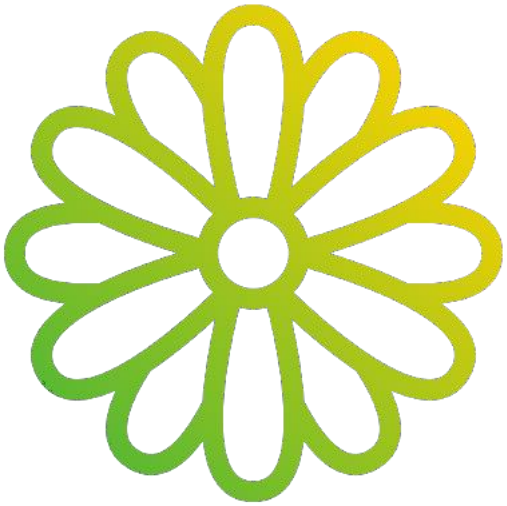
RINSC



Spring 2022

EEE 446 COMPUTER ARCHITECTURE II - Laboratory Report

MODULE #5

Programming & Verification of a Pipelined CPU with Basic Cache Hierarchy

Report Prepared by:

Barış Güzel 2315935

Ege Ereren 2152387

The content of the report represents the work completed by the submitting team only, and no material has been borrowed in any form.

THIS PAGE LEFT BLANK INTENTIONALLY

TABLE OF CONTENTS

1. OBJECTIVES

The objective of this module is to finalize the previously designed pipelined CPU. Any functionality issues and bugs will be fixed for the CPU to pass the benchmarks. Benchmarks will be written in the proposed ISA, and an assembler will be used to convert the code to hex. Verilator tests will be conducted to extract information such as CPI, miss rate, stalls, and power analysis.

2. INTRODUCTION

RINSC is a 32-bit 5-stage pipelined CPU with L1 cache and memory. RINSC stands for RINSC, is not so complex. A recursive abbreviation inspired by GNU. RINSC has 2 ALUs and forwarding capabilities to decrease CPI. D cache is 512bytes, I cache is 256bytes, while the main memory is 4095kB. This final module design is tested with three benchmarks, all making use of the RINSC's capabilities. An ISA is prepared with the CPU called Papatya. Final updates and benchmarks results are shared in this module.

3. DESIGN

RINSC, as the name stands, is designed so that new functionalities and advantages are earned without increasing the complexity of the datapath and ISA. The main philosophy was doing everything efficiently, doing nothing very well. Only three benchmarks are considered while making design decisions, and the CPU is optimized to pass these benchmarks as efficiently as possible. Papatya ISA holds similarities with both RISC-V and MIPS ISAs. There might be better ISAs in terms of abstraction from the machine, but it was a tradeoff between simplicity. This hardware is split into two, datapath and control logic, and the CPU ISA decisions are explained in this chapter.

3.1 Datapath and Control

RINSC has a 5-stage (Fetch, Decode, Execute, Memory and Writeback) pipelined processor with cache hierarchy. A forwarding unit is implemented to prevent Data Hazards. Also, Hazard Detection Unit is implemented to insert stall and flush into the pipeline. ALU operations are defined at the high level. There is a second ALU in the Memory stage to execute the instruction with three registers(multiply-add).

This machine does not have structural hazards. Data and instruction memories are separate. Also, register file access, as discussed before in lab 2, write and read are done on opposite clock edges.

Cache hierarchy consists of one instruction cache, one data cache, and main memory. These units have their controllers. Controllers also communicate with the hazard detection unit in the CPU.

Hazard Detection is responsible for detecting load use data hazards and memory stalls. It was decided to use a flush in branch and jump operations to achieve better performance.

Stall and Flush logic could be implemented with different methods. For example, the most extensive benchmark, merge sort, has many branches. CPU has got some problems with forwarding after branch instructions. Due to some problems, memory and load use stalls can not be separated. Also, forwarding to branch instructions does not work correctly in some cases.

Forwarding Unit

Any destination register becomes a source register in the next cycle; data will be forwarded.

Forwarding A and B:

Forwarding A represents the input A of the first ALU. Forwarding B represents the input B of the first ALU. If the next instruction's destination becomes the source operand, Forwarding A or B should be active. One case is data is forwarded from data memory to ALU. The other case is data is forwarded from the first ALU's output to the first ALU's input.

Forwarding C:

If there is any dependency on the input of the second ALU, forwarding C will be active. For example, LW before MULA instruction and the last LW should be fed into the input of the second ALU.

