Financial Technology with a System on Chip

QueenField

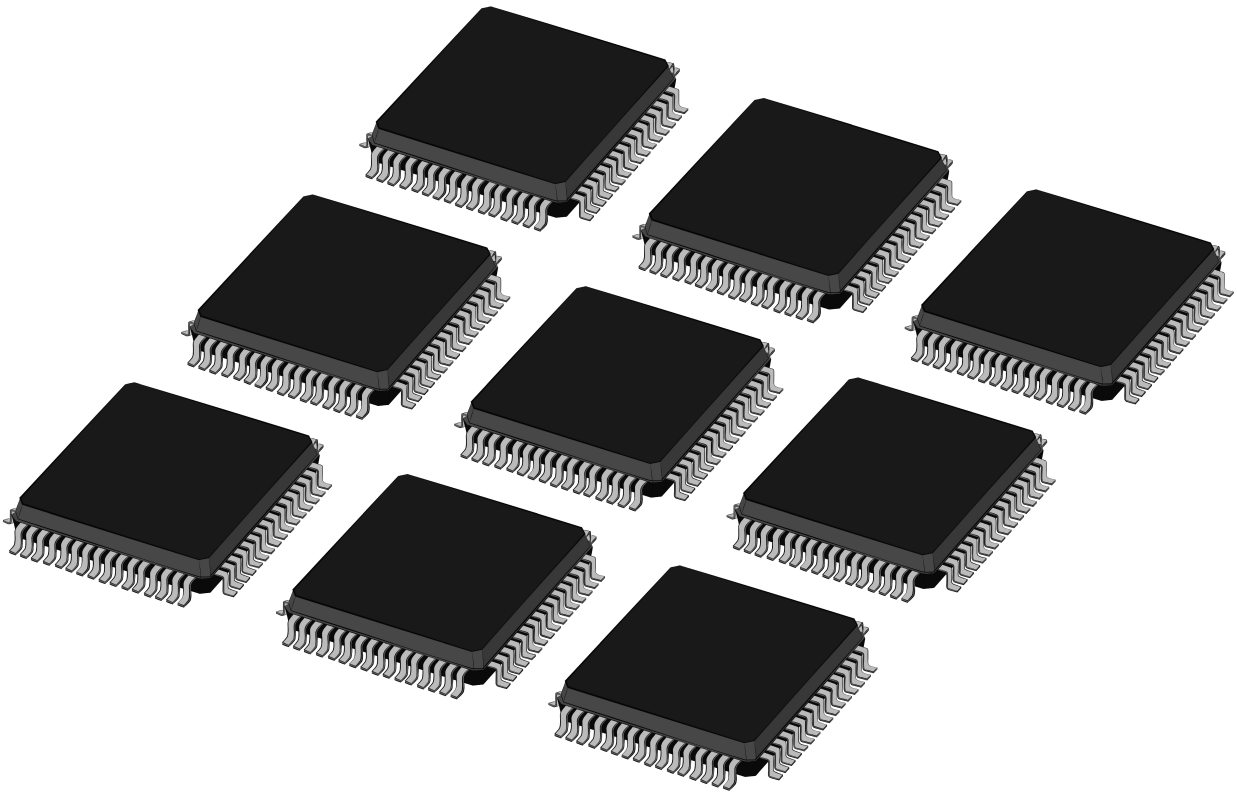

Figure 1: QueenField

0. INTRODUCTION

A System on Chip (SoC) is an integrated circuit that integrates components of a computer system (PU, RAM, GPIO, etc). As they are integrated on a single substrate, SoCs consume much less power and take up much less area than multi-chip designs with equivalent functionality. SoCs are common in the mobile computing, embedded systems and the Internet of Things.

An Automation Financial Method (AFM) is the technology and innovation that aims to compete with Traditional Financial Methods in the delivery of financial services. It is an emerging industry that uses technology to improve activities in finance. AFM is the new applications, processes, products, or business models in the financial services industry, composed of complementary financial services and provided as an end-to-end process via the Internet.

0.0. DO-254

0.0.1. PLANNING PROCESS

0.0.1.1. Planning Process Objectives

0.0.1.2. Planning Process Activities

0.0.2. HARDWARE DESIGN PROCESS

0.0.2.1. Requirements Capture Process

0.0.2.2. Conceptual Design Process

0.0.2.3. Detailed Design Process

0.0.2.4. Implementation Process

0.0.2.5. Production Transition

0.0.2.6. Acceptance Test

0.0.2.7. Series Production

0.0.3. VALIDATION AND VERIFICATION PROCESS

0.0.3.1. Validation Process

0.0.3.2. Verification Process

0.0.3.3. Validation and Verification Methods

0.0.4. CONFIGURATION MANAGEMENT PROCESS

0.0.4.1. Configuration Management Objectives

0.0.4.2. Configuration Management Activities

0.0.4.3. Data Control Categories

0.0.5. PROCESS ASSURANCE

0.0.5.1. Process Assurance Objectives

0.0.5.2. Process Assurance Activities

0.0.6. CERTIFICATION LIAISON PROCESS

0.0.6.1. Means of Compliance and Planning

0.0.6.2. Compliance Substantiation

0.0.7. HARDWARE DESIGN LIFECYCLE DATA

0.0.7.1. Hardware Plans

0.0.7.1.1. Plan for Hardware Aspects of Certification

0.0.7.1.2. Hardware Design Plan

0.0.7.1.3. Hardware Validation Plan

0.0.7.1.4. Hardware Verification Plan

0.0.7.1.5. Hardware Configuration Management Plan

0.0.7.1.6. Hardware Process Assurance Plan

0.0.7.2. Hardware Design Standards and Guidance

0.0.7.2.1. Requirements Standards

0.0.7.2.2. Hardware Design Standards

0.0.7.2.3. Validation and Verification Standards

0.0.7.2.4. Hardware Archive Standards

0.0.7.3. Hardware Design Data

0.0.7.3.1. Hardware Requirements *The requirements specify the functional, performance, safety, quality, maintainability, and reliability requirements for the hardware item being developed.*

0.0.7.3.2. Hardware Design Representation Data *The hardware design representation data provides a definition of the hardware item and is comprised of the set of drawings, documents and specifications used to build the hardware item. The following paragraphs define some typical hardware design data and their content. The type of data, drawings and documents produced for a given hardware design will vary depending on the size, complexity and number of components the hardware item contains.*

0.0.7.3.2.1. Conceptual Design Data

The conceptual design data is the data that describes the hardware item's architecture and functional design.

