to implement. It was found, however, that translation requires numerous intermediary signals in order to maintain calculation precision throughout the HDL compilation process.

It was further found that it is more practical to implement scripting evaluation and netlist generation as a combined single-pass translation flow. Implementing the scripting as a pre-processing step results in various difficulties related to the compiler internal architecture.

The rest of the language practicality was considered from a more empirical perspective. Combinational branching and loops have been implemented for existing HDL compilers, and are therefore considered practical to implement. Similarly, the RTL construct is simply syntactic sugar for the more verbose HDL equivalent. The FSM construct is considered practical to implement by imagining a control-flow graph intermediate step to producing the equivalent RTL.

Simple functions can be implemented by means of substitution, and FSM-based functions can be synthesised by following a handshaking control-flow strategy, as described in section 5.2. Function-related optimisation, however, is a complex topic and left to future work.

Attributes can be implemented as a dynamic dictionary-based data structure prevalent in most modern interpreted languages, and the scripting engine itself can be implemented as an interpreted language.

Chapter 7

Evaluation

Chapter 5 presented the language design, based on the requirements outlined in chapter 4. This chapter evaluates the language by means of a radar-based case study.

The radar firmware is representative of a real radar system, even though various design details have been obfuscated and / or modified to prevent intellectual property violations.

A large portion of the source code presented in this chapter is abbreviated by means of to-do comments (i.e. comments starting with `!!`). The interested reader can download the full ALCHA source of the case study from [GitHub](#) [85].

Section 7.1 presents an overview of the radar system used for the case study. Section 7.2 then presents the ALCHA implementation, while analysing the language from the language objectives point of view. Section 7.3 evaluates the language power of the resulting code-base, according the metrics described in section 3.3.2.

7.1 Project Overview

A radar-based project was chosen because of two main reasons. First, the author is familiar with radar-related firmware and systems and second, radar-related projects tend to be multiple-discipline, thereby offering a wide variety of test cases in a single project.

The project chosen is a simplified frequency modulated continuous wave (FMCW) radar, based on an ADF4159 synthesiser and ADS7056 analogue to digital converter (ADC). The general hardware architecture is presented in figure 7.1, which is roughly in line with the simplified design presented in figure 4.4 in section 4.2.1.3.

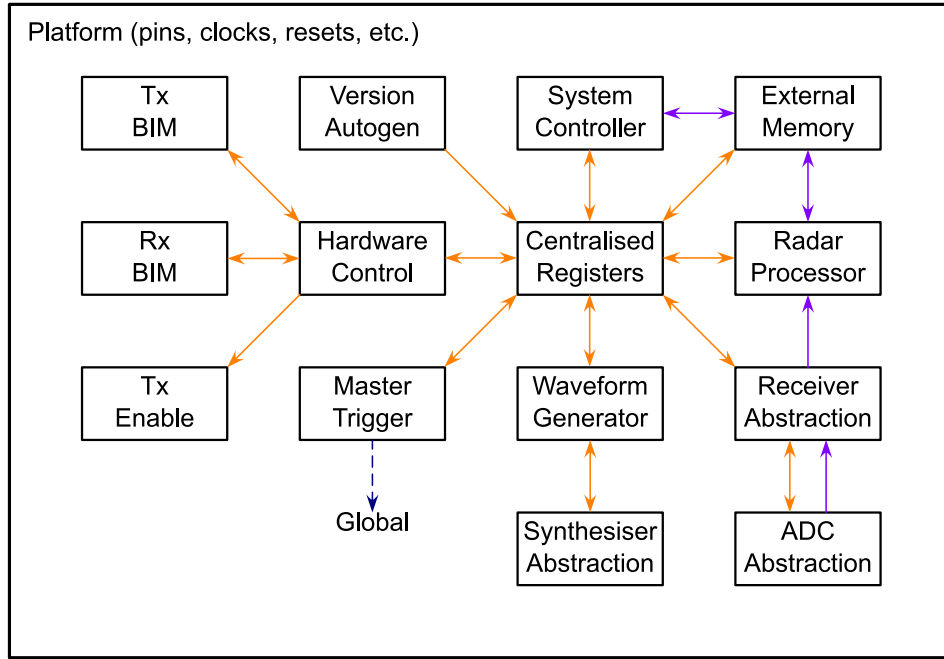


Fig. 7.1. Overview block diagram of the case study project

Orange signals indicate control, whereas purple signals indicate data flow. The data processor stores its output in external memory, where the system controller can fetch it for further processing.

7.1.1 Platform Description

The platform includes pin definitions, clock generation blocks (i.e. phase-locked loops), reset circuits, etc. All other modules have access to the globally-defined structures within the platform section.

7.1.2 Master Trigger and System Timing

There is a single master trigger in this design, globally accessible to all other modules. All timing is derived from this trigger. A simplified timing diagram is presented in figure 7.2. The master trigger is slightly early, so that all filters and other modules can have time to settle and initialise before the real data arrives (t_{delay}). The sweep trigger is derived from this master trigger, which in turn triggers the ADF4159 synthesiser to start the sweep.

The ADC Valid signal informs the packetiser that the ADC is producing valid samples that should be included in the packet. This validating signal is delayed so that the radar

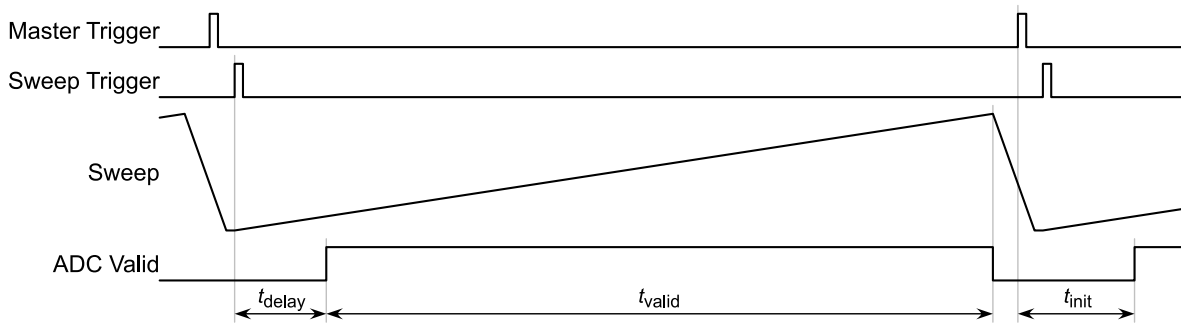


Fig. 7.2. Simplified timing diagram of the case study project

signal has time to propagate to the target and back, as well as taking the ADC latency into account (t_{delay}). The down-sweep is not triggered from the FPGA. The ADF4519 IC does this automatically.

7.1.3 System Controller

The system controller in this case study is an embedded ARM microcontroller instantiated by means of Quartus Platform Designer. Platform Designer is further used to convert the AXI busses of the ARM microcontroller to Avalon busses, which are then exposed to the FPGA side.

7.1.4 Registers Decoder

The system control is based around a centralised registers bank. This means that there is a single register address decoder instead of a smaller one within each submodule. This architecture is useful in highly integrated systems, such as radar, where more than one module requires access to a specific control register.

The system controller makes use of the Avalon bus infrastructure to read and write registers. The Registers module decodes the addresses and distributes / collects the values to / from the various system modules.

7.1.5 Hardware Control

The hardware control block encapsulates the built-in measurement controllers, as well as anything else that must be controlled on the hardware level (such as RF switches). In the most simple case, this includes only a transmit enable switch on the power amplifier.

7.1.6 Waveform Generator

The waveform generator encapsulates the ADF4159 based synthesiser, as well as any related timing controllers. A future iteration of the radar could, for instance, make use of an AD9915 DDS based synthesiser instead, so the encapsulation is advantageous.

7.1.7 Receiver Abstraction

The analogue to digital conversion (ADC) is encapsulated by the receiver abstraction, which is also the first stage of the data chain. It is relatively easy to change the ADC to a different device without changing any other module in the data chain, so the receiver abstraction is also responsible to transform the data from the ADC abstraction to the format expected by the rest of the chain.

The rest of the chain expects packetised data, i.e. one packet per sweep. It is therefore the job of the receiver abstraction to break the data into packets of appropriate size.

7.1.8 Radar Processor

The radar processor, presented in figure 7.3, is the heart of the firmware section of the overall radar digital processor. It contains the range and Doppler FFTs, as well as the intermediate corner-turn required. The resulting map is then alpha-filtered before sending it to the ARM processor by means of a shared section of external memory.

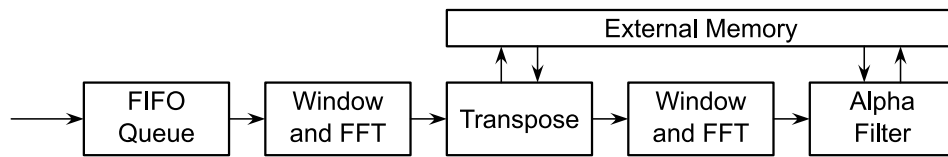


Fig. 7.3. Functional block diagram of the case study digital processing chain

7.2 Implementation in ALCHA

This section presents the findings that resulted from implementing the case study. In the interest of brevity, this section contains only the findings: the fully detailed explanation pertaining to the implementation is presented in appendix D instead. The reader is encouraged to refer to the appendix in order for the findings presented in this section to make sense.

7.2.1 Architectural Abstraction

The `main` module, as presented in listing D.1 is short and concise, successfully hiding the bulk of low level detail in abstraction.

Similarly, each of the subsystems are neatly contained within a relatively short module that instantiates the submodules and hook up the registers. These modules are not cluttered with low level detail, thereby making them easy to follow. This shows that the language objective of architectural abstraction, as discussed in section 4.2.1, is practical within the ALCHA language.

The remainder of this subsection presents various findings of interest related to the ALCHA architectural and programming models.

7.2.1.1 Libraries and Globals

The case study suggests that it is impractical to allow the library units to make use of the globally defined objects directly. The library modules in the case study implementation do not make direct use of global clock and reset signals, because the user might want to use different instances of the library modules in different clock and / or reset domains. These signals are therefore passed to the module instance by means of the class constructor parameters.

Similarly, the `FirmwareVersion` module cannot assume that all projects that make use of it will also make use of the `RegistersDecoder` module for control purposes. The library function therefore cannot automatically add it's own registers to the bus like the project-specific modules do. Adding the registers to the bus is therefore done directly in the `main` module, thereby reducing encapsulation, but the function-driven way in which this is implemented minimises the impact.

For the same reason, `Registers` is instantiated in the `main` module. There is no way for the library function to know exactly what bus to connect the registers to, or even how many instances of `RegisterDecoder` the project requires, so the instance is created in the main module. The amount of code to create the instance is minimal and specifies only the essentials, however, so the impact on main module verbosity is small.

A similar pattern is visible within each of the `SubSystem/...` modules, but this detail is successfully hidden from the main module and encapsulated within each subsystem module by means of the import mechanism.

7.2.1.2 Coding Style

One might argue that the global clock related issues highlighted in section 7.2.1.1 arose because of coding style, and that the restriction is artificial. As a relatively simple solution, different instances of the same class can be put in different clock domains by means of aliases.

If the library module expects a clock signal named `Clk` and a reset signal named `Reset`, for example, the user can make sure that aliases for those signals exist at the point of class instantiation. This would allow the user to put a whole circuit in a specific clock domain by simply defining an alias in a higher level routing module, which would significantly reduce the verbosity of listing D.32.

The `RadarProcessor` module can simply define the appropriate aliases once at the top of the module, and all DSP module instances from then onwards will use the correct global clock and reset signals. Furthermore, because the `RadarProcessor` module is imported into its own namespace, the clock domain aliases would be local to that import.

7.2.1.3 Frameworks

Other library-related issues highlighted in section 7.2.1.1 can be mitigated by implementing a framework, which is a form of architectural abstraction quite popular in higher level software systems¹. The generation of a framework is outside the scope of this case study, but it is fairly simple to imagine how such a framework could be created.

The framework should target a specific type of project, such as a radar. The radar firmware should then be implemented with a specific control scheme and overall architectural pattern that is shared among many products. Consistent naming-convention can then be used to implement a framework of library modules that can assume the presence of certain global structures and objects, thereby simplifying the higher level modules that makes use of them.

7.2.1.4 Namespaces

The ALCHA namespace model have been successfully used to reduce global namespace clutter, while also reducing routing clutter by making objects globally accessible where appropriate.

¹Popular framewoks include Microsoft .NET, Vue.js, Angular JS, jQuery, etc.

Some modules are imported into the global namespace directly, and others into their own namespace. The only truly global signals are the pins, clocks, resets and the master trigger. The only global class instances are `SystemController`, `Registers` and `Firmware`. Everything else is in their own namespace.

This has the added benefit of ensuring that the namespace hierarchy follows the project tree, which partitions the project in a convenient way so that each firmware developer can work on his / her own subsystem without being concerned about namespace clutter or name clashes with the other subsystems.

7.2.1.5 Aliases

The project was implemented on a DE0 Nano SoC development kit, the platform descriptor of which is defined as a generic library function. The pins are therefore specific to the generic development kit, and not the final project.

The `alias` language feature of ALCHA proved instrumental in linking the project-specific pin names to the generic pins defined for the development kit. When the project is moved to the final hardware, the platform descriptor can be updated without the need to touch any of the other modules.

7.2.1.6 Bidirectional Ports on HDL Modules

ALCHA only supports two-valued logic, which poses a problem when connecting bidirectional HDL ports. One example of this is the `DQ` port of the SDRAM driver, which is a part of the ARM processor system.

A secondary, but related problem is that some HDL code exposes differential ports as two separate signals, which is incompatible with the ALCHA model of representing differential signals as a single net.

This incompatibility was a major limitation of the language at the time of implementing the case study project, so the `hdl_map` method was introduced to `pin` objects. This special `pin` object method only accepts arguments that evaluate to `hdl` ports and translates to a standard HDL-style component instance port mapping. Furthermore, this special `hdl_map` method takes an optional second argument used when mapping individual positive and negative components to a differential pin.

7.2.1.7 Object Orientation

Object orientation, especially the inheritance and polymorphism aspects, turned out to be a very useful feature. The `LTC2991` module is implemented as a derived class based on the `I2C` module. Similarly, `AvalonMaster` and `AvalonSlave` modules are derived from `AvalonInterface`.

In the first case, the base class provides low-level functionality (i.e. implementing the I²C protocol). In the latter case, the base class provides the interface signals so that the derived class can be used anywhere that interface is expected.

7.2.1.8 Adding Modules to the Bus

In various parts of the case study implementation, modules can add themselves to the bus. This is true for both Avalon and I²C buses and proved to be a useful design pattern.

Registers are handled in similar fashion, where a module can turn nets into registers within the centralised register controller by means of calling register controller class methods. All administrative detail, such as register addresses, is taken care of by the `RegistersDecoder` module.

7.2.1.9 Pipes and Filters Model

The digital signal processing (DSP) chain was implemented by using a pipes and filters architectural pattern. Two implementation styles were considered: one where the class instance is used directly in the pipes chain (listing [D.31](#)), and a second where the user creates named instances of the DSP modules, which are then piped together (listing [D.32](#)).

Even though more concise, the first option was found to be impractical because the user would have no means to do anything with the module instances after piping them together. The second option is more verbose, but provides the user with handles that can be used to manipulate the modules by, for e.g. turning the parameters into registers.

7.2.2 Control Flow Abstraction

The control flow abstraction features of ALCHA include the `fsm` construct (with its associated structural programming features) and functions.

7.2.2.1 FSM Construct

The `fsm` construct was found to be a bit of a mixed bag. The user must choose carefully whether the `fsm` construct is the correct tool for the job, or whether the `rtl` construct might be more appropriate.

The I2C module, for example, would have been easier to implement using `rtl` than the `fsm` version presented in the implementation. By contrast, the ADF4159 SPI implementation presented in listing D.24 shows the elegance that can be obtained with an `fsm`-based higher level implementation.

The `fsm` construct was, in general, found to be more useful for implementing higher-level modules that have a more linear flow while calling lower-level functions. The `rtl` construct was found to be more appropriate for implementing low-level interfaces that communicate directly with external ICs.

7.2.2.2 Functions

Functions are useful for information hiding and increasing readability of the code. Consider listing 7.1 for example, where the user used code-folding to hide the function contents. It is immediately evident what the code does: just after reset it initialises the module and updates the registers. After reset and initialisation, the registers are updated upon user demand by means of an update-busy handshake mechanism.

Functions do have limitations, however. It would be tempting to refactor the repetitive structure of the `UpdateRegisters` function into something akin to listing 7.2, but this would change the timing of the circuit. Each function call effectively adds one clock cycle latency to the process, because the first statement of the function FSM is executed in the clock-cycle after the calling state (see section 5.2.3.2).

There is no easy solution to this problem. One could consider adding an `inline` keyword to the language, which would do the same thing as in the C family of languages, but that would only work for a small subset of function types (i.e. functions that only contain a single `fsm` construct, and no local signals that require a reset), which will complicate the compiler and possibly confuse the user. For now, the function call overhead is something that the user must be aware of, and the acceptability of the compromise is left to the user.

```

private void Send_SPI(net(8) Address, net(32) Value){
    // ----- 14 Lines -----
}

private void Init(){
    // ----- 10 Lines -----
}

private void UpdateRegisters(){
    // ----- 33 Lines -----
}

fsm(Clk, Reset){
    Init();
    UpdateRegisters();

    loop{
        if(opBusy){
            while(ipUpdate);
            opBusy = 0;
        }
        while(~ipUpdate);
        opBusy = 1;

        UpdateRegisters();
    }
}

```

Listing 7.1. Main state machine implementation of the AD9915 module

```

private void UpdateRegister(net(8) Address, net(32) Port, net(32) Saved){
    fsm{
        if(Port != Saved){
            Saved = Port,
            Send_SPI(Address, Port);
        }else;
    }
}

private void UpdateRegisters(){
    fsm{
        UpdateRegister(0x04, ipFreqLowerLimit, FreqLowerLimit);
        UpdateRegister(0x05, ipFreqUpperLimit, FreqUpperLimit);
        UpdateRegister(0x06, ipStepUp, StepUp);
        UpdateRegister(0x07, ipStepDown, StepDown);
        UpdateRegister(0x08, :(ipSlopeDown, ipSlopeUp), :(SlopeDown, SlopeUp));

        if(RegsChanged){
            opIO_Update = 1;
            opIO_Update = 0,
            RegsChanged = false;
        }else;
    }
}

```

Listing 7.2. Refactored version of the AD9915 UpdateRegisters function

7.2.3 Fixed-point Abstraction

The DSP chain was implemented by using the fixed-point mechanisms. The full-scale attribute of the external interfaces of each DSP block seamlessly takes care of bit-shifts and data format transformations between the various components.

The `Window` module, for instance, uses coefficients that have an auto-generated full-scale value that maximises the efficiency of the ROM bits. The automated fixed-point scaling takes care of the bit-shifts required, regardless of the optimal coefficient full-scale.

7.2.4 Design Constraints

There are a variety of design constraints present in the project. The pin locations, as well as other pin-related attributes, are conveniently specified at the point of declaration in the `Platform` module (or, more specifically, the `DE0-Nano-SoC` module imported by the `Platform` module).

External timing requirements are successfully split between the `Platform` module and peripheral abstraction class. The constraints of the ADS7056 pins, for example, can be seen in listings [D.28](#) and [D.26](#).

The language objective of design constraint integration is therefore adequately satisfied.

7.2.5 Generalisation

This case study made use of two main aspects of the generalisation objective, i.e. high level parametrisation and algorithmic circuit generation.

7.2.5.1 High Level Parametrisation

While implementing the abstraction for the ADF4159 synthesiser IC, the ALCHA scripting engine was used to translate between human-readable constants (such as starting frequency, bandwidth and ramp times) to the rather complicated set of registers required by the IC.

This turned out to be quite a powerful technique. The user can set up hard-coded parameters, or register reset values, by means of high level functions, thereby successfully abstracting the complexities of the external IC register set.

A similar strategy was followed for the `Window` module in order to preload the ROM block with the window coefficients. The window function is called by name (e.g. "`Hann`" or "`Hamming`"), and the scripting engine makes sure that the correctly scaled coefficients are loaded into the ROM block.

7.2.5.2 Algorithmic Circuit Creation

The `RegistersDecoder` module is a good example of algorithmic circuit generation, and the level of abstraction the process can provide. Registers are added to the register pool throughout the design, and the actual register decoder circuitry is auto-generated at the end of the compilation process. Register addresses are automatically assigned, and the resulting C++ classes and L^AT_EX-based documentation is automatically kept up to date by means of the scripting engine.

Similarly, the `MutEx` module allows various modules to attach themselves to the mutex pool. The arbitration circuit is then auto-generated at the end of the compilation cycle.

7.2.5.3 The Finally Function

Various use cases were found where it would be useful if the compiler can automatically call a special class method during the final stages of compilation. Two examples include generating the register bank from the temporary scripting arrays internal to the `RegistersDecoder` module, and generating a mutex controller of appropriate size after all interested modules have connected themselves to the `MutEx` module.

The language was modified by introducing a new keyword: `finally`. The `finally` member function (or method) is called automatically at the end of parsing the source, so that the user does not need to. The case study source was then updated accordingly: the interested reader can refer to the [feature/finally tag](#) in the Git repository [85].

As a potential future feature, this same `finally` function can be used to implement destructor-like behaviour for pure scripting classes, if so desired.

7.3 Language Power

This section evaluates the language power of ALCHA, as compared with an equivalent Verilog implementation. The criteria outlined in section 3.3.2 is used. The Verilog-based project is available in the [HDL/Radar](#) folder of the GitHub repository [85].

The Verilog implementation follows the ALCHA version as closely as possible. Scripting regions that are not possible in Verilog have been implemented by means of TCL or Python based auto-generation scripts, and features not yet implemented in the ALCHA version have been left unimplemented in the Verilog as well.

7.3.1 Code Complexity

Code complexity can be estimated by counting the number of lines of code (LoC). The more lines of code a code-base contains, the more instructions the programmer needed to give the compiler to produce the same resulting circuit.

The project is divided into four sections: the top-level routing and design constraints; Library modules; DSP modules; and other subsystems that do not fit into the previous three categories. The corresponding Git tag used when the lines were counted for this section is [Counting_Lines_of_Code](#).

7.3.1.1 Top-level Routing and Design Constraints

Table 7.1 presents the lines of code required by the Verilog and ALCHA implementations for the top-level routing and design constraints.

One of the language goals of ALCHA is to reduce routing clutter, especially in the top-level module. This is visible in the difference between the `Radar.v` module, and the equivalent `main.alc` and `MasterTrigger.alc` modules.

Another improvement is evident in the pin-mapping and timing constraints. The ALCHA version contains all pin mapping and other project-specific properties as part of the pin definitions in the `DE0-Nano-SoC.alc` module. By contrast, the Verilog implementation must separately define the pins in `DE0-Nano-SoC.vh`, and then map the pins and other properties in `Radar.qsf`.

The `Platform` modules in both ALCHA and Verilog contain mostly aliases and signal mapping, so the sizes are equivalent.

TABLE 7.1
LINES OF CODE: TOP-LEVEL PROJECT

Verilog File	LoC	ALCHA File
Radar.v	74	18 main.alc
		5 MasterTrigger.alc
DE0-Nano-SoC.vh	63	99 DE0-Nano-SoC.alc
Radar.qsf	247	
Radar.sdc	27	
Platform.vh	49	53 Platform.alc
Total	460	175 (38%)

7.3.1.2 Library Modules

Table 7.2 presents the lines of code required by the various library modules.

The implementations are relatively short in both Verilog and ALCHA versions, and therefore equivalent in length with ALCHA showing only a marginal advantage in most cases. The most notable differences are present for modules where the ALCHA implementation can take full advantage of the `fsm` construct, such as the `AD9915` and `I2C` modules.

The `LTC2991` module is largely unimplemented, so the numbers in the table represent only the routing and generalisation overhead, which is much larger in Verilog.

The ALCHA versions of the RAM and ROM modules are quite large, because the full HDL module definition of the RAM or ROM module is followed by a port-mapping section. The Verilog module, by contrast, only requires the port-mapping section.

TABLE 7.2
LINES OF CODE: LIBRARY MODULES

Verilog File	LoC		ALCHA File
AD9915.v	175	127	AD9915.alc
ADF4159.v	168	195	ADF4159.alc
ADF4159.py	102		
ADS7056.v	31	42	ADS7056.alc
AvalonArbiter.v	30	15	AvalonInterface.alc
		30	AvalonMaster.alc
		10	AvalonSlave.alc
DelayedReset.v	16	11	DelayedReset.alc
DualPortRAM.v	44	61	DualPortRAM.alc
DualPortROM.v	48	74	DualPortROM.alc
DualPortROM.py	19		
FullDualPortRAM.v	56	77	FullDualPortRAM.alc
I2C.v	109	64	I2C.alc
LTC2991.v	61	32	LTC2991.alc
MutEx.v	17	21	MutEx.alc
NCO.v	15	16	NCO.alc
PLL_CycloneV.v	70	83	PLL_CycloneV.alc
SinCos.v	65	47	SinCos.alc
TriggerDelay.v	27	22	TriggerDelay.alc
TriggerGen.v	18	13	TriggerGen.alc
WatchDog.v	21	18	WatchDog.alc
Total	1092	958	(88%)

7.3.1.3 DSP Modules

Table 7.3 presents the lines of code required by the various DSP modules. It should be noted that the `FFT`, `Transpose` and `AlphaFilter` modules are not implemented, so the code contained therein represent black-box modules with code only sufficient for the compiler to validate the hierarchical routing.

TABLE 7.3
LINES OF CODE: DSP MODULES

Verilog File	LoC	ALCHA File
FIFO.v	51	35 FIFO.alc
Window.v	69	71 Window.alc
Window.py	41	
RealFFT.v	6	43 FFT.alc
FFT.v	6	
Transpose.v	16	0 Transpose.alc
AlphaFilter.v	16	0 AlphaFilter.alc
Total	205	149 (73%)

The fully-implemented DSP modules (i.e. `FIFO` and `Window`) are both smaller in ALCHA than in Verilog.

7.3.1.4 Other Subsystems

Table 7.4 presents the lines of code used by intermediary routing modules and peripheral subsystems.

The ALCHA version of `SystemController` is larger, as expected. It contains a large `hdl` construct that defines the `SoC_System` module, followed by the port mapping section. The Verilog version, by contrast, requires only the port mapping section, because Quartus already knows what the module looks like.

The `RegistersDecoder` module is not a fair comparison, because the ALCHA version performs a significant amount of automation, whereas the Verilog version is hand-coded. The user could implement an auto-generation tool that mitigates the code duplication present in the Verilog version, thereby significantly reducing the number of lines of code required to implement the registers module.

TABLE 7.4
LINES OF CODE: OTHER SUBSYSTEMS

Verilog File	LoC	ALCHA File
SystemController.vh	124	219 SystemController.alc
RegistersDecoder.v	152	57 RegistersDecoder.alc
FirmwareVersion.tcl	44	13 FirmwareVersion.alc
HardwareControl.v	66	44 HardwareControl.alc
WaveformGenerator.v	45	27 WaveformGenerator.alc
WaveformGenerator.py	8	
ReceiverAbstraction.v	70	35 ReceiverAbstraction.alc
DebugStreamer.v	35	25 DebugStreamer.alc
RadarProcessor.v	179	39 RadarProcessor.alc
		18 Streams.alc
Total	723	477 (66%)

The rest of the modules are consistently smaller in ALCHA than in Verilog. The largest advantage is the pipes-and-filters architecture utilised in the **RadarProcessor** module. All the port-mapping present in the Verilog version is reduced to a single line of code in ALCHA.

7.3.2 Cost of Maintenance

The cost of maintenance can be measured by investigating the amount of code that require modification when implementing a design change. The maintenance effort is then measured based on the Git diff stat, which summarises the number of changes made to each file. Comparisons between the two implementations include both the number of files touched by the modification, as well as the number of changes.

The command-line instruction used is shown below. In other words, the Git tag that implements the change is compared with the node immediately before it, while ignoring changes in spaces. In each test case, the interested reader can view the modification in [the GitHub repository \[85\]](#).

```
git diff --stat --ignore-all-space <tag>~1 <tag>
```

7.3.2.1 Registers

A relatively common maintenance task is to turn previously hard-coded parameters into registers that can then be put under software control. For this test, the six ramp parameters of the waveform generator is modified into registers. The relevant Git tag is [modify/NewRegisters](#) and the results are presented in table 7.5.

TABLE 7.5
COST OF MAINTENANCE: REGISTERS

File	Insertions / Deletions		
WaveformGenerator.alc	8	0	+++++++
RegistersDecoder.v	24	0	+++++++
WaveformGenerator.v	6	6	+++++-----

The ALCHA modification is simple. The `WaveformGenerator` module gets 6 extra lines of code, along with an extra 2 blank lines for increased readability. The register values at reset is taken care of by the initial values assigned in the `ADF4159` class, and the rest of the register infrastructure is automated by the `RegistersDecoder` library module.

The Verilog version requires updates in five places: the structures used to ease register routing throughout the project, the “reset” section, the “read” decoder, the “write” decoder and the port mapping of the `ADF4159` instance. The modification totals 24 new lines and 6 modified lines, compared to the 6 new lines required by the ALCHA implementation.

The impact can be reduced by means of an auto-generation tool used in order to reduce code duplication. Instead of manually defining the register structures and code to handle the reset, read and write sections, the auto-generation tool would do that.

The same modification in that case would reduce to adding the 6 registers to the input of the auto-generation script, along with the 6 modifications to the port mapping, which is a 60% improvement. It is still twice the effort when compared to the ALCHA implementation, however.

7.3.2.2 Changing the Synthesiser

Another common maintenance task is to update the firmware to accommodate a hardware change. Imagine that the user would like to change the ADF4159 based synthesiser to an AD9915 based one. The relevant Git tag is [modify/NewSynthesiser](#) and the results are presented in table 7.6.

TABLE 7.6
COST OF MAINTENANCE: NEW SYNTHESISER

File	Insertions / Deletions		
Platform.alc	9	9	++++-----
WaveformGenerator.alc	27	22	+++++++++++-----
Platform.vh	9	5	++++--
Radar.v	9	5	++++--
RegistersDecoder.v	20	43	+++++++-----
WaveformGenerator.v	25	36	+++++++++++-----

The modification in ALCHA involved updating the pin mapping in the `Platform` module and hooking up the AD9915 instead of the ADF4159 peripheral abstraction in the `WaveformGenerator` module. The Verilog update involved the same modifications as with the ALCHA version, but with a more complicated registers update as well as a rerouting of the device pins through the design hierarchy.

