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-OR1K

2.1.1. Definition

The OpenRISC implementation has a 32/64 bit Microarchitecture, 5 stages data pipeline and an Instruction Set Architecture based on Reduced Instruction Set Computer. Compatible with Wishbone Bus. Only For Researching.

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	or1k_cache_lru
		or1k_fetch_cappuccino
		or1k_icache
		or1k_immu
DECODE	DECODE	or1k_decode
EXECUTE & CONTROL	EXECUTE & WRITE-BACK	or1k_execute_alu
		or1k_execute_ctrl_cappuccino
		or1k_rf_cappuccino
		or1k_wb_mux_cappuccino
MEMORY	MEMORY	or1k_dcache
		or1k_dmmu
		or1k_lsu_cappuccino
		or1k_store_buffer

Typical	Modified	Module
WRITE-BACK	CONTROL	or1k_cfgrs or1k_ctrl_cappuccino or1k_pcu or1k_pic or1k_ticktimer

- 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-OR1K Organization

The CORE-OR1K 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 OR1K implementation was developed in order to provide a better platform for processor component development than previous implementations.

Core	Module description
or1k_core	Top-level, instantiatng bus interfaces, data cache and CPU
. . . or1k_dcache	Data cache implementation
. . . or1k_bus_if_xx	Bus interface, depending on desired bus standard
. . . or1k_cpu	Pipeline implementation wrapper
. . . ..or1k_cpu_xx	Pipeline implementation, depending on configuration
. . . . . .or1k_fetch_xx	Pipeline-implementation-dependent fetch stage
. . . . . .or1k_decode	Generic decode stage
. . . . . .or1k_execute_alu	Generic ALU for execute stage

Core	Module description
.....or1k_lsu_xx	Pipeline-implementation-dependent load/store unit
.....or1k_wb_mux_xx	Pipeline-implementation-dependent writeback stage mux
.....or1k_rf_xx	Pipeline-implementation-dependent register file
.....or1k_ctrl_xx	Pipeline-implementation-dependent control stage

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

2.1.4.1. Basic Parameters

Parameter	Description	Default	Values
OPTION_OPERAND_WIDTH	CPU data and address widths	32	32, 64
OPTION_CPU0	CPU pipeline core	`CAPPUCCINO`	`CAPPUCCINO`
OPTION_RESET_PC	Program Counter upon reset	`0x100`	N

2.1.4.2. Caching Parameters

Parameter	Description	Default	Values
FEATURE_DATACACHE	Enable memory access data caching	`NONE`	`ENABLED`
OPTION_DCACHE_BLOCK_WIDTH	Address width of a cache block	5	`n`
OPTION_DCACHE_SET_WIDTH	Set address width	9	`n`
OPTION_DCACHE_WAYS	Number of blocks per set	2	`n`
OPTION_DCACHE_LIMIT_WIDTH	Maximum address width	32	`n`
OPTION_DCACHE_SNOOP	Bus snooping for cache coherency	`NONE`	`ENABLED`
FEATURE_INSTRUCTIONCACHE	Memory access instruction caching	`NONE`	`ENABLED`
OPTION_ICACHE_BLOCK_WIDTH	Address width of a cache block	5	`n`
OPTION_ICACHE_SET_WIDTH	Set address width	9	`n`
OPTION_ICACHE_WAYS	Number of blocks per set	2	`n`
OPTION_ICACHE_LIMIT_WIDTH	Maximum address width	32	`n`

2.1.4.3. Memory Management Unit (MMU) Parameters

Parameter	Description	Default	Values
FEATURE_DMMU	Enable the data bus MMU	NONE	ENABLED
FEATURE_DMMU_HW_TLB_RELOAD	Enable hardware TLB reload	NONE	ENABLED
OPTION_DMMU_SET_WIDTH	Set address width	6	n
OPTION_DMMU_WAYS	Number of ways per set	1	n
FEATURE_IMMU	Enable the instruction bus MMU	NONE	ENABLED
FEATURE_IMMU_HW_TLB_RELOAD	Enable hardware TLB reload	NONE	ENABLED
OPTION_IMMU_SET_WIDTH	Set address width	6	n
OPTION_IMMU_WAYS	Number of ways per set	1	n

2.1.4.4. System Bus Parameters

Parameter	Description	Default
FEATURE_STORE_BUFFER	Load store unit store buffer	ENABLED
OPTION_STORE_BUFFER_DEPTH_WIDTH	Load store unit store buffer depth	8
BUS_IF_TYPE	Bus interface type	WISHBONE32
IBUS_WB_TYPE	Instruction bus interface	B3_READ_BURSTING
DBUS_WB_TYPE	Data bus interface type option	CLASSIC