0.0.7.3.2.2. Detailed Design Data

0.0.7.3.2.2.1. Top-Level Drawing

0.0.7.3.2.2.2. Assembly Drawings

0.0.7.3.2.2.3. Installation Control Drawings

0.0.7.3.2.2.4. Hardware/Software Interface Data

0.0.7.4. Validation and Verification Data

0.0.7.4.1. Traceability Data

0.0.7.4.2. Review and Analysis Procedures

0.0.7.4.3. Review and Analysis Results

0.0.7.4.4. Test Procedures

0.0.7.4.5. Test Results

0.0.8. ADDITIONAL CONSIDERATIONS

0.0.8.1. Use of Previously Developed Hardware

0.0.8.2. Commercial Components Usage

0.0.8.3. Product Service Experience

0.0.8.4. Tool Assessment and Qualification

0.1. BEST PRACTICES

0.1.1. HARDWARE

0.1.2. SOFTWARE

1. METHODOLOGY

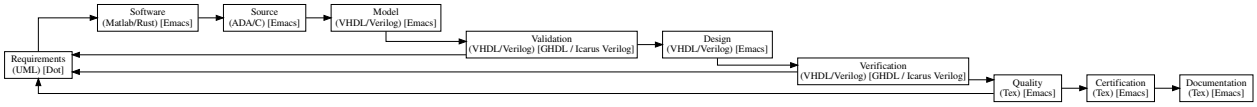

Figure 2: Project Workflow

1.1. Requirements

1.1.1. Structural UML diagrams

1.1.1.1. Class diagram

1.1.1.2. Component diagram

1.1.1.3. Composite diagram

1.1.1.4. Deployment diagram

1.1.1.5. Object diagram

1.1.1.6. Package diagram

1.1.1.7. Profile diagram

1.1.2. Behavioral UML diagrams

1.1.2.1. Activity diagram

1.1.2.2. Communication diagram

1.1.2.3. Interaction diagram

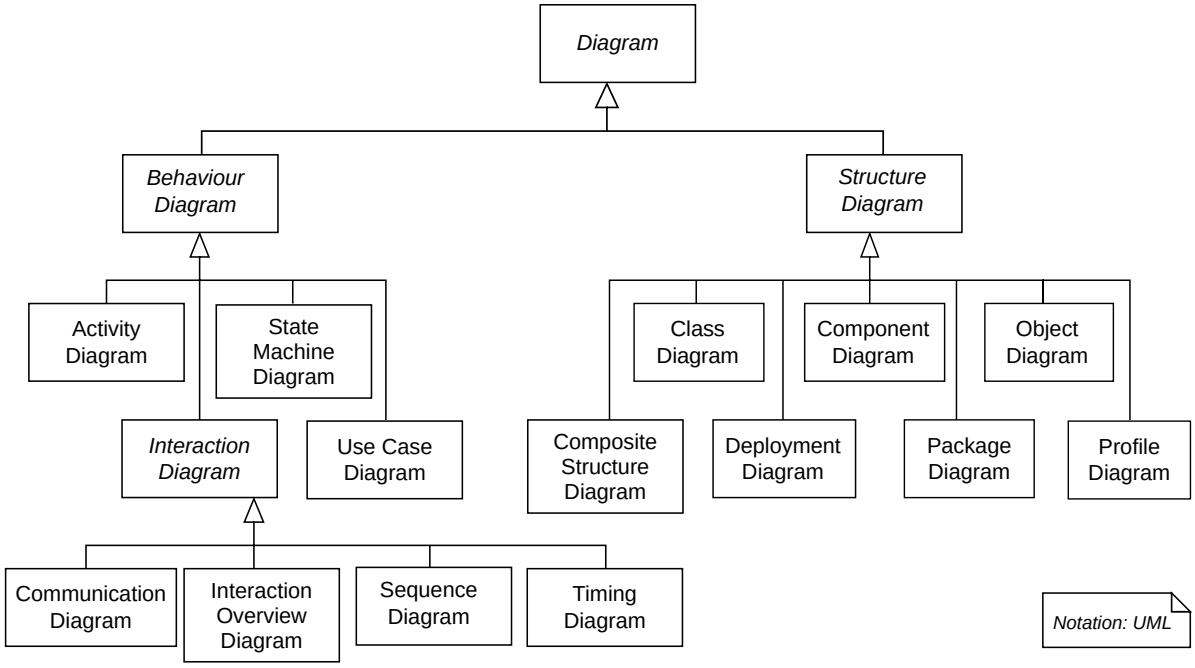

Figure 3: UML Diagrams Overview

1.1.2.4. Sequence diagram

1.1.2.5. State diagram

1.1.2.6. Timing diagram

1.1.2.7. Use diagram

1.2. Source

1.2.1. Ada Language

1.2.2. C Language

1.3. Model

1.3.1. VHDL

1.3.2. Verilog

1.5. Validation

1.5.1. VHDL

1.5.2. Verilog

1.5. Design

1.5.1. VHDL

1.5.2. Verilog

1.6. Verification

1.6.1. OSVVM-VHDL

1.6.1.1. OSVVM Checker

1.6.1.2. OSVVM Stimulus

1.6.1.3. OSVVM Testbench

1.6.2. UVM-Verilog

1.6.2.1. UVM Agent

1.6.2.2. UVM Driver

1.6.2.3. UVM Enviroment

1.6.2.4. UVM Monitor

1.6.2.5. UVM Scoreboard

1.6.2.6. UVM Sequence

1.6.2.7. UVM Sequencer

1.6.2.8. UVM Subscriber

1.6.2.9. UVM Test

1.6.2.10. UVM Testbench

1.6.2.11. UVM Transaction

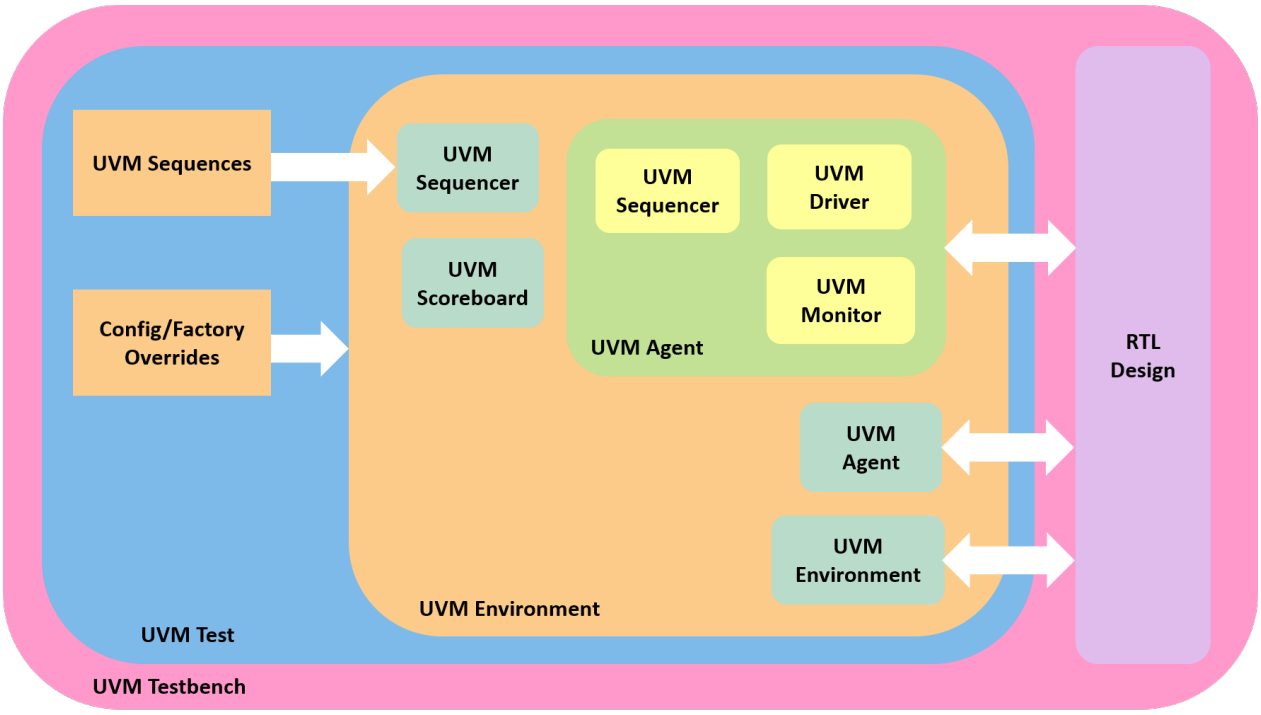

Figure 4: UVM Diagram Overview

2. PROJECTS

2.1. INTERFACE

2.1.1. INSTRUCTION CACHE

2.1.1.1 Instruction INPUTS/OUTPUTS AMBA4 AXI-Lite Bus

2.1.1.1.1. Signals of the Read and Write Address channels

Write Port	Read Port	Size	Direction	Description
AWID	ARID	AXI_ID_WIDTH	Output	Address ID, to identify multiple streams
AWADDR	ARADDR	AXI_ADDR_WIDTH	Output	Address of the first beat of the burst
AWLEN	ARLEN	8	Output	Number of beats inside the burst
AWSIZE	ARSIZE	3	Output	Size of each beat