There are no updates to the QSF or SDC files, because the implementation makes use of a development kit and maps the pins according to the development kit pin names. The `Platform` module provides a pin naming abstraction layer, which implies that there are no platform-level changes. Timing requirement updates are not required either, because the SPI interface in both ADF4159 and AD9915 cases is slow enough that no constraints are required.

The bulk of the difference between the 67 modifications in the ALCHA version versus the 152 modifications in the Verilog version can therefore be attributed to register updates and hierarchical signal routing.

7.3.2.3 Adding Test Points

Although modern firmware development include powerful tools for debugging purposes, it is sometimes required to gain insight into the circuit behaviour while running in-system on real hardware. General-purpose debug pins are often included in the design for this purpose.

For this test, the user is interested in the exact timing of the internal triggers and packet streams, so adds debug routing accordingly. The relevant Git tag is [modify/Debug](#) and the results are presented in table 7.7.

TABLE 7.7
COST OF MAINTENANCE: TEST POINTS

File	Insertions / Deletions		
Platform.alc	2	3	+--
main.alc	8	0	++++
TriggerDelay.alc	1	0	+
TriggerGen.alc	1	0	+
ReceiverAbstraction.alc	4	0	++
FIFO.alc	4	0	++
Window.alc	8	0	++++
FFT.alc	8	0	++++
Transpose.alc	5	0	+++
AlphaFilter.alc	5	0	+++
Platform.vh	1	1	+--
Radar.v	59	3	+++++++++++++++++++++++--
WaveformGenerator.v	3	3	+--
ReceiverAbstraction.v	3	3	+--
RadarProcessor.v	20	20	+++++++-----

The strategies between the two implementations differ, even though the resulting circuits are equivalent. In ALCHA, the user defines a function at the top of the `main` module as shown in listing 7.3. The library modules are then modified to automatically add debug routing as required. The Verilog implementation turns the signal definitions into port definitions and then hook up the debug pins as part of the top-level port-mapping.

This means that the ALCHA version is modified at a lower level, which implies more files. Even though more files have been changed, the changes are all generic and was done with

```

num DebugCounter = 0;
void AddDebug(net Signal){
    Debug[DebugCounter] = Signal;
    print(:["Debug[" , $(DebugCounter), "] <= ", Signal'full_instance_name, "\n"]);
    DebugCounter++;
}

```

Listing 7.3. Helper function implemented to aid in adding debug pins

copy-and-paste, thereby taking minimal effort. The total changes in ALCHA was 49, whereas the Verilog version required 116 modifications.

7.4 Summary

This section made use of a radar-based case study in order to evaluate the ALCHA language against the following criteria:

- Adequate satisfaction of the language objectives, as formulated in section 4.2
- Language power, as defined in section 3.3.2

In addition, the inner loop of the engineering method (see section 3.1.2) was performed *in situ*. As a result, various new language features were introduced, and potential language shortcomings have been presented.

7.4.1 Language Objectives

The case study touched on all five language objectives, as presented in sections 7.2.1 to 7.2.5.

Various forms of abstraction was shown to be practical within the ALCHA language, including “ports and adaptors” and “pipes and filters” as shown by the `main` and `RadarProcessor` modules, respectively. In addition, various forms of encapsulation and information-hiding were showcased.

The control flow abstraction mechanism (i.e. the `fsm` construct) was shown to work well for some applications, but the `rtl` construct should be used instead when it makes sense to do so.

Functions are useful for information hiding and making the code more readable, but the user must be aware of the function call overhead when performing refactoring.

Fixed-point abstraction was successfully used to implement the DSP chain, and found to be useful in reducing the complexity of the implementation. The challenging details related to shifting and truncation is fully abstracted by the ALCHA language, so the user can concentrate on the algorithms instead.

The design constraints in the case study implementation are all applied at the point in the source where it makes the most sense. The pin locations and PCB-related timing constraints are applied in the `Platform` module, whereas the timing constraints related to external ICs are specified as part of the peripheral abstraction module.

Generalisation was also shown to be successful, especially for the algorithmic circuit generation employed for the `RegistersDecoder` and `MutEx` modules.

7.4.2 New Language Features

The version of ALCHA prior to the case study had no way to map bidirectional and differential pins to external HDL modules. As a consequence, the `hdl_map` pin method was introduced.

In addition, a recurring pattern was noticed where a “generate” class method needed to be called at the end of the code-base. The `finally` class method was introduced to perform this function automatically.

7.4.3 Potential Language Shortcomings

Globally-defined objects work well for project-specific modules, but was found ill-suited for general-purpose library modules. An argument was made, however, that the problem can be mitigated by means of coding style and the implementation of a framework.

The language paradigm and programming model is quite different to traditional HDL, especially from an architectural freedom point of view. A person experienced in traditional HDL would find it challenging to fully utilise the features of ALCHA and explore the resulting architectural possibilities. Similarly, traditional best-practices do not necessarily apply when using the ALCHA language. Future work would involve formulating a new set of best-practices specifically applicable to the ALCHA language.

7.4.4 Language Power

Code complexity was estimated by means of counting the lines of code in both ALCHA and Verilog implementations. It was found that importing existing HDL from other languages is expensive. The user must define the external module, and then port-map it. A future version of the ALCHA language should consider changing the mechanism so that the port mapping can be done while defining the module, instead of requiring a separate section.

Apart from importing external HDL modules, the ALCHA implementation has consistently fewer lines of code for all categories considered. The total LoC of the Verilog version is 2 480, whereas the ALCHA version is 1 759.

Next, the cost of maintenance was considered. The project was modified by adding new registers, replacing the ADF4159 synthesiser based waveform generator with an AD9915 DDS based version and adding internal signals to the debug pins. In all three cases, the ALCHA version required less modification than the Verilog version. In total, the Verilog version required 304 modifications, whereas the ALCHA version required 124.

In conclusion, the case study shows that ALCHA is a more efficient language when compared to a Verilog-based project.

Chapter 8

Conclusion

This dissertation described the development of ALCHA, a new firmware development language. Its features aim to increase developer efficiency and reduce the cost of maintenance relating to FPGA firmware. The project set out to answer the research questions listed below, which can now be answered.

- What aspects of current firmware development makes the process difficult to reason about, and therefore challenging for developers?
- What common and tedious development tasks exist, and can they be automated?
- Can low level FPGA development be made more efficient?

8.1 HDL Developer Challenges

Chapter 4 showed that the major aspect to HDL based firmware development that makes it challenging for developers is inadequate abstraction mechanisms. Existing HDLs enforce a programming paradigm that mixes high level and low level details at the same place in the project hierarchy, creating significant code clutter in higher level modules.

The L^AT_EX-like import model of ALCHA allows the code to hide a significant portion of wiring and plumbing in abstraction. When used in conjunction with well-defined interface models, the code becomes more maintainable than the traditional HDL equivalent, especially for systems with a high level of module interoperability.

ALCHA further makes use of an object-oriented model that supports multiple inheritance, which allows for even more flexibility in defining abstraction layers. The developer can create abstractions of lower level modules, as well as bus structures and complex signal types.

One of the abstraction mechanisms that ALCHA promotes is a global object model, especially for clocks, resets and external device pins. The case study suggests that this model is ill-suited for general-purpose library modules, thereby forcing a reduction in encapsulation. This limitation can be mitigated, however, by appropriate use of other language features such as object orientation and structured programming.

8.2 Automation

It was found that a large part of the verbosity present in existing HDL based code can be automated by the compiler. ALCHA hides much of the verbosity of low level register transfer logic (RTL) in abstraction, thereby increasing the readability of the code.

ALCHA supports a dedicated `fsm` construct that supports structured procedural and object-oriented programming models, which significantly reduces the verbosity of finite state machine descriptions. The resulting code is faster to write and easier to read, and therefore more maintainable.

Another task that can be automated is the handshake logic often present when splitting finite state machines over multiple submodules. A procedural programming model was used for this purpose.

The ALCHA procedural programming model makes use of modern concepts, such as automated type inference and duck-typing. The developer can, with relative ease, hide algorithm details and make the resulting code more re-usable and maintainable. Furthermore, ALCHA functions can be used to implement a fork-and-join program flow within finite state machines. These features make the ALCHA procedural model flexible, expressive and powerful.

The third feature related to automation is language support for arbitrary fixed-point arithmetic, which enables the user to develop using real-world units and representation. The ALCHA compiler performs the translation between the real-world literals and the underlying fixed-point representation, as well as between different fixed-point types.

8.3 Development Efficiency

Developer efficiency can be increased by the use of scripting, as illustrated by the TCL and Python scripts prevalent in the industry. By incorporating a scripting front-end, the firmware developer can make use of powerful algorithmic module parametrisation features, as well as dynamic module creation. Modules close to the leaves of the instantiation tree can add themselves to the global system bus, or add their control registers to the register hierarchy, for example.

Maintainability is further increased by means of attributes. Design constraints are defined in the module to which they apply, rather than the traditional project-based central location. Modules can therefore be updated in isolation without worrying about updating the design constraints of all the projects that make use of that module.

The above features related to algorithmic code generation and integrated design constraints promote code reuse, further increasing developer efficiency.

Chapter 7 evaluated the ALCHA language efficiency by considering a case study. The radar-based project was implemented in both ALCHA and Verilog. All five language objectives, as presented in section 4.2, were shown to be satisfied by the ALCHA language, after which the relative language powers were compared.

Code complexity was estimated by means of counting the lines of code, which showed that the ALCHA-based implementation is less complex than the Verilog equivalent. It was further shown that the ALCHA version is less expensive to maintain than the Verilog equivalent. For the case study, the ALCHA based implementation was found to be about half as complex and twice as powerful compared to the Verilog version.

8.4 Compiler Practicality

The practicality of the ALCHA language was confirmed in chapter 6. It was shown that it is practical to parse the ALCHA grammar and implement the automated fixed-point abstraction. The practicality of implementing other features, such as the `fsm` construct and procedural programming model, were considered from a more theoretical perspective.

Due to the flexibility of the ALCHA scripting engine, however, it was found impractical to have a one-to-one match between the ALCHA source code and the resulting HDL. The HDL output from an ALCHA compiler will in all likelihood be nothing more than a representation

of the netlist. It is envisioned that the ALCHA language will eventually be used to generate the netlist directly, instead of using Verilog or VHDL code as an intermediary step.

8.5 Novel Contributions

This dissertation made the following contributions to firmware development research within the field of electronic engineering:

- A software-like and object-oriented programming model that can be applied to FPGA firmware development, without losing low-level control over the circuit description, as presented in section 5.1. Related work was presented in section 2.5, but those typically focus on the module level, rather than the project as a whole. Scala-based solutions (see section 2.5.5) come relatively close to a project-wide solution, but only superficially improves the procedural and structural aspects.
- A significant reduction in boilerplate and complexity when implementing finite state machines, including the ability to apply structured and procedural programming principles and paradigms, as presented in section 5.2. Even though the ALCHA model is inspired by Synflow Cx (see section 2.5.2), ALCHA takes the concept beyond the limitations of the Kahn process network model employed by Cx.
- A means by which fixed-point arithmetic can be automated as an integral component of the hardware description language, as presented in section 5.3. High level synthesis tools provide automatic floating point to fixed point conversion, which aims to perform a similar function, but does not allow the low-level control as provided by ALCHA.

8.6 Practical Applications

Firmware developers targeting entry-level hardware, for example embedded systems, will find ALCHA an effective development tool, especially when leveraging the `fsm` construct and procedural programming model.

Firmware developers working on higher power hardware used by scientific and engineering communities in fields such as astronomy, particle physics and radar will benefit from the decreased development time associated with multiple levels of abstraction and ease of code-reuse that ALCHA provides.

Users of ALCHA do not have to stick to a high level of abstraction. Researchers in the field of computational hardware architectures can make use of ALCHA at a variety of abstraction levels, including combinational, RTL and architectural. The superior target independence of ALCHA should promote large-scale distributed development, such as open-source projects.

8.7 Future Work

An ALCHA prototype compiler is under development [23]. After full implementation, the HDL generation process must be tested for robustness and the generated HDL analysed for efficiency. The full compiler system remains to be tested with real-world, real-time applications.

The ALCHA grammar and syntax should be optimised and refined based on user feedback. The C-based family of languages, for example, is still being refined, even after many decades of existence and use.

There are opportunities for further research related to optimisation, such as automated multi-cycle timing constraints extracted from the `fsm` control-flow graph; and function instance reuse strategies.

The language paradigm and programming model is significantly different to traditional HDL, so a new set of best-practices specifically applicable to the ALCHA language should be developed.

A simulation engine should be developed in order to take advantage of the sequential flow of the `fsm` construct. Instead of simulating the resulting netlist, a faster simulation can be achieved by simulating the control-flow graph of the source.

A future version of ALCHA might include a high level synthesis construct as an integrated feature of the language.

Bibliography

- [1] D. J. Smith, *HDL Chip Design: A Practical Guide for Designing, Synthesizing & Simulating ASICs & FPGAs Using VHDL or Verilog*. Doone Pubns, Mar. 1998, ISBN 0965193438.
- [2] J. Kring, “A Recipe for Spaghetti Code in LabVIEW,” <https://blog.jki.net/a-recipe-for-spaghetti-code-in-labview-part-1>, Feb. 2019.
- [3] S. Bourdeauducq, “Migen Home Page,”
<https://github.com/m-labs/migen>, 2016.
- [4] N. Siret and M. Wipliez, “Cx Documentation,” <https://www.synflow.com/>, 2020.
- [5] Protop Solutions UG, “VHDPlus Home Page,” Online, available at <https://vhdplus.com/>, 2021.
- [6] J. Rajewski, “Alchitry Home Page,” Online, available at <https://alchitry.com/>, 2019.
- [7] —, *Learning FPGAs*. O’Reilly, 2017, ch. 13 Lucid Reference, pp. 183–204, ISBN: 978-1-49-196548-1.
- [8] SpinalHDL, “SpinalHDL – An alternative to standard HDL,” <https://spinalhdl.github.io/SpinalDoc-RTD>, 2020.
- [9] I. Cooper, “The Clean Architecture,” DevTernity Online Video at <https://youtu.be/SxJPQ5qXisw?t=1254>, Dec. 2019.
- [10] S. Cass, “Top programming languages 2020,” <https://spectrum.ieee.org/at-work/tech-careers/top-programming-language-2020>, Jul. 2020, iEEE.
- [11] A. Nieto, V. M. Brea, and D. L. V. no, *Image Processing, Yung-Sheng Chen (Ed.)*. InTech, 2009, ch. 27: An FPGA-based Topographic Computer for Binary Image Processing, Image Processing.

- [12] A. Vishwanath, “Enabling High-Performance Floating-Point Designs,” Intel, White Paper, 2016.
- [13] M. Adhiwiyogo, R. D’Souza, S. Leibson, and R. Shah, “Pushing AI Boundaries with Scalable Compute-Focused FPGAs,” Intel, White Paper, 2019.
- [14] Xilinx, “ZYNQ UltraScale+ RFSocCs,” <https://www.xilinx.com/products/silicon-devices/soc/rfsoc.html>, 2018.
- [15] Intel, “Intel Stratix 10 Product Page,” <https://www.intel.com/content/www/us/en/products/programmable/fpga/stratix-10.html>, 2020.
- [16] N. Diakopoulos and S. Cass, “Interactive: The top programming languages 2020, with the language types filter set to embedded,” <https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020>, Jul. 2020, iEEE.
- [17] *IEEE Standard VHDL Language Reference Manual*, IEEE, Jan. 2009.
- [18] *IEEE Standard for Verilog Register Transfer Level Synthesis*, IEEE, Dec. 2002.
- [19] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley, 2007, ISBN 9780470054376.
- [20] E. Cochlovius, U. Golza, M. Schäfers, and K.-P. Wachsmann, “Experiences with an HDL-based Design Method for Complex Architectures,” in *Sixth Annual IEEE International ASIC Conference and Exhibit*, 1993, pp. 297–300.
- [21] *IEEE Standard for Verilog*, IEEE, Mar. 2001.
- [22] *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*, IEEE, Feb. 2013.
- [23] J. Taylor, “ALCHA SourceForge Wiki,” <https://sourceforge.net/p/alcha/wiki/>, 2018.
- [24] —, “ALCHA Source on GitHub,” <https://github.com/jpt13653903/ALCHA>, 2018.
- [25] J. Taylor and S. Winberg, “ALCHA: New Object Oriented Approach to FPGA Project Development,” in *Proceedings of International Conference on Industrial Technology, IES*. Taipei, Taiwan: IEEE, Mar. 2016, pp. 707 – 712, ISBN: 978-1-4673-8074-4.
- [26] —, “ALCHA: Introducing Arbitrary Fixed-point and Procedural Programming to FPGA Firmware Design,” in *Proceedings of Workshop on Signal Processing Systems*, IEEE. Cape Town, South Africa: IEEE, Oct. 2018.

- [27] —, “ALCHA: An Alternative Programming Model for FPGA Firmware,” in *Proceedings of Radar Conference*, IEEE. Boston, MA, USA: IEEE, Apr. 2019.
- [28] E. W. Dijkstra, “The Humble Programmer,” *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, Oct. 1972.
- [29] M. Shaw, “Toward higher-level abstractions for software systems,” *Data & Knowledge Engineering*, vol. 5, no. 2, pp. 119–128, Jul. 1990.
- [30] Python Software Foundation, “Python Website,” <https://www.python.org/>, 2016.
- [31] B. H. Liskov, “Keynote address – data abstraction and hierarchy,” *ACM SIGPLAN Notices*, vol. 23, no. 5, Jan. 1987.
- [32] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Ph.D. dissertation, University of California, Irvine, 2000.
- [33] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, “Abstractions for software architecture and tools to support them,” *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314–335, Apr. 1995.
- [34] T. Colburn and G. Shute, “Abstraction in Computer Science,” *Minds and Machines*, vol. 17, pp. 169–184, Jun. 2007.
- [35] C. A. R. Hoare, E. W. Dijkstra, and O.-J. Dahl, *Structured Programming*. London and New York: Academic Press, 1972, ISBN: 0-12-200550-3.
- [36] Xilinx, “Vivado Design Suite,” <http://www.xilinx.com/products/design-tools/vivado.html>, 2016.
- [37] Altera, “Quartus II Software,” <https://www.altera.com/products/design-software/fpga-design/quartus-ii/overview.html>.
- [38] M. C. Feathers, *Working Effectively with Legacy Code*. Upper Saddle River, NJ: Prentice Hall, Sep. 2004, ISBN: 0-13-117705-2.
- [39] J. Decaluwe, “MyHDL: From Python to Silicon,” <http://www.myhdl.org/>, 2014.
- [40] Xilinx, “Vivado IP Integrator,” <https://www.xilinx.com/video/hardware/designing-with-vivado-ip-integrator.html>, 2016.
- [41] Altera, “Qsys – Altera’s System Integration Tool,” <http://www.altera.com/products/software/quartus-ii/subscription-edition/qsys/qts-qsys.html>, 2015.
- [42] Xilinx, *AXI Reference Guide*, Jun. 2015, UG1037.

- [43] Altera, *Avalon Interface Specifications*, Dec. 2015, MNL-AVABUSREF.
- [44] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, “A Survey and Evaluation of FPGA High-Level Synthesis Tools,” in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35. IEEE, Oct. 2016, pp. 1591 – 1604.
- [45] Accellera, “SystemC Standards,” <http://accellera.org/downloads/standards/systemc>, 2014.
- [46] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards, “Hardware Synthesis from a Recursive Functional Language,” in *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 2015, pp. 83 – 93.
- [47] P. Coussy and A. Morawiec, *High-Level Synthesis: from Algorithm to Digital Circuit*. Springer, 2008, ISBN 978-1-4020-8587-1, e-ISBN 978-1-4020-8588-8.
- [48] MathWorks, “HDL Coder,” <http://www.mathworks.com/products/hdl-coder/>, 2015.
- [49] Celoxica, “Handel-C Language Reference Manual,” <http://babbage.cs.qc.cuny.edu/courses/cs345/Manuals/HandelC.pdf>, 2005.
- [50] Altera, “Altera SDK for OpenCL,” <http://www.altera.com/products/software/opencl/opencl-index.html>, 2015.
- [51] P. Haglund, O. Mencer, W. Luk, and B. Tai, “PyHDL: Hardware Scripting with Python,” in *Engineering of Reconfigurable Systems and Algorithms*, 2003, pp. 288 – 291.
- [52] A. Mashtizadeh, “PHDL: A Python Hardware Design Framework,” Masters thesis, Massachusetts Institute of Technology, May 2007.
- [53] N. Gasson and N. Audsley, “Synthesis of the SR Programming Language for Complex FPGAs,” in *Proceedings of International Conference on Field Programmable Logic and Applications*. IEEE, Aug. 2009, pp. 617 – 621.
- [54] I. Shcherbakov, C. Weis, N. Wehn, and T. U. Kaiserslautern, “Bringing C++ Productivity to VHDL world: From Language Definition to a Case Study,” in *Proceedings of Forum on Specification and Design Languages*. IEEE, Sep. 2011, pp. 1 – 7.

- [55] Intel FPGA, “[Intel HLS Compiler](https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html),” <https://www.altera.com/products/design-software/high-level-design/intel-hls-compiler/overview.html>, 2018.
- [56] B. C. Schafer and Z. Wang, “[High-Level Synthesis Design Space Exploration: Past, Present, and Future](#),” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2628–2639, Sep. 2020.
- [57] C. Xiao and E. Casseau, “[Improving High-Level Synthesis Effectiveness Through Custom Operator Identification](#),” in *Proceedings of International Symposium on Circuits and Systems (ISCAS)*. IEEE, Jun. 2014, pp. 161 – 164.
- [58] P. Arató and G. Suba, “[A Data Flow Graph Generation Method Starting from C Description by Handling Loop Nest Hierarchy](#),” in *Proceedings of International Symposium on Applied Computational Intelligence and Informatics (SACI)*. IEEE, May 2014, pp. 269 – 274.
- [59] W.-J. Fang and A. C. H. Wu, “[Integrating HDL Synthesis and Partitioning for Multi-FPGA Designs](#),” in *Design & Test of Computers*, vol. 15. IEEE, Jun. 1998, pp. 65 – 72.
- [60] H. Zheng, S. T. Gurumani, L. Yang, D. Chen, and K. Rupnow, “[High-Level Synthesis with Behavioral-Level Multicycle Path Analysis](#),” in *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33. IEEE, Dec. 2014, pp. 1832 – 1845.
- [61] Q. Liu, G. A. Constantinides, K. Masselosy, and P. Y. K. Cheung, “[Compiling C-like Languages to FPGA Hardware: Some Novel Approaches Targeting Data Memory Organisation](#),” in *Proceedings of BCS International Academic Conference — Visions of Computer Science*, 2008.
- [62] L. Gallo, A. Cilardo, D. Thomas, S. Bayliss, and G. A. Constantinides, “[Area Implications of Memory Partitioning for High-Level Synthesis on FPGAs](#),” in *Proceedings of International Conference on Field Programmable Logic and Applications*. IEEE, Sep. 2014, pp. 1 – 4.
- [63] S. A. Edwards, “[The Challenges of Synthesizing Hardware from C-Like Languages](#),” in *Design & Test of Computers*, vol. 23. IEEE, May 2006, pp. 375 – 386.
- [64] R. A. Bergamaschi, “[Bridging the Domains of High-Level and Logic Synthesis](#),” in *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21. IEEE, Aug. 2002, pp. 582 – 596.

- [65] D. Ménard, N. Hervé, O. Sentieys, and H. Nguyen, “[High-Level Synthesis under Fixed-Point Accuracy Constraint](#),” *J. Electrical and Computer Engineering*, vol. 2012, pp. 906 350:1–906 350:14, 2012.
- [66] MathWorks, “[Simulink](https://uk.mathworks.com/products/simulink.html),” <https://uk.mathworks.com/products/simulink.html>, 2021.
- [67] National Instruments, “[Application Design Patterns: State Machines](#),” <https://www.ni.com/en-za/support/documentation/supplemental/16/simple-state-machine-template-documentation.html>, Dec. 2020.
- [68] ALDEC, “[FPGA Design – Graphical/Text Design Entry](#),” https://www.aldec.com/en/solutions/fpga_design/graphical_text_design_entry, Dec. 2020.
- [69] W. B. Frakes and K. Kang, “[Software Reuse Research: Status and Future](#),” in *Transactions on Software Engineering*, vol. 31. IEEE, Jul. 2005, pp. 529 – 536.
- [70] A. G. Schmidt, G. Weisz, and M. French, “[Evaluating Rapid Application Development with Python for Heterogeneous Processor-Based FPGAs](#),” in *Proceedings of the 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Napa, CA, USA: IEEE, Apr. 2017, pp. 121–124.
- [71] J. A. Bower, W. N. Cho, and W. Luk, “[Unifying FPGA Hardware Development](#),” in *Proceedings of International Conference on Field-Programmable Technology*. IEEE, Dec. 2007, pp. 113 – 120.
- [72] N. Siret and M. Wipliez, “[Tame Your FPGA](https://synflow.gitlab.io/docs/),” <https://synflow.gitlab.io/docs/>, 2020.
- [73] G. Kahn, “[The Semantics of a Simple Language for Parallel Programming](#),” *Information Processing*, vol. 74, pp. 471–475, 1974.
- [74] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “[Chisel: Constructing Hardware in a Scala Embedded Language](#),” in *Design Automation Conference*, IEEE. San Francisco, California, USA: ACM, Jun. 2012, pp. 1212–1221.
- [75] SpinalHDL, “[SpinalHDL Home Page](https://spinalhdl.github.io/SpinalDoc-RTD/),” <https://spinalhdl.github.io/SpinalDoc-RTD/>, 2016.
- [76] O. Port and Y. Etsion, “[DFiant: A Dataflow Hardware Description Language](#),” in *27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–4.
- [77] D. Braha and O. Maimon, “[The design process: properties, paradigms, and structure](#),” *IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans*, vol. 27, no. 2, pp. 146–166, Mar. 1997.

- [78] I. Lakatos, “Falsification and the Methodology of Scientific Research Programmes,” in *Can Theories be Refuted?* Springer, 1976, pp. 205–259.
- [79] K. R. Popper, *The Logic of Scientific Discovery*, English ed. Hutchinson, 1959, English ed., ISBN 0-41-527843-0.
- [80] S. Golson and L. Clark, “Language Wars in the 21st Century: Verilog versus VHDL – Revisited,” Trilobyte: Synopsys Users Group, Tech. Rep., 2016.
- [81] D. van Heesch, “Doxygen,” <https://www.doxygen.nl/index.html>, 2018.
- [82] K. Kennedy, C. Koelbel, and R. Schreiber, “Defining and Measuring the Productivity of Programming Languages,” in *International Journal of High Performance Computing Applications*, vol. 18. SAGE, Winter 2004, pp. 441 – 448.
- [83] “Git diff Documentation,” <https://git-scm.com/docs/git-diff>, 2021.
- [84] C. Sadowski and T. Zimmermann, Eds., *Rethinking Productivity in Software Engineering*. Springer, 2019, ISBN 978-1-4842-4221-6.
- [85] J. Taylor, “ALCHA Case Studies on GitHub,” <https://github.com/jpt13653903/ALCHA-Case-Studies>, 2021.
- [86] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995, ISBN: 0-20-163361-2.
- [87] D. L. Lanning and T. M. Khoshgoftaar, “Modeling the Relationship Between Source Code Complexity and Maintenance Difficulty,” in *Computer*, vol. 27. IEEE, Sep. 1994, pp. 35 – 40.
- [88] *Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model*, ISO/IEC 7498-1:1994, Jun. 1996.
- [89] J. Postel, “Transmission Control Protocol,” Internet Requests for Comments, RFC Editor, STD 7, Sep. 1981, <http://www.rfc-editor.org/rfc/rfc793.txt>. [Online]. Available: <http://www.rfc-editor.org/rfc/rfc793.txt>
- [90] J. R. Heath and S. Durbha, “Methodology for Synthesis, Testing, and Verification of Pipelined Architecture Processors from Behavioral-Level-Only HDL Code and a Case Study Example,” in *Proceedings of SoutheastCon*. IEEE, Mar. 2001, pp. 143 – 149.
- [91] S. Sutherland, “Modeling FIFO Communication Channels Using SystemVerilog Interfaces,” in *Proceedings of Synopsys Users Group*, Boston, 2004.
- [92] D. L. Parnas, “On the Criteria to Be Used in Decomposing Systems into Modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

- [93] K. Forsberg and H. Mooz, “System Engineering for Faster, Cheaper, Better,” in *INCOSE International Symposium*, vol. 9, no. 1. Wiley Online Library, 1999, pp. 924–932.
- [94] S. Hammond and D. A. Umphress, “Test driven development: the state of the practice,” in *Proceedings of the 50th Annual Southeast Regional Conference*. ACM-SE, Mar. 2012, pp. 158–163.
- [95] L. Besaw, “Berkeley UNIX System Calls and Interprocess Communication,” Berkeley, Tech. Rep., 1994.
- [96] A. Cockburn, “Hexagonal architecture,” <https://alistair.cockburn.us/hexagonal-architecture/>, 2005.
- [97] R. Woods, J. McAllister, G. Lightbody, and Y. Yi, *FPGA-based Implementation of Signal Processing Systems*. Wiley, 2008, ISBN: 978-0-470-03009-7.
- [98] P. Coussy, G. Lhairech-Lebreton, and D. Heller, “Multiple Word-Length High-Level Synthesis,” *EURASIP J. Emb. Sys.*, vol. 2008, 2008.
- [99] Altera, *Intel Quartus Prime Timing Analyzer Cookbook*, Nov. 2017, MNL-01035.
- [100] S. Hadjis, A. Canis, R. Sobue, Y. Hara-Azumi, H. Tomiyama, and J. Anderson, “Profiling-Driven Multi-Cycling in FPGA High-Level Synthesis,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Grenoble: IEEE, Mar. 2015, pp. 31 – 36, ISBN 978-3-9815-3704-8, INSPEC 15076207.
- [101] A. Feller, R. Noto, and A. M. Smith, “Standard Cell Approach for Generating Custom CMOS/SOS Devices Using a Fully Automatic Layout Program,” *IEEE Circuits and Systems Magazine*, vol. 3, pp. 9 – 13, Sep. 1981.
- [102] C. Fulks, “Using TCL Scripts to Accelerate FPGA Development,” Intuitive Research and Technology Corporation, Huntsville, Alabama, Tech. Rep., Feb. 2014.
- [103] A. Kishore and J. Tucker, “JSON5 – JSON for Humans,” Project home page, available at <https://json5.org/>, 2018.
- [104] *Committee Draft, Programming Languages – C*, ISO/IEC N1570, Apr. 2011.
- [105] *Working Draft, Standard for Programming Language C++*, ISO/IEC N3797, Oct. 2013.
- [106] Python Software Foundation, “Python Website,” <https://docs.python.org/3.8/>, 2016.
- [107] Berkeley, “Chisel/FIRRTL Documentation,” <https://www.chisel-lang.org/>, 2020.