2.1.4.5. Hardware Unit Configuration Parameters

Parameter	Description	Default
FEATURE_TRACEPORT_EXEC	Traceport hardware interface	NONE
FEATURE_DEBUGUNIT	Hardware breakpoints and debug unit	NONE
FEATURE_PERFCOUNTERS	Performance counters unit	NONE
OPTION_PERFCOUNTERS_NUM	Performance counters to generate	0
FEATURE_TIMER	Internal OpenRISC timer	ENABLED
FEATURE_PIC	Internal OpenRISC PIC	ENABLED
OPTION_PIC_TRIGGER	PIC trigger mode	LEVEL
OPTION_PIC_NMI_WIDTH	Non maskable interrupts width	0
OPTION_RF_CLEAR_ON_INIT	clearing all registers on initialization	0
OPTION_RF_NUM_SHADOW_GPR	Number of shadow register files	0
OPTION_RF_ADDR_WIDTH	Address width of the register file	5
OPTION_RF_WORDS	Number of registers in the register file	32
FEATURE_FASTCONTEXTS	Fast context switching of register sets	NONE
FEATURE_MULTICORE	`coreid` and `numcores` SPR registers	NONE
FEATURE_FPU	FPU, for cappuccino pipeline only	NONE
OPTION_FTOI_ROUNDING	Rounding behavior for `lf.ftoi.s`	CPP
FEATURE_BRANCH_PREDICTOR	Branch predictor implementation	SIMPLE

Note: *C/C++ double to integer conversion assumes truncation (rounding*

toward zero). The default (CPP) value of OPTION_FTOI_ROUNDING forces toward zero rounding mode exclusively for lf.ftoi.s instruction regardless of rounding mode bits of FPCSR. While with IEEE value lf.ftoi.s performs conversion in according with rounding mode bits of FPCSR. And these bits are nearest-even rounding mode by default. All other floating point instructions always perform rounding in according with rounding mode bits of FPCSR.

2.1.4.6. Exception Handling Options

Parameter	Description	Default
FEATURE_DSX	Enable setting the SR[DSX] flag	ENABLED
FEATURE_RANGE	Enable checking and raising range exceptions	ENABLED
FEATURE_OVERFLOW	Enable checking and raising overflow exceptions	ENABLED

2.1.4.7. ALU Configuration Options

Parameter	Description	Default
FEATURE_MULTIPLIER	Specify the multiplier implementation	THREESTAGE
FEATURE_DIVIDER	Specify the divider implementation	SERIAL
OPTION_SHIFTER	Specify the shifter implementation	BARREL
FEATURE_CARRY_FLAG	Enable checking and setting the carry flag	ENABLED

2.1.4.8. Instruction Enabling Options

Parameter	Description	Default
FEATURE_MAC	l.mac* multiply accumulate instructions	NONE
FEATURE_SYSCALL	l.sys OS syscall instruction	ENABLED
FEATURE_TRAP	l.trap instruction	ENABLED
FEATURE_ADDC	l.addc add with carry flag instruction	ENABLED
FEATURE_SRA	l.sra shirt right arithmetic instruction	ENABLED
FEATURE_ROR	l.ror* rotate right instructions	NONE
FEATURE_EXT	l.ext* sign extend instructions	NONE
FEATURE_CMOV	l.cmov conditional move instruction	ENABLED
FEATURE_FFL1	l.f[fl]1 find first/last set bit instructions	ENABLED
FEATURE_ATOMIC	l.lwa and l.swa atomic instructions	ENABLED
FEATURE_CUST1	l.cust* custom instruction	NONE
FEATURE_CUST2	l.cust* custom instruction	NONE
FEATURE_CUST3	l.cust* custom instruction	NONE
FEATURE_CUST4	l.cust* custom instruction	NONE
FEATURE_CUST5	l.cust* custom instruction	NONE
FEATURE_CUST6	l.cust* custom instruction	NONE

Parameter	Description	Default
FEATURE_CUST7	`l.cust*` custom instruction	`NONE`
FEATURE_CUST8	`l.cust*` custom instruction	`NONE`

2.2. OPENRISC ARCHITECTURE

2.2.1. Library

2.2.2. Toolchain

2.2.3. Software

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 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 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
```