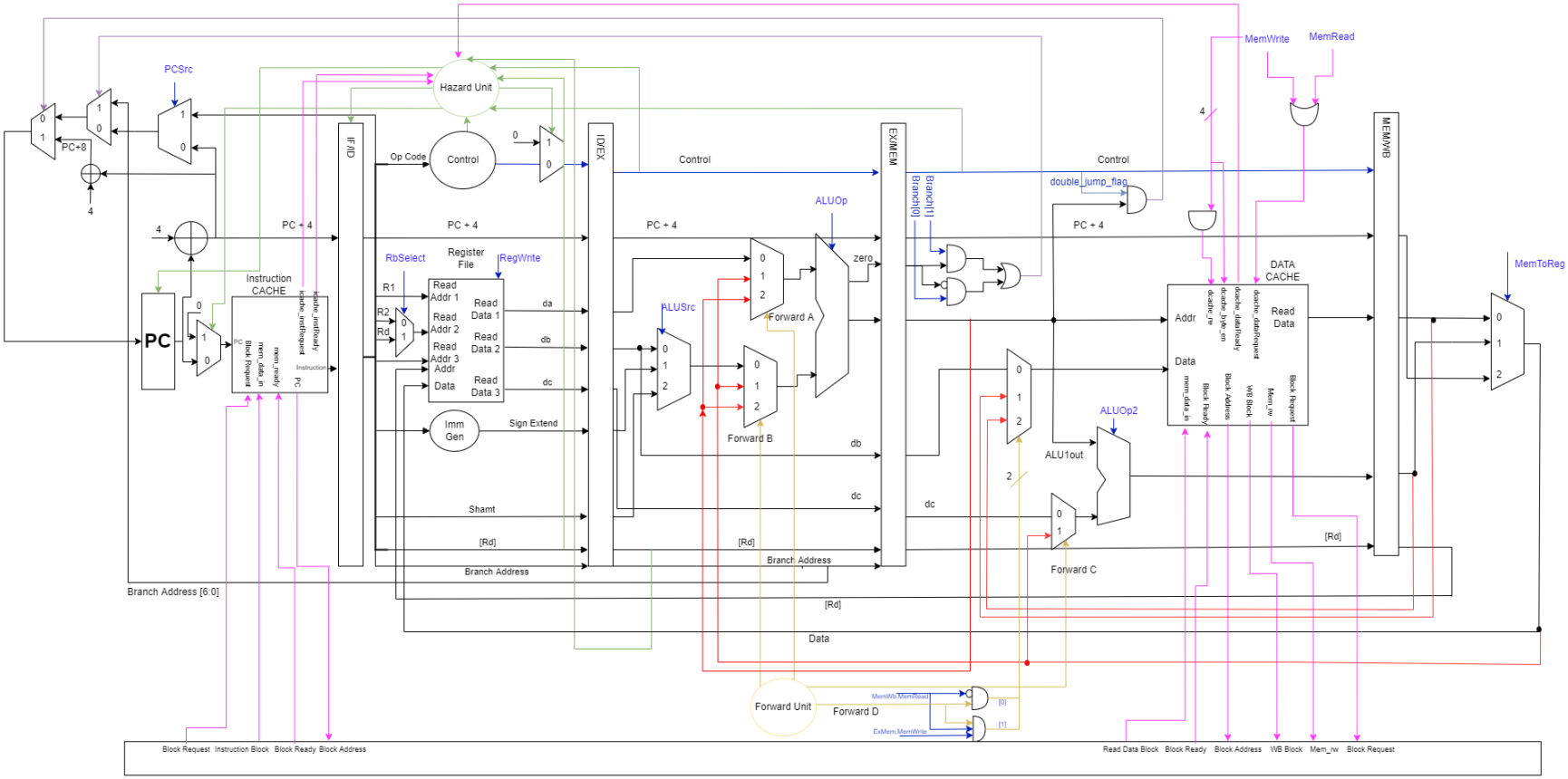
Forwarding D:

If there is memory to memory copies like load after store or add after store, forwarding D should be active. Takes the data from the output of the second ALU and puts it into the input of the data memory.

The only input to the control block is the opcode. The control block decides on the necessary signals with the opcode. Control signals are shown in blue fonts in the datapath figure.

Table 1 Control Signals of Each Instruction

Instr.	Type	AluSrc	AluOp	AluOp2	MemRead	MemWrite	RegWrite	MemToReg	RbSelect	PCSrc	MemSign Extend	Jump_flag	Branch Flag
NOP	R	00	0000	0	0000	0000	0	00	0	0	0	0	00
ADD	R	00	0000	0	0000	0000	0	01	0	0	0	0	00
SUB	R	00	0001	0	0000	0000	1	01	0	0	0	0	00
MUL	R	00	0010	0	0000	0000	1	01	0	0	0	0	00
AND	R	00	0101	0	0000	0000	1	01	0	0	0	0	00
OR	R	00	0100	0	0000	0000	1	01	0	0	0	0	00
XOR	R	00	0011	0	0000	0000	1	01	0	0	0	0	00
SLT	R	00	1001	0	0000	0000	1	01	0	0	0	0	00
SLL	R	10	0110	0	0000	0000	1	01	0	0	0	0	00
SRL	R	10	1000	0	0000	0000	1	01	0	0	0	0	00
SRA	R	10	0111	0	0000	0000	1	01	0	0	0	0	00
MULA	R	00	0010	1	0000	0000	1	01	0	0	0	0	00
SB	I	01	0000	0	0000	0001	0	00	1	0	0	0	00
SH	I	01	0000	0	0000	0011	0	00	1	0	0	0	00
SW	I	01	0000	0	0000	1111	0	00	1	0	0	0	00
LB	I	01	0000	0	0001	0000	1	00	1	0	0	0	00
LBS	I	01	0000	0	0001	0000	1	00	1	0	1	0	00
LH	I	01	0000	0	0011	0000	1	00	1	0	0	0	00
LHS	I	01	0000	0	0011	0000	1	00	1	0	1	0	00
LW	I	01	0000	0	1111	0000	1	00	1	0	0	0	00
BEQ	I	00	0001	0	0000	0000	0	00	1	0	0	0	01
BNE	I	00	0001	0	0000	0000	0	00	1	0	0	0	10
ADDI	I	01	0000	0	0000	0000	1	01	0	0	0	0	00
SUBI	I	01	0001	0	0000	0000	1	01	0	0	0	0	00
MULI	I	01	0010	0	0000	0000	1	01	0	0	0	0	00
ORI	I	01	0100	0	0000	0000	1	01	0	0	0	0	00
XORI	I	01	0011	0	0000	0000	1	01	0	0	0	0	00
ANDI	I	01	0101	0	0000	0000	1	01	0	0	0	0	00
SLTI	I	01	1001	0	0000	0000	1	01	0	0	0	0	00
JAL	J	00	0000	0	0000	0000	1	10	0	1	0	1	00



Main Memory

Figure 1 Datapath Schematic

3.2 ISA

This ISA has fixed 32-bit instructions. It has 32 bits wide registers, and all registers are defined as 5-bit address fields. There are three main instruction formats.

All opcodes and Rd destination registers are located in the same fields in each type's instruction. This ISA has arithmetic logic, memory access, conditional branch, and jump instructions. Only integer operations are supported. Remainders such as in divisions are not calculated.

The opcode is 8 bits. 255 instructions can fit in this design. Less than 255 instructions are implemented. For readability and compiler concerns, it is kept at 8 bits. There is a consistency between the opcodes. The first 3 bits are grouping the instructions. For readability and compilers, branches, arithmetics, store, and load are grouped again. R3 field is co-used by the shamt field. Only instruction that uses the field as r3 is mula. Func4 field has been left blank for further development. For example, func4 can be used to group opcodes to save some opcodes for new instructions.

The program counter is 12 bits since the main memory can be accessed with 12 bit PC. Furthermore, direct addressing is used in branch and jump operations. In branch instructions, two conditions were selected, equal and not equal. There are enough instructions to implement any possible loop and conditional operations with these instructions. ALU can do addition, subtraction, multiplication, division, or, xor, shifting, and less than operations. It is used in branch and immediate calculations.