- [108] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “[Design Patterns: Abstraction and Reuse of Object-Oriented Design](#),” in *ECOOP’93 – Object-Oriented Programming*, ser. Lecture Notes in Computer Science, N. O.M., Ed. Berlin, Heidelberg: Springer, 1993, vol. 707, pp. 406–431.
- [109] J. Stephenson, *Design Guidelines for Optimal Results in FPGAs*, Altera Corporation, 2005.
- [110] M. Karnaugh, “[The Map Method for Synthesis of Combinational Logic Circuits](#),” *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, vol. 72, no. 5, pp. 593–599, Nov. 1953.
- [111] K. Chapman, “[Get Smart About Reset: Think Local, Not Global](#),” Xilinx white paper, available at http://www.xilinx.com/support/documentation/white_papers/wp272.pdf, 2008.
- [112] Altera, “[Altera Cyclone V Handbook](#),” https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cv_5v2.pdf, 2014.
- [113] M. Hutton, “[Understanding How the New Intel HyperFlex FPGA Architecture Enables Next-Generation High-Performance Systems](#),” Intel, White Paper, 2017.
- [114] A. Davidson, “[A New FPGA Architecture and Leading-Edge FinFET Process Technology Promise to Meet Next-Generation System Requirements](#),” Intel, White Paper, 2017.
- [115] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with Verilog Design*, 3rd ed. McGraw-Hill, Feb. 2013, ISBN 978-0073380544.
- [116] A. Vallecillo and M. Gogolla, “[Typing Model Transformations Using Tracts](#),” in *ICMT – Theory and Practice of Model Transformations*, ser. Lecture Notes in Computer Science, Z. Hu and J. de Lara, Eds. Berlin, Heidelberg: Springer, 2012, vol. 7307.
- [117] R. Chugh, P. M. Rondon, and R. Jhala, “[Nested refinements: a logic for duck typing](#),” *ACM SIGPLAN Notices*, vol. 47, no. 1, Jan. 2012.
- [118] F. Kefallonitis, “[Name mangling demystified](#),” int 0x80, White Paper, Oct. 2007.
- [119] *C# 6.0 draft Language Specification*, Microsoft, Nov. 2017.
- [120] D Language Foundation, “[D Language Documentation](#),” <http://dlang.org/documentation.html>, 2016.
- [121] *Vivado Design Suite User Guide – Synthesis*, Xilinx, Apr. 2017, UG901.

- [122] W3C, “Extensible Markup Language (XML) 1.0 (Fifth Edition),” <http://www.w3.org/TR/REC-xml/>, Nov. 2008.
- [123] R. Scoville, “Tips for Incremental Compilation And LogicLock,” Cornell Engineering, Tech. Rep., Feb. 2012.
- [124] *Synopsys Design Constraints*, Synopsys, 2015, <https://www.synopsys.com/Community/Interoperability/Pages/TapinSDC.aspx>.
- [125] T. Granlund and the GMP development team, “GNU MP: The GNU Multiple Precision Arithmetic Library,” <http://gmplib.org/>, 2015, version 6.1.0.
- [126] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélicier, and P. Zimmermann, “MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, Jun. 2007.
- [127] *The JSON Data Interchange Syntax*, 2nd ed., ECMA International, Rue du Rhône, Geneva, Dec. 2017, ECMA-404.
- [128] T. Bray, “The JavaScript Object Notation (JSON) Data Interchange Format,” Internet Requests for Comments, Internet Engineering Task Force, STD 90, Dec. 2017.
- [129] R. DeLine and M. Fähndrich, “Typestates for Objects,” in *Proceedings of ECOOP – Object-Oriented Programming*, vol. 3086. Springer, 2004, pp. 465 – 490.
- [130] E. Darulova and V. Kuncak, “Towards a Compiler for Reals,” *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 2, pp. 8:1–8:28, Mar. 2017.
- [131] P. Lucas, “The Structure of Formula-Translators,” *ALGOL Bull.*, no. Sup 16, pp. 1–27, Sep. 1961.
- [132] A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation, and Compiling*. USA: Prentice-Hall, Inc., 1972, ISBN 978-0-13-914556-8.
- [133] O. Hummel and S. Burger, “A Pragmatic Means for Measuring the Complexity of Source Code Ensembles,” in *Proceedings of Emerging Trends in Software Metrics*. IEEE, May 2013, pp. 76 – 79.
- [134] J. E. Hirsch, “An Index to Quantify an Individual’s Scientific Research Output,” in *Proceedings of the National Academy of Sciences of the United States of America*, vol. 102. CrossMark, Nov. 2005, pp. 16 569 – 16 572.

-
- [135] I. Heitlager, T. Kuipers, and J. Visser, “A Practical Model for Measuring Maintainability,” in *null*. IEEE, 2007, pp. 30–39.
 - [136] MathWorks, “MATLAB,” <http://www.mathworks.com/products/matlab/>, 2015.
 - [137] D. E. Simon, *An Embedded Software Primer*. Addison-Wesley, Jul. 1999, ISBN: 0-201-61569-X.
 - [138] Intel, “NIO S II Design Suite,” <https://www.intel.com/content/www/us/en/products/programmable/processor/nios-ii/design-tools.html?wapkw=nios%20II>.
 - [139] RISC-V, “RISC-V Home Page,” <https://riscv.org/>.
 - [140] NXP Semiconductors, *I²C-Bus Specification and User Manual, Rev. 6*, Apr. 2014, UM10204.
 - [141] J. W. Cooley and J. W. Tukey, “An Algorithm for the Machine Calculation of Complex Fourier Series,” *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.
 - [142] P. Runeson, “A Survey of Unit Testing Practices,” *IEEE Software*, vol. 23, no. 4, pp. 22–29, Aug. 2006.
 - [143] S. Sutherland and T. Fitzpatrick, “UVM Rapid Adoption: A Practical Subset of UVM,” in *Proceedings of DVCon*. Accellera, Mar. 2015.

Appendix A

Evaluation Metrics

In order to objectively evaluate the ALCHA language there is a need for consistent metrics. This section explores various software-related evaluation metrics available in the literature.

A.1 Code Complexity

The first candidate considered for measuring code complexity was the hm-index as proposed by Hummel *et al.* [133]. It is based on the bibliometric h-index [134]. The index is calculated by arranging the code units (typically ‘classes’ in software) in descending order of so-called fan-out. Fan-out is defined as the number of method calls per code unit. The hm-index is then equal to the first entry in the table where the rank is at least as large as the fan-out. An hm-index of N means that: “There are at least N classes with N method calls each.”

This was found to be difficult to apply to firmware, especially HDL. The code units of HDL are entities (or modules). There is no concept of method calls – the closest analogy is sub-circuit instantiations, but that does not provide an adequate measure of complexity.

Most firmware systems will have an enormously complex top-level entity, with many sub-circuits and plenty of routing. Most entities from that point down, however, will have very few sub-circuits. The rationale for this was presented in section 4.2.1.3. The hm-index will therefore generally be on the low end: typically 3 or 4, regardless of complexity.

There are other means to map the hm-index to firmware (such as considering the number of state transitions, or number of port connections, or number of signals), but it was found that, in general, the hm-index is not a suitable measure of HDL-based firmware complexity.

A.2 Code Maintainability

The next candidate is a measure of maintainability as presented by Lanning *et al.* [87]. The paper provides a statistical model which can be used to correlate code complexity with maintenance difficulty, but this dissertation is more interested in the metric used to measure maintenance effort.

Maintenance effort is measured in terms of the number of lines of code that require modification for any given functionality update or bugfix. This is quite a useful metric, and can readily be applied to firmware in both HDL and ALCHA forms.

A related metric used for measuring code maintainability is the degree of duplication [135]. Any language property that forces the user to duplicate sections of code should be eliminated.

A.3 Language Productivity

Kennedy *et al.* provides a means by which programming language productivity can be measured. Language productivity is measured by means of two metrics: relative power and relative efficiency. Relative power is the ratio of implementation time of the same problem in two different languages. Relative efficiency is the ratio of runtime of the two implementations. [82]

There are two means by which language productivity can be increased. Either by increasing the relative power of an existing highly efficient language, or by increasing the relative efficiency of a highly powerful language. [82]

One of the key goals of the ALCHA language is to have no loss of low level control. In principle, a circuit implemented in ALCHA will execute in exactly the same number of clock cycles as the equivalent circuit in traditional HDL. The concept of language efficiency therefore falls away, because the ALCHA language is specifically designed to have equivalent efficiency.

A similar argument can be made regarding synthesised circuit area. The ALCHA language is designed so that the user has sufficient low level control to minimise area in the same way he / she would in traditional HDL. This leaves language power as the parameter of choice.

Appendix B

ALCHA Design

This appendix supplies supplementary information to chapter 5. It presents language design decisions that is important to the overall design of ALCHA, but does not contribute to satisfying the research objectives directly.

B.1 Changes to the Baseline

Verilog was chosen as the baseline language in section 3.2.1. Some basic modification was made to the Verilog baseline in order to make the language easier to use and less verbose. These modifications are well established in various forms throughout the software world, and is therefore not discussed in detail. The interested reader is referred to appendix F for more information.

B.1.1 Blocks

Verilog blocks are denoted by the `begin` and `end` keywords. This is rather verbose to type and does not enhance the visual structure of the code.

Furthermore, it is easy to add bracket-matching to an IDE or text editor, but challenging to get it to work with `begin-end` blocks. It was therefore decided to replace the block keywords with C-style braces.

B.1.2 Concatenation

Vector concatenation cannot use braces any more because those are now used for blocks of code. A bracketed notation is desirable above something like an addition operator because it neatly keeps the concatenated vector as a visual unit. The syntax decided on is: “:(A, B, C)”, where A, B and C are vectors, or arrays of vectors with equal array dimensions (i.e. ALCHA supports vectorised expressions).

This is closely related to array concatenation, which cannot use the same syntax. There is a useful difference between a vector and an array in Verilog: a vector is a collection of bits that represent a value, whereas an array is a collection of values. The syntax for array concatenation is “[A, B, C]”, where A, B and C are arrays.

B.1.3 Array Builders

Array builders are related to array concatenation, but instead of concatenating the elements of existing arrays, it builds a new array and includes the objects provided as nested elements within the new array. The syntax is “[A, B, C]”, where A, B and C can be anything.

Another form of array building is the ‘range’, which is common in Python. The ALCHA syntax is “A..B” or “A..B:C”. The semantics are the same as those used by the Matlab “[A:B]” and “[A:C:B]” constructs, except that an inverted range automatically implies a step size of -1.

B.1.4 Array Slicing

Array slicing is loosely based on the D programming language. The array index accepts any numerical array, and the result is an array that contains the indices specified in the index array. This is quite a powerful construct, allowing the user significant freedom in array slicing.

B.1.5 Vectorised Expressions

Vectorised expressions (as supported by Matlab [136]) is an immensely powerful language construct, especially for a highly parallel architecture such as an FPGA. It was therefore decided that any expression in ALCHA will support vectorised operation.

If, for example, two arrays are added together, the result will be an array of the resulting element-wise addition. Similarly, if an array is added to a scalar, that scalar will be distributed to all the elements of the array.

Vectorised expressions are extended to function calls as well. If a function expects a scalar parameter, but the user provides an array instead, the resulting operation is equivalent to the user calling the function individually for each element in the array.

B.1.6 Complex Arithmetic

The ALCHA scripting engine supports complex numbers, which is useful in calculating filter constants, for instance.

B.1.7 Stringification

In Python it is very convenient to be able to say `print(A)`, where `A` can be anything. It is a useful debugging tool. ALCHA supports a similar notion by means of the `$` operator. Anything prefixed by the `$` operator is turned into the string representation of itself. Formatting is supported by means of the syntax “`$( A, format_string )`”.

Interpolated string literals (such as f-strings in Python) are considered a future feature. For the time being, array concatenation can be used for the purpose, as in: “`:[ "x = ", $x ]`”.

B.1.8 Loops

For-loops are changed from the C-like syntax of Verilog to a more modern ‘for-each’ concept. The syntax is: “`for ( A in B ) { ... }`”, where `A` is the loop iterator variable (implicitly defined) and `B` is an array of any kind.

A similar enhancement is the `loop` keyword, which either loops forever (if no parameter is given), or loops the given number of times (if a parameter is given). Many applications require a given number of iterations without the need for an explicit iterator.

B.1.9 Comments

Verilog supports C-style comments of both types, i.e. line and block comments [104]. ALCHA extends comment support to D-style nested comments [120], as well as todo comments.

Todo comments are semantically equivalent to line comments, except that they cause the compiler to print out a ‘todo’ message (along with source code filename and line number). The syntax is: “`!! Comment Text`”, which was inspired by the preface of [137] pg. xvii.

B.1.10 Unicode Names

The ALCHA language considers any Unicode character above U+7F, apart from recognised space and newline characters, as valid identifier characters (see appendix E.1 for details). A developer can therefore use variable and class identifiers in his / her native language.

B.1.11 Extended Character Names

An ALCHA string literal supports Unicode directly (the file format is assumed to be UTF-8). In addition, the string can contain an HTML name escape sequence, e.g. “`\⟶`”, which is a long rightwards arrow, or “`→`”.

B.2 Guidelines

This section investigates design guidelines that are important to keep in mind, but not directly related to key objectives and goals as presented in section 4.1.

B.2.1 Platform Independence

Various developers make use of different platforms, such as Windows, Linux or MacOS. Additionally, the file and folder structure in use could be in any of a number of languages.

ALCHA source-code should be able to seamlessly switch between platforms. This is especially important when considering an international collaboration.

The easiest solution is to use Linux as an example. All file names and source files are encoded in UTF-8, and all paths are Unix-style (i.e. uses a forward slash to separate folder levels within a path).

Absolute paths are generally frowned-upon, but should nevertheless be supported by the language – especially paths specified on the compiler command-line, such as library paths.

The convention on the ALCHA command-line is that the command-line parser will quietly replace all Windows-style drive letters so that `C:\MyFolder\MyFile.alc` becomes `/c/MyFolder/MyFile.alc`. Internals to the ALCHA compiler (including all source files) only uses the latter format.

Line-endings are another issue that requires consideration. The three major operating systems all use different line-endings. The ALCHA scanner is line-ending agnostic and will accept all three forms as equivalent, in addition to other forms presented in [appendix E.1](#).

B.2.2 Target Independence

In order to be useful, library components must be as vendor- and device-independent as possible. Even the same device on a different PCB can sometimes prove problematic.

The ALCHA language should therefore be designed so that the user can make use of a new device with minimal changes to the source code, and no changes to the core language.

The easiest means by which this can be achieved is to allow the user to write his / her own abstractions and not provide too much fancy device integration as part of the core language. Class-based object orientation is one of the easiest solutions.

There can, of course, be a language-provided standard library in order to support abstraction of the most common use-cases, but even this should not form part of the core language.

B.2.3 Expandability

ALCHA is a young language, and still under development. It should therefore be relatively easy to add additional features not yet considered.

Let us assume, for example, that the ALCHA language automates state machine encoding by always using one-hot [\[1\]](#). Some time in the future, a user makes a feature-request that it should be possible to change the default encoding to, say, Johnson [\[1\]](#).

The language should be designed such that this unforeseen feature is easy to implement. In other words, it should leave room for adding features that change compiler behaviour without influencing the core language syntax.

The mechanism employed for compiler behaviour (such as the state machine encoding example above) is global attributes. Other mechanism are also possible, such as new keywords.

B.3 Miscellaneous Features

This section presents various features of ALCHA that does not fit into any specific category.

B.3.1 Groups and Classes

ALCHA supports object orientation by allowing the user to define a class. That defined class can then later be used to create various instances of that class.

Groups, by contrast, are simply a collection of statements. They provide a convenient means by which the user can apply the same set of attributes to a large number of declarations, or put a section into a different namespace.

Class declarations have to be named, whereas groups can be either named or anonymous. The user can create various instances of classes, whereas a group is it's own instance. A group can be seen as a local namespace.

Groups and classes can be nested within each other, to an arbitrary level. A class defined within a group is declared within the group namespace, whereas a group defined within a class creates a local namespace within each instance of that class.

B.3.2 Import Search Paths

Whenever the user references a file, typically by means of the `import` statement, the compiler must search for that file. The paths are searched in the following order:

1. The same path as the file that contains the `import` statement.
2. The same path as the 'main' source file (i.e. the one used on the command-line of the compiler).
3. The 'include paths' specified on the command-line of the compiler, which function the same as the GCC C compiler.

This searching algorithm ensures that components first search their local folder, then the project folder, and then the library folders.

B.3.3 Existing Code Integration

ALCHA should integrate well with existing HDL code. There are multiple use-cases where this is important, as summarised below:

- Legacy firmware modules.
- Device-specific resources, such as PLL, DSP or RAM blocks¹.
- Systems generated by vendor tools such as Quartus Platform Designer or Vivado IP Integrator.
- Modules generated by other third-party EDA or HLS tools.

ALCHA achieves this by means of the `hdl` construct, which is presented in more detail in section [F.15.1](#).

¹This category also include hard IP such as PCIe controllers, DDR4-based external DRAM controllers, high speed serial transceivers, etc.

Appendix C

SPI Example

```
module ADXL345 #(
    parameter Clock_Frequency = 50e6,
    parameter BAUD             = 5e6
)(
    input Clk, Reset,

    output reg [15:0]X, // 2's Complement
    output reg [15:0]Y,
    output reg [15:0]Z,

    // Physical device interface
    output reg nCS, SClk, SDI,
    input      SDO
);
//-----
```

Listing C.1. SPI interface example in Verilog (part 1)

```
localparam Clock_Div    = $floor(Clock_Frequency/BAUD/2.0 + 0.5);
localparam Clock_Div_N = $clog2(Clock_Div);

reg [(Clock_Div_N-1):0] Clock_Count;
wire Clock_Enable = ~|Clock_Count;

always @(posedge Clk) begin
    if(Clock_Enable) Clock_Count <= Clock_Div - 1'b1;
    else              Clock_Count <= Clock_Count - 1'b1;
end
//-----
```

Listing C.2. SPI interface example in Verilog (part 2)

```
reg Local_Reset;

reg [ 4:0]BitCount;
reg [15:0]Data; // R/W, MB, Address, Byte

typedef enum{
    Setup,
    ReadX, ReadY, ReadZ,
    Transaction
} STATE;

STATE State;
STATE RetState; // Used for function calls
```

Listing C.3. SPI interface example in Verilog (part 3)

```
always @(posedge Clk) begin
    Local_Reset <= Reset;
    //-----

    if(Local_Reset) begin
        nCS    <= 1'b1;
        SClk   <= 1'b1;
        SDI    <= 1'bX;
        State  <= Setup;

        BitCount <= 5'hX;
        Data     <= 16'hX;
        X        <= 0;
        Y        <= 0;
        Z        <= 0;
    end
    //-----
```

Listing C.4. SPI interface example in Verilog (part 4)

```

end else if(Clock_Enable) begin
  case(State)
  Setup: begin
    // SPI 4-wire; Full-res; Right-justify; 4g Range
    Data    <= {2'b00, 6'h31, 8'b0000_1001};
    BitCount <= 5'd16;
    State    <= Transaction;
    RetState <= ReadX;
  end
  //-----

  ReadX: begin
    Z    <= {Data[7:0], Data[15:8]};
    Data <= {2'b11, 6'h32, 8'd0};
    BitCount <= 5'd24;
    State <= Transaction;
    RetState <= ReadY;
  end
  //-----

  ReadY: begin
    X    <= {Data[7:0], Data[15:8]};
    Data <= {2'b11, 6'h34, 8'd0};
    BitCount <= 5'd24;
    State <= Transaction;
    RetState <= ReadZ;
  end
  //-----

  ReadZ: begin
    Y    <= {Data[7:0], Data[15:8]};
    Data <= {2'b11, 6'h36, 8'd0};
    BitCount <= 5'd24;
    State <= Transaction;
    RetState <= ReadX;
  end
  //-----

```

Listing C.5. SPI interface example in Verilog (part 5)

```

Transaction: begin
  if(nCS) begin
    nCS <= 1'b0;

    end else if(SClk) begin
      if(BitCount == 0) begin
        nCS <= 1'b1;
        State <= RetState;
      end else begin
        SClk <= 1'b0;
      end
      {SDI, Data} <= {Data, SD0};
      BitCount <= BitCount - 1'b1;

    end else begin
      SClk <= 1'b1;
    end
  end
endcase
end
endmodule
//-----

```

Listing C.6. SPI interface example in Verilog (part 6)

Appendix D

Case Study Implementation in ALCHA

This appendix supplies supplementary information to section 7.2. It presents the ALCHA implementation of the case study. The interested reader can also follow the full source code listing in the [Alcha folder](#) in the GitHub repository [85].

The exact details of the radar being implemented are beyond the scope of this case study. This said, it is still important to have a high-level understanding of what is being implemented. The most important radar parameters are presented in table D.1. The parameters have been calculated by neglecting many practical considerations, and are therefore approximate. They are nonetheless valid for the purposes of this case study.

D.1 Main Module

The ALCHA main module is intended to represent the overview of the system. As such, it follows figure 7.1 as closely as possible, as presented in listing D.1.

D.2 Platform Description

This case study assumes that development is done on a development kit, which will later be ported to a custom-developed hardware platform. The development kit is useful in various different projects, and must therefore be defined generically and can be part of a library.

TABLE D.1
RADAR PARAMETERS

Parameter	Value
ADC sampling rate	2.5 MSps
Instrumented Range	1.25 km
Range resolution	1 m
FMCW sweep bandwidth	150 MHz
Sweep time (active)	1 ms
Sweep time (range-fill)	8.34 μ s
Down-sweep	90 μ s
PRI	1.1 ms
Active samples per sweep	2 500 samples
Range FFT size (input)	4 096 samples
Number of range bins	2 048 bins
Doppler FFT size	128 bins
Range bin size	610 mm

```

import "Platform/Platform";
import "SubSystems/SystemController";
import "Library/Interfaces/RegistersDecoder";

RegistersDecoder(SystemController.LightWeightBus) Registers;
//-----

import "Library/Misc/FirmwareVersion";
FirmwareVersion(1,0) Firmware;

Registers.ReadOnly(Firmware.Version);
Registers.ReadOnly(Firmware.Date );
Registers.ReadOnly(Firmware.Time );
Registers.ReadOnly(Firmware.GitHash);
//-----

import "SubSystems/MasterTrigger";
import "SubSystems/HardwareControl" as Hardware;
import "SubSystems/WaveformGenerator" as Waveform;
import "SubSystems/ReceiverAbstraction" as Receiver;
import "SubSystems/RadarProcessor" as Processor;
//-----

Registers.GenerateRegs ();
Registers.GenerateCpp ("../Software/Registers/Registers");
Registers.GenerateLaTeX("../Documentation/Registers");

```

Listing D.1. The complete main.alc, showing the top-level code

D.2.1 Pin Mapping and Naming

The final hardware platform is project-specific, and will therefore have project-specific pin and net names on the PCB. The strategy followed here is to import the development kit pin descriptions from a library, and then use aliases to map the project-specific names to the general-purpose pins of the development kit. An example is presented in listings D.2 and D.3.

```
group Arduino{
  input pin    <location = "AH7"> nReset;
  pin(16) <location = [ "AG11", "AH9", "AH12", "AH11",
                        "AG16", "AF15", "AE15", "AF17",
                        "AH8", "AG8", "U13", "U14",
                        "AG9", "AG10", "AF13", "AG13" ]> IO;
}
```

Listing D.2. Excerpt from DE0-Nano-SoC.alc showing the Arduino pin mapping

```
import "Library/Platforms/DE0-Nano-SoC" as DE0;
//-----

group Synth{ // ADF4159
  group SPI{
    alias SClk  = DE0.Arduino.IO[0];
    alias Data  = DE0.Arduino.IO[1];
    alias Latch = DE0.Arduino.IO[2];
  }
  alias Trigger = DE0.Arduino.IO[3];
  alias MuxOut  = DE0.Arduino.IO[4];
}

group I2C{ // LTC2991
  alias SClk = DE0.Arduino.IO[15];
  alias Data = DE0.Arduino.IO[14];

  SClk'pullup = true;
  SClk.driver = 0;
  SClk.enable = 0;

  Data'pullup = true;
  Data.driver = 0;
  Data.enable = 0;
}
```

Listing D.3. Excerpt from Platform.alc showing some project-specific pin mapping

When the time comes to port the development platform to the final product, the firmware developer can simply define the pins in `Platform.alc` directly (using the same names), and recompile.

D.2.2 Unused Pins

The design does not make use of all the pins defined by the DE0-Nano-SoC platform descriptor. This is not a problem, because the ALCHA compiler will simply trim those nodes from the

netlist and not use them at all. The Quartus compiler will then assign weak pull-up to all unused pins, which is typically a safe state.

D.2.3 Clocking and Reset Sequencing

The platform descriptor is also responsible with generating the system clock and reset signals, which is presented in listing D.4.

```
import "Library/Clocking_and_Reset/PLL_CycloneV";

PLL_CycloneV(DE0.Clock1, ResetKey, [120, 2.5]) PLL;
alias DspClock      = PLL.Output[0];
alias ControlClock = PLL.Output[1];
//-----

import "Library/Clocking_and_Reset/DelayedReset";

net HPS_Reset      = DelayedReset(ControlClock, ResetKey | ~PLL.Locked, 1);
net HPS_FPGA_Reset = 0; // Assigned by SystemController
net MasterReset    = DelayedReset(ControlClock,
                                   HPS_FPGA_Reset | ResetKey | ~PLL.Locked, 100);
```

Listing D.4. Clock and reset generation

The `PLL_CycloneV` module generates a PLL instance by means of the Quartus-provided `altera_pll` megafunction. The result is a DSP clock of 120 MHz and a control clock of 2.5 MHz. The clock objects are declared as aliases in order to preserve the clock-related attributes of the PLL members.

The `DelayedReset` module is used to generate a longer pulse than the input reset signals provide, specified in milliseconds.

D.3 System Controller

The system controller chosen for this case study is the on-chip ARM processor of the Cyclone V SoC IC. This said, the ALCHA implementation should not be written in a way that limits the design to this type of controller. It should be relatively easy for future incarnations of the project to reduce component cost and system power usage by, for instance, utilising a much less powerful soft microcontroller, such as a NIOS II or RISC-V [138, 139], or even a finite state machine controlled by means of UDP messages over an Ethernet interface.

The means by which to achieve this independence is to make sure that the interfaces are generic. The ARM processor was instantiated by means of Platform Designer¹, which makes

¹A tool within the Quartus design suite, formally known as QSys.

it easy to expose control interfaces to the FPGA side by means of the Avalon standard [43]. The system is presented in figure D.1.

Connections	Name	Description	Export	Base	End	IRQ
	HPS	Arria V/Cyclone V Hard Processor System				
	f2h_cold_reset_req	Reset Input	hps_cold_reset_req			
	f2h_debug_reset_req	Reset Input	hps_debug_reset_req			
	f2h_warm_reset_req	Reset Input	hps_warm_reset_req			
	h2f_user2_clock	Clock Output	<i>Double-click to export</i>			
	memory	Conduit	memory			
	hps_io	Conduit	hps_io			
	h2f_reset	Reset Output	<i>Double-click to export</i>			
	f2h_sdram0_clock	Clock Input	<i>Double-click to export</i>			
	f2h_sdram0_data	Avalon Memory Mapped Slave	sdram			
	h2f_axi_clock	Clock Input	<i>Double-click to export</i>			
	h2f_axi_master	AXI Master	<i>Double-click to export</i>			
	h2f_lw_axi_clock	Clock Input	<i>Double-click to export</i>			
	h2f_lw_axi_master	AXI Master	<i>Double-click to export</i>			
	f2h_irq0	Interrupt Receiver	<i>Double-click to export</i>		IRQ 0	IRQ 31
	f2h_irq1	Interrupt Receiver	<i>Double-click to export</i>		IRQ 0	IRQ 31
	HPS_Clk	Clock Bridge				
	in_clk	Clock Input	<i>Double-click to export</i>			
	out_clk	Clock Output	hps_clk_out			
	HPS_Reset	Reset Bridge				
	clk	Clock Input	<i>Double-click to export</i>			
	in_reset	Reset Input	<i>Double-click to export</i>			
	out_reset	Reset Output	hps_reset_out			
	JTAG_Master	JTAG to Avalon Master Bridge				
	clk	Clock Input	<i>Double-click to export</i>			
	clk_reset	Reset Input	<i>Double-click to export</i>			
	master	Avalon Memory Mapped Master	<i>Double-click to export</i>			
	master_reset	Reset Output	<i>Double-click to export</i>			
	JTAG_UART	JTAG UART Intel FPGA IP				
	clk	Clock Input	<i>Double-click to export</i>			
	reset	Reset Input	<i>Double-click to export</i>			
	avalon_jtag_slave	Avalon Memory Mapped Slave	<i>Double-click to export</i>	0x0004_0000	0x0004_0007	
	irq	Interrupt Sender	<i>Double-click to export</i>			
	LightWeightBus	Avalon-MM Clock Crossing Bridge				
	m0_clk	Clock Input	light_weight_bus_clk			
	m0_reset	Reset Input	light_weight_bus_reset			
	s0_clk	Clock Input	<i>Double-click to export</i>			
	s0_reset	Reset Input	<i>Double-click to export</i>			
	s0	Avalon Memory Mapped Slave	<i>Double-click to export</i>	0x0000_0000	0x0003_ffff	
	m0	Avalon Memory Mapped Master	light_weight_bus			
	StreamBuffer	On-Chip Memory (RAM or ROM) Intel FPGA IP				
	clk1	Clock Input	<i>Double-click to export</i>			
	s1	Avalon Memory Mapped Slave	<i>Double-click to export</i>	0x0000_0000	0x0000_3fff	
	reset1	Reset Input	<i>Double-click to export</i>			
	s2	Avalon Memory Mapped Slave	streambuffer			
	clk2	Clock Input	streambuffer_clk			
	reset2	Reset Input	streambuffer_reset			

Fig. D.1. Platform Designer view of the System Controller subsystem

D.3.1 Importing the SoC System HDL

Platform Designer produces an auto-generated Verilog (or VHDL) module that can be instantiated in a higher level of the design. Within the case study implementation, the HDL module is pulled into an ALCHA module by means of the `hdl` construct, which in turn is abstracted by means of a class. The basic form of the `SystemController` module is shown in listing D.5.

```

import "Library/Interfaces/AvalonMaster"
import "Library/Interfaces/AvalonSlave"
//-----

class SystemController_Class{
  private{ // SoC System Instance
    hdl("../Verilog/SoC_System/SoC_System.qsys") SoC_System_HDL(){
      !! SoC System Port Declarations
    }
    SoC_System_HDL() SoC_System;
  }
  //-----

  private{ // HPS Pins
    !! HPS Pin Mapping
  }
  //-----

  public{ // Public Interfaces
    SoC_System.hps_cold_reset_req_reset_n = !HPS_Reset;
    SoC_System.hps_warm_reset_req_reset_n = !HPS_Reset;
    SoC_System.hps_debug_reset_req_reset_n = !HPS_Reset;

    HPS_FPGA_Reset = SoC_System.hps_reset_out_reset;
    //-----

    !! Light Weight Bus;
    //-----

    !! SDRAM;
    //-----

    !! StreamBuffer;
  }
}
SystemController_Class() SystemController;

```

Listing D.5. Basic form of the System Controller module

For the purposes of this case study, it is assumed that all Avalon infrastructure (such as multiple masters connecting to the same SDRAM slave, clock crossing, etc.) is handled by Platform Designer, even though the same can be done by means of an ALCHA class.