AWBURST	ARBURST	2	Output	Type of the burst
AWLOCK	ARLOCK	1	Output	Lock type, to provide atomic operations
AWCACHE	ARCACHE	4	Output	Memory type, progress through the system
AWPROT	ARPROT	3	Output	Protection type
AWQOS	ARQOS	4	Output	Quality of Service of the transaction
AWREGION	ARREGION	4	Output	Region identifier, physical to logical
AWUSER	ARUSER	AXI_USER_WIDTH	Output	User-defined data
AWVALID	ARVALID	1	Output	xVALID handshake signal
AWREADY	ARREADY	1	Input	xREADY handshake signal

2.1.1.1.2. Signals of the Read and Write Data channels

Write Port	Read Port	Size	Direction	Description
WID	RID	AXI_ID_WIDTH	Output	Data ID, to identify multiple streams
WDATA	RDATA	AXI_DATA_WIDTH	Output	Read/Write data
--	RRESP	2	Output	Read response, current RDATA status
WSTRB	--	AXI_STRB_WIDTH	Output	Byte strobe, WDATA signal
WLAST	RLAST	1	Output	Last beat identifier
WUSER	RUSER	AXI_USER_WIDTH	Output	User-defined data
WVALID	RVALID	1	Output	xVALID handshake signal
WREADY	RREADY	1	Input	xREADY handshake signal

2.1.1.1.3. Signals of the Write Response channel

Write Port	Size	Direction	Description
BID	AXI_ID_WIDTH	Input	Write response ID, to identify multiple streams
BRESP	2	Input	Write response, to specify the burst status
BUSER	AXI_USER_WIDTH	Input	User-defined data
BVALID	1	Input	xVALID handshake signal
BREADY	1	Output	xREADY handshake signal

2.1.1.2. Instruction INPUTS/OUTPUTS AMBA3 AHB-Lite Bus

Port	Size	Direction	Description
HRESETn	1	Input	Asynchronous Active Low Reset
HCLK	1	Input	System Clock Input
IHSEL	1	Output	Instruction Bus Select
IHADDR	PLEN	Output	Instruction Address Bus
IHRDATA	XLEN	Input	Instruction Read Data Bus
IHWDATA	XLEN	Output	Instruction Write Data Bus
IHWRITE	1	Output	Instruction Write Select
IHSIZE	3	Output	Instruction Transfer Size
IHBURST	3	Output	Instruction Transfer Burst Size
IHPROT	4	Output	Instruction Transfer Protection Level
IHTRANS	2	Output	Instruction Transfer Type
IHMASTLOCK	1	Output	Instruction Transfer Master Lock
IHREADY	1	Input	Instruction Slave Ready Indicator
IHRESP	1	Input	Instruction Transfer Response

2.1.1.3. Instruction INPUTS/OUTPUTS Wishbone Bus

Port	Size	Direction	Description
rst	1	Input	Synchronous Active High Reset
clk	1	Input	System Clock Input
iadr	AW	Input	Instruction Address Bus
idati	DW	Input	Instruction Input Bus
idato	DW	Output	Instruction Output Bus
isel	DW/8	Input	Byte Select Signals
iwe	1	Input	Write Enable Input