4. BENCHMARKS AND TESTS

4.1 Assembler

An assembler program, written in python3, converts benchmarks code to hex format. It is forked from optiMIPS-Assembler, a basic assembler for MIPS ISA. It can convert one instruction to hex or a program file to hex. Many different structural and functional error checks are done in the assembler, increasing the benchmark codes' reliability before testing the CPU. Assembler code is shown in APPENDIX C.

4.2 Benchmark Algorithms

As discussed, the CPU is designed to pass three benchmarks. These benchmarks are basic but industry-standard in testing processing units. Each benchmark code is firstly written in C language, and algorithms are optimized to be suitable for writing in the systems' assembly language. C codes are shown in APPENDIX C.

4.2.1 IAXPY

In the Integer a-x-p-y, IAXPY benchmark, two 2-dimensional vectors are loaded to registers, and the following calculation is done where A and B are the vectors. Sizes are defined with the value n. Array sizes can be maximum of 20x20, and k is a 16-bit integer.

$$B[i][j] = k.A[i][j] + B[i][j] \ldots (1)$$

Benchmark code is written as a pointer for array A is starting from address 0xA0 and B from right after the end of the first array.

$$\&B = 0xA0+n^2 \ldots (2)$$

Benchmark is run with n= 2,4,8 and k= 3, -3. Benchmark consists of 17 instructions, 10 instructions are called inside the loop, iterating nxn times. Both arrays are filled with the same numbers. Numbers 1 to 16 are filled back-to-back, and the results loaded to array B are multiplied by four input values. These test values are selected to create readable results. As an example:

A[0][1] = 2 and B[0][1] = 2 => B[0][0] = kx A[0][0] + B[0][0] => 8= 3x2+2

Odd n values created dram address call errors. Incorrect or inconsistent results are acquired from the tests.

4.2.2 Sorter + CRC

The merge sort algorithm is used to sort 16-bit numbers. The maximum input size is 400 numbers. After sorting each number appended with *CRC-16-CCITT* 16-bit (remainder) string in the remaining 16-bit of the 32-bit word. Generally, merge sort is done in a recursive fashion. A pointer and/or a call instruction are needed to write an effective recursive algorithm in assembly. In the proposed ISA, there is no jump from register instruction or there isn't a stack-like unit designed in the datapath. The merge sort algorithm is updated to be used iteratively since the complexity of both iterative and recursive methods is nlogn.

While instructions required to run the CRC algorithm, such as signed loads, and xor, are functioning, debugging the merge sort algorithm limited the time to work on the rest of the algorithm.

The merge sort algorithm fails after two iterations of the code. Assuming the forwarding system is failing at some point in the loop, and branch addresses are being forwarded incorrectly. But this bug is inconsistent to conclude that it is the reason.

4.2.3 Text Parse

Text parse benchmark takes string input, removes adjacent space characters, and replaces them with one space (0x20). There are 6 different ASCII space characters such as tabs, different new lines, vertical spaces etc. (0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x20) Benchmark is run in iterative approach. It scans the string whenever a consecutive space

character is found, replaces it with 0x20, and starts again from that point of the string to the end, including the newly replaced space character. The following test is conducted. The string is loaded to memory starting from address 0xA0. Benchmark took 25 instructions and 1 nop. Nop is added due to the same reason merge sort failed, memory stalls, and forwarding clashing.

Input: {'b', 'a', 'r', 8, 8, 11 ,'i', 32, 32, 's', 32, 'g', 10, 'u', 10,10, 'z', 'e', 'l'}

Expected Output: {'b', 'a', 'r', 8, 32, 11 ,'i', 32, 's', 32, 'g', 10, 'u', 32, 'z', 'e', 'l'}

Different space characters and strings are tested. All tests are passed successfully.

4.3 Verilator Tests

Verilator is used to test CPU in every step of the design process. Benchmarks are loaded to main memory, and Verilator test code sim_main.cpp is run to test each benchmark. Verilator test code is included in APPENDIX D. Verilator program can print registers, d-cache blocks and main memory. Whenever D-cache blocks are updated, Verilator prints the changed nonzero blocks and prints at the end of the program depending on the program and expected result. Verilator code reads register or memory values and compares with expected results before printing statistics shown in the following. As discussed in Chapter 4.2, IAXPY and Text Parse tests are passed successfully. The Sorter+CRC benchmark is not completed. In IAXPY tests, CPI is minimized when the array sizes are bigger than 8x8. In smaller sizes, I cache stalls, exceeds the total number of instructions executed cycles, with that increases the CPI matching d cache block sizes with array size decreased CPI further. For 8x8 arrays, 8 block size is the best option. It is similar in the text parse benchmark but less sensitive to block sizes.

Table 2 Verilator Test Results

Benchmark	Instr. Count	CC	CPI	ET (ns)	Energy (nJ)	CC Memory Stall	Estimated % functionality verified
IAXPY	199	544	2.734	1.04	0.91	259	90%
Sorter+CRC							20%
Text Parse	376	1149	3.059	2.2	1.93	222	100%

4.4 Quartus II Verification

Quartus simulation is done by DRAM address size 11. Address size 12 is used at first. However, this was too big for this FPGA. It could not fit. When the address size change, only logic elements are changed. The number of memory bits stayed the same.

Moreover, the simulation does not show the number of ram blocks. We think this design does not reserve internal memory. Maybe some special implementation is needed to use the ram blocks and memory bits.

Fmax is decreased after cache implementation. Also, power dissipation and the total cost are increased. FPGA resources are heavily used due to the design growth. This was expected after cache hierarchy was implemented. Screen captures are included in APPENDIX E.

Table 3 Quartus Simulation Results

Number of Logic Elements	59,923
Total Registers	18,235
Total Memory Bits	6,055
Embedded Multiplier Elements	6
Fmax (MHz)	49.24
Total Thermal Power Dissipation (mW)	876.39
Core Dynamic Power (mW)	677.85
Core Static Power (mW)	157.41
I/O Thermal Power (mW)	41.13

5. CONCLUSION

RINSC was mostly successful. 2 of 3 benchmarks are fully passed, and one benchmark is functioning, but branching logic has bugs to fix. Pipelined CPU and controlling a CPU was a good practice in addition to the Computer Architecture course material. Many issues are encountered and fixed while developing the system, again increasing the understanding of pipelined CPUs and memory hierarchies. The second challenge was adding a big hardware, like cache and implementing it without losing any functionality. In terms of further updates and fixes; firstly memory stall logic should be fixed. Second criticism can be that making use of the pipeline stages more (pulling the branch decision before in the pipeline) and using second ALU more in ISA design. Github link of the project can be found in references.