D.3.2 HDL Source Files

The Quartus design suite provides a convenient mechanism that encapsulates the full collection of source files required by the QSys system: the user can include the entire system by specifying only the `Soc_System.qsys` file.

An earlier iteration of the ALCHA language design specified that the compiler should be able to parse Verilog or VHDL files in order to automatically generate the port definitions from the HDL source. This proved impractical, as shown with this example. There is no HDL file specified directly, and therefore no HDL file for the ALCHA compiler to parse. It is more generic to let the user specify the ports.

D.3.3 Bidirectional Ports

ALCHA only supports two-valued logic, which poses a problem when connecting bidirectional HDL ports. This was a major limitation of the language at the time of implementing the case study project, so the `hdl_map` method was introduced to `pin` objects. This special pin object method only accepts arguments that evaluate to `hdl` ports.

Furthermore, this special `hdl_map` method takes an optional second argument used when mapping individual positive and negative components to a differential pin, as presented in the SDRAM mapping example in listing D.6.

```

HPS_Pins.DDR3.nReset      .hdl_map(SoC_System.memory_mem_reset_n);
HPS_Pins.DDR3.Clock      .hdl_map(SoC_System.memory_mem_ck,
                                SoC_System.memory_mem_ck_n );
HPS_Pins.DDR3.ClockEnable .hdl_map(SoC_System.memory_mem_cke );
HPS_Pins.DDR3.nChipSelect .hdl_map(SoC_System.memory_mem_cs_n );
HPS_Pins.DDR3.Bank        .hdl_map(SoC_System.memory_mem_ba );
HPS_Pins.DDR3.Address     .hdl_map(SoC_System.memory_mem_a );
HPS_Pins.DDR3.DataMask    .hdl_map(SoC_System.memory_mem_dm );
HPS_Pins.DDR3.nRowAddressStrobe .hdl_map(SoC_System.memory_mem_ras_n );
HPS_Pins.DDR3.nColumnAddressStrobe .hdl_map(SoC_System.memory_mem_cas_n );
HPS_Pins.DDR3.nWriteEnable .hdl_map(SoC_System.memory_mem_we_n );
HPS_Pins.DDR3.Data        .hdl_map(SoC_System.memory_mem_dq );
HPS_Pins.DDR3.DataStrobe  .hdl_map(SoC_System.memory_mem_dqs,
                                SoC_System.memory_mem_dqs_n );
HPS_Pins.DDR3.OnDieTermination .hdl_map(SoC_System.memory_mem_odt );
HPS_Pins.DDR3.RZQ         .hdl_map(SoC_System.memory_oct_rzqin );

```

Listing D.6. Example showing how the `hdl_map` function is used

D.3.4 Avalon Master Interface

In a typical design, the Avalon infrastructure would be implemented within the Platform Designer environment. For the purposes of testing the ALCHA language, however, it was decided to implement a simple Avalon master class that can be used to attach various slaves on-demand. Listing D.7 presents the Avalon base class, used to encapsulate the various signals, and listing D.8 presents the Avalon master derived class.

```
class AvalonInterface(num Width, num Depth, num MaxBurstCount = 1){
    num AddressWidth    = ceil(log2(Depth));
    num NumBytes        = ceil(Width/8);
    num BurstCountWidth = ceil(log2(MaxBurstCount + 1));

    net(AddressWidth)    Address;
    net(NumBytes)        ByteEnable = 2**NumBytes - 1;
    net(BurstCountWidth) BurstCount = 1;
    net                  WaitRequest = 1;

    net(Width)           WriteData;
    net                  Write = 0;

    net                  Read = 0;
    net(Width)           ReadData;
    net                  ReadValid = 0;
}
```

Listing D.7. Implementation of the `AvalonInterface` base class

The Avalon interface class presented in listing D.7 shows one advantage over traditional HDL: the fact that the class members are agnostic to direction. This class can be used anywhere there is an Avalon interface, and the various nets take the direction as appropriate to the situation.

This implementation is suboptimal for purposes of brevity. It is used to illustrate one important feature of ALCHA: any slave module that makes use of an Avalon interface can attach itself to this master at will, without modifying the `SystemController` module. The slave module simply calls the `Attach` method, with a reference to itself as argument. The `BaseAddress` attribute can then be used to auto-generate C++ (or other) software code that requires knowledge of the absolute byte address.

With these Avalon interfaces in place, the various Avalon interfaces exposed by the SoC System can now be abstracted, as presented in listings D.9 to D.11.

The `LightWeightBus` is exposed as a clock-crossed 16-bit address subset of the full lightweight bridge [85]. The `BaseAddress` attribute is updated with the absolute offset within the address space of the ARM processor, and the interface is put within the `ControllerClock` clock domain.

```

import "Library/Interfaces/AvalonInterface"
//-----

class AvalonMaster(net Clock, net Reset, num Width, num Depth):
    AvalonInterface(Width, Depth){
        'BaseAddress = 0;

    private
        num Used                = 0;
        net ReadingWaitRequest  = 0;
        net InterfacesWaitRequest = 0;
    }
    //-----

    ReadData      = 0;
    WaitRequest = ReadingWaitRequest | InterfacesWaitRequest;
    //-----

    void Attach(AvalonInterface Interface){
        assert(Interface.Width == Width);
        assert(Interface.Depth < Depth - Used);

        Interface.Address      = Address - Used;
        Interface.ByteEnable    = ByteEnable;
        InterfacesWaitRequest |= Interface.WaitRequest;

        net AddressValid = (Address >= Used)
                        & (Address < (Used + Interface.Depth));

        Interface.WriteData = WriteData;
        Interface.Write      = AddressValid & Write & !WaitRequest;

        Interface.Read = AddressValid & Read & !WaitRequest;
        ReadData      |= Interface.ReadValid ? Interface.ReadData : 0;
        ReadValid      |= Interface.ReadValid;

        Interface.'BaseAddress = 'BaseAddress + Used*(Width/8);
        Used += Interface.Depth;
    }
    //-----

    rtl(Clock, Reset){
        if (Read & !WaitRequest) ReadingWaitRequest = 1;
        else if(ReadValid)        ReadingWaitRequest = 0;
    }
}

```

Listing D.8. Implementation of the AvalonMaster class

```

AvalonMaster(ControllerClock, MasterReset, 32, 0x10000) LightweightBus;
LightWeightBus.'BaseAddress = 0xC000_0000;

SoC_System.light_weight_bus_clk_clk      = ControllerClock;
SoC_System.light_weight_bus_reset_reset  = MasterReset;

!! Assign SoC_System members to LightweightBus members, and vice versa

```

Listing D.9. General form of encapsulating the light weight bus

```
AvalonSlave(SoC_System.hps_clk_out_clk,
            SoC_System.hps_reset_out_reset,
            256, 0x_200_0000, 128) SDRAM;

SDRAM.Clock'frequency = 100e6;

!! Assign SoC_System members to SDRAM members, and vice versa
```

Listing D.10. General form of encapsulating the SDRAM interface

The **SDRAM** interface is a direct interface to the ARM's external DDR memory. It supports a maximum burst size of 128 (or 4 kiB, which is one page). The idea is to force the operating system kernel to reserve a section of memory for use by the FPGA.

A caching system is required to clock-cross between the DSP and SDRAM clock domains, as well as hide the latency involved in accessing the external memory, but this is implemented elsewhere. The SDRAM interface is clocked at 100 MHz, which is sourced from the SDRAM PLL within the ARM processor, which maximises the RAM bandwidth.

```
AvalonInterface(16, 1p13) StreamBuffer;
StreamBuffer'BaseAddress = 0xFF20_0000;

SoC_System.streambuffer_clk_clk      = DspClock;
SoC_System.streambuffer_clken       = 1;
SoC_System.streambuffer_reset_reset = MasterReset;
SoC_System.streambuffer_chipselect  = 1;

SoC_System.streambuffer_address      = StreamBuffer.Address;
SoC_System.streambuffer_byteenable   = 0b11;

SoC_System.streambuffer_writedata    = StreamBuffer.WriteData;
SoC_System.streambuffer_write        = StreamBuffer.Write;

StreamBuffer.ReadData = SoC_System.streambuffer_readdata;
StreamBuffer.ReadValid = 1;
```

Listing D.11. General form of encapsulating the stream buffer interface

The **StreamBuffer** is one port of an internal memory interface. The ARM connects to the other port, using a 64-bit interface, by means of the high-performance bridge. The FPGA side is 16-bit wide so that it provides a natural means by which to stream the 14-bit ADC samples. This is mainly used for debug purposes: the output of the full FPGA-side DSP chain is stored in external memory, which is directly accessible by the ARM processor.

D.4 Registers Decoder

The central registers module is implemented in such way that each module can add its own registers to the bus. This is not necessarily the best architecture to use in a real system, but does illustrate a useful feature of the ALCHA language. The source is presented in listings D.12 and D.13.

```
class RegistersDecoder(Bus): AvalonInterface(32, 0x1000){
    Bus.Attach(this);
    //-----

    private{
        'RdRegisters = [];
        'WrRegisters = [];
        'LiveRegisters = [];
        num Count = 0;
    }
    //-----

    public{
        // Register direction as seen by the controller
        void ReadOnly(Register){
            Register'Address = Count++;
            'RdRegisters.append(Register);
        }
        void Writeable(Register){
            Register'Address = Count++;
            'WrRegisters.append(Register);
        }
        void Live(RdRegister, WrRegister, WrStrobe){
            RdRegister'Address = Count;
            WrRegister'Address = Count;
            WrStrobe 'Address = Count;
            'LiveRegisters.append(< Read  = RdRegister,
                                   Write = WrRegister,
                                   Strobe = WrStrobe >);

            Count++;
        }
    }
}
```

Listing D.12. Implementation of the `RegistersDecoder` class (part 1)

The strategy is to keep arrays of objects within local attributes of the class instance. The other modules can then call the various functions, depending on what type of register is being added². This strategy is dependent on the fact that the ALCHA language stores references to objects when assigning them to attributes.

The `GenerateRegs` function should be called at the end of the main module, after all the other modules have added their registers to the `Registers` module. All the `for` loops are scripting loops³ and are evaluated during compile-time, thereby generating the cases of the

²A 'live' register is something that the FPGA-side is in control of, but can be overwritten by the system controller. One example is a real-time clock, which the controller can set, but the FPGA advances at some appropriate rate.

³A scripting loop is any loop where the iterator is of non-synthesisable type.

```

net(32) Resize(x){
    result := x; // Raw-assign the bits
    num N = x'width;
    if((x'fullscale < 0) & (N < 31)) // Sign-extend
        result(31..(N+1)) := (x(N))^(31-N);
}

void GenerateRegs(){
    rtl(Bus.Clock, Bus.Reset){
        WaitRequest = 0;

        switch(Address){
            for(Register in 'RdRegisters){
                case(Register'Address) ReadData = Resize(Register);
            }
            for(Register in 'WrRegisters){
                case(Register'Address) ReadData = Resize(Register);
            }
            for(Register in 'LiveRegisters){
                case(Register'Read'Address) ReadData = Resize(Register'Read);
            }
        }
        ReadValid = Read;

        if(Write){
            switch(Address){
                for(Register in 'WrRegisters){
                    case(Register'Address) Register := WriteData;
                }
                for(Register in 'LiveRegisters){
                    case(Register'Write'Address) Register'Write := WriteData;
                }
            }
            for(Register in 'LiveRegisters){
                Register'Strobe = ((Address == Register'Strobe'Address) & Write);
            }
        }
    }
}

//-----

void GenerateCpp(string Filename){
    !! Generate the C++ source and header files
    !! Base the structure on the Register'identifier attribute, which is the
    !! full name of the object (including the full name-space path)
}

//-----

void GenerateLaTeX(string Filename){
    !! Generate the LaTeX documentation files
}
} } }

```

Listing D.13. Implementation of the RegistersDecoder class (part 2)

switch statement. All registers are assigned raw, i.e. ignoring the fixed-point format, and sign-extension and / or truncation is performed automatically.

The `GenerateC++` and `GenerateLaTeX` functions can similarly be used to generate the C++ headers and $\text{\LaTeX}$ documentation, based on the attributes of the registers. The details of this are outside the scope of this case study.

D.5 Master Trigger

The master trigger module is relatively straightforward. Listing D.14 presents the implementation. Once again the appropriate control register is added to the bus at the place where it is used, ensuring that the resulting code is easy to maintain.

```
import "Library/Timing/TriggerGen"
//-----

TriggerGen(ControlClock, MasterTrigger) MasterTriggerGen;
Registers.Writable(MasterTriggerGen.Period);
alias MasterTrigger = MasterTriggerGen.Output;
```

Listing D.14. Implementation of the Master Trigger module

The `TriggerGen` module is equally straightforward, as presented in listing D.15.

```
class TriggerGen(net Clock, net Reset){
    net(32) Period = 1e-3 * Clock'frequency; // Default to 1 ms

    net    Output = 0;
    net(32) Count  = 0;

    rtl(Clock, Reset){
        if(Count >= Period){
            Count  = 1;
            Output = 1;
        }else{
            Count++;
            Output = 0;
        }
    }
}
```

Listing D.15. Implementation of the TriggerGen module

D.6 Hardware Control

The hardware control module is used to encapsulate all the peripheral abstractions not directly related to the radar data. Examples include the built-in measurement (BIM) involved in system health, as well as various RF switches. Listing D.16 shows the module source.

```

import "Library/Misc/Mutex"
import "Library/Timing/TriggerDelay"
import "Library/Peripherals/ADC_and_BIM/LTC2991"
//-----

Mutex(ControlClock, MasterReset) I2C_Mutex;
//-----

LTC2991(ControlClock, MasterReset, 0b000) TxBIM;

TxBIM.Quiet = !TxEnable;
TxBIM.MakeRegisters(Registers);
TxBIM.ConnectBus(I2C.SClk, I2C.Data);

I2C_Mutex.Add(TxBIM);
//-----

LTC2991(ControlClock, MasterReset, 0b001) RxBIM;

RxBIM.Quiet = !TxEnable;
RxBIM.MakeRegisters(Registers);
RxBIM.ConnectBus(I2C.SClk, I2C.Data);

I2C_Mutex.Add(RxBIM);
//-----

TriggerDelay(ControlClock, MasterReset, MasterTrigger) TriggerDelay_PA;

Registers.Writeable(TriggerDelay_PA.Enable);
Registers.Writeable(TriggerDelay_PA.Delay );
Registers.Writeable(TriggerDelay_PA.Length);

TxEnable = TriggerDelay_PA.Output;
//-----

I2C_Mutex.Generate();

```

Listing D.16. Source listing of the `HardwareControl` module

D.6.1 Inheritance

The LTC2991 module is implemented as a derived class of the I2C base, as shown in listings D.17 and D.18.

```
class I2C(net Clk, net Reset, num Baud_kHz = 50){
    input net Quiet = 0;
    output net Error = 0;
    //-----

private{
    num Baud_Cycles = round(Clk'frequency / (Baud_kHz*3));
    num Baud_N      = ceil(log2(Baud_Cycles));
    net(Baud_N) Baud_Count;

    rtl(Clk){
        if(Baud_Count) Baud_Count--;
        else          Baud_Count = Baud_Cycles-1;
    }
    net Baud_Trigger = ~|Baud_Count;

    net ClkIn , ClkOut = 1;
    net DataIn, DataOut = 1;
}
//-----

void ConnectBus(pin Clock, pin Data){
    Clock.enable |= ~ClkOut;
    Data .enable |= ~DataOut;
    rtl(Clk){
        ClkIn = Clock.pad;
        DataIn = Data .pad;
    }
}
//-----

net(8) Transaction(
    net(8) TxData,
    bool Start = false,
    bool R_nW = false,
    bool Ack = true,
    bool Stop = false
){
    net(8) Data;

    // If ClkIn != ClkOut it means that the slave is clock-stretching
    fsm(Clk, Reset, Baud_Trigger & (ClkIn == ClkOut) & !Quiet){
        !! The I2C state machine based on the command in the function parameters
    }
}
}
```

Listing D.17. Source summary of the I2C module

The I²C lines can be made quiet while the radar is sensitive to interference by raising the Quiet line. The registers required to control the module is effected by using the MakeRegisters function, which makes it relatively easy to add and remove the module from the system in one central place.

```

import "Library/Comms/I2C/I2C"
//-----

class LTC2991(net Clk, net Reset, net(3) Address, num Baud_kHz = 50):
    I2C(Clk, Reset, Baud_kHz){
        input net( 5) Enable = 0b11111; // Register 01
        input net(32) Control = 0;      // Registers 09..06

        enum TYPE{ Voltage, Differential, Temperature }
        void SetType(num Channel, TYPE Type){
            assert(Channel >= 0 & Channel < 4);

            switch(Type){
                case(Voltage      ) Control[(4*Channel+3)..(4*Channel)] = 0;
                case(Differential) Control[(4*Channel+3)..(4*Channel)] = 1;
                case(Temperature ) Control[(4*Channel+3)..(4*Channel)] = 2;
            }
        }

        output net(-15) Vcc;
        output net(-15) V[8];
        output net(-15) InternalTemp;

        group MutEx{
            output net Request;
            input  net Grant = Request; // Default behaviour without a controller
        }

        void MakeRegisters(Registers){
            Registers.ReadOnly(Error);
            // Enable and Control registers are generally hard-coded

            Registers.ReadOnly(Vcc);
            for(n in 0..7) Registers.ReadOnly(V[n]);
            Registers.ReadOnly(InternalTemp);
        }

        fsm(Clk, Reset){
            !! Implement using the Transaction function in I2C base class
        }
    }
}

```

Listing D.18. Source summary of the LTC2991 module

D.6.2 Finite State Machine Construct

The I2C module presented in listing D.17 makes use of the `fsm` construct, the contents of which is presented in listing D.19.

```

net(8) Data;

// If ClkIn != ClkOut it means that the slave is clock-stretching
fsm(Clk, Reset, Baud_Trigger & (ClkIn == ClkOut) & !Quiet){
  // Start / Restart
  Data = TxData;
  if(Start | ClkOut){
    if(!ClkOut & !DataOut) DataOut = 1;
    if(!ClkOut & DataOut) ClkOut = 1;
    if( ClkOut & DataOut) DataOut = 0;
    if( ClkOut & !DataOut) ClkOut = 0;
  }

  // Transfer Data
  for(n in 0..7){
    if(R_nW) DataOut = 1;
    else DataOut = Data[7];
    ClkOut = 1;
    ClkOut = 0, Data = :(Data[6..0], DataIn);
  }

  // Ack
  if(R_nW){
    result = Data, DataOut = !Ack;
    ClkOut = 1;
    ClkOut = 0;
  }else{
    DataOut = 1;
    ClkOut = 1;
    ClkOut = 0,
    if(DataIn != Ack){
      Error = 1,
      return;
    }else{
      Error = 0;
    }
  }

  // Stop
  if(Stop){
    if( ClkOut & DataOut) ClkOut = 0;
    if(!ClkOut & DataOut) DataOut = 0;
    if(!ClkOut & !DataOut) ClkOut = 1;
    if( ClkOut & !DataOut) DataOut = 1;
  }
}

```

Listing D.19. Source listing of the I2C FSM

In this specific case it could be argued that a more traditional RTL implementation would be the superior option, especially for the start and stop sequences. The exact timing of the circuit is not evident at first glance, and is most likely quite difficult for a compiler to translate to an efficient netlist. For instance, the start sequence can be translated (by a human developer) to the Verilog presented in listing D.20.

```

case(State)
  IdleState: begin
    Data = TxData;
    if(Go) begin
      Busy = 1;
      if(Start || ClkOut) State <= StartState;
      else                State <= TransferState;
    end
  end
//-----

  StartState: begin
    case({ClkOut, DataOut})
      2'b00: DataOut <= 1'b1;
      2'b01: ClkOut  <= 1'b1;
      2'b11: DataOut <= 1'b0;
      2'b10: begin
        ClkOut <= 1'b0;
        State  <= TransferState;
      end
    endcase
  end
//-----

// ...
endcase

```

Listing D.20. Verilog equivalent of the I2C start sequence

The human implementation makes use of a single **Start** state to handle the entire sequence, making use of the sequence itself as part of the state variable. The ALCHA compiler, on the other hand, must implement the conditional fence⁴ in a way that the end-result mimics the behaviour described. It is difficult for the compiler to see the pattern in the output and make use of that, so will most likely duplicate code in multiple states in order to obtain the same effect.

The user can write the ALCHA in a slightly different way to make the netlist more obvious, as presented in listing D.21, which solves most of the translation issues.

```

// Start / Restart
Data = TxData;
if(Start | ClkOut){
  loop{
    switch(:(ClkOut, DataOut)){
      case(0b00) { DataOut = 1;      }
      case(0b01) { ClkOut  = 1;      }
      case(0b11) { DataOut = 0;      }
      case(0b10) { ClkOut  = 0, break; }
    }
  }
}

```

Listing D.21. Source listing of the I2C FSM

⁴A ‘fence’ is an instruction for the state machine to execute the next bit of code in the next clock cycle. These fences are ‘conditional’ because they are in the **if** statement main body, but not the **else**. This means that the following statements must sometimes be executed in the same clock cycle, and other times in the next clock cycle.

This example shows that, even though the ALCHA `fsm` construct can make finite state machines easier to implement in some scenarios, the user should still be aware of what the compiler is actually doing, as well as the likely resulting netlist. The language does not eliminate the need for low-level knowledge and experience.

D.6.3 Mutual Exclusion

There are multiple modules connected to the I²C bus, which must be arbitrated. The I²C standard specifies that arbitration should be performed by collision-detection and random back-off [140], but this has been found to be inefficient and can lead to starvation. A `MutEx` module is used for arbitration instead.

Instead of generating special signals and connecting them to the `MutEx` module instance, the user can simply call the `Add` function to add the module to the `MutEx` group. Duck-typing is used to make the `Add` function generic, so that any module that has the appropriate nets defined can be added to the `MutEx` group. The `MutEx` module itself is presented in listing D.22.

Similarly to the `Registers` module, the `MutEx` module requires the user to call the `Generate` function at the end of the code-base (or at least after the last module has been added to the `MutEx` pool. This seems to be a recurring pattern, so the ALCHA language would benefit from a feature that automatically runs a piece of code when the parser has reached the end of the code-base.

The language was modified by introducing a new keyword: `finally`. The `finally` member function (or method) is called automatically at the end of parsing the source, so that the user does not need to. The case study source was then updated accordingly: the interested reader can refer to the `feature/finally` tag in the git repository [85].

D.7 Waveform Generator

The waveform generator is based on an ADF4159 synthesiser, which is a library function. The aim is to implement the library function as generic as possible, so that the same module can be used in as many use-cases as possible. The general class structure is presented in listing D.23, with more detail in [the git repository](#) [85].

```

class MutEx(net Clk, net Reset){
    'Modules = [];
    //-----

    void Add(Module){
        'Modules.append(Module);
    }
    //-----

    void Generate(){
        num N = 'Modules'.length;
        assert(N > 0);

        net(N) Request;
        net(N) Grant = 0;
        net(N) Device = 1;

        rtl(Clk, Reset){
            if(Request & Device){
                Grant = Device;
            }else{
                Grant = 0;
                Device = ~(Device[N-2..0], Device[N-1]);
            }
        }

        // Cannot assign directly, because one is an array of objects
        // and the other a vector of bits
        for(n in 0..(N-1)){
            Request[n] = 'Modules[n].MutEx.Request;
            'Modules[n].MutEx.Grant = Grant[n];
        }
    }
}

```

Listing D.22. Implementation of the MutEx module

```

class ADF4159(net Clk, net Reset, num Baud_kHz = 1000){
public{ // Ports
    !! Define the device parameters as optional module ports

    // Control interface
    input net Update;
    output net Busy = 1;
    input net Trigger;
}
private{ // Local nets of the device pins
    net SPI_Clk = 0;
    net SPI_Data;
    net SPI_LE = 1;
}
void ConnectPins(pin SClk, pin Data, pin LE, pin TxData){
    SClk = SPI_Clk;
    Data = SPI_Data;
    LE = SPI_LE;
    TxData = Trigger & RampOn;

    !! Assign the device-specific timing parameters
}
private{ // Convenience variables
    !! Define various helper variables
}
void SetStart(num StartFreq){
    !! Set the registers associated with the start frequency
}
void SetRamp(num Bandwidth, num UpTime, num DownTime){
    !! Set the registers associated with ramp generation
}
void SetRefFreq(num RefFreq, bool FeedbackVcoDiv2 = true){
    !! Set the registers associated with reference and feedback options
}
private{ // Parameters to device Registers mapping
    !! Map individual registers to the device R0 to R7
}
private{ // Baud Counter
    !! Generate the Baud_Trigger signal
}
private void SendSPI(net(32) Register){
    !! Function used to set the device register
}
fsm(Clk, Reset, Baud_Trigger){
    !! FSM used to update the device registers
}
}

```

Listing D.23. Class overview of the ADF4519 module

D.7.1 Helper Functions

Sometimes the user would want to control the device by means of registers, and other times the user would want to hard-code parameters based on more abstract concepts. In other words, sometimes the control registers must be hooked up to the control bus, thereby controlling the parameters by means of a software function, and other times the user would want to hard-code the parameters, in which case a scripting function to calculate the parameters would be quite useful.

The ramp definition, for instance, is controlled by means of `DeviationWord`, `DeviationOffset` and `StepWord` registers, among others. If the ramp is to be hard-coded in firmware it is more convenient to specify the sweep bandwidth and ramp durations instead, in which case the user can make use of the `SetRamp` function.

D.7.2 Pins

The `ADF4159` module is a library function, and therefore has no knowledge of the project-specific pin names. It must, however, assign device-specific timing parameters to the pins.

One solution to this bootstrapping issue is to make use of private nets within the module, and then connect those nets to the pins by means of a `ConnectPins` function. The user of the module can then assign pins as required. The same strategy was followed with the I²C pins in listing [D.17](#).

D.7.3 SPI State Machine

The SPI finite state machine detail is presented in listing [D.24](#). This works better than the I²C state machine used in listing [D.17](#). Both functionality and timing detail is clear from the source, with a significant reduction in verbosity when compared with an HDL equivalent.

D.7.4 Container Module

There is a need for a container module that handles the project-specific aspects of the module instance. This is presented in listing [D.25](#).

```
private void SendSPI(net(32) Register){
    net(32) Data;
    SPI_Data = Data[31];

    fsm{
        Data = Register,
        SPI_LE = 0;

        loop(32){
            SPI_Clk = 1;
            SPI_Clk = 0, Data <<= 1;
        }

        SPI_LE = 1;
    }
}

fsm(Clk, Reset, Baud_Trigger){
    loop{
        Busy = 0,
        while(!Update);
        ;

        Busy = 1,
        SendSPI(R7);
        SendSPI(R6_1);
        SendSPI(R6_0);
        SendSPI(R5_1);
        SendSPI(R5_0);
        SendSPI(R4_1);
        SendSPI(R4_0);
        SendSPI(R3);
        SendSPI(R2);
        SendSPI(R1);
        SendSPI(R0);

        while(Update);
    }
}
```

Listing D.24. Code listing of the SPI function of the ADF4519 module

```

import "Library/Timing/TriggerDelay"
import "Library/Peripherals/Synth_and_DDS/ADF4159"
//-----

TriggerDelay(ControlClock, MasterReset, MasterTrigger) SynthTrigger;

Registers.Writable(SynthTrigger.Enable);
Registers.Writable(SynthTrigger.Delay );
Registers.Writable(SynthTrigger.Length);
//-----

ADF4159(ControlClock, MasterReset) SynthInst;

Registers.Writable(SynthInst.CP_CurrentSetting);
Registers.Writable(SynthInst.RampOn);
Registers.Writable(SynthInst.Update);
Registers.ReadOnly (SynthInst.Busy);

SynthInst.SetRefFreq(100e6, true);
SynthInst.SetStart (9.5e9);
SynthInst.SetRamp (150e6, 1e-3, 50e-6);

Registers.Writable(SynthInst.Integer );
Registers.Writable(SynthInst.Fraction);

SynthInst.Trigger = SynthTrigger.Output;

SynthInst.ConnectPins(Synth.SPI.SClk,
                     Synth.SPI.Data,
                     Synth.SPI.Latch,
                     Synth.Trigger);

```

Listing D.25. Implementation of the WaveformGenerator module

The architectural pattern used in this example is dependent on the way ALCHA handles reset values. The `SetStart` function is used to set the `Integer` and `Fraction` nets, which are then connected to the register bank. The `Registers` module does not enforce a reset value, so whatever the value of the net is when the `Registers.Generate` function is called is used as the reset value.

D.8 Receiver Abstraction

The receiver abstraction module has two main components: the module used to obtain data from the ADS7056 ADC and the packetiser used to break the continuous stream of data into sweep-based packets. There is a further side-line component: the debug stream used to stream packets of raw ADC samples to the ARM-based controller. The latter runs via the `StreamBuffer` presented in section D.3.