Port	Size	Direction	Description
istb	1	Input	Strobe Signal/Core Select Input
icyc	1	Input	Valid Bus Cycle Input
iack	1	Output	Bus Cycle Acknowledge Output
ierr	1	Output	Bus Cycle Error Output
iint	1	Output	Interrupt Signal Output

2.1.2. DATA CACHE

2.1.2.1. Data INPUTS/OUTPUTS AMBA4 AXI-Lite Bus

2.1.2.1.1. Signals of the Read and Write Address channels

Write Port	Read Port	Size	Direction	Description
AWID	ARID	AXI_ID_WIDTH	Output	Address ID, to identify multiple streams
AWADDR	ARADDR	AXI_ADDR_WIDTH	Output	Address of the first beat of the burst
AWLEN	ARLEN	8	Output	Number of beats inside the burst
AWSIZE	ARSIZE	3	Output	Size of each beat
AWBURST	ARBURST	2	Output	Type of the burst
AWLOCK	ARLOCK	1	Output	Lock type, to provide atomic operations
AWCACHE	ARCACHE	4	Output	Memory type, progress through the system
AWPROT	ARPROT	3	Output	Protection type
AWQOS	ARQOS	4	Output	Quality of Service of the transaction
AWREGION	ARREGION	4	Output	Region identifier, physical to logical
AWUSER	ARUSER	AXI_USER_WIDTH	Output	User-defined data
AWVALID	ARVALID	1	Output	xVALID handshake signal
AWREADY	ARREADY	1	Input	xREADY handshake signal

2.1.2.1.2. Signals of the Read and Write Data channels

Write Port	Read Port	Size	Direction	Description
WID	RID	AXI_ID_WIDTH	Output	Data ID, to identify multiple streams
WDATA	RDATA	AXI_DATA_WIDTH	Output	Read/Write data
--	RRESP	2	Output	Read response, current RDATA status
WSTRB	--	AXI_STRB_WIDTH	Output	Byte strobe, WDATA signal
WLAST	RLAST	1	Output	Last beat identifier
WUSER	RUSER	AXI_USER_WIDTH	Output	User-defined data
WVALID	RVALID	1	Output	xVALID handshake signal
WREADY	RREADY	1	Input	xREADY handshake signal

2.1.2.1.3. Signals of the Write Response channel

Write Port	Size	Direction	Description
BID	AXI_ID_WIDTH	Input	Write response ID, to identify multiple streams
BRESP	2	Input	Write response, to specify the burst status
BUSER	AXI_USER_WIDTH	Input	User-defined data
BVALID	1	Input	xVALID handshake signal
BREADY	1	Output	xREADY handshake signal

2.1.2.2. Data INPUTS/OUTPUTS AMBA3 AHB-Lite Bus

Port	Size	Direction	Description
HRESETn	1	Input	Asynchronous Active Low Reset
HCLK	1	Input	System Clock Input
DHSEL	1	Output	Data Bus Select
DHADDR	PLEN	Output	Data Address Bus
DHRDATA	XLEN	Input	Data Read Data Bus
DHWDATA	XLEN	Output	Data Write Data Bus
DHWRITE	1	Output	Data Write Select
DHSIZE	3	Output	Data Transfer Size
DHBURST	3	Output	Data Transfer Burst Size
DHPROT	4	Output	Data Transfer Protection Level
DHTRANS	2	Output	Data Transfer Type
DHMASTLOCK	1	Output	Data Transfer Master Lock
DHREADY	1	Input	Data Slave Ready Indicator
DHRESP	1	Input	Data Transfer Response

2.1.2.3. Data INPUTS/OUTPUTS Wishbone Bus

Port	Size	Direction	Description
rst	1	Input	Synchronous Active High Reset
clk	1	Input	System Clock Input
dadr	AW	Input	Data Address Bus
ddati	DW	Input	Data Input Bus
ddato	DW	Output	Data Output Bus
dsel	DW/8	Input	Byte Select Signals
dwe	1	Input	Write Enable Input
dstb	1	Input	Strobe Signal/Core Select Input
dcyc	1	Input	Valid Bus Cycle Input
dack	1	Output	Bus Cycle Acknowledge Output
derr	1	Output	Bus Cycle Error Output
dint	1	Output	Interrupt Signal Output

2.2. FUNCTIONALITY

2.2.1. Structure

```
class traditional_classes {
   private:
      int number_pu;
      int number_soc;
      int number_mpsoc;

   public:
      void traditional_method_0();  // method 0
      void traditional_method_1();  // method 1
      void traditional_method_2();  // method 2
      void traditional_method_3();  // method 3
};
```

2.2.1.1. Philosophers T-DNC/NTM-SoC

```
class traditional_philosophers : private traditional_classes {
   private:
      int number_p_pu;
      int number_p_soc;
      int number_p_mpsoc;

   public:
      void traditional_method_p0();  // method 0
      void traditional_method_p1();  // method 1
      void traditional_method_p2();  // method 2
      void traditional_method_p3();  // method 3
};
```

2.2.1.1.1. PU-NTM

2.2.1.1.2. SoC-NTM

2.2.1.2. Soldiers T-DNC/NTM-SoC

```
class traditional_soldiers : private traditional_classes {
   private:
      int number_s_pu;
      int number_s_soc;
      int number_s_mpsoc;

   public:
      void traditional_method_s0();  // method 0
      void traditional_method_s1();  // method 1
      void traditional_method_s2();  // method 2
      void traditional_method_s3();  // method 3
};
```

2.2.1.2.1. PU-NTM

2.2.1.2.2. SoC-NTM

2.2.1.3. Workers T-DNC/NTM-SoC

```
class traditional_workers : private traditional_classes {
   private:
      int number_w_pu;
      int number_w_soc;
      int number_w_mpsoc;

   public:
      void traditional_method_w0();  // method 0
      void traditional_method_w1();  // method 1
      void traditional_method_w2();  // method 2
      void traditional_method_w3();  // method 3
};
```

