1. INTRODUCTION

A Processing Unit (PU) is an electronic system within a computer that carries out instructions of a program by performing the basic arithmetic, logic, controlling, and I/O operations specified by instructions. Instruction-level parallelism is a measure of how many instructions in a computer can be executed simultaneously. The PU is contained on a single Metal Oxide Semiconductor (MOS) Integrated Circuit (IC).

2. PROJECTS

2.1. CORE-RISCV

2.1.1. Definition

The RISC-V implementation has a 32/64/128 bit Microarchitecture, 6 stages data pipeline and an Instruction Set Architecture based on Reduced Instruction Set Computer. Compatible with AMBA and Wishbone Buses. For Researching and Developing.

Processing Unit	Module description
riscv_pu	Processing Unit
. . . riscv_core	Core
. . . riscv_imem_ctrl	Instruction Memory Access Block
. . . riscv_biu - imem	Bus Interface Unit (Instruction)
. . . riscv_dmem_ctrl	Data Memory Access Block
. . . riscv_biu - dmem	Bus Interface Unit (Data)

2.1.2. RISC Pipeline

In computer science, instruction pipelining is a technique for implementing instruction-level parallelism within a PU. Pipelining attempts to keep every part of the processor busy with some instruction by dividing incoming instructions into a series of sequential steps performed by different PUs with different parts of instructions processed in parallel. It allows faster PU throughput than would otherwise be possible at a given clock rate.

Typical	Modified	Module
FETCH	FETCH	riscv_if
. . .	PRE-DECODE	riscv_id
DECODE	DECODE	riscv_id

Typical	Modified	Module
EXECUTE	EXECUTE	riscv_execution
MEMORY	MEMORY	riscv_memory
WRITE-BACK	WRITE-BACK	riscv_wb

- IF – Instruction Fetch Unit : Send out the PC and fetch the instruction from memory into the Instruction Register (IR); increment the PC to address the next sequential instruction. The IR is used to hold the next instruction that will be needed on subsequent clock cycles; likewise the register NPC is used to hold the next sequential PC.
- ID – Instruction Decode Unit : Decode the instruction and access the register file to read the registers. This unit gets instruction from IF, and extracts opcode and operand from that instruction. It also retrieves register values if requested by the operation.
- EX – Execution Unit : The ALU operates on the operands prepared in prior cycle, performing one functions depending on instruction type.
- MEM – Memory Access Unit : Instructions active in this unit are loads, stores and branches.
- WB – WriteBack Unit : Write the result into the register file, whether it comes from the memory system or from the ALU.

2.1.3. CORE-RISCV Organization

The CORE-RISCV is based on the Harvard architecture, which is a computer architecture with separate storage and signal pathways for instructions and data. The implementation is heavily modular, with each particular functional block of the design being contained within its own HDL module or modules. The RISCV implementation was developed in order to provide a better platform for processor component development than previous implementations.

Core	Module description
riscv_core	Core
... riscv_if	Instruction Fetch
... riscv_id	Instruction Decoder
... riscv_execution	Execution Unit
.....riscv_alu	Arithmetic & Logical Unit
.....riscv_lsu	Load Store Unit
.....riscv_bu	Branch Unit
.....riscv_mul	Multiplier Unit
.....riscv_div	Division Unit
... riscv_memory	Memory Unit

Core	Module description
...riscv_wb	Data Memory Access (Write Back)
...riscv_state	State Unit
...riscv_rf	Register File
...riscv_bp	Correlating Branch Prediction Unit
.....riscv_ram_1r1w	RAM 1RW1
.....riscv_ram_1r1w_generic	RAM 1RW1 Generic
...riscv_du	Debug Unit

In a Harvard architecture, there is no need to make the two memories share characteristics. In particular, the word width, timing, implementation technology, and memory address structure can differ. In some systems, instructions for pre-programmed tasks can be stored in read-only memory while data memory generally requires read-write memory. In some systems, there is much more instruction memory than data memory so instruction addresses are wider than data addresses.