D.8.1 ADC Abstraction

Listings D.26 and D.27 present the implementation of the ADC abstraction. It makes use of the `rtl` construct instead of the `fsm` construct used in previously presented modules.

```
class ADS7056(net Clk, net Reset){
    // Maximum SPI clock is 60 MHz, and gives a 2.5 MSps sampling rate
    assert(Clk'frequency <= 120e6);

    private{
        net nCS = 1;
        net SClk = 0;
        net Data;

        net(14) Shift;
        net( 6) Count = 0;
    }

    output group Output{
        net(14) Data;
        net Valid = 0;
    }

    void ConnectPins(pin nCS, pin SClk, pin SDO){
        nCS = this.nCS;
        SClk = this.SClk;
        Data = SDO;

        SClk'create_generated_clock = <divide_by = 2, source = Clk>;

        // The state machine takes care of the actual sample and hold timing, so
        // this just needs to make sure it's not ridiculously out of place. The
        // constraints below ensures that the signals change within a 5 us window,
        // 3 ns after the clock edge.
        nCS'set_output_delay'max += 0e-9; // Setup
        nCS'set_output_delay'min += -3e-9; // Hold

        // The actual IC delay is 2.5 to 10 ns, but the state machine ensures that
        // the data is sampled just before making SClk high.
        SDO'set_input_delay'min += 2.5e-9;
        SDO'set_input_delay'max += 10.0e-9;

        SDO'set_multicycle_path'to = Clk;
        SDO'set_multicycle_path'setup = 2;
        SDO'set_multicycle_path'hold = 1;
    }
}
```

Listing D.26. Implementation of the ADS7056 module

The timing constraints are assigned to the pins when the higher level module connects them. The PCB trace lengths are not included, because those are specified in the `Platform` module instead, as shown in listing D.28.

```

rtl(Clk, Reset){
  if(Count == 47) Count = 0;
  else          Count++;

  switch(Count){
    case(0      ) nCS = 0;
    case(1..36) SClk = ~SClk;
    case(37     ) nCS = 1;
    default      SClk = 0; // Reset in case something went wrong
  }

  if(SClk == 0) Shift = :(Shift[13..0], Data);

  if(Count == 32){
    Output.Data = Shift;
    Output.Valid = 1;
  }else{
    Output.Valid = 0;
  }
}
}

```

Listing D.27. Implementation of the ADS7056 module, continued

```

group ADC{ // ADS7056
  alias nCS = DE0.Arduino.IO[5];
  alias SClk = DE0.Arduino.IO[6];
  alias Data = DE0.Arduino.IO[7];

  SClk'frequency = 60e6;

  nCS'set_output_delay'clock      = SClk;
  nCS'set_output_delay'delay      = 0e-3; // Source-synchronous
  nCS'set_output_delay'uncertainty = 50e-12;

  Data'set_input_delay'clock      = SClk;
  Data'set_input_delay'delay      = 100e-3 / 2e8;
  Data'set_input_delay'uncertainty = 50e-12;
}

```

Listing D.28. Declaration of the ADC pins in the Platform module

D.8.2 Packetiser

The packetiser module takes the output stream of the ADC module and breaks it into packets. The full receiver abstraction is presented in listing D.29. It is relatively straightforward and self-explanatory. The debug streamer is pulled into its own namespace in order to avoid namespace clutter.

```
import "Library/Timing/TriggerDelay"
import "Library/Peripherals/ADC_and_BIM/ADS7056"
//-----

ADS7056(DspClock, MasterReset) ADC;
//-----

TriggerDelay(ControlClock, MasterReset, MasterTrigger) PacketTrigger;

Registers.Writeable(PacketTrigger.Enable);
Registers.Writeable(PacketTrigger.Delay );
Registers.Writeable(PacketTrigger.Length);

net PacketTrigger_DSP;
rtl(DspClock) PacketTrigger_DSP = PacketTrigger.Output;
//-----

group Packet{
    net      SoP;
    net      EoP;
    net(13, -1) Data;
    net      Valid = 0;
}

private net(12) n;
private num      N = 2500;

fsm(DspClock, MasterReset){
    loop{
        n = 0,
        Packet.Valid = 0,
        while(!PacketTrigger_DSP);
        ;

        while(n < N){
            Packet.SoP      = (n == 0),
            Packet.EoP      = (n == N-1),
            Packet.Data      := (~ADC.Output.Data(13), ADC.Output.Data(12..0)),
            Packet.Valid     = ADC.Output.Valid,
            if(ADC.Output.Valid) n++,
            ;
        } } }

import "DebugStreamer" as DebugStreamer;
```

Listing D.29. Implementation of the ReceiverAbstraction module

One notable nuance of the ALCHA language is the position of the semicolon, or fence. The first part of the FSM loop waits for the trigger, but the first valid output must happen after the trigger. The user must therefore include a fence after the `while(!PacketTrigger_DSP);` line, otherwise the first iteration of the following loop will happen in the same clock cycle,

thereby implying two drivers for the `n` net.

Similarly, the nested if statement must end in a comma, rather than a semicolon, so that the fence is outside the if statement body.

D.8.3 Debug Stream

The debug streamer module is presented in listing D.30. It is written such that it makes the write pointer read by the system controller always point to the sample just after the EoP of the previous one, thereby ensuring that the controller always reads whole packets. It also makes use of the raw-assign operator, because the packet data is signed, but the buffer data is defined as unsigned.

```
alias Buffer = SystemController.StreamBuffer;
//-----

net(Buffer.Address'width) WrAddress = 0;
net(Buffer.Address'width) RdAddress = 0;

Registers.ReadOnly (WrAddress);
Registers.Writable(RdAddress);
//-----

Buffer.Address = 0;

fsm(DspClock, MasterReset){
  loop{
    loop{ // Synchronise to SoP
      if(Buffer.Write) Buffer.Address++,
      else             WrAddress = Buffer.Address,

      Buffer.WriteData := :(Packet.Data, 0@2), // Extend to 16-bit

      if(Packet.Valid & Packet.SoP){
        Buffer.Write = 1,
        break;
      }else{
        Buffer.Write = 0;
      }
    }

    loop{ // Write the packet
      Buffer.WriteData := Packet.Data,
      Buffer.Write      = Packet.Valid,
      if(Packet.Valid){
        Buffer.Address++,
        if(Packet.EoP) break;
      }
    }
  }
}
//-----
```

Listing D.30. Implementation of the DebugStreamer module

D.9 Radar Processor

Figure 7.3 presents the overview of the DSP chain. The chain input (from the `ReceiverAbstraction` module) is queued, after which the range and Doppler FFTs are taken. The result is then filtered. The ARM processor reads the result directly from the shared external memory in order to perform detection and tracking, which is beyond the scope of this case study.

D.9.1 Programming Model

It would be very convenient if the DSP chain could be programmed similarly to a Linux command-line, i.e. by piping the stages together, thereby following the “pipes and filters” architectural pattern presented in section 4.2.1.2. This can be achieved by means of operator-overloading, as illustrated in listing D.31.

```
import "Library/DSP/Streams"
import "Library/DSP/FIFO"
import "Library/DSP/Window"
import "Library/DSP/FFT"
import "Library/DSP/RealFFT"
import "Library/DSP/Transpose"
import "Library/DSP/AlphaFilter"
//-----

net (18, 1) Alpha;
Registers.Writable(Alpha);

// The Alpha Filter does not produce an output stream
Receiver.Packet | FIFO      (DspClock, MasterReset, 8192);
                 | Window   (DspClock, MasterReset, 2500, "Hann");
                 | FFT      (DspClock, MasterReset, 4096);
                 | Transpose (DspClock, MasterReset, 2048, 128);
                 | Window   (DspClock, MasterReset, 128, "Hann");
                 | FFT      (DspClock, MasterReset, 128);
                 | AlphaFilter(DspClock, MasterReset, 128, 2048, Alpha);
```

Listing D.31. Implementation of the `RadarProcessor` module

One problem with this approach is that none of the DSP module instances have handles. If the user wants to turn the FIFO length into a register for monitoring purposes, there is nothing to base the register on. To solve this, the user would have to follow a different pattern, such as listing D.32.

The class body implements the infrastructure used to perform the calculation, whereas the pipe operator overload function implements the calculation itself. This model makes it possible for the implementation to automatically infer all data widths and types as required.

```

import "Library/DSP/Streams"
import "Library/DSP/FIFO"
import "Library/DSP/Window"
import "Library/DSP/FFT"
import "Library/DSP/Transpose"
import "Library/DSP/AlphaFilter"
//-----

FIFO      (DspClock, MasterReset, 8192      ) Queue;
Window    (DspClock, MasterReset, 2500, "Hann") RangeWindow;
FFT        (DspClock, MasterReset, 4096      ) RangeFFT;
Transpose  (DspClock, MasterReset, 2048, 128 ) CornerTurn;
Window     (DspClock, MasterReset, 128, "Hann") DopplerWindow;
FFT        (DspClock, MasterReset, 128      ) DopplerFFT;
AlphaFilter(DspClock, MasterReset, 128, 2048 ) Filter;
//-----

Registers.ReadOnly (Queue.NumItems);
Registers.Writeable(Filter.Alpha);
//-----

// The Alpha Filter does not produce an output stream
Receiver.Packet | Queue | RangeWindow | RangeFFT | CornerTurn
                | DopplerWindow | DopplerFFT | Filter;

```

Listing D.32. Implementation of the RadarProcessor module

With the exception of the stream between the window and associated FFT, each intermediary stream contains data that is represented by a fixed-point format with unity full-scale. This standardises the modules in such a way that it is easy to change the data widths as required without influencing the scaling. The compiler fixed-point functionality takes care of the resulting change in bit-shift operations.

D.9.2 FIFO Queue

The queue is placed as early in the DSP chain as possible so that it buffers the smallest amount of data possible. All DSP processing steps after the queue grows the data rate. Listing D.33 shows the general overview of the FIFO implementation. The data width, and therefore the RAM width, is automatically inferred from the input stream.

The overloaded pipe operator is used to call the `Run` function, and thereby implement the pipes and filters architecture mentioned in section D.9.1. All DSP modules follow this same pattern. The data is stored into the queue using raw-assignments in order to be independent of fixed-point formatting.

```

import "../Memory/DualPortRAM"
//-----

class FIFO(net Clk, net Reset, num Length){
  DataStream Run(Input){
    num Width = Input.Width;

    DualPortRAM(Clk, Reset, Width+3, Length) RAM;

    result = DataStream(Width); // Create new stream instance for the output

    RAM.WrAddress = 0;
    RAM.WrData    := :(Input.SoP, Input.EoP, Input.Data);
    RAM.WrEnable  = Input.Valid;

    RAM.RdEnable  = Output.Ready;
    RAM.RdAddress = 0;

    rtl(Clk, Reset){ // Input
      if(Input.Valid) RAM.WrAddress++;
    }

    net Valid = 0; // Takes care of the RAM read latency
    num(RAM.RdAddress'width) NumItems = RAM.WrAddress - RAM.RdAddress;

    rtl(Clk, Reset, result.Ready){ // Output
      :(result.SoP, result.EoP, result.Data) := RAM.RdData;
      result.Valid = Valid;

      if(NumItems != 0){
        Valid = 1;
        RAM.RdAddress++;
      }else{
        Valid = 0;
      }
    } } } }
  DataStream operator| (Input, FIFO Queue){
    return Queue.Run(Input);
  }
}

```

Listing D.33. Implementation of the FIFO module

D.9.3 Window

The window can be used for both real and complex data, but with real coefficients in both cases. The abbreviated implementation is shown in listing D.34. The coefficients are scaled such that the maximum range of the 18-bit format is used while allowing the FFT to correctly scale the output with bit-shifts only.

```
import "../Memory/DualPortROM"
//-----

class Window(net Clk, net Reset, num Length, string Function){
    DualPortROM(Clk, Reset, 18, Length) ROM;

    private{
        num n[Length] = 0..(Length-1);

        switch(Function){
            case("Hann") ROM.Initial = (sin(pi*n/Length))*2;
            !! Other windows
            default ROM.Initial = 1; // Rectangular
        }

        num Sum = 0;
        for(c in ROM.Initial) Sum += c;
        ROM.Initial /= Sum;

        num Max = 1e-24;
        for(c in ROM.Initial){ if(Max < c) Max = c; }

        num Fullscale = 1;
        while(Fullscale > Max) Fullscale /= 2;
        while(Fullscale < Max) Fullscale *= 2;

        ROM.Initial = round(ROM.Initial * (1p18 / Fullscale));
        net(Width, Fullscale) Coefficient := ROM.Data_A;
    }
    auto Run(Input, bool Complex){
        !! Apply the window to the stream
    }
}

DataStream operator| (DataStream Input, Window Instance){
    return Instance.Run(Input, false);
}

IQ_Stream operator| (IQ_Stream Input, Window Instance){
    return Instance.Run(Input, true);
}
```

Listing D.34. Implementation of the Window module

The class instance creates the coefficients ROM as specified by the window type in the constructor. The pipe operator overload function implements the multiplication operation. There are two versions of the pipe operator overload function: one for real streams and another for complex streams.

D.9.4 FFT

The FFT module is intended to implement the Cooley–Tukey FFT algorithm [141]. The relatively low sampling rate of the system means that the FFT for this application does not need to be pipelined. An in-place FFT can be performed instead. This happens in 3 stages: load the data into a buffer, then run the butterfly operation over N samples $\log_2(N)$ times, and finally stream out the result.

Dual-port internal memory is used for the buffer, so that two samples can be read every second clock-cycle, and two butterfly results can be written every other clock cycle. Each butterfly sequence therefore takes N clock-cycles, where N is the number of points of the FFT. The total time to load, execute and output is therefore $N(\log_2(N) + 2)$ clock cycles, which is 478 μs at the 120 MHz DSP clock of the system (for $N = 4\,096$). This is much faster than the 1.1 ms PRI, and therefore practical to implement in this way, even after considering any latency-related dead-time.

Similarly, the Doppler FFT takes 9.6 μs for each of the 2 048 FFTs. This implies a total of about 20 ms for one burst of 128 sweeps, which is much faster than the 140.8 ms it takes to transmit a full burst. There is therefore no need to parallelise the process.

The details of implementation are outside the scope of this case study, because it is a relatively straightforward state-machine that keeps track of timing and location related to reads and writes. The fixed-point scaling is interesting, however, and presented in listing D.35. The twiddle factors ROM is initialised similar to the Window function in section D.9.3, and therefore not repeated here.

The window function is designed such that normal FFT scaling would result in an output stream with unity full-scale. In order to keep precision, however, each butterfly operation is performed such that the FFT scaling is unity. The input stream is therefore scaled with `NumPoints`, but because this is guaranteed to be power-of-two the fixed-point engine will reduce the operation to a compile-time scaling of the fixed-point format.

The output stream has unity full-scale representation, whereas the buffer is capable of storing values larger than one. This is a result of the window scaling. Even with this mismatch, the output state-machine can simply assign the values stored in the buffer to the output stream, and the compiler fixed-point engine will sort out the shifting and scaling.

In order to prevent strange overflow behaviour, the `overflow` attribute of the output stream is set to `"clip"`, which will cause the fixed-point engine to synthesise a clipping circuit.

```

import "../Memory/FullDualPortRAM"
//-----

class FFT(net Clk, net Reset, num NumPoints){
  assert(2**round(log2(NumPoints)) == NumPoints);

  private{
    DualPortROM    (Clk, Reset, 36, NumPoints) TwiddleFactors;
    FullDualPortRAM(Clk, Reset, 80, NumPoints) Buffer;

    !! Other private declarations
  }
  //-----

  IQ_Stream Run(Input, bool ComplexInput){
    result = IQ_Stream(31, -1);
    result.I'overflow = "clip";
    result.Q'overflow = "clip";

    num BufferFullScale;
    if(ComplexInput) BufferFullScale = Input.I  'fullscale * NumPoints;
    else              BufferFullScale = Input.Data'fullscale * NumPoints;

    struct ComplexData{
      net(39, BufferFullScale) I, Q;
    }
    ComplexData Buffer_WrData_A, Buffer_WrData_B;
    ComplexData Buffer_RdData_A, Buffer_RdData_B;

    Buffer.WrData_A := :(Buffer_WrData_A.I, Buffer_WrData_A.Q);
    Buffer.WrData_B := :(Buffer_WrData_B.I, Buffer_WrData_B.Q);
    :(Buffer_RdData_A.I, Buffer_RdData_A.Q) := Buffer.RdData_A;
    :(Buffer_RdData_B.I, Buffer_RdData_B.Q) := Buffer.RdData_B;

    !! Scale the input data with NumPoints and
    !! zero-pad as required

    !! Run the engine with each butterfly scaled by 0.5
    !! i.e. WrData = (Even + Twiddle*Odd) / 2

    !! Stream the output in bit-reversed order
    !! For real input data, only stream out the first half of the result
  }
}

DataStream operator| (DataStream Input, FFT Engine){
  return Engine.Run(Input, false);
}

IQ_Stream operator| (IQ_Stream Input, FFT Engine){
  return Engine.Run(Input, true);
}

```

Listing D.35. Implementation overview of the FFT module

D.9.5 Transpose

The transpose would be implemented by using the same pattern as the modules presented above. The input stream is placed in external memory by using a page-based write cache to match the memory transactions to natural page sizes and therefore maximise RAM bandwidth.

The output stream is formed by performing a strided read from the external memory and placing it in a cache that perform an automatic prefetch of the next range-bin before the Doppler FFT requires it. This minimises wait-time caused by the RAM read operation.

The `SystemController` module exposes an `SDRAM` interface, which is an instance of the `AvalonSlave` class. Multiple masters must connect to this interface, which is achieved by means of arbitration implemented within the `AvalonSlave` class. The class exposes a `Attach` function used to attach various masters.

D.9.6 Alpha Filter

The alpha filter would be implemented by using a page-based read cache with auto-prefetch functionality, so that the filter is ready by the time the Doppler FFT streams out its result. Pages are 4 kiB in size, whereas each Doppler FFT output is 1 kiB (128 samples of 8 bytes each: 32-bit I and 32-bit Q). Each page therefore contains the data of four range bins.

When the Doppler FFT streams the output the internal memory is updated in-place, after which the resulting page is written back to external memory before the next page is fetched.

A typical Cyclone V based system has an external RAM bandwidth of 3.2 GiB/s (at 800 MT/s with a 32-bit data width). This means that, neglecting RAM latency, a page can be read or written in about 1.3 μ s, which is much faster than the 9.6 μ s latency of the Doppler FFT. There is therefore no need to perform double-buffering.

D.10 Version Autogen

Listing [D.36](#) presents the implementation of the `FirmwareVersion` module. It makes use of compiler constants and a shell system call in order to obtain the build time-stamp and Git hash. The resulting nets are then added to the register bank in the `main` module.

```
class FirmwareVersion(num Major, num Minor){
  private num HexToNum(string S){
    result = 0;
    for(s in S){
      if (s >= "0" & s <= "9") result = (0x10*result) | ((s - "0") );
      else if(s >= "a" & s <= "f") result = (0x10*result) | ((s - "a") + 10);
      else if(s >= "A" & s <= "F") result = (0x10*result) | ((s - "A") + 10);
      // Simply ignore invalid characters
    }
  }

  output{
    net(32) Version = (Major << 16) | Minor;
    net(32) Date    = HexToNum(__DATE__);
    net(28) Time    = HexToNum(__TIME__);
    net(32) GitHash = HexToNum(shell("git rev-parse --short=8 HEAD"));
  }
}
```

Listing D.36. Implementation of the `FirmwareVersion` module

Appendix E

ALCHA Grammar

This appendix serves as a reference to the ALCHA grammar, in EBNF. The language is further described in appendix F.

E.1 Scanner

```
WhiteSpace = Space | Newline | Comment;
Comment    = LineComment | BlockComment | NestedComment | ToDoComment;

Space      = ? U+0020 ? | ? U+0009 ? | ? U+00A0 ? | ? U+1680 ? | ? U+2000 ?
            | ? U+2001 ? | ? U+2002 ? | ? U+2003 ? | ? U+2004 ? | ? U+2005 ?
            | ? U+2006 ? | ? U+2007 ? | ? U+2008 ? | ? U+2009 ? | ? U+200A ?
            | ? U+202F ? | ? U+200B ? | ? U+205F ? | ? U+3000 ? | ? U+FEFF ? ;
Newline    = ? U+0A U+0D ? | ? U+0D U+0A ? | ? U+0A ? | ? U+0D ? | ? U+0B ?
            | ? U+0C ? | ? U+85 ? | ? U+2028 ? | ? U+2029 ? ;

Byte       = ? Any 8-bit unit from the input stream ?;

B = "0" | "1";
O = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7";
D = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
H = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
    | "a" | "b" | "c" | "d" | "e" | "f" | "A" | "B" | "C" | "D" | "E" | "F";

NonDigit = ?"a".."z"? | ?"A".."Z"? | "_"
           | (? Any Unicode >= U+80 ? -Space -Newline);
```

```
LineComment = "/" { ? Any byte sequence ? -Newline } Newline;
ToDoComment = "!!" { ? Any byte sequence ? -Newline } Newline;
BlockComment = "/*" { ? Any byte sequence ? -* / } "*/";
NestedComment = "+/" { NestedComment | (? Any byte sequence ? -"/") } "+/";
```

```

Token      = String | Literal | Keyword | Identifier | Operator;

Special    = "__FILE__" | "__LINE__" | "__DATE__" | "__TIME__"
            | "__CLASS__" | "__FUNCTION__" | "__NAMESPACE__";

Keyword    = "void"      | "auto"      | "operator" | "unary"
            | "pin"      | "net"      |
            | "byte"     | "char"     | "num"      | "string"
            | "bool"     | "true"     | "false"
            | "class"    | "enum"     | "struct"   | "group" | "alias"
            | "this"     | "result"   | "finally"
            | "input"    | "output"
            | "public"   | "private"  | "protected"
            | "if"       | "else"     | "for" | "in" | "while" | "loop"
            | "switch"   | "case"     | "default"
            | "import"   | "as"
            | "return"   | "break"    | "continue" | "goto"
            | "func"     | "inline"
            | "rtl"      | "fsm"      | "hdl"
            | "stimulus" | "emulate"  | "sequence" | "assert" | "wait" ;

Operator   = "++" | "--" | "'" | "..." | ":" | "." | ".{"
            | "&" | "~&" | "|" | "~|" | "^" | "~^" | "~" | "!"
            | ":( " | ":[" | "`"
            | "+" | "-" | "*" | "/" | "%" | "**"
            | "<<" | ">>" | "<" | ">" | "<=" | ">=" | "==" | "!="
            | "?" | "?: " | "?."
            | "=" | ":@" | "~=" | "+=" | "-=" | "*=" | "/=" | "%="
            | "^=" | "&=" | "|=" | "**=" | "<<=" | ">>="
            | "(" | ")" | "[" | "]" | "{" | "}" | "," | ";"
            | "$" | "$("
            | "@" | "#" | "##" | "[" | "*" | "[->" | "[=" | "|->" | "|=">
            | "|" | "&&" | "&&&" ;

```

```

Literal    = Binary | Octal | Hexadecimal | Decimal [Exponent] ["i" | "j"];

Identifier = (NonDigit {D | NonDigit}) -Keyword;

String     = "'" { (Byte -'"' -"\") | EscapeSequence} "'" [String];

EscapeSequence = "\n" | "\t" | "\v" | "\b" | "\r" | "\f"
               | "\a" | "\\ " | "\?" | "\'" | "\'"
               | ("\" x" H H) | ("\u" H H H H) | ("\U" H H H H H H H H)
               | ("\ " 0 {0})
               | ("\"&" ? Any of the HTML-5 character names ? ";"");

Binary     = "0b" ((B | "_" ) {B | "_" } ["." (B | "_" ) {B | "_" }]);
Octal      = "0o" ((0 | "_" ) {0 | "_" } ["." (0 | "_" ) {0 | "_" }]);
Decimal    =      D      {D | "_" } ["." (D | "_" ) {D | "_" } ] ;
Hexadecimal = "0x" ((H | "_" ) {H | "_" } ["." (H | "_" ) {H | "_" }]);

Exponent   = ("p" | "P" | "e" | "E") ["-" | "+"] D {D};

```

E.2 Declarations

```

Module = [Statements];

Statements = Statement {Statement};
Statement = Label | Definition | ClassDefinition | EnumDefinition
           | Alias | Import | Struct | Group | AccessDirectionGroup
           | IfStatement | For | While | Loop | Switch | Jump | GoTo
           | FunctionCall | NamespacePush | Assignment
           | RTL | FSM | HDL
           | Stimulus | Emulate | ForkJoin | Assert | Sequence | ";";

Label = Identifier ":"; (* Only useful within an fsm construct *)

Definition = ( BaseType | TypeIdentifier ) [ParameterList]
            [AttributeList] IdentifierList;

ClassDefinition = "class" [AttributeList] Identifier
                [ "(" [DefParameterList] ")" ]
                [
                  ":" TypeIdentifier [ParameterList]
                  { "," TypeIdentifier [ParameterList] }
                ]
                "{ " [Statements] " }";

AccessDirectionGroup = ( ( AccessSpecifier [DirectionSpecifier] ) |
                        ( DirectionSpecifier ) ) StatementBlock;
AccessSpecifier      = "public" | "private" | "protected";
DirectionSpecifier   = "input" | "output";

```

```

BaseType = "pin" | "net"      (* Synthesisable types *)
          | "void" | "auto"   (* Function return types *)
          | "num" | "byte"    (* Scripting numeric types *)
          | "char" | "string" (* Scripting text types *)
          | "bool"           (* Scripting boolean type *)
          | "func" ;         (* Function pointer *)

IdentifierList = FunctionDef | (VariableDef {"," VariableDef} ";");

VariableDef    = Identifier {ArrayDefinition} [Initialiser];
FunctionDef    = ["inline"] ( Identifier | OperatorName ) {ArrayDefinition}
                ("[" [DefParameterList] ")")
                [ "{" [Statements] "}" ] ;
OperatorName   = ( "operator" | "unary" ) Operator ;

ArrayDefinition = "[" [Expression] "]";
Initialiser     = ("=" | ":=") Expression;

DefParameterList = DefParameter {"," DefParameter} [","];
DefParameter     = [DirectionSpecifier] [BaseType | TypeIdentifier]
                  Identifier {"[" "]" } [Initialiser];

AttributeList    = "<" AttributeAssignment {"," AttributeAssignment} ["," ">";
AttributeAssignment = Identifier "=" Primary;

EnumDefinition = "enum" Identifier "{" Identifier {"," Identifier} ["," ">";

TypeIdentifier = Identifier {"." Identifier}; (* Namespace.TypeName *)

```

E.3 Expressions

```

Expression    = BitwiseOR      ( [ "?" Cast ":" Cast ] | [ ":" Cast ] );
BitwiseOR     = BitwiseXOR     {("&" | "~&") BitwiseXOR };
BitwiseXOR    = BitwiseAND     {("&" | "~&") BitwiseAND };
BitwiseAND    = Equality       {("&" | "~&") Equality };
Equality      = Relational     {("==" | "!=") Relational };
Relational    = Shift         {("<" | ">" | "<=" | ">=") Shift };
Shift         = Additive       {("<<" | ">>") Additive };
Additive      = Multiplicative {("+", "-") Multiplicative};
Multiplicative = Exponential   {("*", "/", "%") Exponential };
Exponential   = Replication    { "**" Replication };

Replication    = Stringification ["`" Primary]; (* Vector replication *)

(* When used as a function, the second parameter must evaluate to a string *)
Stringification = ["$"] Reduction
                  | "$(" Reduction [", " Expression] ")";

Reduction = ["+", "&", "~&", "|", "~|", "^", "~^", "!"] Range;

Range = Unary [".."] Unary [":" Unary];

Unary = {"-", "~", ":", "++", "--"} Postfix;

```

```

Postfix = Cast { Array           | (* Slice *)
                  ParameterList  | (* Function call *)
                  "." Identifier | (* Member reference *)
                  "?." Identifier | (* Safe member reference *)
                  "'." Identifier | (* Attribute reference *)
                  "++"           | (* Increment *)
                  "--"           | (* Decrement *)
                  "!"            | (* Factorial *)
                };

Cast    = Primary [CastEpr];
CastEpr = "@" ( Identifier | Literal
              | (" ExpressionList ") );

(* The "'" prefix chooses the global scope *)
Primary = ([ "'" ] Identifier) | Literal | "true" | "false"
          | "this" | "result"
          | VectorConcat | ArrayConcat | Array | String
          | AttributeList | (" Expression ");

(* These perform vector or array concatenation *)
VectorConcat = ":" (" ExpressionList ");
ArrayConcat  = "[" (" ExpressionList ");

(* This builds an array (an array of arrays when nested) *)
Array = "[" [ExpressionList] ";";

ExpressionList = Expression { "," Expression } [ "," ];
ParameterList  = "(" [Parameter { "," Parameter } [ "," ] ] ")";
Parameter      = [Identifier ("=" | ":=")] Expression;

```


E.4 Statements

```

StatementBlock = Statement | ("{" [Statements] "}");

Alias    = "alias" Identifier "=" Expression ",";
Import  = "import" String ["as" Identifier] ",";
Struct  = "struct" [AttributeList] [Identifier] "{"
        { (Definition | Struct) "; " }
        "}";
Group   = "group"   [AttributeList] [Identifier] "{" [Statements] "}";

(* The result from "Postfix" or "Concatenation" is checked for syntactic
   validity before use as a name-space, class name or assignment target. *)
FunctionCall = Postfix ("," | ";");
NamespacePush = Postfix ".{" [Statements] "}";
Assignment    = Postfix AssignmentOperator Expression ("," | ";");

AssignmentOperator = "=" | ":@" | "~=" | "+=" | "-=" | "*=" | "/=" | "**="
                    | "%=" | "&=" | "|=" | "^=" | "<<=" | ">>=" ;

IfStatement = "if" "(" Expression ")" StatementBlock ["else" StatementBlock];

For = "for" "(" Identifier "in" Range ")" StatementBlock;

(* For while loops, Expression must evaluate to a boolean *)
While = "while" "(" Expression ")" StatementBlock;

(* For loop loops, Expression must evaluate to an integer *)
Loop = "loop" "(" Expression ")" StatementBlock;

```

```

(* For case statements, Expression must evaluate to a constant or range *)
Switch = "switch" "(" Expression ")" "{"
    {"case" "(" ExpressionList ")" StatementBlock}
    ["default" StatementBlock]
    "}";

(* For "break" and "continue", the expression indicates the number of levels,
   which is 1 by default. *)
Jump = ("return" | "break" | "continue") [Expression] ";";
GoTo = "goto" Identifier ";";

(* The first parameter must evaluate to a clock net. If the RTL is inside
   a function, all parameters are inherited from the calling state machine. *)
RTL = "rtl" [AttributeList] [ParameterList] StatementBlock;

(* The first parameter must evaluate to a clock net and the second parameter
   to a reset net. If the FSM is inside a function, the parameters are
   inherited from the calling state machine. *)
FSM = "fsm" [AttributeList] [ParameterList] StatementBlock;

(* The strings form a list of file names to include in the project, the
   Assignments form a list of HDL parameter assignments and the Definitions
   are port definitions. *)
(* The "Assignment" is obtained through "Other" *)
HDL = "hdl" [AttributeList] "(" String {" "," String} ["," "]" Identifier
    [{" {Assignment } "} ]
    [{" {Definition | Stimulus} "} ]";

```

E.5 Verification

The verification-related grammar will be completed when implementation of the prototype compiler gets to the verification features. The end result will be very similar to SystemVerilog. A preliminary overview is provided in [appendix F.16](#).