6. REFERENCES

Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach*. Morgan Kaufmann.

Mahmood, Hadeel & Omran, Safaa. (2014). Pipelined MIPS processor with cache controller using VHDL implementation for educational purposes. 82-87. 10.1109/ICECCPCE.2013.6998739.

Muhtaroglu, A. (2022). *MODULE #3 ISA Delivery and Verification on Fully Functional 32-bit Pipelined Integer CPU*

Muhtaroglu, A. (2022). *MODULE #5 Programming & Verification of a Pipelined CPU with Basic Cache Hierarchy*

RINSC. (2022). *Curba/riscv_pipelinedCPU: METU NCC computer architecture II course. GitHub. https://github.com/Curba/riscv_pipelinedCPU*

APPENDIX A ISA DESIGN

This ISA has fixed 32-bit instructions. CPU has 32 bits wide registers, and all registers are defined as 5 bit address field. There are three main instruction formats.

All opcodes and Rd destination registers are kept in the same positions in the ISA structure. This ISA has arithmetic-logic, memory access, conditional branch and jump instructions. Only integer operations are supported.

R Type

R-type instructions have three registers and shamt or four register. R type is used for doing operations between two registers and writing the result in another register (Rd). Fourth register field can be shamt field for the shift operations or register for mula. Func4 field is left blank for now.

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	Rc/Shamt	Func4	Op

I Type

I-type instructions have two register and 14-bit immediate field. I-type consist of immediate arithmetic, memory access and conditional branch instructions. Instructions that require one register and immediate operations are defined in I-type as well. There is one ImmGen logic that sign extends 14 bits value to 32 bits. Immediate field is also used for address calculation which are direct addressing and index addressing.

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	Op

J Type

J-type has only jump instructions. It has one register field in order to store the current PC location before jump.

31 27	26 8	7 0
Rd	Immediate 19	Op

Definitions

Registers R0 = 00000 and R31 = 11111 are reserved for special purposes. R0 is zero register and R31 is jump register. Any other register from 1 to 30 are general purpose registers.

Program Counter (PC) = 7 bits

SignExtImm = { (18){ Instruction [21]}, Immediate}

BranchAddr = {Immediate [6:0]}

JumpAddr = {Immediate [6:0]}

List of instructions and their descriptions are listed below.

R Type

NOP

31 27	26 22	21 17	16 12	11 8	7 0
00000	00000	00000	0 0000	0000	0000 0000

Syntax: NOP

Desc: No operation. Does not change any user-visible state.

PC = PC + 4

ADD

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	0 0000	0000	0000 1000

Syntax: ADD Rd, Ra, Rb

Desc: Adds the 32-bit registers Ra and Rb and stores the result in rd.

(Signed Operation -2,147,483,648< R[Ra],R[Rb],R[Rd] < +2,147,483,647)

R[Rd] = R[Ra] + R[Rb]

PC = PC+ 4

SUB

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	0 0000	0000	0001 0000

Syntax: SUB Rd, Ra, Rb

Desc: Subtracts the 32-bit registers Ra and Rb and stores the result in rd.

(Signed Operation -2,147,483,648< R[Ra],R[Rb],R[Rd] < +2,147,483,647)

R[Rd] = R[Ra] - R[Rb]

PC = PC+ 4

MUL

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	0 0000	0000	0001 1000

Syntax: MUL Rd, Ra, Rb

Desc: Multiplies the 16-bit values Ra and Rb and stores the result in rd.

(Signed Operation -2,147,483,648< R[Rd] < +2,147,483,647)

(Signed Operation -32,767< R[Ra]< +32,767)

R[Rd] = R[Ra] * R[Rb]

PC = PC+ 4

AND

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	0 0000	0000	0010 1000

Syntax: AND Rd, Ra, Rb

Desc: Bitwise AND on registers Ra and Rb and stores the result in rd.

R[Rd] = R[Ra] & R[Rb]

PC = PC+ 4

OR

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	0 0000	0000	0011 0000

Syntax: OR Rd, Ra, Rb

Desc: Bitwise OR on registers Ra and Rb and stores the result in rd.

R[Rd] = R[Ra] | R[Rb]

PC = PC+ 4

XOR

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	0 0000	0000	0011 1000

Syntax: XOR Rd, Ra, Rb

Desc: Bitwise XOR on registers Ra and Rb and stores the result in rd.

R[Rd] = R[Ra] ^ R[Rb]

PC = PC+ 4

SLT

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	0 0000	0000	0100 0000

Syntax: SLT Rd, Ra, Rb

Desc: Sets 1 to Rd if Ra less than Rb, otherwise sets 0

R[Rd] = (R[Ra] < R[Rb]) ? 1 : 0

PC = PC+ 4

SLL

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	Shamt	0000	0100 1000

Syntax: SLL Rd, Ra, Shamt

Desc: Logical Shift Left

R[Rd] = R[Ra] << {(27){1'b0}},Shamt}

PC = PC+ 4

SRL

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	Shamt	0000	0101 0000

Syntax: SRL Rd, Ra, Shamt

Desc: Logical Shift Right

R[Rd] = R[Ra] >> {(27){1'b0}},Shamt}

PC = PC+ 4

SRA

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	Shamt	0000	0101 1000

Syntax: SRA Rd, Ra, Shamt

Desc: Arithmetic Shift Right

R[Rd] = R[Ra] >>> {(27){1'b0}},Shamt}

PC = PC+ 4

MULA

31 27	26 22	21 17	16 12	11 8	7 0
Rd	Ra	Rb	Rc	0000	0000 0111

Syntax: MULA Rd, Ra, Rb, Rc

Desc: Multiply Add Operation

(Signed Operation -2,147,483,648< R[Rc],R[Rd] < +2,147,483,647)

(Signed Operation -32,768< R[Ra],R[Rb] < +32,767)