2.1.4. Parameters

Parameter	Type	Default	Description
JEDEC_BANK	Integer	0x0A	JEDEC Bank
JEDEC_MANUFACTURER_ID	Integer	0x6E	JEDEC Manufacturer ID
XLEN	Integer	64	Data Path Width
PLEN	Integer	64	Physical Memory Address Size
PMP_CNT	Integer	16	Physical Memory Protection Entries
PMA_CNT	Integer	16	Physical Menory Attribute Entries
HAS_USER	Integer	1	User Mode Enable
HAS_SUPER	Integer	1	Supervisor Mode Enable
HAS_HYPER	Integer	1	Hypervisor Mode Enable
HAS_RVM	Integer	1	“M” Extension Enable
HAS_RVA	Integer	1	“A” Extension Enable
HAS_RVC	Integer	1	“C” Extension Enable
HAS_BPU	Integer	1	Branch Prediction Unit Control Enable
IS_RV32E	Integer	0	Base Integer Instruction Set Enable
MULT_LATENCY	Integer	1	Hardware Multiplier Latency
ICACHE_SIZE	Integer	16	Instruction Cache size
ICACHE_BLOCK_SIZE	Integer	64	Instruction Cache block length
ICACHE_WAYS	Integer	2	Instruction Cache associativity
ICACHE_REPLACE_ALG	Integer	0	Instruction Cache replacement
DCACHE_SIZE	Integer	16	Data Cache size
DCACHE_BLOCK_SIZE	Integer	64	Data Cache block length
DCACHE_WAYS	Integer	2	Data Cache associativity
DCACHE_REPLACE_ALG	Integer	0	Data Cache replacement algorithm

Parameter	Type	Default	Description
HARTID	Integer	0	Hart Identifier
PC_INIT	Address	'h200	Program Counter Initialisation Vector
MNMIVEC_DEFAULT	Address	PC_INIT-'h004	Machine Mode Non-Maskable
MTVEC_DEFAULT	Address	PC_INIT-'h040	Machine Mode Interrupt Address
HTVEC_DEFAULT	Address	PC_INIT-'h080	Hypervisor Mode Interrupt Address
STVEC_DEFAULT	Address	PC_INIT-'h0C0	Supervisor Mode Interrupt Address
UTVEC_DEFAULT	Address	PC_INIT-'h100	User Mode Interrupt Address
BP_LOCAL_BITS	Integer	10	Number of local predictor bits
BP_GLOBAL_BITS	Integer	2	Number of global predictor bits
BREAKPOINTS	Integer	3	Number of hardware breakpoints
TECHNOLOGY	String	GENERIC	Target Silicon Technology

2.1.5. Instruction INPUTS/OUTPUTS Bus

Port	Size	Direction	Description
ins_stb	1	Input	Strobe
ins_stb_ack	1	Output	Strobe acknowledge
ins_d_ack	1	Output	Data acknowledge
ins_adri	PLEN	Input	Start address
ins_adro	PLEN	Output	Response address
ins_size	3	Input	Syze
ins_type	3	Input	Type
ins_prot	3	Input	Protection
ins_lock	1	Input	Locked access
ins_d	XLEN	Input	Write data
ins_q	XLEN	Output	Read data
ins_ack	1	Output	Acknowledge
ins_err	1	Output	Error

2.1.6. Data INPUTS/OUTPUTS Bus

Port	Size	Direction	Description
dat_stb	1	Input	Strobe
dat_stb_ack	1	Output	Strobe acknowledge
dat_d_ack	1	Output	Data acknowledge
dat_adri	PLEN	Input	Start address
dat_adro	PLEN	Output	Response address
dat_size	3	Input	Syze
dat_type	3	Input	Type
dat_prot	3	Input	Protection

Port	Size	Direction	Description
dat_lock	1	Input	Locked access
dat_d	XLEN	Input	Write data
dat_q	XLEN	Output	Read data
dat_ack	1	Output	Acknowledge
dat_err	1	Output	Error

2.2. INSTRUCTION CACHE

A PU cache is a hardware cache used by the PU to reduce the average cost (time or energy) to access instruction/data from the main memory. A cache is a smaller, faster memory, closer to a core, which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches.

2.2.1. Instruction Organization

Instruction Memory	Module description
riscv_imem_ctrl	Instruction Memory Access Block
... riscv_membuf	Memory Access Buffer
.....riscv_ram_queue	Fall-through Queue
... riscv_memmisaligned	Misalignment Check
... riscv_mmu	Memory Management Unit
... riscv_pmachk	Physical Memory Attributes Checker
... riscv_pmpchk	Physical Memory Protection Checker
... riscv_icache_core	Instruction Cache (Write Back)
.....riscv_ram_1rw	RAM 1RW
..... .riscv_ram_1rw_generic	RAM 1RW Generic
... riscv_dext	Data External Access Logic
... riscv_ram_queue	Fall-through Queue
... riscv_mux	Bus-Interface-Unit Mux
riscv_biu	Bus Interface Unit