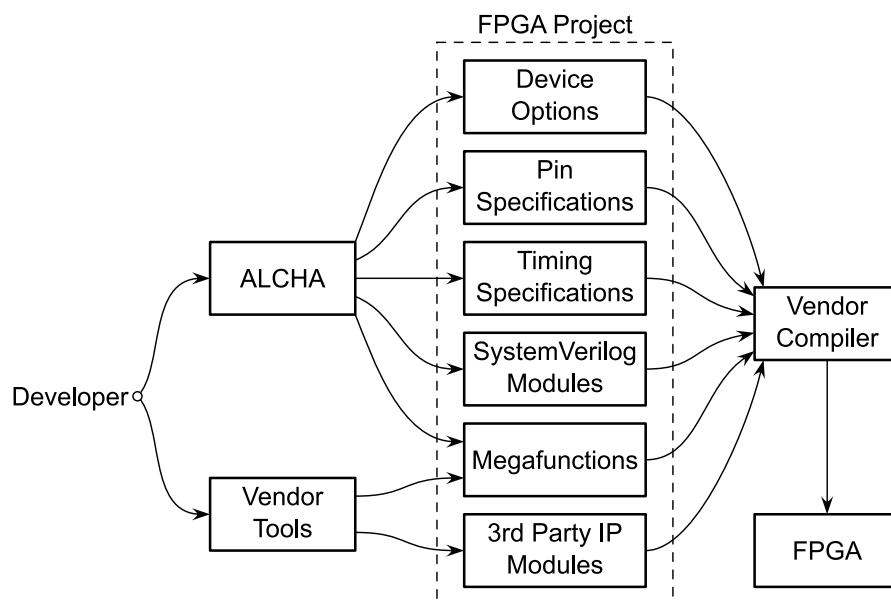
Appendix F

ALCHA Specification

This appendix is an abbreviated language specification based on the EBNF grammar presented in appendix E.

F.1 Introduction

ALCHA (architectural level computational hardware abstraction) is a command-line tool to generate vendor-specific FPGA projects from largely vendor-neutral text-based source code. The image below provides a rough overview of the intended tool-chain.



ALCHA is aimed at unifying the various aspects of FPGA firmware design (RTL design, finite state machines, timing and design constraints, as well as scripting) into a single language.

ALCHA is designed to be as portable, concise and expressive as possible, without the loss of low-level control. It provides a development platform that promises reduced development time and maintenance effort.

F.2 Lexical

The scanner reads UTF-8 formatted files. The UTF-8 formatting is retained, even though the input stream is searched for Unicode space and newline characters. At this stage, white-space and comments are ignored, adjacent strings are concatenated and escape sequences within strings are evaluated. The lexical EBNF is presented in appendix E.1.

F.2.1 Literals

ALCHA literals are all stored as infinite-precision rational numbers (by means of the [GNU MP library](#) [125]). An imaginary literal can be specified by using either a *j* or *i* postfix.

Literals can be binary (*0b* prefix), octal (*0o* prefix), hexadecimal (*0x* prefix) or decimal (no prefix). Literals may contain an optional decimal or binary exponent, by means of a *e* or *p* postfix, respectively.

All literals can be cast to a specific fixed-point format by means of the *@* operator. Within numerical literals, the underscore character (*_*) is ignored. Character literals are string literals with only one character. Below are some examples:

```
123.456           // decimal constant
123.456@ (16, 128) // Decimal constant cast to 16 bits with full-scale range [0, 128)
0b101.0101       // Binary constant
0o123.456        // Octal constant
0xABC.DEF        // Hexadecimal constant
123_456_e7        // Decimal constant with decimal exponent (10^7)
123_456_p7        // Decimal constant with binary exponent (2^7)
0x123_ABC_p7      // Hexadecimal constant with binary exponent (2^7)
7.3 + 8.9j        // Decimal complex constant
"A"              // Unicode character constant
"ABC"            // Unicode string literal (character array)
```

Imaginary and complex numbers are useful when using compile-time scripting to initialise FIR filter constants, among other uses.

Unicode characters can be used directly within strings, such as “ ω ” and “ $\alpha\beta\gamma$ ”. Certain non-printable characters can be used by means of escaping. ALCHA escape sequences are similar to D escape sequences. The following table provides a summary:

Sequence	Character
<code>\a</code>	Alarm (U+07)
<code>\b</code>	Backspace (U+08)
<code>\f</code>	Form-feed (U+0C)
<code>\n</code>	New-line (U+0A)
<code>\r</code>	Carriage return (U+0D)
<code>\t</code>	Horizontal tab (U+09)
<code>\v</code>	Vertical tab (U+0B)
<code>\\</code>	Backslash (U+5C)
<code>\'</code>	Single quotation mark (U+27)
<code>\"</code>	Double quotation mark (U+22)
<code>\?</code>	Question mark (U+3F)
<code>\nnn</code>	up to 11-digit octal Unicode
<code>\xHH</code>	2-digit hexadecimal Unicode
<code>\uHHHH</code>	4-digit hexadecimal Unicode
<code>\UHHHHHHHH</code>	8-digit hexadecimal Unicode
<code>\&nnnn;</code>	Named character, where <code>nnnn</code> is any of the HTML-5 character names

F.2.2 Identifiers

Non-digits in ALCHA are defined as the ‘_’ character, as well as any character in the ranges ‘a’ to ‘z’, ‘A’ to ‘Z’ and any Unicode character above U+7F, as long as it is not a white-space or newline character.

Digits in ALCHA are defined as any character in the range ‘0’ to ‘9’.

Identifiers start with a non-digit, which may then be followed with any digit or non-digit.

F.2.3 Predefined Constants

The following constants are predefined with double precision. They can be redefined by the developer, following the scoping rules.

Symbol	Description	Value
pi or π (U+0C30)	π	3.141 592 653 ...
e	Natural base	2.718 281 828 ...
i or j	Imaginary constant	$\sqrt{-1}$
__YEAR__	The current year	
__MONTH__	The current month	(1..12)
__DAY__	The current day of the month	(1..31)
__HOUR__	The current hour	(0..23)
__MINUTE__	The current minute	(0..59)
__SECOND__	The current second	(0..61)
__WEEKDAY__	The current day of the week	(1..7 $\Rightarrow$ Monday..Sunday)
__YEARDAY__	The current day of the year	(1..366)

F.2.4 Special Keywords

Alcha supports the following special keywords:

Keyword	Description
__FILE__	A string constant containing the current source file name
__LINE__	A literal constant equal to the current source line number
__DATE__	A string constant containing the current date, in “yyyy-mm-dd” format
__TIME__	A string constant containing the current time, in “HH-mm-ss” format
__CLASS__	A string constant containing the current class name. If not in a class, “Global” is returned.
__FUNCTION__	A string constant containing the current function name. If not in a function, “Global” is returned.
__NAMESPACE__	A string constant containing the current name-space. If not in a name-space, “Global” is returned.

F.2.5 Syntax Highlighting Scripts

Syntax highlighting scripts are available from [the repository](#).

F.3 Grammar

The parser takes tokens from the scanner and builds an abstract syntax tree. Nothing is evaluated at this point, not even definitions. Refer to appendix [E](#) for more details.

F.4 Modules

F.4.1 Target Independence

The intended target (simulation, Altera project, Xilinx project, etc.) is a feature of the compiler. A different target is selected by using either a different compiler, or by means of a compiler switch.

In order to make the code target independent, compile-time scripting can be used to implement conditional compilation. The code could, for instance, contain a header similar to the code below:

```
'target_type    = "Project";
'target_vendor  = "Altera";
'target_series  = "Cyclone V GX";
'target_device  = "5CGXFC5C6F27C7N";
'target_board   = "Cyclone V GX Starter Kit";
```

This header creates global attributes, which contain various string objects. This can then be used later in the code for conditional compilation, as follows:

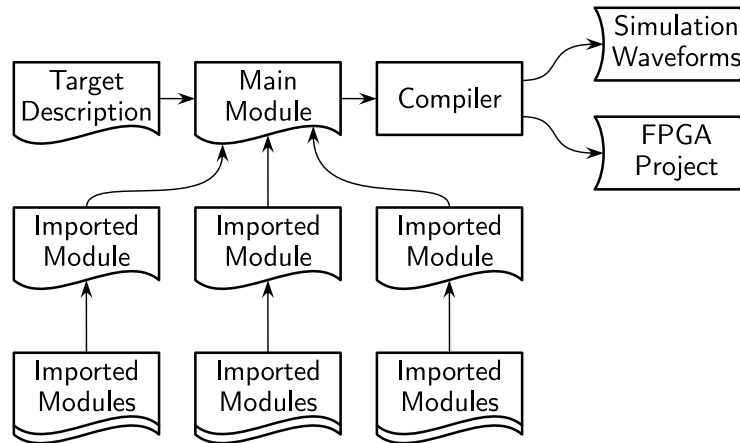
```
if('target_type == "Simulation"){
    // Some simulation-specific code
}else if('target_vendor == "Altera"){
    // Some Altera-specific code
}else if('target_vendor == "Xilinx"){
    // Some Xilinx-specific code
}
```

Some global attributes are standardised by the language, such as the `target_` series above. Others, in order to promote portability, should be standardised within a developer community.

F.4.2 Import and Name-space

Conceptually, ALCHA processes the source as a whole. It is convenient, however, to break the source into a hierarchical structure of files, as presented in the figure below. In the

context of ALCHA, a “module” refers to a source file, rather than an HDL module. An ALCHA module might or might not compile to a corresponding HDL module, depending on the source contents.



The `import` statement is used to import a child module into a parent module. There are two forms, as shown below. The first form imports the contents of “UART.alc” into the same name-space as the parent module, as if the file contents appeared in place of the `import` statement. The second form imports the contents of “Ethernet.alc” into the “Eth” name-space. It is illegal to import two modules into the same name within the same module. In both forms, the imported modules have direct access to objects in the parent module name-space.

```

import "UART";
import "Ethernet" as Eth;

```

This name-space model allows the developer to define an environment in a parent module. This environment is visible to all child modules, without the need of ports or parameter passing. The target and pin definitions are examples of where this is useful. As another example, the developer might, for instance, define a `Round` function, which is used by child modules whenever signals must be rounded. The rounding behaviour of the design can then be changed by simply modifying the behaviour of this `Round` function.

Specifying the class instance, or name-space, can become tedious. Instead, the developer can temporarily push a name-space onto the scope stack by means of a special syntax, as shown below. The `A.{` construct pushes the `A` name-space onto the stack, whereas the `B.{` construct pushes the `B` name-space. The closing curly braces pop the name-spaces from the stack.


```

class S1{net(8) r, g, b, a;}
class S2{net(8) x, y, z, w;}
S1() A;
S2() B;

// Instead of saying:
:[A.r, A.g, A.b, A.a] = :[B.x, B.y, B.z, B.w] + 1234;

// Say:
A.{
  B.{
    :[r,g,b,a] = :[x,y,z,w] + 1234;
  }
}

```

When referring to a symbol, the search progresses as follows:

1. The name-space stack is searched, from top to bottom, without following parent nodes of the name-space tree
2. The current node of the name-space tree is searched
3. The name-space tree is traversed from node to parent, following the tree towards the root (which is the global namespace)

F.5 Declarations

F.5.1 Nets

F.5.1.1 Definition

Nets relate to physical wires and / or registers on the FPGA. They are specified by means of the **net** keyword. If no format is specified, a net is a single-bit unsigned integer, which also represents a boolean. The format is specified by means of optional instantiation parameters. Some examples are shown below:

```

net      a; // Single-bit unsigned integer
net(8)   b; // 8-bit unsigned integer
net(7, -128) c; // 8-bit signed integer

net(4, 4) d; // 4-bit unsigned fixed point with 2 integer bits
           // and 2 fraction bits -- in the range [0, 4)
net(4, -4) e; // 5-bit signed fixed-point in the range [-4, 4)

```

Signed nets have one more bit than specified in the format. This is a convenient convention for fixed-point digital signal processing, but can be confusing.

F.5.1.2 Initialisers

All nets, pins and variables can take an optional initialiser, as illustrated below:

```
net(8 ) a = 7;          // The binary value "0000 0111"
net(8, 4) pi = 355/113; // The binary value "1100 1001"
```

F.5.1.3 Timing Constraints

Net attributes can be used to specify external peripheral timing requirements. These timing attributes are specified as if there are no PCB or trace delays, and follows the same model as the industry-standard Synopsis Design Constraints format. An example is provided below:

```
// PCB trace delays in the pin definitions:
pin(4)<voltage = 3.3, capacitance = 10e-12,
    set_input_delay = <clock = Clock, min = 500e-12, max = 1e-9>,
    set_output_delay = <clock = Clock>
    location = ["U7", "T7", "V8", "T8"]> SD_DAT;

// Peripheral specifications in the driver class:
net(4) DataIn;
net(4) DataOut;

// Assign the nets to the pin
DataIn      = SD_DAT.pad;
SD_DAT.driver = DataOut;
SD_DAT.enable = DataEnable;

// Add device-side timing constraints on top of the existing PCB delays
SD_DAT'set_output_delay'min    -= 5e-9;    // Hold timing
SD_DAT'set_output_delay'max    += 5e-9;    // Setup timing
SD_DAT'set_input_delay'delay    += 14e-9;   // IC delay
SD_DAT'set_input_delay'uncertainty += 100e-12; // Delay uncertainty
```

F.5.1.4 Bus Connections

ALCHA does not support high-impedance nets directly. Pins can be set to high-impedance by disabling the driver (see the Pins section for details). In order to emulate a bus that has a set of tri-state drivers, the nets should be gated through AND gates and then combined through an OR gate to drive the bus.

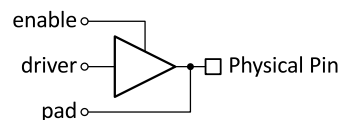
F.5.2 Pins

F.5.2.1 Definition

Pins in ALCHA are specified by means of the **pin** keyword. Pins can have various attributes. A short example is presented below:

```
pin<frequency = 50e6, standard = "2.5V", location = "H12"> Clock;
pin<                standard = "1.2V", location = "P11"> Key;
pin<                standard = "2.5V", location = "F7" > LED;
```

A pin is a structured object that consists of three nets: the **pad** (physical FPGA pin), the **driver** (input to the pin buffer) and the **enable** (driver enable). In order to set a pin to output, the driver must be enabled by assigning a constant high to the **enable** net. Bidirectional pins are created by assigning an expression to the **enable** net.



When, at the time of circuit synthesis, the **enable** net is undefined, the compiler infers the direction of the pin. If there is an expression assigned to the **driver** net, the pin is an output and the driver is enabled. If the **driver** net is also undefined, the pin is an input.

When the developer assigns directly to the pin name, the expression is actually assigned to the **driver** net. When the pin is read, the **pad** net is read. This is sometimes problematic, as illustrated in the following example:

```

// Define I2C bus pins (initialisers assign to the driver net)
pin S_Clk  = 1;
pin S_Data = 0;

// Assign pin direction
S_Data.enable = 0;

// Add a device to the bus (wrong)
S_Clk  &= ThisClock; // Equivalent to S_Clk .driver = S_Clk .pad & ThisClock;
S_Data |= ~ThisData; // Equivalent to S_Data.driver = S_Data.pad | ~ThisData;

// Add a device to the bus (correct)
S_Clk .driver &= ThisClock;
S_Data.enable |= ~ThisData;

// Add another device to the bus
S_Clk.driver  &= ThatClock;
S_Data.enable |= ~ThatData;

// Read the Data line
pin LED = S_Data; // Equivalent to LED.driver = S_Data.pad

```

When ignoring the first (wrong) assignment, the code above is equivalent to the following Verilog:

```

module TopLevel(
    output S_Clk,
    inout  S_Data,
    output LED
);

wire  S_Data_enable;
assign S_Data = S_Data_enable ? 1'b0 : 1'bZ;

assign S_Clk      = 1'b1 & ThisClock & ThatClock;
assign S_Data_enable = 1'b0 | ~ThisData | ~ThatData;

assign LED = S_Data;

```

Typically, the compiler will remove the and-with-one and or-with-zero. It is kept here to show the relationship between the ALCHA and Verilog more clearly.

F.5.2.2 Location Assignment

The `location` attribute is used to assign the pin to a physical pin number of the target device. In the case of a differential pair, the pin location is specified in “PP-NN” format, where PP is the positive pin number and NN the negative pin number.

When specifying pin locations for bit-vectors or pin arrays, the `location` attribute contains a comma-separated list of pins. Pin locations are specified most-significant bit first, as follows:

```
// Assign H9 to LED(7), H8 to LED(6), ..., L7 to LED(0)
pin(8)<location = ["H9", "H8", "B6", "A5", "E9", "D8", "K6", "L7"]> LED;
```

Pin arrays are handled in similar fashion. The first (0-index) pin location(s) are specified first, then the next (1-index), etc. All the locations are comma-separated.

The same location can be assigned to different pin objects. This is useful when the physical board can have different functions on the same pin, such as a choice between single-ended or LVDS. This is illustrated below:

```
pin(4) <
  standard      = "LVDS",
  termination   = true,
  location      = ["L12-K11", "H18-H17", "M11-L11", "N12-M12"],
> HSMC_RX;

group <standard = "LVCMOS">{
  pin(4) <location = ["L12", "H18", "M11", "N12"]> HSMC_RX_p;
  pin(4) <location = ["K11", "H17", "L11", "M12"]> HSMC_RX_n;
}
```

F.5.3 Groups

Often there are attributes that apply to many objects. In this case, the definitions can be grouped. All child definitions inherit the attributes of the group. When a child definition includes an attribute that is already defined in the group, the child definition takes precedence. Below is an example of a named group for pin definitions.

```
group<standard = "3.3V", capacitance = 10e-12> SD{
  pin <location = "AB6" > CLK;
  pin <location = "W8" > CMD;
  pin(4)<location = ["U7", "T7", "V8", "T8"]> DAT;
}
```

In this case, the pins that are actually defined are ‘SD.CLK’, ‘SD.CMD’ and ‘SD.DAT’. The group can also be anonymous, as given below. In this case, each pin is a child of the parent group object (global, in this case).

```
group <standard = "3.3V", frequency = 50e6>{
  pin<location = "R20"> CLOCK_B5B;
  pin<location = "N20"> CLOCK_B6A;
}
```

Nets, class instances, derived clocks, etc. can be grouped in similar fashion.

F.5.4 Structures

ALCHA structures are “packed”, which makes it possible to treat the structure as if it is a bit-vector. It is therefore possible, for example, to assign a multi-member structure to the input port of a RAM block and then assign the output of the RAM block to a different instance of the same structure. The user does not need to use concatenation in the process.

Structures are defined by means of the **struct** keyword, as follows:

```
struct double{  
    net    Sign;  
    net(11) Exponent;  
    net(52) Mantissa;  
}
```

These structures follow the same rules as SystemVerilog structures, i.e. they are specified from most-significant to least-significant bit. An instance of a structure can be used as if it is a bit-vector.

F.5.5 Enumerations

An enumeration type can be defined by means of the **enum** keyword, as illustrated below.

```
enum STATE {Idle, Writing, Done, Others}  
STATE State;  
  
switch(State){  
    case(Idl)e{  
        // Do some stuff  
        State = Writing;  
    }  
    case(Writing){  
        // Do some stuff  
        State = Done;  
    }  
    case(Done){  
        // Do some stuff  
        State = Idle;  
    }  
    default{  
        // Do some stuff  
    }  
}
```

F.5.5.1 Base Type and Encoding

By default, the numerical constants associated with the enumeration start at 0 for the first element and increase by 1 for each element. The user can, however, assign arbitrary constants to the enumeration members. Ordinarily, enumerations are equivalent to the non-synthesisable type `num`. The enumeration can, however, be used to define a synthesisable enumeration, as shown below.

```
enum STATE {Idle, Writing, Done, Others}
pin(STATE) PinState; // Pin enumeration
net(STATE) SigState; // Net enumeration
```

In this case, the number of bits, or width, of the net (or pin array) is the number of bits required to uniquely identify each enumeration value. In the above example, this is 2 bits. If the enumeration was declared with the ‘encoding = “one-hot”‘ attribute, the vectors would be 4 bits wide.

F.5.5.2 ALCHA Enumeration Semantics

A variable declared as an enumeration can only take values from that enumeration. If the LHS of an assignment is an enumeration instance, the compiler pushes that enumeration onto the name-space stack. If an enumeration is used outside a variable of that type, the values of the enumeration can be referenced through its type. It is illegal to assign anything else to an enumeration type. It is legal to compare an enumeration to other types, however. The code below shows various examples of legal and illegal statements.

```
enum Enum{A, B, C}

num a;
Enum e, n;

a = A;           // Illegal: A does not exist in this name-space
a = Enum.A;      // Legal  : A represents an integer value
e = B;           // Legal  : assigning a value to the enumeration
n = e;           // Legal  : the enumeration types are the same
e = 2;           // Illegal: must assign a value from the defined list
a = e;           // Legal  : e is automatically cast to an integer
if(e == B){...}  // Legal  : comparing enumeration values
if(a == B){...}  // Illegal: B does not exist in this name-space
if(e == 2){...}  // Legal  : e is automatically cast to an integer
```

F.5.6 Scripting Data Types

The table below summarises ALCHA scripting variable types:

Keyword	Description
bool	Boolean type that takes true or false values only
num	Infinite precision, rational, signed complex numbers (see GNU MP and GNU MPFR)
byte	8-bit unsigned integer typically used for binary file I/O
char	32-bit Unicode character typically used for text file I/O
string	Syntactic sugar for a char array

The user can freely interchange variables of **bool** and **num** types. The value of **true** is 1 and the value of **false** is 0. When assigning a **num** to a **bool**, the equivalent statement is **b = (n != 0)**.

When a floating-point is assigned to a **num** type, the same precision as the floating-point is used to determine the size of the components of the rational number.

More on scripting in section [F.13](#).

F.5.7 Built-in Members

Various ALCHA types have built-in attributes. The list below is not exhaustive.

Member	Types	Description
width	pin , net	The number of bits in the vector. Returns 1 for scalars.
length	All array types	The number of elements in the array.
frequency	Clock nets	The frequency of the clock, in Hz.

F.6 Expressions

Combinational statements in ALCHA are evaluated in order. In terms of HDL, the statements are considered blocking.

Assignments come in two flavours. Normal assignments use the **=** operator and take the fixed-point format into account, whereas raw assignments use the **:=** operator and assigns the raw bits only.

During normal assignment, the source is type-cast to the target type before assignment. If the target does not have sufficient bits to store the source, the source is truncated. When most-significant bits are removed, the compiler issues a warning. When least-significant bits are removed, no warning is issued. It is up to the developer to ensure that correct rounding is applied, if required.

In order to prevent accidental variable creation through typos, all variables, nets, etc. must be defined before use. It is useful, however, to be able to define a variable based on the type of the first expression assigned to it. In that case, define it with `auto` type. Once defined in such a manner, that variable cannot change its type, and redefining it is illegal.

F.6.1 Operators

The following table summarises ALCHA operators, in order of precedence:

Group	(...)	Grouping
Array	[A, B, C]	Build array from elements A, B and C
Dictionary	<K=V, A=B>	Build dictionary from key-value pairs
Concatenate	:(A, B, C)	Bit-wise concatenate the vectors A, B and C
	: [A, B, C]	Concatenate arrays A, B and C
Cast	A @ N	Casts A to an N-bit integer
	A @(N, s)	Casts A to an N-bit fixed-point in the range [0, s)
	A @(N, -s)	Casts A to an (N+1)-bit fixed-point in the range [-s, s)
	A @Class	Casts A to Class, provided Class is in A's inheritance graph
	A @(N.Class)	Casts A to Class, where Class is in namespace N
Post-fix	A++	Increment after use
	A--	Decrement after use
	A!	Factorial
	A.B	Access B in namespace (or class instance) A
	A'B	Access the attribute B of A
	A(B)	Call function A with parameter(s) B
	A[B]	Slice array A with elements in array B
Unary	++A	Increment before use
	--A	Decrement before use
	-A	Negate
	~A	Bitwise NOT
	:A	Cast A to raw bits (unsigned integer)
	'A	Access the global attribute A
Range	A .. B	Builds an array of elements from A to B, inclusive
	A .. B : C	Builds an array of elements from A to B, in steps of C
Reduction	+A	Add-reduce (an array)
	&A	AND-reduce
	~&A	NAND-reduce
	A	OR-reduce
	~ A	NOR-reduce
	^A	XOR-reduce
	~^A	XNOR-reduce
	!A	Logical compliment

Replicate	A 'B	Bit-wise replicate A, B times
Exponential	A ** B	Raise A to the power of B
Arithmetic	A * B	Multiply
	A / B	Divide
	A % B	Modulus
	A + B	Add
	A - B	Subtract
Shift	A << B	Shift A left by B bits
	A >> B	Shift A right by B bits
Relational	A > B	Greater than
	A < B	Less than
	A >= B	Greater than or equal to
	A <= B	Less than or equal to
	A == B	Equal to
	A != B	Not equal to
Bit-wise	A & B	Bit-wise AND
	A ~& B	Bit-wise NAND
	A ^ B	Bit-wise XOR
	A ~^ B	Bit-wise XNOR
	A & B	Bit-wise OR
	A ~ B	Bit-wise OR
Conditional	A ? B : C	If A, then B, else C
	A ?: C	If A, then A, else C
Assignment	A = B	Normal assign (automatically casts to the target fixed-point format)
	A := B	Raw assign (assigns bit-for-bit, ignoring the fixed-point format)
	A ~= B	Append array B to A (equivalent to A = : [A, B] and only applicable to dynamic arrays, which are available during scripting)
	A += B	Add and assign (normal assign)
	A -= B	Subtract and assign (normal assign)
	A *= B	Multiply and assign (normal assign)
	A /= B	Divide and assign (normal assign)
	A %= B	Modulus and assign (normal assign)
	A ^= B	Exclusive OR and assign (raw assign)
	A &= B	Bit-wise AND and assign (raw assign)
	A = B	Bit-wise OR and assign (raw assign)
	A ^= B	Bit-wise XOR and assign (raw assign)
	A <<= B	Shift left and assign (normal assign)
	A >>= B	Shift right and assign (normal assign)

The unary, arithmetic, shift and relational operators take the fixed-point format of the operands into account, whereas the other operators do not. Arithmetic operators result in a fixed-point number which is of such a format that no bits are lost in the calculation. Other operators result in an unsigned integer, regardless of input format.

The target of an assignment can be an array, array slice or concatenation.

The usual flow-control structures (`if`, `for`, `while`, etc.) are supported. More detail is provided later.

F.6.2 Global Attribute Definition and Access

Defining an attribute puts it in the current namespace. Accessing a namespace runs the hierarchy from the current namespace toward the root of the namespace tree, which is the global attribute. The current namespace attributes therefore shadow the global attributes. In addition, global attributes are read-only from within a namespace.

F.6.3 Net use Before Assignment

When describing FPGA firmware it is often necessary to use a net before a value is assigned. This could occur when using parametrised classes, or clocked structures with external combinational feedback. The code below provides two examples:

```
class Adder(N){ // Parametrisation is presented in the scripting section
    net(N) A, B;
    net(N) Y;
    Y = A + B;
}
Adder(8) MyAdder; // Creates an instance of an 8-bit adder and assigns the
                  // object to the MyAdder reference. At this point, "Y" is
                  // a reference to the expression "Y = A + B", but "A" and "B"
                  // are null references.

net(8) x, y, z;
MyAdder.{ // Pushes the name-space "MyAdder" onto the name-space stack
    A = x; // Assign the expression "x" to the reference "A"
    B = y; // Assign the expression "y" to the reference "B"
    z = Y; // Assign the expression "Y" to the reference "z"
}
```

The Verilog equivalent of the above is:

```
module Adder #(parameter N)(
    input  [N-1:0]A,
    input  [N-1:0]B,
    output [N-1:0]Y
);
    assign Y = A + B;
endmodule

module TopLevel;
    wire [7:0]x;
    wire [7:0]y;

    Adder #(8) MyAdder(x, y, z);
endmodule
```

And here is another example:

```
net(8) Counter;
UserClk = (Counter == 125); // "Counter" has no value yet
rtl(CPUClock){
    if(UserClk) Counter = 0;
    else      Counter++;
}
```

ALCHA is intended for describing hardware, so the defined **net** types represent physical hardware nets. When using that net in an assignment, the ALCHA user is making connections, not making a copy from one RAM location to another. This paradigm allows for use-before-assign semantics.

ALCHA does, however, allow circular assignments. These are performed by using the following algorithm:

1. Evaluate the right-hand-side (RHS) expression to produce an expression tree
2. Inspect the left-hand-side (LHS) and the expression tree to determine whether or not the statement is circular
3. If it is circular, update the expression tree to use the current expression assigned to the LHS net, instead of the net itself
4. Assign the expression tree to the LHS net
5. Perform garbage collection on any resulting dangling circuits

F.7 Statements

F.7.1 Control Flow Structures

F.7.1.1 if

If statement syntax is presented below:

```
if(A){  
    // if A is non-zero, do this  
}else{  
    // otherwise do this  
}
```

If the condition is purely scripting (i.e. it does not contain any synthesisable variables), the if-statement is equivalent to conditional compilation (the `#if ... #else ... #endif` construct of the C preprocessor).

F.7.1.2 switch

Switch-statement syntax is presented below:

```
switch(A){  
    case(B, C, D){  
        // if A is equal to B, C or D, do this  
    }  
    case(E){  
        // if A is equal to E, do this  
    }  
    case(1..8){  
        // If A is in the range 1 to 8, do this  
    }  
    default{  
        // otherwise do this  
    }  
}
```

The expressions in the `case` statements (B, C, D and E in the above example) must evaluate to unique values. During the scripting stage of compilation, these can be dynamic (because the values are known during compile-time). During the synthesis stage, the expressions must evaluate to constants or ranges.

ALCHA does not infer latches. Modern FPGAs do not have latches, so it makes little sense to implement a language structure that does. Within combinational circuits, ALCHA assigns “don’t care” to any net not assigned a value in all possible evaluations of `if`- or `switch`-statements. Within clocked structures, the net keeps its previous value unless explicitly assigned.