2.2.1.3.1. PU-NTM

2.2.1.3.2. SoC-NTM

2.2.2. Behavior

2.3. REGISTERS

2.4. INTERRUPTIONS

3. ORGANIZATION

3.1. Mechanics

3.2. Information

3.2.1. Bit

3.2.2. Logic Gate

3.2.2.1. YES/NOT Gate

3.2.2.2. AND/NAND Gate

3.2.2.3. OR/NOR Gate

3.2.2.4. XOR/XNOR Gate

3.2.3. Combinational Logic

3.2.3.1. Arithmetic Circuits

3.2.3.2. Logic Circuits

3.2.4. Finite State Machine

3.2.5. Pushdown Automaton

3.3. Neural Network

3.3.1. Feedforward Neural Network

3.3.2. Long Short Term Memory Neural Network

3.3.3. Transformer Neural Network

3.4. Turing Machine

3.4.1. Neural Turing Machine

3.4.1.1. Feedforward Neural Turing Machine

3.4.1.2. LSTM Neural Turing Machine

3.4.1.3. Transformer Neural Turing Machine

3.4.2. Differentiable Neural Computer

3.4.2.1. Feedforward Differentiable Neural Computer

3.4.2.2. LSTM Differentiable Neural Computer

3.4.2.3. Transformer Differentiable Neural Computer

3.5. Computer Architecture

3.5.1. von Neumann Architecture

3.5.1.1. Control Unit

3.5.1.2. ALU

3.5.1.3. Memory Unit

3.5.1.4. I/O Unit

3.5.2. Harvard Architecture

3.5.2.1. Control Unit

3.5.2.2. ALU

3.5.2.3.Memory Unit

3.5.2.4.I/O Unit

3.6. Advanced Computer Architecture

3.6.1. Processing Unit

3.6.1.1. SISD

3.6.1.2. SIMD

3.6.1.3. MISD

3.6.1.4. MIMD

3.6.2. System on Chip

3.6.2.1. Bus on Chip

3.6.2.2. Network on Chip

4. HARDWARE WORKFLOW

1. System Level (SystemC/SystemVerilog)

The System Level abstraction of a system only looks at its biggest building blocks like processing units or peripheral devices. At this level the circuit is usually described using traditional programming languages like SystemC or SystemVerilog. Sometimes special software libraries are used that are aimed at simulation circuits on the system level. The IEEE 1685-2009 standard defines the IP-XACT file format that can be used to represent designs on the system level and building blocks that can be used in such system level designs.

2. Behavioral & Register Transfer Level (VHDL/Verilog)

At the Behavioural Level abstraction a language aimed at hardware description such as Verilog or VHDL is used to describe the circuit, but so-called behavioural modeling is used in at least part of the circuit description. In behavioural modeling there must be a language feature that allows for imperative programming to be used to describe data paths and registers. This is the always -block in Verilog and the process -block in VHDL.

A design in Register Transfer Level representation is usually stored using HDLs like Verilog and VHDL. But only a very limited subset of features is used, namely minimalistic always blocks (Verilog) or process blocks (VHDL) that model the register type used and unconditional assignments for the datapath logic. The use of HDLs on this level simplifies simulation as no additional tools are required to simulate a design in Register Transfer Level representation.

3. Logical Gate

At the Logical Gate Level the design is represented by a netlist that uses only cells from a small number of single-bit cells, such as basic logic gates (AND, OR, NOT, XOR, etc.) and registers (usually D-Type Flip-flops). A number of netlist formats exists that can be used on this level such as the Electronic Design Interchange Format (EDIF), but for ease of simulation often a HDL netlist is used. The latter is a HDL file (Verilog or VHDL) that only uses the most basic language constructs for instantiation and connecting of cells.

4. Physical Gate

On the Physical Gate Level only gates are used that are physically available on the target architecture. In some cases this may only be NAND, NOR and NOT gates as well as D-Type registers. In the case of an FPGA-based design the Physical Gate Level representation is a netlist of LUTs with optional output registers, as these are the basic building blocks of FPGA logic cells.

5. Switch Level

A Switch Level representation of a circuit is a netlist utilizing single transistors as cells. Switch Level modeling is possible in Verilog and VHDL, but is seldom used in modern designs, as in modern digital ASIC or FPGA flows the physical gates are considered the atomic build blocks of the logic circuit.

1. Settings → Apps → Apps & features → Related settings, Programs and Features → Turn Windows features on or off → Windows Subsystem for Linux
2. Microsoft Store → INSTALL UBUNTU

Front-End and Back-End Library type:

```
sudo apt update
sudo apt upgrade

sudo apt install bison cmake flex freeglut3-dev libcairo2-dev libgsl-dev \
libncurses-dev libx11-dev m4 python-tk python3-tk swig tcl tcl-dev tk-dev tcsh
```

Synthesizer Library type:

```
sudo apt update
sudo apt upgrade

sudo apt -y install build-essential clang bison flex \
libreadline-dev gawk tcl-dev libffi-dev git make gnat \
graphviz xdot pkg-config python3 libboost-system-dev \
libboost-python-dev libboost-filesystem-dev zlib1g-dev
```

4.1. FRONT-END OPEN SOURCE TOOLS

4.1.1. Modeling System Level of Hardware

A System Description Language Editor is a computer tool that allows to generate software code. A System Description Language is a formal language, which comprises a Programming Language (input), producing a Hardware Description (output). Programming languages are used in computer programming to implement