R[Rd] = R[Ra] * R[Rb] + R[Rc]

PC = PC+ 4

I Type

SB

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0000 1001

Syntax: SB Imm (Ra), Rd

Desc: Stores one byte to location (Imm + Ra)

M[[R[Ra] + SignExtImm](7:0)] = R[Rd] (7:0)

PC = PC+ 4 -8192<Imm< 8191

SH

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0001 0001

Syntax: SH Imm (Ra), Rd

Desc: Stores half word to location (Imm + Ra)

M[[R[Ra] + SignExtImm](15:0)] = R[Rd] (15:0)

PC = PC+ 4 -8192<Imm< 8191

SW

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0001 1001

Syntax: SW Imm (Ra), Rd

Desc: Stores one word to location (Imm + Ra)

M[R[Ra] + SignExtImm] = R[Rd]

PC = PC+ 4 -8192<Imm< 8191

LB

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0010 0001

Syntax: LB Rd, Imm (Ra)

Desc: Loads one byte from location (Imm + Ra)

R[Rd] = {24'b0, (M [R[Ra]] + SignExtImm])}

PC = PC+ 4 -8192<Imm< 8191

LBS

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0100 1001

Syntax: LBS Rd, Imm (Ra)

Desc: Loads one byte for signed numbers from location (Imm + Ra).

(Signed Operation)

R[Rd] = {24'b1, (M [R[Ra]] + SignExtImm])}

PC = PC+ 4 -8192<Imm< 8191

LH

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0010 1001

Syntax: LH Rd, Imm (Ra)

Desc: Loads half word from location (Imm + Ra)

R[Rd] = {16'b0, (M [R[Ra]] + SignExtImm])}

PC = PC+ 4 -8192<Imm< 8191

LHS

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0010 1001

Syntax: LH Rd, Imm (Ra)

Desc: Loads half word for signed numbers from location (Imm + Ra).

(Signed Operation)

R[Rd] = {16'b1, (M [R[Ra]] + SignExtImm])}

PC = PC+ 4 -8192<Imm< 8191

LW

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0011 0001

Syntax: LH Rd, Imm (Ra)

Desc: Loads one word from location (Imm + Ra)

R[Rd] = (M [R[Ra]] + SignExtImm])

PC = PC+ 4 -8192<Imm< 8191

BEQ

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0011 1001

Syntax: BEQ Rd, Ra, Imm

Desc: Branches to Imm PC if Zero Flag is 1

(Signed Operation -2,147,483,647< R[Ra], R[Rd] < +2,147,483,646)

PC = (R[Rd] == R[Ra]) ? BranchAddr: PC+4 -8192<Imm< 8191

BNE

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0100 0001

Syntax: BNE Rd, Ra, Imm

Desc: Branches to Imm PC if Zero Flag is 0

(Signed Operation -2,147,483,647< R[Ra], R[Rd] < +2,147,483,646)

PC = (R[Rd] != R[Ra])) ? PC+4: BranchAddr -8192<Imm< 8191

ADDI

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0000 0011

Syntax: ADDI Rd, Ra, Imm

Desc: Adds immediate value to Ra register and writes to Rd

(Signed Operation -2,147,483,647< R[Ra], R[Rd] < +2,147,483,646)

R[Rd] = R[Ra] + SignExtImm

PC = PC+ 4 -8192<Imm< 8191

SUBI

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0000 1011

Syntax: SUBI Rd, Ra, Imm

Desc: Subtracts immediate value from Ra register and writes to Rd

(Signed Operation -2,147,483,647< R[Ra], R[Rd] < +2,147,483,646)

R[Rd] = R[Ra] – SignExtImm

PC = PC+ 4 -8192<Imm< 8191

MULI

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0001 0011

Syntax: MULI Rd, Imm

Desc: Multiplies Ra with immediate value and writes to Rd

(Signed Operation -2,147,483,647< R[Ra], R[Rd] < +2,147,483,646)

R[Rd] = R[Ra] * SignExtImm

PC = PC+ 4 -8192<Imm< 8191

ORI

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0001 1011

Syntax: ORI Rd, Ra, Imm

Desc: Does or operation with Ra and immediate and writes to Rd

R[Rd] = R[Ra] Imm

PC = PC+ 4 -8192<Imm< 8191

XORI

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0010 0011

Syntax: XORI Rd,Ra, Imm

Desc: Does xor operation with Ra and immediate and writes to Rd

R[Rd] = R[Ra] ^ Imm

PC = PC+ 4 -8192<Imm< 8191

ANDI

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0010 1011

Syntax: ANDI Rd, Ra, Imm

Desc: Does and operation with Rd and immediate and writes to Rd

R[Rd] = R[Ra] & Imm

PC = PC+ 4 -8192<Imm< 8191

SLTI

31 27	26 22	21 8	7 0
Rd	Ra	Immediate 14	0011 0011

Syntax: SLTI Rd, Ra, Imm

Desc: Sets Rd as 1 if Ra< Imm

(Signed Operation -2,147,483,647< R[Ra], R[Rd] < +2,147,483,646)

R[Rd] = (R[Ra]< SignExtImm) ? 1 : 0

PC = PC+ 4 -8192<Imm< 8191

J Type

JAL

31 27	26 8	7 0
Rj (0 0011)	Immediate 19	0000 0100

Syntax: JAL Imm

Desc: Jumps to immediate PC value and stores last PC to Rj (Jump Register)