F.7.1.3 while

The while-loop syntax is presented below. The body of the loop is evaluated for as long as `A` is not zero. For combinational circuits and pure RTL, the value of `A` must be known at compile-time. Within a finite state machine (FSM) structure, `A` can be a synthesisable expression.

```
while(A){  
    // while A is non-zero, do this  
}
```

F.7.1.4 for

The for-loop syntax is presented below. `A` must evaluate to an array. For every element of the array, `x` becomes a reference to that element and the loop is evaluated.

```
for(x in A){  
    // for every element of array A, set x to that element and do this  
}
```

The loop is evaluated in the same order as the elements of the array, not in parallel. When the elements of `A` is a synthesisable type, and the loop occurs during a combinational statement, the result could potentially be a parallel combinational circuit. During compilation, however, the loop is still evaluated sequentially, following normal combinational-statement blocking assignment rules. This is illustrated below:

```
net(8) A, B;  
  
// Counts the number of bits of A that are high and assigns the answer to B  
B = 0;  
for(x in 0..7) B += A[x];
```

F.7.1.5 loop

The loop-loop syntax is presented below:

```
loop(N){
    // Do this N times
}
loop{
    // Do this forever (or until reset)
}
```

The evaluation of loops in FSM structures depend on the use of commas or semicolons. If the statement within the loop body ends in a comma, the loop is effectively unrolled into the state in which it is declared. If, on the other hand, the statement within the loop body ends in a semicolon (thereby telling the compiler to go to the next state), the loop is evaluated one clock-cycle at a time.

A loop loop can therefore be used to implement a “wait” state, as follows:

```
net LED = 0;

fsm(Clk, Reset){
    loop{ // Main loop (do this forever)
        LED = ~LED, // Toggle the LED and
        loop(20_000); // Go to the next state 20 000 times
    }
}
```

This is equivalent to the following Verilog:

```
reg LED;
reg [14:0]Count;

always @(posedge Clk) begin
    if(Reset) begin
        LED <= 0;
        Count <= 15'd_20_000;

    end else if(Count == 15'd_20_000) begin
        LED <= ~LED;
        Count <= 15'd1;

    end else begin
        Count <= Count + 1'b1;
    end
end
```


F.7.1.6 return

The **return** statement returns from a function. If the function was called combinationally, the return statement simply ends evaluation of the rest of the function body (thereby not synthesising the rest of the circuit). If the return statement occurs within a clocked structure, the clocked structure goes into an idle state, either waiting for the next call (if it was called from a state machine) or halting until the next system reset (if it was called combinationally).

```
num Func(A, B){
    return A+B; // Equivalent to "Func = A+B; return;"
}
void Proc(A, B){
    // Do some stuff
    if(whatever) return; // return early
    // do some more stuff
}
```

F.7.1.7 break and continue

The **break** and **continue** keywords are used to jump out of, or within, loops. Both of these take an optional integer expression argument to indicate the number of loop levels. For instance:

```
loop{
    for(j in 1..100){
        while(Busy){
            if(Timeout) break 3; // Continues at "Error code"
            continue;           // skips the rest of this loop body
                                // and does the next iteration of this loop

            continue 2; // skips the rest of the loop body of this and the "for"
                        // loop and does the next iteration of the "for" loop

            break;        // Breaks out of the "while" loop and does the rest of
                        // the "for" loop body;
        }
    }
}
// Error code
```

F.7.2 Alias

It is often the case where the same expression is used many times, yet cannot be simply assigned to a variable. In this case, it is convenient to define an alias for the expression, as

follows:

```
alias C    = Namespace.ClassInstance;
alias M    = SomeClass.Member;
alias Clk  = SYS_GLOBAL_CLK;

C.Member = 3;
M = 5;
fsm(Clk){
    // Some statements...
}
```

It is important to note, however, that the alias is evaluated first, before used in the target expression. The following two lines are equivalent:

```
alias sum = A + B; X = Y * sum;
X = Y * (A + B);
```

F.8 Arrays

ALCHA supports fixed-length arrays for synthesisable types, and variable-length arrays for scripting types. Multiple-dimensional arrays can be defined in terms of “arrays of arrays”.

F.8.1 Defining Arrays

Each new dimension is defined by means of a positive integer in square brackets, representing the number of elements in that dimension. An example is presented below:

```
net(8) A[16];      // A 16-element array of 8-bit unsigned integers
net(8) B[3][4][5]; // A 3-dimensional array of 8-bit unsigned integers
```

Arrays containing scripting variables or object references can be defined as dynamic by omitting the size specifier:

```
num    A[]; // A dynamic array of numbers
MyClass C[]; // A dynamic array of MyClass references
```

Furthermore, arrays defined as attributes have both dynamic size and arbitrary element types:

```
'A = [ MyClassInstance, 3, "A String" ];
```

F.8.2 Slices

Array elements can be addressed by means of slices. The bits are modelled as the deepest dimension. The element indices are zero-based, i.e. index 0 is the least significant bit of a bit-vector, or the first element in an array. An empty slice (“`[]`”) can be used to indicate “all elements”.

Any bit-slice removes the fixed-point format. The result of a bit-slice is always an unsigned integer with as many bits as in the slice. Some examples are presented below:

```
net(16, 8) A[16];
A[5]           // Element 5 of A (the 6th element).
A[] [15..13]    // All the elements of A, but only the integer bits
A[3,7,2][12..0] // Elements 3, 7 and 2 of A (in that order), and only the fraction bits
A[0..15:3]      // Every third element of A (0, 3, 6, 9, 12 and 15)

A[5,3,7][6,7,2] // Bits 6, 7 and 2 (in that order) of elements 5, 3 and 7 (in that order)
                // The result is a 3-element array of 3-bit integers
```

The indices can also be specified by means of a number array, as follows:

```
net(8) A[16];
num    B[5] = [4, 2, 7, 1, 12];
A[B]    // Elements 4, 2, 7, 1 and 12 of A, in that order
A[B, 0..15] // Elements 4, 2, 7, 1, 12 and 0 to 15 of A
```

F.8.3 Array Literals

Array literals are useful for vectorised operations. There are many forms, which are illustrated in the example below:

```
[0, 1, 2, 3]    // 4-element array
[0..3]          // Same as above
[0..4:2]        // The elements 0, 2 and 4
[3..0]          // The elements 3, 2, 1 and 0
[0..3, 4..0:-2] // The elements 0, 1, 2, 3, 4, 2 and 0
```

F.8.4 Concatenation

Arrays can be concatenated, resulting in a larger array:

```
num A[] = [0, 1, 2, 3]; // 4-element array
num B[] = [4..7];      // Another 4-element array
num C[] = [A, B];      // 2-element array of 4-element arrays
num D[] = :[A, B];     // Equivalent to [0..7]
```

It is often useful to add one element to the tail of the array. Array objects have the built-in `append` function for this purpose:

```
num A[] = [0..3];
A.append(4);
```

F.8.5 Vectorised Statements

Array slices can be used in vectorised statements. Vectorised operations apply the operation to each element of the array as if it was specified separately for each element. This works for scalar functions as well. Below are some examples:

```
auto Add(A, B){
    return A + B;
}

net(8) A[16], B[16], Y[16];
Y = A + B; // Perform an element-wise addition
Y = Add(A, B); // Equivalent to the addition above

Y[0..15] = A[15..0] + B[0..15:2, 1..15:2];
// Equivalent to:
Y[ 0] = A[15] + B[ 0];
Y[ 1] = A[14] + B[ 2];
Y[ 2] = A[13] + B[ 4];
...
Y[13] = A[ 2] + B[11];
Y[14] = A[ 1] + B[13];
Y[15] = A[ 0] + B[15];
```

When an array is used as the condition of an `if` statement or `while` loop, the array is AND-reduced. This allows a concise means by which to compare all elements in an array (or string). Any other intention must be written explicitly in terms of a loop structure.

When a scalar is involved in a vectorised operation, the scalar is repeated for every instance of the operation. This can be used, for instance, to assign a scalar to every element of an array.

Net and numeric arrays can also be add-reduced, which returns the sum of the elements.

F.9 Functions

ALCHA provides a procedural programming model for combinational circuits, state machines and scripting. There are various categories of functions, resulting in different calling conventions.

The `result` keyword is used as a reference to the target net (or variable) for the return value. This has implications for functions that contain clocked structures, as the state machine (or RTL) has control over the value of the target net, even though the function has not returned yet.

The parameter types do not need to be defined in the function, but can be inferred from the input parameters when the function is called (this is also known as duck-typing). When the types are specified, the parameter types form part of the function name, so that many functions of the same name, but different parameter types, can be defined.

Parameters may also carry direction specifiers (i.e. `input` or `output`) in order to constrain their use. Input parameters cannot be written to by the function body, and output parameters overwrite the expressions assigned to the objects used during the function call. If not specified, the direction is inferred from use.

F.9.1 Combinational Functions

Functions in ALCHA are defined in similar fashion as C, as illustrated in the example below.

```
net(8) Add(net(8) A, net(8) B){  
    result = A + B;  
}  
net(8) x, y, z;  
z = Add(x, y);
```

Functions also support default parameters and named parameter assignment, as illustrated below:

```
auto MyFunction(A, B, C = 7, D = 2){
    // Function body
}

net(8) X = MyFunction(B = 2, A = 8, D = 4); // C defaults to 7
```

Combinational functions can be called from anywhere, including the bodies of clocked structures. Functions can also be called in vectorised form, which is especially useful when combined with array slicing. The adder above can be called as follows:

```
net(8) x[4], y[4], z[4];
z = Add(x, y);

// This is equivalent to:
z[0] = Add(x[0], y[0]);
z[1] = Add(x[1], y[1]);
z[2] = Add(x[2], y[2]);
z[3] = Add(x[3], y[3]);
```

Arrays and scalars can be mixed in the call, as follows:

```
net(8) x, y[4], z[4];
z = Add(x, y);

// This is equivalent to:
z[0] = Add(x, y[0]);
z[1] = Add(x, y[1]);
z[2] = Add(x, y[2]);
z[3] = Add(x, y[3]);
```

If the function takes an array as a parameter, the same rules apply. The function can be called with a higher-dimension array, as follows:

```
net(18) Dot(A[], B[]){
    result = A[1] * B[1];
    result += A[2] * B[2];
    result += A[3] * B[3];
    result += A[4] * B[4];
}

net(18) Y[16];
net( 8) A[4][16], B[4][16];
Y = Dot(A, B);
```

To return an array, the function name should be defined as an array, as follows:

```
net(16, -8) Mult[4][4](A[][], B[][]){
    int i, j, k;
    for(i in 0..3){
        for(j in 0..3){
            result[i][j] = 0;
            for(k in 0..3){
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```

The loops above are discussed in greater detail in the Scripting section. Such combinational loops should be used with care, as very little code can result in very large circuits.

F.9.2 Clocked Functions

Clocked functions contain optional combinational code, as well as an embedded state machine or RTL. The embedded clocked structure's clock and reset can, potentially, be different to that of the calling clocked structure. These could either be global signals, or passed through a function parameter.

The function body is implemented as few times as possible, typically in a sub-module. If many different states call the same function, the function body is implemented only once. It is possible to call the same function many times from the same state, in which case the function is implemented multiple times.

The call is controlled through hand-shaking signals. The resulting function module has implicit **Go** (input) and **Busy** (output) signals. These are handled by the compiler and hidden from the developer. When multiple different functions are called from the same state, the state machine must wait for all the functions to finish (i.e. make their **Busy** lines low) before continuing to the next state.

The combinational code of the function body, as well as the function reset code, is evaluated during the calling state. The **Go** line is also pulled high during that state.

The handshaking signals are handled differently when the function and calling structure reside within the same clock-domain, or when there is a clock-domain crossing required. When there is a clock-domain crossing, the calling state machine waits for the **Busy** to go high before releasing the **Go** signal. The function waits for the **Go** to go low before releasing the **Busy** signal. These hand-shaking signals are properly clock-domain crossed when required.

When there are no clock-domain crossing required, the calling state machine pulses the **Go** signal for a single cycle instead of following the full hand-shaking algorithm described above. This is a timing optimisation.

The **Busy** signal further acts as a clock-enable signal of the function body. The output signals of the function keep their state until the next call.

F.9.3 RTL Functions

RTL structures do not have the concept of an instruction sequence. In order to return from a function that contains an RTL structure, a **return** statement must exist somewhere in the RTL body, which releases the **Busy** signal and halts execution of the RTL structure.

Furthermore, RTL structures do not support the handshaking algorithm described above. The developer must implement them manually.

When calling a function from an RTL structure, the same rules apply as for calling them from combinational logic.

F.9.4 Example

In order to illustrate the use of functions within state machines, here follows an example of a simple SPI interface in ALCHA, as well as its equivalent Verilog:

```

pin<frequency = 50e6> Clk;

pin Reset;

group SPI{
    pin Clk, Data, Latch;
}
//-----

void SPI_Send(net(32) Data){
    net( 5) bit;
    net(32) Shift;

    SPI.Clk  = 0;
    SPI.Latch = 0;
    SPI.Data  = Shift[31];

    fsm(Clk, Reset){
        SPI.Latch = 0,
        Shift      = Data;

        for(bit in 31..0){
            SPI.Clk = 1;

            SPI.Clk = 0,
            Shift <<= 1,
            if(!bit) SPI.Latch = 1,
            ;
        }
        SPI.Latch = 0;
    }
}
//-----

// Initialise some SPI-based peripheral
fsm(Clk, Reset){
    SPI_Send(0x12345678);
    SPI_Send(0xFEDCBA98);
    SPI_Send(18271662);
    SPI_Send(0b11010011:0x123456);
}
//-----

```

Which is equivalent to:

```
module SPI_Send(  
    input      Clk,  
    input      Reset,  
  
    input      Go,  
    output reg Busy,  
  
    input [31:0]Data,  
  
    output reg SPI_Clk,  
    output      SPI_Data,  
    output reg SPI_Latch  
);  
//-----  
  
reg [ 4:0]bit;  
reg [31:0]Shift;  
  
reg      tReset;  
reg [1:0]State;  
localparam GetData = 2'd0;  
localparam Sending = 2'd1;  
localparam Done     = 2'd2;  
//-----  
  
assign SPI_Data = Shift[31];  
//-----
```

```

always @(posedge Clk) begin
    tReset <= Reset;

    if(tReset) begin
        SPI_Clk    <= 0;
        SPI_Latch <= 0;
        Busy       <= 0;
        State      <= GetData;
    //-----

    end else if(Busy) begin
        case(State)
            GetData: begin
                SPI_Latch <= 0;
                Shift     <= Data;
                bit        <= 5'd31;
                State      <= Sending
            end
        //-----

        Sending: begin
            if(~SPI_Clk) begin
                SPI_Clk <= 1'b1;

                end else begin // SPI_Clk is high
                    SPI_Clk <= 0;
                    Shift    <= {Shift[30:0], 1'b0};
                    if(~|bit) begin
                        SPI_Latch <= 1'b1;
                        State      <= Done;
                    end
                    bit <= bit - 1'b1;
                end
            end
        //-----

        Done: begin
            SPI_Latch <= 0;
            Busy       <= 0;
        end
        //-----

        default::;
    endcase
    State <= State + 1'b1;
    //-----

    end else if(Go) begin
        SPI_Clk    <= 0;
        SPI_Latch <= 0;
        Busy       <= 1'b1;
        State      <= GetData;
    end
end

endmodule

```

```

module TopLevel(
    input Clk,
    input Reset,

    output SPI_Clk,
    output SPI_Data,
    output SPI_Latch
);
//-----

    reg      SPI_Send_Go;
    wire     SPI_Send_Busy;
    reg [31:0]Data;

    SPI_Send SPI_Send_Instance(
        .Clk      (Clk),
        .Reset    (Reset),

        .Go       (SPI_Send_Go),
        .Busy     (SPI_Send_Busy),

        .Data     (Data),

        .SPI_Clk  (SPI_Clk),
        .SPI_Data (SPI_Data),
        .SPI_Latch(SPI_Latch)
    );
//-----

    reg      tReset;
    reg [2:0]State;
    localparam Send1 = 3'd0;
    localparam Send2 = 3'd1;
    localparam Send3 = 3'd2;
    localparam Send4 = 3'd3;
    localparam Done  = 3'd4;
//-----

```

```

always @(posedge Clk) begin
    tReset <= Reset;

    if(tReset) begin
        SPI_Send_Go <= 0;
        State      <= Send1;
    //-----

    end else begin
        case(State)
            Send1: begin
                Data      <= 32'h12345678;
                SPI_Send_Go <= 1'b1;
                State      <= Send2;
            end
        //-----

            Send2: begin
                if( SPI_Send_Go ) SPI_Send_Go <= 0;
                else if(!SPI_Send_Busy) begin
                    Data      <= 32'hFEDCBA98;
                    SPI_Send_Go <= 1'b1;
                    State      <= Send3;
                end
            end
        //-----

            Send3: begin
                if( SPI_Send_Go ) SPI_Send_Go <= 0;
                else if(!SPI_Send_Busy) begin
                    Data      <= 32'd18271662;
                    SPI_Send_Go <= 1'b1;
                    State      <= Send4;
                end
            end
        //-----

            Send4: begin
                if( SPI_Send_Go ) SPI_Send_Go <= 0;
                else if(!SPI_Send_Busy) begin
                    Data      <= {8'b11010011, 24'h123456};
                    SPI_Send_Go <= 1'b1;
                    State      <= Done;
                end
            end
        //-----

            default: SPI_Send_Go <= 0;
        endcase
    end
end

endmodule
//-----

```

The hand-shaking has been simplified because the two clocked structures are in the same clock domain. This Verilog code has been hand-written in order to show the details of the ALCHA calling convention. Compiler-generated code might look very different, but is equivalent in functionality.

F.9.5 Calling Clocked Functions from Combinational Circuits

It is possible to call a clocked function from a combinational circuit. In that case, the function body is evaluated as if it occurred in place of the function call, with the signal being assigned to appearing in place of the function name.

F.10 Synchronous Circuits

F.10.1 Clock Networks

F.10.1.1 Definition

A clock in ALCHA is any synthesisable node with a **frequency** attribute. In order to generate a clock by means of a PLL, use the **hdl** construct to instantiate the appropriate mega-function. Some examples of clock definitions are presented below:

```
// An input pin
pin<frequency = 100e6> ClkPin;

// An RTL circuit
net<frequency = 50e6> Clk_50;
rtl(ClkPin) Clk_50 = ~Clk_50;

// A combinational circuit
net<frequency = 100e6> Clk_LowPower = ClkPin & (~Sleep);

// A phase-locked loop megafunction
hdl() altera_pll(
    reference_clock_frequency = "100 MHz";

    number_of_clocks          = 1;
    operation_mode             = "direct";

    output_clock_frequency0    = "250 MHz";
    duty_cycle0                = 50;
    phase_shift0               = "0 ps";

    pll_type                   = "General";
    pll_subtype                 = "General";
    fractional_vco_multiplier = "false";
){
    net refclk = ClkPin;
    net rst    = GlobalReset;
    net locked;
    net fbclk  = 0;
    net fboutclk;

    net<frequency = 250e6> outclk;
}
altera_pll() PLL_250; // Use "PLL_250.outclk" as the clock
```

ALCHA further supports clock-enable type clocks, which are generally used for lower frequency circuits that must be synchronised to a higher frequency clock domain. The clock enable net must also carry the `frequency` attribute, so that the compiler knows how to set up the associated multi-cycle timing requirements.

```
net(14) ClkCounter;

net<frequency = 100e6> Clk;
net<frequency = 9600 > ClkEnable = ~|ClkCounter;

rtl(Clk){
    if(ClkCounter) ClkCounter--;
    else           ClkCounter = (Clk'frequency / ClkEnable'frequency)- 1;
}
rtl(Clk, Reset, ClkEnable){
    // The RS-232 controller code goes here.
}
```

F.10.1.2 Global Clock Networks

Most modern FPGAs contain dedicated high-fanout, low-skew clock tree networks. ALCHA depends on the downstream vendor compiler to automatically assign clocks to the available clock networks. This automated global clock network assignment can be bypassed by explicitly specifying the ‘`global = true`’ attribute, which applies the associated design constraint.

F.10.1.3 Built-in Properties

All clocks (including clock pins) have a `frequency` member (of type `num`) that can be used by scripts to calculate clock-frequency dependent parameters. A button denouncer might, for instance, be defined as follows:

```
net Debouncer(net Clk, net Input){
    num Count = Clk'frequency * 20e-3; // 20 ms dead-time
    num N      = ceil(log2(Count));

    net(N) Counter = 0;

    rtl(Clk){
        if(~|Counter){
            if(Debouncer != Input){
                Debouncer = Input;
                Counter    = Count - 1;
            }
        }else{
            Counter--;
        }
    }
}
```

F.10.2 Register Transfer Level

Most HDL designers are familiar with RTL design. In ALCHA, the `rtl` construct can be used to describe RTL logic. It takes one parameter: the clock, which must be a synthesisable type with a frequency attribute. All statements within an `rtl` construct are non-blocking. Below is an example:

```
pin<frequency = 50e6> Clk;
pin                    Reset;

net A, B;

net(27) Count = 0; // Initialiser => reset value

rtl(Clk, Reset){
    Count++;

    if(&Count){
        A = B;
        B = A;
    }
}
```

The equivalent Verlog of the above ALCHA is:

```

reg      tReset;
reg      A, B;
reg [26:0]Count;

always @(posedge Clk) begin
    tReset <= Reset;

    if(tReset) Count <= 0;
    else      Count <= Count + 1'b1;

    // No initialiser on A or B, so not part of the reset circuit
    if(&Count) begin
        A <= B;
        B <= A;
    end
end
end

```

F.10.3 Finite State Machines

ALCHA provides a concise FSM description syntax by means of the `fsm` construct. It takes two parameters: the clock and the reset. The reset is optional: if a reset signal is not specified, an auto-generated power-on reset signal is used.

Each statement within an `fsm` construct that ends in a comma (,), is considered to be in the same state, or cycle, as the next statement. A semicolon (;) ends a cycle (or state). All statements are considered non-blocking. A simple example is provided below:

```

pin<frequency = 50e6> Clk;
pin                      Reset;

net(8) A, B, C;

B = 123;

fsm(Clk, Reset){
    loop{
        A = B + C, B = C - A;

        if(A < B) C++,
        else      C--,
        ;
    }
}

```

This state machine has two states and translates to the following Verilog:

```

reg[7:0] A, B, C;

reg tReset;
reg State;

always @(posedge Clk) begin
    tReset <= Reset

    if(tReset) begin
        B    <= 8'd123;
        State <= 0;

    end else begin
        case(State)
            1'b0: begin
                A    <= B + C;
                B    <= C - A;
                State <= 1'b1;
            end

            1'b1: begin
                if(A < B) C <= C + 1'b1;
                else    C <= C - 1'b1;
                State <= 1'b0;
            end

            default::;
        endcase
    end
end

```

All reset signals are localised to the state machine. All signals that enter the **fsm** construct with a value (that must evaluate to a run-time constant) are reset to that value.

The ALCHA **if** and **while** statements follow the same syntax as in C. The state (or cycle) boundaries are controlled by means of semicolons (;).

The **for** loop is used to iterate through elements in an array, as follows:

```

net( 8) x;
net(10) A;

fsm(Clk){
    for(x in 0..200){
        A = 3 * x + 7;
    }
}

```

This translates to:

```
reg [7:0]x;
reg [9:0]A;
reg      State;

always @(posedge Clk) begin
    tReset <= Reset;

    if(tReset) begin
        x      <= 0;
        State <= 0;

    end else begin
        case(State)
            1'b0: begin
                A <= 2'd3 * x + 3'd7;
                x <= x + 1'b1;
                if(x == 8'd200) State <= 1'b1;
            end

            default::;
        endcase
    end
end
```

The `loop` loop is used to repeat something. Without a parameter, a `loop` loop will run forever (or until a `break` statement is encountered). With a parameter, the ‘`loop(N)`’ loop will run for N iterations.

F.11 Classes

ALCHA supports a simple class structure that has a single constructor form.

F.11.1 Constructor

The body of the class is the constructor. Any statements within the body of the class are evaluated within the name-space of the class. Any declarations within the class body declare class members, be it nets, scripting variables or functions.

F.11.2 Parameters

Class parameters are essentially syntactic sugar for members that must be assigned during class instantiation. Class parameters, just like function parameters, are assigned by reference, and are therefore bidirectional. Direction is inferred from use. The following forms are equivalent:

```
net x, y, z;

class Class1(net A, net B, net C){
    !! Class body
}
Class1(x, y, z) C1;

class Class2{
    net A, B, C;
    !! Rest of class body
}
Class2() C2;
C2.A = x;
y = C2.B;
C2.C = z;
```

F.11.3 Member Functions

ALCHA classes can have member functions. These behave the same as any other ALCHA function, except that they have direct access to the class members (instead of accessing members through a class instance).

If the function is defined without a body, that function is pure virtual, thereby turning the parent class into an abstract class. The inheriting class is expected to implement the function.

F.11.4 Finally Member Functions

The `finally` member function is called automatically at the end of the code-base, in order of the class instance declarations. This is useful when the class is used to auto-generate a piece of code that is dependent of future module instances. The actual generation can be triggered once the compiler reaches the end of the code-base.

F.11.5 Reference to Self

ALCHA makes use of the `this` keyword in order to refer to the current instance. It has the same semantics as in C++. This keyword is optional when referencing members from within the class body.

F.11.6 Access Specifiers

Any class member can be restricted with an access specifier block. The meaning of the concepts are the same as in C++.

```
private  num N = 3; // A private member
protected net Clock; // A protected member
public   net Reset; // A public member

private{ // Private region
    num Width;
    num Depth
}

// Anything unspecified is public
void Delay(num Seconds){
    !! Function body
}
```

Direction of ports is generally inferred from use. If a member is only ever read in the class, the direction is inferred as `input`. If, on the other hand, the class body writes to the member, it is inferred as `output`. The user can override this behaviour (and enforce the resulting rule) by explicitly using the `input` or `output` keywords:

```
class MyClass{
    input{ // Input region
        net Clock;
        net Reset;
    }
    output net(32) Count = 0; // Output port

    rtl(Clock, Reset) Count++;
}
```

A member that has been defined as `output` is read-only (i.e. no external statement can modify it), and a member that has been defined as `input` is write-only (i.e. no internal class statement can modify it).

F.11.7 Instantiation

A variable defined with a class type is actually a reference to that class. Calling the class, as if it is a function, executes the constructor and returns a reference to a new instance of that class. The class instance is automatically destroyed when no more references to it exists (similar to Python). This mechanism enables expressions such as:

```
class vec4(x, y, z, w){  
  num dot(vec4 V){  
    dot = V.x*x + V.y*y + V.z*z + V.w*w;  
  }  
};  
vec4 A = vec4(1, 2, 3, 4);  
num B = A.dot(vec4(9, 8, 7, 6));
```

Using this syntax is often tedious and verbose, so a shortened version is available. The following two statements are equivalent:

```
vec4 A = vec4(1, 2, 3, 4);  
vec4(1, 2, 3, 4) A;
```

F.11.8 Abstract Signal Types

When calling a function, the **result** keyword becomes a reference to the target variable, and the parameters are references to the input parameters. This can be used to define abstractions of streaming processors, for instance. An example of how a digital signal processing chain might be implemented is presented below:

```
class STREAM(num Width = 8){
    net(Width) Data;
    net      Valid;
    net      Ready;
}

STREAM Filter(STREAM Input){
    // Creates a new stream object which has double the width of the input.
    Filter = STREAM(Input.Data'width * 2);

    Input.Ready = Filter.Ready; // Implements back pressure

    rtl(Clk){ // Uses global clock
        if(Filter.Ready & Input.Valid){
            // Implement the body of the filter here, controlling the
            // output data and valid fields as required.
        }
    }
}

STREAM Decimate(STREAM Input, num N = 128){
    // Similarly defined as the filter above
}

STREAM FFT(STREAM Input){
    // Implements a streaming FFT
}

// Hook up the chain:
ADC() Input;
STREAM Output = FFT(Decimate(Filter(Input)));
```

F.11.9 Inheritance

ALCHA classes support multiple inheritance. The parent classes are specified by means of a comma-separated list after a colon. The ADC class used in the example above might, for instance, be defined as follows:


```

group<standard = "CMOS", voltage = "2.5 V"> ADC{
  group Data{
    pin    <location = "B7"> Clock = 0; // From FPGA to ADC
    pin(14)<location = ["A8", "C9", "E17", "D11", "G3", "C14", "B16",
                      "B4", "H2", "A25", "E19", "F7", "G21", "A11"]> Data;
  }
  group SPI{
    pin<location = "K7"> Clock = 1; // These initialisers are reset conditions
    pin<location = "L2"> Data  = 1;
    pin<location = "N9"> Latch = 1;
  }
}

class ADC(): STREAM(14){ // Inherits from the STREAM class
  void SetRegister(Address, Data){
    fsm(GlobalClock, GlobalReset){
      // A state-machine that implements the ADC's SPI interface.

      // This state-machine can be called externally to the ADC class as
      // well as from within.
    }
  }

  fsm(GlobalClock, GlobalReset){
    // The state machine used to set up the ADC and receive data.

    // This state machine calls the above SetRegister function in order to
    // set the various ADC registers to default values.
  }
}

```

F.11.10 Polymorphism

References to class instances must be of a type that exists in that instance's inheritance tree. Any other type-cast will result in a compiler error. The root of the inheritance tree is defined as the type of the class instance. The tree branches towards the parent classes, with the original ancestors on the leaves of the tree.

All ALCHA member functions are virtual. In order to call the function of a specific class in the inheritance tree, the class must be referenced by name. The dot operator (".") is used for this purpose.

If there are more than one candidate function, the compiler issues an error. The user must cast the class to make the call unambiguous. The tree is traversed towards the root in order to find another candidate, but not back towards the other leaves.

F.11.11 Attributes

Whenever a class is instantiated, the instance can be assigned various attributes (in similar fashion to pins and nets). A typical attribute that might be assigned is the `location`, which indicates where on the FPGA the class instance should be placed. The exact details of this has not been finalised yet, but it is envisioned that the developer can define a rectangle in normalised coordinates, similar to the example below:

```
class SomeModule{
    // Some members
}
SomeModule()<region = "X44, Y145, W40, H5"> TheInstance;
```

F.12 Operator Overloading

Operator overloading is implemented similar to C++, but with a distinction between `operator` and `unary` operators:

```
class C{...}

C operator+ (C a, C b){
    !! Function body to return the sum of a and b
}

net(32) unary& (C a){
    !! Function body to return the &-reduction of class C
}
```

F.13 Scripting Features

ALCHA provides a powerful scripting platform to control the generation of firmware. Any scripting variable (or scripting expression) can be used where synthesisable code expects a constant, thereby providing powerful parametrisation features. Scripting can further be used for conditional compilation based on various parameters.

F.13.1 Algorithmic Code Generation

Like [Migen](#), ALCHA supports algorithmic code generation. During compilation, the abstract syntax tree is evaluated in a single pass. Scripting statements are evaluated on-the-fly, and the netlist is produced directly.

The user can, for example, use a scripting **for** loop to generate a **switch**-statement based look-up table, as illustrated in the code below. The scripting interpreter will duplicate the part of the AST containing the **case** assignment, updating the constants during each iteration. The same feature can be used to implement conditional compilation.