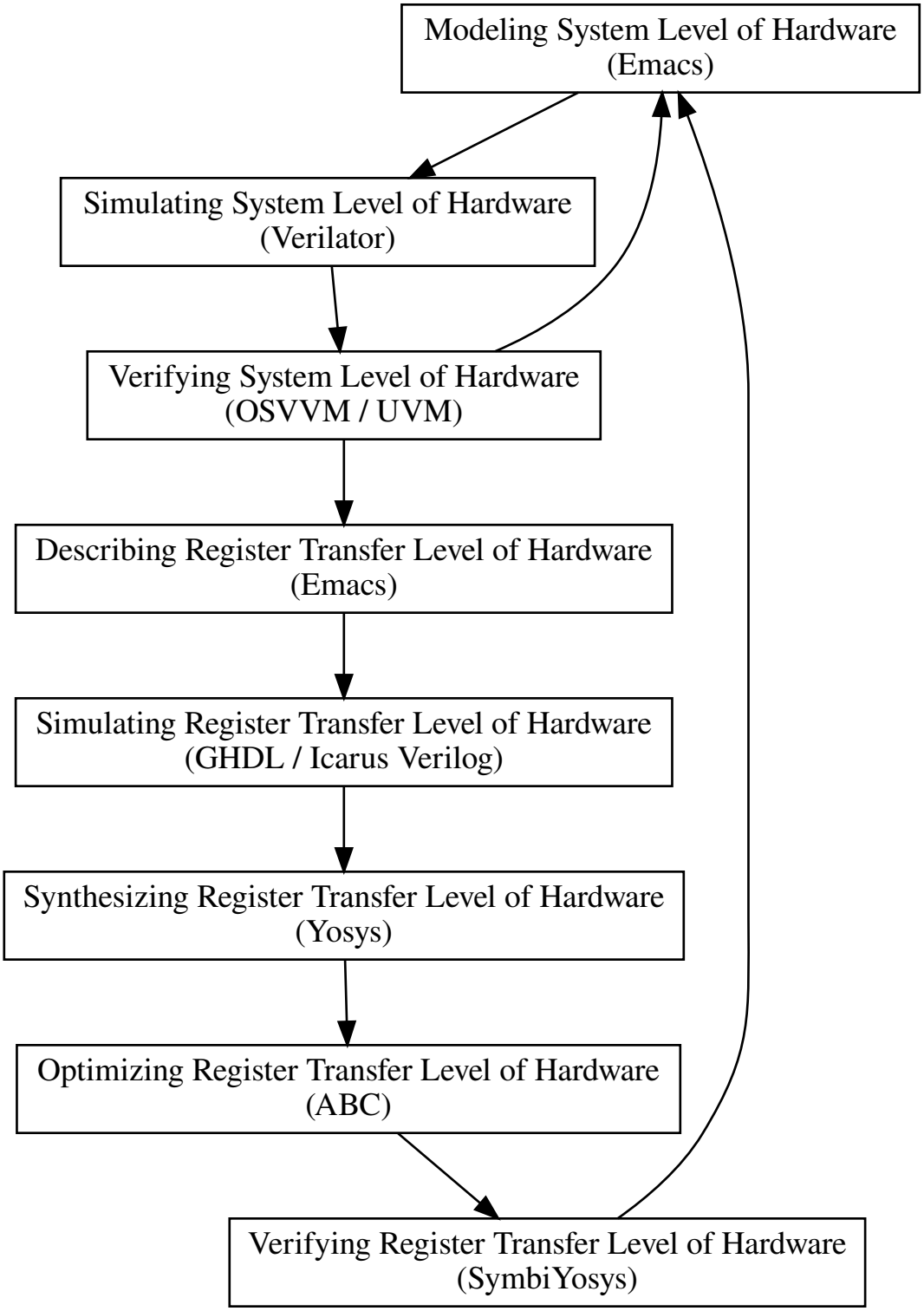

Figure 5: Front-End

algorithms. The description of a programming language is split into the two components of syntax (form) and semantics (meaning).

System Description Language Editor

type:

```
git clone https://github.com/emacs-mirror/emacs
```

4.1.2. Simulating System Level of Hardware

A System Description Language Simulator (translator) is a computer program that translates computer code written in a Programming Language (the source language) into a Hardware Description Language (the target language). The compiler is primarily used for programs that translate source code from a high-level programming language to a low-level language to create an executable program.

SystemVerilog System Description Language Simulator

type:

```
git clone http://git.veripool.org/git/verilator

cd verilator
autoconf
./configure
make
sudo make install

cd sim/verilog/regression/wb/vtor
source SIMULATE-IT

cd sim/verilog/regression/ahb3/vtor
source SIMULATE-IT

cd sim/verilog/regression/axi4/vtor
source SIMULATE-IT
```

4.1.3. Verifying System Level of Hardware

A UVM standard improves interoperability and reduces the cost of repurchasing and rewriting IP for each new project or Electronic Design Automation tool. It also makes it easier to reuse verification components. The UVM Class Library provides generic utilities, such as component hierarchy, Transaction Library Model or configuration database, which enable the user to create virtually any structure wanted for the testbench.

SystemVerilog System Description Language Verifier

type:

```
git clone https://github.com/QueenField/UVM
```

4.1.4. Describing Register Transfer Level of Hardware

A Hardware Description Language Editor is any editor that allows to generate hardware code. Hardware Description Language is a specialized computer language used to describe the structure and behavior of digital logic circuits. It allows for the synthesis of a HDL into a netlist, which can then be synthesized, placed and routed to produce the set of masks used to create an integrated circuit.

Hardware Description Language Editor

type:

```
git clone https://github.com/emacs-mirror/emacs
```

4.1.5. Simulating Register Transfer Level of Hardware

A Hardware Description Language Simulator uses mathematical models to replicate the behavior of an actual hardware device. Simulation software allows for modeling of circuit operation and is an invaluable analysis tool. Simulating a circuit's behavior before actually building it can greatly improve design efficiency by making faulty designs known as such, and providing insight into the behavior of electronics circuit designs.

VHDL Hardware Description Language Simulator

type:

```
git clone https://github.com/ghdl/ghdl

cd ghdl
./configure --prefix=/usr/local
make
sudo make install

cd sim/vhdl/regression/wb/ghdl
source SIMULATE-IT

cd sim/vhdl/regression/ahb3/ghdl
source SIMULATE-IT

cd sim/vhdl/regression/axi4/ghdl
source SIMULATE-IT
```

Verilog Hardware Description Language Simulator

type:

```
git clone https://github.com/steveicarus/iverilog

cd iverilog
sh autoconf.sh
./configure
make
sudo make install

cd sim/verilog/regression/wb/iverilog
source SIMULATE-IT

cd sim/verilog/regression/ahb3/iverilog
source SIMULATE-IT

cd sim/verilog/regression/axi4/iverilog
source SIMULATE-IT
```

4.1.6. Synthesizing Register Transfer Level of Hardware

A Hardware Description Language Synthesizer turns a RTL implementation into a Logical Gate Level implementation. Logical design is a step in the standard design cycle in which the functional design of an electronic circuit is converted into the representation which captures logic operations, arithmetic operations, control flow, etc. In EDA parts of the logical design is automated using synthesis tools based on the behavioral description of the circuit.