2.2.2 Instruction INPUTS/OUTPUTS AMBA4 AXI-Lite Bus

2.2.2.1. 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

Write Port	Read Port	Size	Direction	Description
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

2.2.2.2. 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

2.2.2.3. 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

2.2.3. Instruction INPUTS/OUTPUTS AMBA3 AHB-Lite Bus

Port	Size	Direction	Description
HRESETn	1	Input	Asynchronous Active Low Reset
HCLK	1	Input	System Clock Input

Port	Size	Direction	Description
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

2.2.4. 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
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

2.3. DATA CACHE

2.3.1. Data Organization

Data Memory	Module description
riscv_dmem_ctrl	Data Memory Access Block
. . . riscv_membuf	Memory Access Buffer
. . . . .riscv_ram_queue	Fall-through Queue
. . . riscv_memmisaligned	Misalignment Check
. . . riscv_mmu	Memory Management Unit

Data Memory	Module description
...riscv_pmachk	Physical Memory Attributes Checker
...riscv_pmpchk	Physical Memory Protection Checker
...riscv_dcache_core	Data Cache (Write Back)
.....riscv_ram_1rw	RAM 1RW
.....riscv_ram_1rw_generic	RAM 1RW Generic
...riscv_dext	Data External Access Logic
...riscv_mux	Bus-Interface-Unit Mux
riscv_biu	Bus Interface Unit

2.3.2. Data INPUTS/OUTPUTS AMBA4 AXI-Lite Bus

2.3.2.1. 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

2.3.2.2. 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

2.3.2.3. 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

2.3.3. 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

2.3.4. 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

Port	Size	Direction	Description
dack	1	Output	Bus Cycle Acknowledge Output
derr	1	Output	Bus Cycle Error Output
dint	1	Output	Interrupt Signal Output

2.4. RISC-V ARCHITECTURE

2.4.1. Library

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
```

2.4.2. Toolchain

type:

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

cd riscv-gnu-toolchain

./configure --prefix=/opt/riscv
sudo make

./configure --prefix=/opt/riscv
sudo make linux

./configure --prefix=/opt/riscv --enable-multilib
sudo make linux

./configure --prefix=$RISCV
sudo make linux
sudo make report-linux
```

2.4.3. Software

type:

```
export PATH=$PATH:/opt/riscv/bin

cd software
```

```
rm -rf tests
rm -rf riscv-tests

mkdir tests
mkdir tests/dump
mkdir tests/hex

git clone --recursive https://github.com/riscv/riscv-tests
cd riscv-tests

autoconf
./configure --prefix=/opt/riscv/bin
make

cd isa

source ../../elf2hex.sh

mv *.dump ../../tests/dump
mv *.hex ../../tests/hex

cd ..

make clean
```

3. 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.

3.1. FRONT-END OPEN SOURCE TOOLS

3.1.1. Modeling System Level of Hardware

A System Description Language Editor is a computer tool 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 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
```

3.1.2. 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
```

3.1.3. 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
```

3.1.4. 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
```

3.1.5. 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
```

3.1.6. 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
```

3.1.7. 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
```

3.1.8. 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
```

3.2. BACK-END OPEN SOURCE TOOLS

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
```

Back-End Workflow Qflow

type:

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

cd qflow
./configure
make
sudo make install

mkdir qflow
cd qflow
```

3.2.1. 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
```

3.2.2. 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
```

3.2.3. 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
```

3.2.4. 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
```

3.2.5. 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
```

3.2.6. 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
```

3.2.7. 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
```

3.2.8. 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
```

4. CONCLUSION

4.1. FOR WINDOWS USERS!

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

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
```

4.1.1. Front-End

type:

```
sudo apt install verilator
sudo apt install iverilog
sudo apt install ghdl

cd /mnt/c/../sim/verilog/regression/wb/iverilog
source SIMULATE-IT

sudo apt install yosys

cd /mnt/c/../synthesis/yosys
source SYNTHESIZE-IT
```

4.1.2. Back-End

type:

```
mkdir qflow
cd qflow

git clone https://github.com/RTimothyEdwards/magic
git clone https://github.com/rubund/graywolf
git clone https://github.com/The-OpenROAD-Project/OpenSTA
git clone https://github.com/RTimothyEdwards/qrouter
git clone https://github.com/RTimothyEdwards/irsim
git clone https://github.com/RTimothyEdwards/netgen
git clone https://github.com/RTimothyEdwards/qflow

cd /mnt/c/../synthesis/qflow
source FLOW-IT
```