```
net( 5) x;  
net(32) y;  
num    n;  
  
switch(x){  
  for(n in 0..31){  
    case(n) y = (2**32-1) >> (31-n);  
  }  
}
```

F.13.2 Mathematical Functions

ALCHA scripting has the following built-in mathematical functions:

Function	Description
<code>sin</code>	Sine (radians)
<code>cos</code>	Cosine (radians)
<code>tan</code>	Tangent (radians)
<code>asin</code>	Arc-sine
<code>acos</code>	Arc-cosine
<code>atan</code>	Arc-tangent
<code>sinh</code>	Hyperbolic sine
<code>cosh</code>	Hyperbolic cosine
<code>tanh</code>	Hyperbolic tangent
<code>asinh</code>	Hyperbolic arc-tangent
<code>acosh</code>	Hyperbolic arc-cosine
<code>atanh</code>	Hyperbolic arc-tangent
<code>log</code>	Natural logarithm
<code>log2</code>	Logarithm to base 2
<code>log10</code>	Logarithm to base 10
<code>exp</code>	Natural exponent
<code>pow</code>	Exponent (takes 2 parameters)
<code>round</code>	Round to nearest integer
<code>fix</code>	Drops the part after the decimal point
<code>floor</code>	Round to negative infinity
<code>ceil</code>	Round to positive infinity
<code>rand</code>	Random number between 0 and 1, inclusive
<code>real</code>	The real part of a complex number
<code>imag</code>	The imaginary part of a complex number
<code>abs</code>	The magnitude of a complex number
<code>angle</code>	The phase of a complex number, in radians
<code>fft</code>	Returns the fast Fourier transform of an array of <code>num</code>

Fast Fourier transforms (FFTs) are implemented by means of the [FFTW library](#). The implication is that all FFTs are double-precision, not the arbitrary precision available in the `num` type in general.

F.13.3 I/O Functions

ALCHA supports two types of file access: binary and text-based. Binary files make use of the `byte` base type, whereas text files make use of the `char` base type. A `byte` is an 8-bit one-to-one mapping to the physical file bytes. A `char` is a UTF-32 representation of a UTF-8 encoded file. The code below shows an example of how file I/O is used in ALCHA.

```
byte Data[] = read("MyDataFile.dat");
char Log [] = ""; // An array of UTF-32 characters

byte data;
for(data in Data){
    !! Do some stuff, possibly appending to the Log array
}
textwrite("MyLogFile.log", Log); // Writes the log to file, in UTF-8 format.
```

The `read` function acts as if it reads the entire file into the array provided. The actual implementation, however, might involve caching strategies implemented by the scripting interpreter. The `write` function has no such luxury, because the user must define the entire array contents before writing it to the file.

In order to support large files for writing, the user can make use of the `append` family of functions instead. It is up to the scripting interpreter to decide whether or not to keep the file open between function calls. The ALCHA I/O related functions are listed below:

Function	Description
<code>read</code>	Reads the given file into a <code>byte</code> array buffer
<code>write</code>	Writes the given <code>byte</code> array to a file
<code>append</code>	Appends the given <code>byte</code> array to a file
<code>textread</code>	Reads the given UTF-8 file into a UTF-32 <code>char</code> array buffer
<code>textwrite</code>	Writes the given UTF-32 <code>char</code> array to a UTF-8 file
<code>textappend</code>	Appends the given UTF-32 <code>char</code> array to a UTF-8 file
<code>print</code>	Prints a string to the command-line

The core ALCHA language does not have functions to read specific file formats, such as CSV, XML or JSON, as these can be provided by means of libraries. A similar library function can be developed to, for example, read in a PCB netlist and dynamically build the pin definitions. It is impractical to support all such file formats as part of the core language, but essential to provide sufficient functionality that the user can, with relative ease, develop a parser for any arbitrary file format.

F.13.4 Conversion Functions

While reading and writing text-based files, it is often required to convert between numerical variables (of type `num`) and strings. The functions presented below can be used for this purpose:

Function	Description
<code>\$x</code>	Converts variable <code>x</code> to a string.
<code>\$(x, S)</code>	Converts expression <code>x</code> to a string. The base and other parameters are controlled by means of the optional format string <code>S</code> , which takes a C <code>printf</code> -style format string.
<code>eval(S)</code>	Converts a string <code>S</code> to a number. The scripting interpreter makes use of the same expression engine used in Engineering Calculator .

It is similarly useful to be able to convert `byte` arrays to `num` types, and vice versa. A `byte` array can be assigned directly to a `num` type, and vice versa, in which case the array is evaluated in little Endian order. The code below shows some examples.

```

byte b[4] = [1, 2, 3, 4]; // b[0] = 1; b[1] = 2; ...
num n;

n = b;          // assigns 0x04030201 to n
n = b[0..3]    // assigns 0x04030201 to n
n = b[3..0]    // assigns 0x01020304 to n
n = b[3, 1]    // assigns 0x00000204 to n

n              = 0x05060708;
b              = n; // assigns b[0] = 8; b[1] = 7 ...
b[3..0] = n; // assigns b[0] = 5; b[1] = 6 ...

```

F.13.5 Dynamic Arrays

Scripting arrays can be declared as dynamic by specifying empty square brackets. The array size is adjusted depending on the assignment. While building the contents of a text file, for instance, the code might look as follows:

```
char Buffer[] = "";
// Some code...
Buffer.append("Header\n");
// Some more code...
Buffer.append("Some Body...\n");
// Some more code...
Buffer.append("Some Body...\n");
// Some more code...
Buffer.append("Some Body...\n");
textwrite("My Log Buffer.log", Buffer);
```

F.13.6 Shell Commands

ALCHA scripting is not sufficient for all scenarios of circuit parametrisation. When the design contains a processor, for instance, the user might want the ALCHA compiler to generate a C include-file with all the register addresses, call the C compiler, and then import the resulting executable file into a ROM block on the FPGA.

External commands can be called by means of the `shell` function. It returns the error code of the function called. The `shell` function is a blocking call, so that the external tool can finish before the ALCHA script interpreter continues. The code below shows an example.

```
!! Dynamically build the register addresses here

textwrite("RegDefs.h", RegDefs);
if(shell("gcc main.c -o bin/MyCPU")) error("Cannot compile MyCPU");
byte MyCPU[] = read("bin/MyCPU");

!! Initialise the CPU ROM here
```

Another use for this mechanism is to call an external scripting tool to plot data. This could be useful to visualise the data used to initialise an instance of ROM (especially useful for FIR filter coefficients).

F.14 Advanced Attributes

Attributes are dynamic dictionaries. The user can assign any attribute to any object, and attributes can take any type, including object references and nested attributes:

```
MyClass() MyInstance;
'SomeAttribute = MyInstance; // Attribute containing a reference to an object
'Properties = < Name1 = Value1, Name2 = Value2 >;
```

F.14.1 Logic Regions

In large designs, it is often useful to specify the physical location of each entity. This reduces the solution space of the place-and-route problem, thereby reducing the compilation time. In an Altera project, this is done as shown below.

```
set_global_assignment -name LL_WIDTH 40 -section_id MyDSP
set_global_assignment -name LL_HEIGHT 5 -section_id MyDSP
set_global_assignment -name LL_ORIGIN X44_Y145 -section_id MyDSP
set_global_assignment -name LL_ENABLED ON -section_id MyDSP
set_global_assignment -name LL_AUTO_SIZE OFF -section_id MyDSP
set_global_assignment -name LL_STATE LOCKED -section_id MyDSP
set_global_assignment -name LL_RESERVED OFF -section_id MyDSP
set_global_assignment -name LL_CORE_ONLY OFF -section_id MyDSP
set_global_assignment -name LL_PR_REGION OFF -section_id MyDSP

set_instance_assignment -name LL_MEMBER_OF MyDSP \
                        -to *MyDSP* \
                        -section_id MyDSP
set_instance_assignment -name LL_PRIORITY 1 \
                        -to *MyDSP* \
                        -section_id MyDSP
```

The ALCHA equivalent of this is shown below, which is significantly less verbose. The region is defined by means of an origin coordinate, width and height. The origin is on the bottom-left of the bounding box. If no coordinate is specified, the compiler sets the LL_STATE property to FLOATING. If no width or height is specified, the compiler sets the LL_AUTO_SIZE property to ON. The name of the region is chosen, by the compiler, based on the name of the object the attribute is assigned to.

```
DPS(/* Parameters */) <region = "X44, Y145, W40, H5"> MyDSP;
```


F.14.2 Partition Assignments

Xilinx Vivado and Altera Quartus both allow the user to break large designs into partitions. The vendor compiler only recompiles a partition if the source files that describe the contents of that partition have changed, which reduces compilation time. Partitions come in one of three flavours, depending of what stage of compilation is being re-used:

- **Source:** The partition is always compiled from the source files – nothing is reused.
- **Synthesis:** The synthesis netlist is reused from a previous compile, and the place-and-route stage is processed again.
- **Fitter:** The placed and routed result from the previous compile is reused. This saves significant compilation time, but sometimes makes it more difficult to optimally place the components of other partitions.

ALCHA attributes can be used to perform this partitioning, thereby making it easier to move modules from one partition to another. Any module or net can be placed within a design partition, which is referenced by name. The code below shows an example where the DFE and DataPacker modules are placed in the same partition, and the DSP module in another. The DFE and DataPacker modules uses the fitter netlist, whereas the DSP module uses the synthesis netlist.

```
// Define a default -- also applies to the "Top" partition
'partition_type = "Fitter";

DFE(<partition = "DFE_Packer"> MyDFE;

// This one has a type other than the default
DSP(<partition = "DSP", partition_type = "Synthesis"> MyDSP;

DataPacker(<partition = "DFE_Packer"> MyDataPacker;
```

F.14.3 Compiler Behaviour

In addition to specifying various aspects of design constraints, ALCHA attributes can also be used to locally modify compiler behaviour.

F.14.3.1 Encoding Strategy

The ALCHA grammar does not allow the user to specify the encoding strategy employed by enumerations and state machines. The default encoding for enumerations is binary (i.e. start at zero and continue counting for each member). The default encoding for FSMs are compiler-specific, because state machine encoding can be optimised in various ways. This behaviour can be modified by means of the `encoding` attribute. Valid encodings include “binary”, “Gray”, “Johnson” and “one-hot”.

F.14.3.2 Overflow and Rounding

By default ALCHA expressions follow the same truncation, overflow and rounding rules as Verilog does. Large vectors are assigned to shorter vectors by stripping the most-significant end, rounding is achieved through truncation and there are no checks for overflow conditions.

If this behaviour must be changed for the current application, the user of traditional HDL must manually implement the rounding, clipping and sanity-check circuits. In ALCHA, the user has the option to either implement it manually, or make use of attributes to locally modify the compiler behaviour.

The “rounding” attribute can take one of two values: “truncate” and “nearest”. The “overflow” attribute can take one of two values: “wrap” and “clip”.

Where overflow and rounding are concerned, assignment behaviour follows the attribute of the target net, and type-casting behaviour follows the attribute of the current scope. Without type-casting, expressions are evaluated, as far as possible, without bit-loss.

The concept is illustrated below. The rounding operation includes an addition, which might overflow. If the target width is not enough to store the carry bit, the compiler uses the current “overflow” mode to handle the situation.

```
net(8, 16) A, B;
net(8, 1) C, D;
net(8, 4) X = 0.11; // binary "00.000111"

A'rounding = "nearest";
// B is using default "truncate" rounding

C'overflow = "clip";
// D is using the default "wrap" overflow

A = X; B = X; C = X; D = X;
```

The Verilog equivalent of the above is:

```
wire [ 3:-4] A, B;
wire [-1:-8] C, D;
wire [ 1:-6] X = 8'b00_000111;

// Round to nearest:
assign A = {2'd0, X[1:-4]} + X[-5];
// Truncate:
assign B = {2'd0, X[1:-4]};

always @(*) begin // Clipping circuit
    if(X[1]) begin
        if(&X[0:-1]) C = {X[-1:-6], 2'd0};
        else      C = 8'h80;
    end else begin
        if(!X[0:-1]) C = 8'h3F;
        else      C = {X[-1:-6], 2'd0};
    end
end

// Stripping the most significant bits
assign D = {X[-1:-6], 2'd0};
```

F.15 High-level Structures

F.15.1 Megafunctions and External HDL Modules

An FPGA project often makes use of FPGA features that are not directly supported by ALCHA. Such features might include, among others, PCI-Express interfaces, DDR memory interfaces and embedded processors. These are called “megafunctions”, after the Altera nomenclature. Megafunctions are both vendor and device specific.

Megafunctions and other external HDL modules can be imported into the ALCHA project by means of the `hdl` construct, as illustrated below. Imported HDL modules are not included in simulation.

```

hdl("Library/RS232.v", "Library/RS232_Rx.v", "Library/RS232_Tx.v") RS232_V(
    CountBits = 5; // Parameters
    Count0_5 = 8;
    Count1 = 17;
    Count1_5 = 25;
){
    net    nReset;
    net    Clk;

    net(8) TxData;
    net    Send;
    net    Busy;

    net    DataReady;
    net(8) RxData;
    net    Ack;

    net    Tx;
    net    Rx;
}

class RS232(net Clock, net Reset, num BAUD): RS232_V(
    ceil(log2(round(Clock'frequency / BAUD * 1.5))),
    round(Clock'frequency / BAUD / 2),
    round(Clock'frequency / BAUD),
    round(Clock'frequency / BAUD * 1.5)
){
    nReset = ~Reset;
    Clk    = Clock;
}

pin<frequency = 50e6> Clk;
pin                Reset;
pin                RS232_Tx;
pin                RS232_Rx;

RS232(Clk, Reset, 9600) MyUART;
RS232_Tx = MyUART.Tx;
MyUART.Rx = RS232_Rx;

```

Whether the nets defined in the body of the construct describe input, output or bidirectional ports is optionally inferred from their use. The compiler keeps track of expressions involving that net. If it occurs only on the left-hand side of assignments, it is an input port. If it occurs only on the right-hand side of assignments, it is an output port. If it occurs on both sides, it is bidirectional.

Bidirectional ports pose a problem, because ALCHA does not support high-impedance values on nets. In order to map bidirectional HDL module pins to ALCHA pins, the special `hdl_map` method can be used. An example of assigning I2C pins to an HDL module is presented below:

```
hdl("Library/LTC2991.v") LTC2991_Driver(){
    // Various other ports...

    net I2C_Clk;
    net I2C_Data;
}

pin LTC2991_Clk;
pin LTC2991_Data;

LTC2991_Driver() LTC2991_Driver_Inst;

LTC2991_Clk .hdl_map(LTC2991_Driver_Inst.I2C_Clk );
LTC2991_Data.hdl_map(LTC2991_Driver_Inst.I2C_Data);
```

This `hdl_map` method is only valid for mapping pins to ports defined with an `hdl` construct.

F.16 Simulation and Verification

The ALCHA language has been designed to have simulation and verification features that are as close to the [SystemVerilog industry standard](#) as possible. Minor modifications were applied in order to reduce verbosity and fit in with the rest of the ALCHA language.

F.16.1 Simple Test Benches

It is impractical to perform simulation-based verification at a system level. It is more common to design the modules, or units, of a system with well-defined interfaces, and then test those units prior to integration into the larger system. This practise is called unit-testing [142].

In order to ease maintenance, it is often useful to include the unit test in the same source file as the module it is meant to test. When applied to firmware, the design is synthesised to a circuit netlist, while ignoring all simulation and verification constructs. After synthesis, the resulting netlist is simulated and verified by using the unit test.

F.16.1.1 Stimulus-only Test Benches

In order to perform a unit test, the module under test must have its inputs stimulated by test signals. An ALCHA `stimulus` construct is comparable to the body of a Verilog `initial` block. The delay operators (`#`, `##` and `@`) mean the same thing in both languages. There is no equivalent to a Verilog `always` block, because the same thing can be achieved by using a

loop.

```

group<waveform = "green">{ net(8) A = 0, B = 0, C = 0; }
// Simple stimulus
stimulus(1e-9){ // Use nanoseconds as the basic time-step
  A = 5;        // At the start of simulation, make A = 5
  #5 B = 7;      // After 5 time-steps, make B = 7
  @(posedge Clk); // Wait until the positive edge of the Clk signal
  loop #7 C++;   // Forever afterwards, wait 7 time-steps and increment C
}
// Clock-based stimulus
stimulus(1e-9, Clk){ // Use nanoseconds as the basic time-step,
                    // and Clk as the default clock
  #7 A = 5;        // Wait 7 ns, then make A = 5
  ##3 B = 7;       // After 3 rising-edges of Clk, make B = 7
  loop ##1 C++;    // Forever afterwards, wait 1 clock edge and increment C
}

```

The `waveform` attribute informs the simulator that the signal should be included in the log and displayed on the waveform viewer. Signals without this attribute are not logged, which saves time and memory. The value can be used to format the visual representation. In this case, the waveform should be displayed in green.

The parameters of the `stimulus` construct takes the place of the Verilog `timescale` directive and `default clocking` declaration. The latter is optional, in which case the most recent `@(posedge ...)` statement is used to determine the current clock used by the `##` operator.

Stimulus constructs can be defined anywhere, including class and function bodies. The stimulus construct is ignored by the synthesis tool, but implemented as a concurrent process by the simulation tool. Every instance of the function (or class) then has its own instance of that stimulus process.

F.16.1.2 Concurrent Processes

It is often useful to split stimulus execution into multiple threads without using a completely separate `stimulus` block. Verilog implements this concept by means of the `fork-join` construct. The Verilog `disable` statement can be implemented by means of a `break` statement in ALCHA.

```
stimulus(1e-9){
  // fork ... join_all
  { /* The first fork */ } && { /* The second fork */ } ;

  // fork ... join_any
  { /* The first fork */ } || { /* The second fork */ } ;

  // fork ... join_none
  { /* The first fork */ } || { /* The second fork */ } || {};
  /* The rest of the stimulus body */

  // Wait for Condition, or a delay, and stop the other thread when done
  {wait(Condition); break;} || {#100 break;} ;
}
```

F.16.1.3 Simulation Scripting

Scripting statements within simulation constructs are run during simulation time, not compile-time. This makes it possible to write advanced test-benches, including file I/O, casting **net** types to **num** types, making use of transcendental mathematical functions, etc. File I/O can be used to implement data injection and monitoring. The results can be logged to file and compared with expected results.

F.16.2 Model Based Verification

Model-based verification is a design methodology where the user would first write an emulation of the intended module functionality. This is generally easier to do than writing the synthesisable code, because the user has access to advanced scripting features.

The user would then develop the synthesisable equivalent of the emulation (or model). During verification, the same stimulus is injected into both synthesisable and emulation modules, and the results compared. Differences are highlighted as errors. For digital signal processing applications, however, a small margin of error is generally allowed.

F.16.2.1 Emulating Modules

Emulation has another use: it is often, if not always, faster to emulate a module, that has been previously verified, than simulating the full synthesised netlist. The **emulate** construct can be used for this purpose, as well as model-based verification.

It has the same syntax and semantics as the `stimulus` construct, but allows the user to overwrite synthesised nets with simulated signals. The `stimulus` construct can only assign to nets that have constant-expressions assigned to them (the constant-expression is taken as the value at the start of simulation). The `emulate` construct can assign to any net, thereby overwriting the RTL behaviour.

```
class FFT{
  net(31, -1) Input    [256];
  net(31, -1) Output_I[256];
  net(31, -1) Output_Q[256];

  net Go, Busy;

  !! RTL implementation of the FFT...

  // This is run during simulation, instead of the RTL above
  emulate(1e-9, Clk){
    loop{
      ##1;
      if(Go){
        Busy = 1;
        num X = fft(Input) / 256;

        ##8; // The latency of the FFT engine
        Output_I = real(X);
        Output_Q = imag(X);

        while(Go) ##1;
        Busy = 0;
      }
    }
  }
}
```

The user can make use of conditional compilation to switch between the emulated and simulated models.

F.16.2.2 Emulating Imported HDL

The same construct can be used to emulate imported HDL. The ALCHA simulator does not support the simulation of external VHDL or Verilog code. The user must include an emulator if those modules must be included in the simulation. The example below shows an example of how to emulate an Altera ROM block.


```

hdl() altsyncram(
    num widthad_a = 12,
    num width_a   = 16,
    num numwords_a = 2**widthad_a,
    char init_file[],
    !! Other "altsyncram" parameter declarations go here
){
    input net          clock0;
    input net(widthad_a) address_a;
    output net(width_a ) q_a;
    !! Other "altsyncram" port declarations go here
}

class CPU_ROM(net(16) Opcodes[4096]){
    !! Use scripting to write Opcodes to "CPU_ROM.mif"

    // Instantiate the ROM block
    private altsyncram ROM = altsyncram(init_file = "CPU_ROM.mif");

    // Emulate the ROM block during simulation
    ROM.{ // For convenience, push the ROM namespace onto the stack
        emulate(1e-9, clock0){
            loop ##1 q_a = Opcodes[address_a];
        }
    }

    // Declare local signals (for the convenience of users of this class)
    net    Clk;
    net(12) Address;
    net(16) Data;

    // Connect the ports
    ROM.clock0    = Clk;
    ROM.address_a = Address;
    Data          = ROM.q_a;
}

```

F.16.3 Assertion Based Verification

One of the more powerful methods of verification, provided by SystemVerilog, is assertion-based verification. The verification engineer would write a list of rules, based on the system specification. The verification tool then checks that those rules hold true for every step of the simulation.

F.16.3.1 Simple Assertions

The simplest way to write an assertion is to include it in a `stimulus` construct. It checks the state of the system at that point in the simulation, by means of a semicolon-separated list of boolean expressions.

```

net(8) Adder(net(8) A, net(8) B, net C){
    : (C, Adder) = A + B;
}
net    c;
net(8) a, b, y = Adder(a, b, c);

stimulus(1e-9){
    #1 a = 0x00; b = 0x00; assert{y == 0x00; c == 0; };
    #1 a = 0xFF; b = 0x01; assert{y == 0x00; c == 1; };
    #1 a = 0x01; b = 0xFF; assert{y == 0x00; c == 1; };
    #1 a = 0xFF; b = 0xFF; assert{y == 0xFE; c == 1; };
    #1 a = 0x7F; b = 0x80; assert{y == 0xFF; c == 0; };
}

```

Another option is to write the assertions outside the stimulus construct. Assertion statements of this form would typically take the form shown below. The `|->` operator is used to specify rules of the form: “when the left-hand-side is true, then the right-hand-side must also hold true”.

```

assert{
    a == 0x00 && b == 0x00 |-> y == 0x00 && c == 0;
    a == 0xFF && b == 0x01 |-> y == 0x00 && c == 1;
    a == 0x01 && b == 0xFF |-> y == 0x00 && c == 1;
    a == 0xFF && b == 0xFF |-> y == 0xFE && c == 1;
    a == 0x7F && b == 0x80 |-> y == 0xFF && c == 0;
}

```

F.16.3.2 Sequence Based Assertions

SystemVerilog sequences can be compared to regular expressions. The verification tool matches the given sequence expression against the simulated waveforms. A hand-shake transaction might be specified as shown below. As with SystemVerilog, the signals are sampled one simulation time-step before the clock edge being tested, thereby ensuring that the system state is consistent with a hardware implementation.

```

// A simple sequence:
sequence Handshake{
    Go ##[1..5] Busy ##[1.. 10] !Go ##[1..100] !Busy;
}

// A sequence-based assert:
assert(1e-9, Clk){
    if(~Reset){
        Go      |-> Handshake;
        rose(Go ) |-> !Busy;
        fell(Busy) |-> !Go;
        (Go & !Busy) |> stable(Data);
    }
}

```

In this case, the sequence inherits the time-step and clock signal from the assertions that calls it. It is also possible to specify a time-step and, optionally, a clock signal in the sequence declaration, which will override the assertion time-step and clock signals.

Sequences are useful in other aspects as well. The entire sequence is essentially a boolean expression, albeit over multiple clock-cycles. This means that it can be used within a `stimulus` construct `wait` statement. The `wait` statement will then wait until the sequence has occurred, instead of a simple condition.

F.16.3.3 Advanced Sequences

SystemVerilog sequences can be combined by using the `and`, `or` and `intersect` keywords. The ALCHA equivalents are the `&&`, `||` and `&&&` operators, respectively. These operators are generally used, in other languages, to distinguish between bit-wise and logical binary operations. This is not required in ALCHA, because it does not support logical operations directly: the user is expected to OR-reduce the operands and then use the binary operator instead.

In order to make the sequence expression syntax more expressive, SystemVerilog provides the user with helper functions, as well as repetition operators. These include `$rose(...)`, `$fell(...)`, `$stable(...)`, `$past(...)`, `[* ... ]`, `[-> ... ]` and `[= ... ]`. ALCHA supports the same functions and operators, but without the `$` prefix to the functions.

F.16.4 Formal Verification

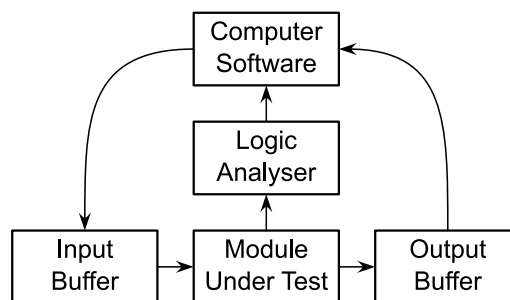
Formal verification is related to assertion-based verification. Assertions make sure that the design adheres to the specified rules for every iteration of the simulation. Formal verification runs the process in reverse. It starts with the rules specified by the assertion statements, and attempts to find a case that violates those rules, without a simulation.

Formal verification is possible within the current language design, but outside the scope of this stage of the design. It can be implemented as a feature of the simulation tool.

F.16.5 Hardware in the Loop Verification

There are limitations to simulation that inhibit the testing of some systems. Large systems take a long time to simulate: much longer than it takes to compile to firmware and load the resulting circuit onto the FPGA. In addition, simulations generally do not include low-level timing, because the computational power required makes the simulation impractical. It is more common to simulate only the behaviour of the circuit and trust that the timing analysis of the vendor-provided compilation tool provides sufficient evidence that the circuit will work correctly.

Both of these problems can be eliminated by means of hardware-in-the-loop verification. The design under test is compiled to firmware and loaded onto the FPGA, together with infrastructure to allow a computer-based test-bench to inject stimulus and read the result.



The logic analyser can be a laboratory instrument, embedded logic analyser¹ within the FPGA firmware, or a combination of the two. Within the ALCHA language, the user can add the `logic_analyser` attribute to any signal, which will cause the compiler to automatically generate an embedded logic analyser project file that includes that signal in the sample set.

¹Embedded logic analysers are part of modern vendor-provided tools. Xilinx Vivado includes ChipScope, and Altera Quartus provides SignalTap.

The input and output buffers are more challenging to automate, because the implementation depends on the specific use case. The user is therefore expected to make use of user-developed classes (which can be packaged into libraries) to implement these buffers and associated communication interfaces.

F.16.6 Coverage

Many modern verification tools support reporting both code and functional coverage. These reports are used to determine two things: dead code, and insufficient test cases. If the verification run shows that a piece of code never executes, the developer should decide to either remove it from the code, exclude it from code coverage checks or include a test case that executes it.

The same holds true for assertion rules. It is up to the verification engineer to create stimuli that make use of all the assertion test cases. A coverage report informs the verification engineer whether or not this is the case.

F.16.6.1 Code Coverage

Code coverage is the simpler of the three coverage types. The process logs a counter on every statement in the source code, and reports the number of times each statement was executed during simulation. This also includes branch coverage, which tests that every possible branch of `if` and `case` statements have been tested.

```
group<code_coverage = true>{  
    !! All the code that should be tested for coverage goes here  
}  
  
stimulus(1e-9, Clk){  
    !! Describe the stimulus here  
}
```

F.16.6.2 Expression Coverage

Expression coverage tests that all possible combinations and permutations of an expression has been tested. This is generally reported as a percentage, because it is often impractical to test every possibility. Any expression that carry the `expression_coverage` attribute is included in the expression coverage report.

F.16.6.3 Functional Coverage

Functional coverage is more complex. The process logs a counter for every functionality that the test engineer specified. Functions include typical valid use cases, edge cases and boundary conditions, protocols, etc.

In the case of a simplified floating-point operation, each input operand, as well as the output, can fall into one of six categories: NAN, $-\infty$, < 0 , 0 , > 0 , ∞ . This means that there are 6^3 (216) possible combinations of functionality. The example below shows the ALCHA syntax for specifying functional coverage for this example.

```
// Instances of the Float class
Float A = Float();
Float B = Float();
Float Y = Float();

// Instance of the FPU_Div class
FPU_Div(Clk, A, B, Y) MyDiv;

// Define coverage bins
coverbins FloatingPointFunctions(x){
    NAN      = x.exponent == 0xFF & x.fraction > 0;
    NegInfnty = x.sign == 1 & x.exponent == 0xFF & x.fraction == 0;
    Negative  = x.sign == 1 & x.exponent > 0 & x.exponent < 0xFF;
    Zero      = x.exponent == 0 & x.fraction == 0;
    Positive  = x.sign == 0 & x.exponent > 0 & x.exponent < 0xFF;
    PosInfnty = x.sign == 0 & x.exponent == 0xFF & x.fraction == 0;
}

// Define a coverage group
covergroup(Clk) MyDivCoverage{
    FloatingPointFunctions(A);
    FloatingPointFunctions(B);
    FloatingPointFunctions(Y);
}
```

The `coverbins` construct defines a list of bins, defined by boolean expressions, that applies to the input parameters. There is only one parameter (`x`) in the example, but there can in fact be any number of parameters. The simulator issues an error when the current system state falls into more than one bin, or no bins.

The boolean expression used to define bins also supports sequences, which can be used to test protocol coverage.

The `covergroup` construct is then used to group multiple `coverbins` constructs into a functional coverage description. In the example, it defines the 216 bins that cover all combinations of input and output categories.

F.16.7 Verification Frameworks

There are various simulation methodologies and industry standards. One of the more popular ones is UVM (Universal Verification Methodology) [143]. It is an Accellera standard, implemented by means of a SystemVerilog library of classes.

The UVM methodology itself is outside the scope of the ALCHA design. It is possible, however, to implement an ALCHA based UVM library that is equivalent to the current SystemVerilog version.

F.16.8 Finite State Machines

Traditional simulation involves breaking the source-code down into event-driven structures. In the case of synthesisable Verilog, the entire **always** block is evaluated for any event included in the sensitivity list. VHDL follows the same concept for evaluating synthesisable processes. Although modern simulation tools are heavily optimised, the deeply-nested branched structure used to describe state machines still poses a limitation on the time-efficiency of simulation.

The ALCHA **fsm** construct describes a sequential program flow, which is more natural to CPU-based execution. It could be possible to simulate the behaviour of an ALCHA-based finite state machine by compiling it to machine-code. The details of this concept is the topic of future work.