R[Rj] = PC

JumpAddr = {Immediate [11:0]}

PC = JumpAddr -262,145≤ Imm ≤ 262144

APPENDIX B SYSTEM VERILOG FILES

Top.sv

```
`include "config.sv"
`include "constants.sv"

module top (
    input logic         clock,
    input logic         reset,
    output logic [7:0]  Op
);

    logic [3:0] MemRead;
    logic [3:0] MemWrite;
    logic PCSrc;
    logic [1:0] MemToReg;
    logic RegWrite;
    logic [3:0] ALUOp;
    logic [1:0] ALUSrc;
    logic ALUOp2;
    logic RbSelect;
    logic MemSignExtend;
    logic [1:0] branch_flag;
    logic jump_flag;
    logic double_jump_flag;


    logic [`DRAM_ADDRESS_SIZE-1:0]  icache_PC;
    logic                           icache_instrRequest;
    logic [`DRAM_WORD_SIZE-1:0]     icache_instruction;
    logic                           icache_instrReady;

    //Data Side
    logic [`DRAM_ADDRESS_SIZE-1:0]  dcache_address;
    logic                           dcache_dataRequest;
    logic                           dcache_rw;
    logic [`DRAM_WORD_SIZE-1:0]     dcache_writeData;
    logic [`DRAM_WORD_SIZE/8-1:0]   dcache_byte_en;
    logic [`DRAM_WORD_SIZE-1:0]     dcache_readData;
    logic                           dcache_data_ready;
    logic                           transfer_in_progress;

control ctr(
    .Op         (Op),
    .AluSrc     (ALUSrc),
    .AluOp      (ALUOp),
    .PCSrc      (PCSrc),
    .MemRead    (MemRead),
    .MemWrite   (MemWrite),
    .MemToReg   (MemToReg),
    .RegWrite   (RegWrite),
    .AluOp2     (ALUOp2),
    .RbSelect   (RbSelect),
```

```
    .MemSignExtend  (MemSignExtend),
    .branch_flag    (branch_flag),
    .jump_flag      (jump_flag),
    .double_jump_flag  (double_jump_flag)
);



datapath datapath(
    .clk        (clock),
    .reset      (reset),
    .MemRead    (MemRead),
    .MemWrite   (MemWrite),
    .MemToReg   (MemToReg),
    .RegWrite   (RegWrite),
    .branch_flag    (branch_flag),
    .jump_flag      (jump_flag),
    .MemSignExtend  (MemSignExtend),
    .PCSrc      (PCSrc),
    .ALUSrc     (ALUSrc),
    .ALUOp      (ALUOp),
    .ALUOp2     (ALUOp2),
    .RbSelect   (RbSelect),
    .Op         (Op),
    .double_jump_flag  (double_jump_flag),

    //Instruction Side
    .icache_PC              (icache_PC),
    .icache_instrRequest    (icache_instrRequest),
    .icache_instruction     (icache_instruction),
    .icache_instrReady      (icache_instrReady),

    //Data Side
    .dcache_address         (dcache_address),
    .dcache_dataRequest     (dcache_dataRequest),
    .dcache_rw              (dcache_rw ),
    .dcache_writeData       (dcache_writeData),
    .dcache_byte_en         (dcache_byte_en),
    .dcache_readData        (dcache_readData),
    .dcache_data_ready      (dcache_data_ready),

    .transfer_in_progress   (transfer_in_progress)
);

memory memory(
    .clock                  (clock),
    .reset                  (reset),

    //Instruction Side
    .icache_PC              (icache_PC), // cpu request address (CPU-
>cache)
    .icache_instrRequest    (icache_instrRequest), // cpu request valid
(CPU->cache)
    .icache_instruction     (icache_instruction), // data to CPU
(cache->CPU)
```

```
    .icache_instrReady      (icache_instrReady), // data to CPU ready
(cache->CPU)

    //Data Side
    .dcache_address         (dcache_address), // cpu request address
(CPU->cache)
    .dcache_dataRequest     (dcache_dataRequest), // cpu request valid
(CPU->cache)
    .dcache_rw              (dcache_rw ), // cpu R/W request (CPU-
>cache)
    .dcache_writeData       (dcache_writeData), // cpu request data
(CPU->cache)
    .dcache_byte_en         (dcache_byte_en), // cpu request byte
enable (CPU->cache)
    .dcache_readData        (dcache_readData), // data to CPU (cache-
>CPU)
    .dcache_data_ready      (dcache_data_ready), // data to CPU ready
(cache->CPU)
    .transfer_in_progress   (transfer_in_progress)
    );
endmodule
```

memory.sv

```
// Memory Top Level Unit

`include "config.sv"
`include "constants.sv"

module memory(input  clock, reset,
    //Instruction Side
    input [`DRAM_ADDRESS_SIZE-1:0]  icache_PC, // cpu request address
(CPU->cache)
    input                           icache_instrRequest, // cpu request
valid (CPU->cache)
    output [`DRAM_WORD_SIZE-1:0]    icache_instruction, // data to CPU
(cache->CPU)
    output                          icache_instrReady, // data to CPU
ready (cache->CPU)

    //Data Side
    input [`DRAM_ADDRESS_SIZE-1:0]  dcache_address, // cpu request
address (CPU->cache)
    input                           dcache_dataRequest, // cpu request
valid (CPU->cache)
    input                           dcache_rw, // cpu R/W request (CPU-
>cache)
    input [`DRAM_WORD_SIZE-1:0]     dcache_writeData, // cpu request
data (CPU->cache)
    input [`DRAM_WORD_SIZE/8-1:0]   dcache_byte_en, // cpu request byte
enable (CPU->cache)
    output [`DRAM_WORD_SIZE-1:0]    dcache_readData, // data to CPU
(cache->CPU)
    output                          dcache_data_ready, // data to CPU
ready (cache->CPU)
    output                          transfer_in_progress
```

```
    );

    logic mem_ready_icache;
    logic zeros;

    assign zeros = 1'b0;

icache_controller icache_controller (
    .clock                  (clock),
    .reset                  (reset),
    .icache_address         (icache_PC), // cpu request address (CPU-
>cache)
    .icache_valid           (icache_instrRequest), // cpu request valid
(CPU->cache)
    .icache_data_out        (icache_instruction), // data to CPU
(cache->CPU)
    .icache_data_ready      (icache_instrReady),// data to CPU ready
(cache->CPU)

    .mem_data_in            (icache_instructionBlock), // memory read
data (memory->cache)
    .mem_ready              (icache_blockReady), // memory read data
ready (memory->cache)

    .mem_address            (icache_blockAddress), // cache request
address (cache->memory)
    .mem_valid              (icache_blockRequest) // request to memory
valid (cache->memory)

    );


dcache_controller dcache_controller (
    .clock              (clock),
    .reset              (reset),
    .dcache_address     (dcache_address), // cpu request address (CPU-
>cache)
    .dcache_data_in     (dcache_writeData), // cpu request data (CPU-
>cache)
    .dcache_byte_en     (dcache_byte_en), // cpu request byte enable
(CPU->cache)
    .dcache_rw          (dcache_rw), // cpu R/W request (CPU->cache)
    .dcache_valid       (dcache_dataRequest), // cpu request valid
(CPU->cache)

    .dcache_data_out    (dcache_readData), // data to CPU (cache->CPU)
    .dcache_data_ready  (dcache_data_ready),// data to CPU ready
(cache->CPU)

    .mem_data_in        (dcache_readBlock), // memory read data
(memory->cache)
    .mem_ready          (dcache_blockReady), // memory read data ready
(memory->cache)

    .mem_address        (dcache_blockAddress), // cache request address
(cache->memory)
```

```
    .mem_data_out       (dcache_wbBlock), // memory write data (cache-
>memory)
    .mem_rw             (dcache_mem_rw), // R/W request to memory
(cache->memory)
    .mem_valid          (dcache_blockRequest) // request to memory
valid (cache->memory)
    );


logic icache_blockRequest, icache_blockReady;
logic [`DRAM_ADDRESS_SIZE-1:0] icache_blockAddress;
logic [`DRAM_WORD_SIZE-1:0]     icache_instructionBlock
[`DRAM_BLOCK_SIZE-1:0];

logic dcache_blockRequest, dcache_blockReady, dcache_mem_rw;
logic [`DRAM_ADDRESS_SIZE-1:0] dcache_blockAddress;
logic [`DRAM_WORD_SIZE-1:0]     dcache_readBlock [`DRAM_BLOCK_SIZE-
1:0];
logic [`DRAM_WORD_SIZE-1:0]     dcache_wbBlock [`DRAM_BLOCK_SIZE-1:0];

/* verilator lint_off PINMISSING */
dram_controller dram_controller(
    .clock                      (clock),
    .reset                      (reset),

    .dram_port1_request         (icache_blockRequest),
    .dram_port1_address         (icache_blockAddress),
    .dram_port1_read_data       (icache_instructionBlock),
    .dram_port1_acknowledge     (icache_blockReady),
    .dram_port1_we              (zeros),

    .dram_port2_address         (dcache_blockAddress),
    .dram_port2_request         (dcache_blockRequest),
    .dram_port2_read_data       (dcache_readBlock),
    .dram_port2_write_data      (dcache_wbBlock),
    .dram_port2_acknowledge     (dcache_blockReady),
    .dram_port2_we              (dcache_mem_rw),

    .dram_busy                  (transfer_in_progress)
    );
/* verilator lint_on PINMISSING */
endmodule
```