Verilog Hardware Description Language Synthesizer

type:

```
git clone https://github.com/YosysHQ/yosys
```

```
cd yosys
make
sudo make install
```

```
cd synthesis/yosys
source SYNTHESIZE-IT
```

VHDL Hardware Description Language Synthesizer

type for Plugin:

```
git clone https://github.com/ghdl/ghdl-yosys-plugin
```

```
cd ghdl-yosys-plugin
make GHDL=/usr/local
sudo yosys-config --exec mkdir -p --datdir/plugins
sudo yosys-config --exec cp "ghdl.so" --datdir/plugins/ghdl.so
```

```
cd synthesis/yosys
source SYNTHESIZE-IT
```

4.1.7. Optimizing Register Transfer Level of Hardware

A Hardware Description Language Optimizer finds an equivalent representation of the specified logic circuit under specified constraints (minimum area, pre-specified delay). This tool combines scalable logic optimization based on And-Inverter Graphs (AIGs), optimal-delay DAG-based technology mapping for look-up tables and standard cells, and innovative algorithms for sequential synthesis and verification.

Verilog Hardware Description Language Optimizer

type:

```
git clone https://github.com/YosysHQ/yosys
```

```
cd yosys
make
sudo make install
```

```
cd synthesis/yosys
source SYNTHESIZE-IT
```

4.1.8. Verifying Register Transfer Level of Hardware

A Hardware Description Language Verifier proves or disproves the correctness of intended algorithms underlying a hardware system with respect to a certain formal specification or property, using formal methods of mathematics. Formal verification uses modern techniques (SAT/SMT solvers, BDDs, etc.) to prove correctness by essentially doing an exhaustive search through the entire possible input space (formal proof).

Verilog Hardware Description Language Verifier

type:

```
git clone https://github.com/YosysHQ/SymbiYosys
```

4.2. BACK-END OPEN SOURCE TOOLS

Library

type:

```
sudo apt update
sudo apt upgrade
```

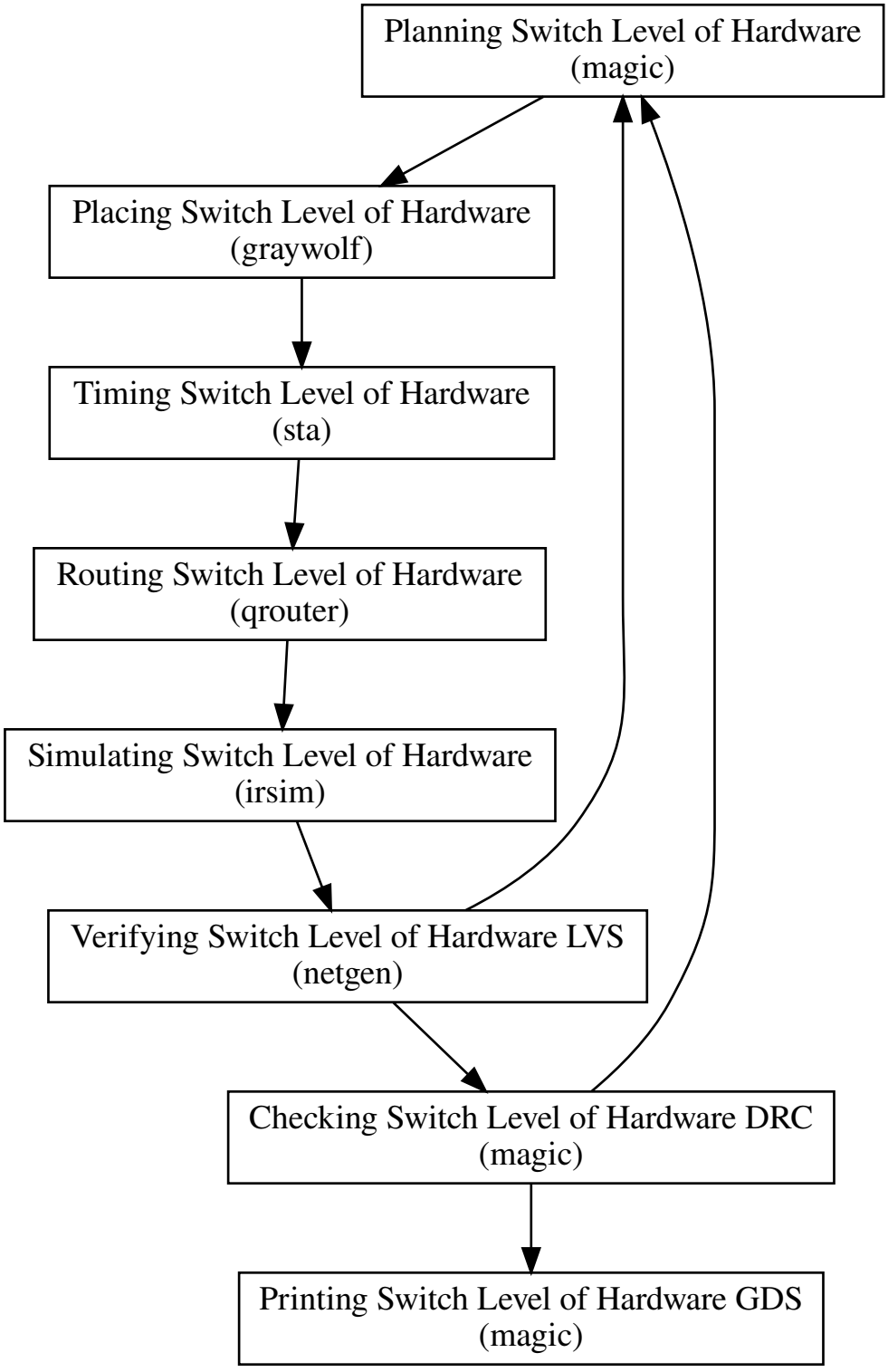

Figure 6: Back-End

```
sudo apt install bison cmake flex freeglut3-dev libcairo2-dev libgsl-dev \
libncurses-dev libx11-dev m4 python-tk python3-tk swig tcl tcl-dev tk-dev tcsh
```

Back-End Workflow Qflow

type:

```
git clone https://github.com/RTimothyEdwards/qflow

cd qflow
./configure
make
sudo make install
```

```
mkdir qflow
cd qflow
```

4.2.1. Planning Switch Level of Hardware

A Floor-Planner of an Integrated Circuit (IC) is a schematic representation of tentative placement of its major functional blocks. In modern electronic design process floor-plans are created during the floor-planning design stage, an early stage in the hierarchical approach to Integrated Circuit design. Depending on the design methodology being followed, the actual definition of a floor-plan may differ.