datapath.sv

```
`include "config.sv"
`include "constants.sv"

module datapath(input logic clk, reset,
                input logic [3:0] MemWrite,
                input logic [1:0] MemToReg,
                input jump_flag,
                input logic RegWrite, PCSrc, MemSignExtend,
                input logic [1:0] branch_flag,
```

```
                input logic [3:0] MemRead,
                input logic [3:0] ALUOp,
                input logic [1:0] ALUSrc,
                input logic RbSelect,
                input logic ALUOp2,
                input logic double_jump_flag,

                input logic icache_instrReady, // data to CPU ready
(cache->CPU)
                input logic [`DRAM_WORD_SIZE-1:0]   icache_instruction,
// data to CPU (cache->CPU)
                output logic [`DRAM_ADDRESS_SIZE-1:0]  icache_PC, //
cpu request address (CPU->cache)
                output logic icache_instrRequest, // cpu request valid
(CPU->cache)

                input logic[`DRAM_WORD_SIZE-1:0]    dcache_readData, //
data to CPU (cache->CPU)
                input logic dcache_data_ready, // data to CPU ready
(cache->CPU)
                output logic[`DRAM_ADDRESS_SIZE-1:0]  dcache_address,
// cpu request address (CPU->cache)
                output logic dcache_dataRequest, // cpu request valid
(CPU->cache)
                output logic dcache_rw, // cpu R/W request (CPU->cache)
                output logic [`DRAM_WORD_SIZE-1:0]  dcache_writeData,
// cpu request data (CPU->cache)
                output logic[`DRAM_WORD_SIZE/8-1:0]   dcache_byte_en,
// cpu request byte enable (CPU->cache)

                input logic transfer_in_progress,
                output logic [7:0] Op);


    logic [11:0]PC;
    logic [11:0] PCSTART; //starting address of instruction memory
    assign PCSTART = 0;
/*
    // Instruction memory internal storage, input address and output
data bus signals
    logic [7:0] instmem [127:0];
    logic [6:0] instmem_address;
    logic [31:0] instmem_data;
    */

    // Data memory internal storage, input address and output data bus
signals
    logic [11:0] datamem;
    logic [11:0] datamem_address;
    logic [31:0] datamem_data;
    logic [31:0] datamem_write_data;

    //Forwarding Parameters
    logic [1:0] ForwardingA;
    logic [1:0] ForwardingB;
    logic  ForwardingC;
```

```
    logic ForwardingD;
    logic stall_flag;
    logic PCenable;
    logic IfIdEN;
    logic flush;

    logic [1:0]branchId;
    logic [1:0] branchex;
    assign  branchId = IdEx.branch_flag;
    assign  branchex = ExMem.branch_flag;
    logic mem_stall_flag;
    logic normal_stall;

    always_comb  begin// data hazard detection and forward , control
hazard detection and flush

            if((dcache_dataRequest && !dcache_data_ready) ||
(icache_instrRequest && !icache_instrReady) || transfer_in_progress)
                mem_stall_flag = 1;
            else
                mem_stall_flag = 0;

        if((IdEx.MemRead != 4'b0000)&&((IdEx.rd ==
IfId.instruction[26:22])
            ||(IdEx.rd == IfId.instruction[21:17])))
                normal_stall = 1;
            else
                normal_stall =0;

        if(((IdEx.MemRead != 4'b0000)&&((IdEx.rd ==
IfId.instruction[26:22])
            ||(IdEx.rd == IfId.instruction[21:17])))
            || (dcache_dataRequest && !dcache_data_ready)
            || (icache_instrRequest && !icache_instrReady) ||
transfer_in_progress)
            begin
            stall_flag = 1;
            PCenable = 0;
            IfIdEN = 0;
            end
        else begin
            stall_flag = 0;
            PCenable = 1;
            IfIdEN = 1;
            end

        if(jump_flag != 0 || branch_flag != 0 || IdEx.branch_flag != 0
|| ExMem.branch_flag != 0)begin
            flush = 1;
            PCenable = (jump_flag != 0 || branch_src != 0) ? 1:0;
            end
        else
            flush = 0;
        end
```

```
    // IF/ID Pipeline staging register fields can be represented using
structure format of System Verilog
    // You may refer to the first field in the structure as
IfId.instruction for example
    struct packed{
        logic [31:0] instruction;
        logic [11:0] PCincremented;
    } IfId;

    //Cache Logic
    assign icache_instrRequest = 1;
 //   assign IfIdEN = icache_instrRequest;
    assign icache_PC = PC;

    always @ (posedge clk) begin