Floor-Planner

type:

```
git clone https://github.com/RTimothyEdwards/magic

cd magic
./configure
make
sudo make install
```

4.2.2. Placing Switch Level of Hardware

A Standard Cell Placer takes a given synthesized circuit netlist together with a technology library and produces a valid placement layout. The layout is optimized according to the aforementioned objectives and ready for cell resizing and buffering, a step essential for timing and signal integrity satisfaction. Physical design flow are iterated a number of times until design closure is achieved.

Standard Cell Placer

type:

```
git clone https://github.com/rubund/graywolf

cd graywolf
mkdir build
cd build
cmake ..
make
sudo make install
```

4.2.3. Timing Switch Level of Hardware

A Standard Cell Timing-Analizer is a simulation method of computing the expected timing of a digital circuit without requiring a simulation of the full circuit. High-performance integrated circuits have traditionally been

characterized by the clock frequency at which they operate. Measuring the ability of a circuit to operate at the specified speed requires an ability to measure, during the design process, its delay at numerous steps.

Standard Cell Timing-Analizer

type:

```
git clone https://github.com/The-OpenROAD-Project/OpenSTA

cd OpenSTA
mkdir build
cd build
cmake ..
make
sudo make install
```

4.2.4. Routing Switch Level of Hardware

A Standard Cell Router takes pre-existing polygons consisting of pins on cells, and pre-existing wiring called pre-routes. Each of these polygons are associated with a net. The primary task of the router is to create geometries such that all terminals assigned to the same net are connected, no terminals assigned to different nets are connected, and all design rules are obeyed.

Standard Cell Router

type:

```
git clone https://github.com/RTimothyEdwards/qrouter

cd qrouter
./configure
make
sudo make install
```

4.2.5. Simulating Switch Level of Hardware

A Standard Cell Simulator treats transistors as ideal switches. Extracted capacitance and lumped resistance values are used to make the switch a little bit more realistic than the ideal, using the RC time constants to predict the relative timing of events. This simulator represents a circuit in terms of its exact transistor structure but describes the electrical behavior in a highly idealized way.

Standard Cell Simulator

type:

```
git clone https://github.com/RTimothyEdwards/irsim

cd irsim
./configure
make
sudo make install
```

4.2.6. Verifying Switch Level of Hardware LVS

A Standard Cell Verifier compares netlists, a process known as LVS (Layout vs. Schematic). This step ensures that the geometry that has been laid out matches the expected circuit. The greatest need for LVS is in large analog or mixed-signal circuits that cannot be simulated in reasonable time. LVS can be done faster than simulation, and provides feedback that makes it easier to find errors.

Standard Cell Verifier

type:

```
git clone https://github.com/RTimothyEdwards/netgen

cd netgen
./configure
make
sudo make install
```

```
cd synthesis/qflow
source FLOW-IT
```

4.2.7. Checking Switch Level of Hardware DRC

A Standard Cell Checker is a geometric constraint imposed on Printed Circuit Board (PCB) and Integrated Circuit (IC) designers to ensure their designs function properly, reliably, and can be produced with acceptable yield. Design Rules for production are developed by hardware engineers based on the capability of their processes to realize design intent. Design Rule Checking (DRC) is used to ensure that designers do not violate design rules.

Standard Cell Checker

type:

```
git clone https://github.com/RTimothyEdwards/magic

cd magic
./configure
make
sudo make install
```

4.2.8. Printing Switch Level of Hardware GDS

A Standard Cell Editor allows to print a set of standard cells. The standard cell methodology is an abstraction, whereby a low-level VLSI layout is encapsulated into a logical representation. A standard cell is a group of transistor and interconnect structures that provides a boolean logic function (AND, OR, XOR, XNOR, inverters) or a storage function (flipflop or latch).

Standard Cell Editor

type:

```
git clone https://github.com/RTimothyEdwards/magic

cd magic
./configure
make
sudo make install
```

5. SOFTWARE WORKFLOW

5.1. BACK-END OPEN SOURCE TOOLS

type:

```
sudo apt install autoconf automake autotools-dev curl python3 libmpc-dev \
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf \
libtool patchutils bc zlib1g-dev libexpat-dev
```

5.1.1. MSP430

5.1.1.1. MSP430 GNU C/C++

5.1.1.2. MSP430 GNU Go

5.1.2. OpenRISC

5.1.2.1. OpenRISC GNU C/C++

5.1.2.2. OpenRISC GNU Go

5.1.3. RISC-V

5.1.3.1. RISC-V GNU C/C++ type:

```
git clone --recursive https://github.com/riscv/riscv-gnu-toolchain

cd riscv-gnu-toolchain

./configure --prefix=/opt/riscv-elf-gcc
sudo make clean
sudo make

./configure --prefix=/opt/riscv-elf-gcc
sudo make clean
sudo make linux

./configure --prefix=/opt/riscv-elf-gcc --enable-multilib
sudo make clean
sudo make linux
```

5.1.3.2. RISC-V GNU Go type:

```
git clone --recursive https://go.googlesource.com/go riscv-go
cd riscv-go/src
./all.bash
cd ../..
sudo mv riscv-go /opt
```

5.2. FRONT-END OPEN SOURCE TOOLS

5.2.1. MSP430

5.2.2. OpenRISC

5.2.3. RISC-V

type:

```
sudo apt install device-tree-compiler libglib2.0-dev libpixman-1-dev pkg-config
```

5.2.3.1. Hardware Engineers Compiler: Spike Building Proxy Kernel

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

git clone --recursive https://github.com/riscv/riscv-pk
```

```
cd riscv-pk
mkdir build
cd build
../configure --prefix=/opt/riscv-elf-gcc --host=riscv64-unknown-elf
make
sudo make install
```

Building Spike

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

git clone --recursive https://github.com/riscv/riscv-isa-sim

cd riscv-isa-sim
mkdir build
cd build
../configure --prefix=/opt/riscv-elf-gcc
make
sudo make install
```

5.2.3.2. Software Engineers Compiler: QEMU

type:

```
export PATH=/opt/riscv-elf-gcc/bin:${PATH}

git clone --recursive https://github.com/qemu/qemu

cd qemu
./configure --prefix=/opt/riscv-elf-gcc \
--target-list=riscv64-softmmu,riscv32-softmmu,riscv64-linux-user,riscv32-linux-user
make
sudo make install
```