        if(IfIdEN)begin
        IfId.instruction <= (flush) ? 0:icache_instruction[31:0];
        IfId.PCincremented <= PC+12'b100;
        end
    end

    //decode
    logic [18:0] JumpAddress;

    assign Op = IfId.instruction[7:0];
    assign JumpAddress = IfId.instruction [26:8];

    // Register File description
    logic [31:0] RF[31:0];
    logic [31:0] da;            //Read Ra
    logic [31:0] db;            //Read Rb
    logic [31:0] dc;            //Read Rb
    logic [31:0] RF_WriteData;  //Write data
    logic [31:0] RF_WriteAddr;  //Write address
    logic double_jump;

    // Register Logic
    assign da = RF[IfId.instruction[26:22]] ;
    assign db = (RbSelect)?
RF[IfId.instruction[31:27]]:RF[IfId.instruction[21:17]] ;
    assign dc = RF[IfId.instruction[16:12]];

    always_comb
        case(MemWb.MemToReg)
            2'b00: RF_WriteData = MemWb.datamem_data;
            2'b01: RF_WriteData = MemWb.Alu2out;
            2'b10: RF_WriteData = {{(20){1'b0}},MemWb.PCincremented};
           default: RF_WriteData = MemWb.datamem_data;
         endcase

    assign RF_WriteAddr = {{(27){1'b0}},MemWb.rd};

    always @(negedge clk) begin
        if (MemWb.RegWrite) begin
            RF[RF_WriteAddr] <= RF_WriteData;
        end
```

```
    end

    struct packed{
        logic [4:0] ra;
        logic [4:0] rb;
        logic [4:0] rc;
        logic [1:0] ALUSrc;
        logic [3:0] ALUOp;
        logic ALUOp2;
        logic MemSignExtend;
        logic [1:0]MemToReg;
        logic [3:0] MemRead;
        logic [3:0] MemWrite;
        logic RegWrite;
        logic [11:0] PCincremented;
        logic [31:0] da;
        logic [31:0] db;
        logic [31:0] dc;
        logic [31:0] signextend;
        logic [4:0] rd;
        logic [4:0] shamt;
        logic [1:0] branch_flag;
        logic [13:0] branch_addr;
        logic RbSelect;
        logic double_jump_flag;
    } IdEx;

    always @ (posedge clk) begin
        if(mem_stall_flag == 0 )begin
        IdEx.MemSignExtend <= MemSignExtend;
        IdEx.ALUSrc <= ALUSrc;
        IdEx.ALUOp <= ALUOp;
        IdEx.ALUOp2 <= ALUOp2;
        IdEx.MemRead <= MemRead;
        IdEx.MemWrite <= MemWrite;
        IdEx.MemToReg <= MemToReg;
        IdEx.RegWrite <= RegWrite;
        IdEx.PCincremented <= IfId.PCincremented;
        IdEx.branch_addr <= IfId.instruction [21:8];
        IdEx.branch_flag <= branch_flag;
        IdEx.da <= da;
        IdEx.db <= db;
        IdEx.dc <= dc;
        IdEx.shamt <= IfId.instruction[16:12];
        IdEx.rd <= IfId.instruction[31:27];
        IdEx.signextend <= { {(18){IfId.instruction
[21]}},IfId.instruction [21:8] };
        IdEx.ra <= IfId.instruction[26:22];
        IdEx.rb <= (RbSelect) ?
IfId.instruction[31:27]:IfId.instruction[21:17];
        IdEx.rc <= IfId.instruction[16:12];
        IdEx.RbSelect <= RbSelect;
        IdEx.double_jump_flag <= double_jump_flag;
        end
        else begin
    IdEx.double_jump_flag <= 0;*/
```

```
        end
    end

    // Execute Stage Variables
    logic [31:0] alu1in_a;
    logic [31:0] alu1in_b;
    logic [31:0] alu1in_b_mux;
    logic [31:0] Alu1out;
    logic zero_flag;

    logic [4:0]exmemrd;
    logic [4:0]idexra;
    logic [4:0]idexrb;
    logic [1:0]exmembranchflag;
    assign exmemrd = ExMem.rd;
    assign idexra=IdEx.ra;
    assign idexrb= IdEx.rb;
    assign exmembranchflag= ExMem.branch_flag;

    logic debugmemwbregwrite;
    logic debugexmemregwrite;
    logic [1:0] debugbranch;
    logic [4:0] debugmemwbrd;
    logic [4:0] debugexmemrd;
    logic [4:0] debugidexra;
    logic [3:0] debugmemwbmemread;
    logic [3:0] debugexmemmemwrite;
    assign debugmemwbregwrite = MemWb.RegWrite;
    assign debugmemwbrd = MemWb.rd;
    assign debugexmemrd = ExMem.rd;
    assign debugidexra = IdEx.ra;
    assign debugexmemregwrite = ExMem.RegWrite;
    assign debugbranch = ExMem.branch_flag;
    assign debugmemwbmemread = MemWb.MemRead;
    assign debugexmemmemwrite = ExMem.MemWrite;


    always_comb begin
      if ((ExMem.RegWrite)&&(ExMem.rd !=0) && ((ExMem.rd != IdEx.ra &&
ExMem.rd == IdEx.ra) || (ExMem.branch_flag == 0 && ExMem.rd ==
IdEx.ra)))
            ForwardingA = 2'b10;
        else if (MemWb.RegWrite && MemWb.rd !=0 && ExMem.rd != IdEx.ra
&& MemWb.rd == IdEx.ra)
            ForwardingA = 2'b01;
        else
            ForwardingA = 2'b00;

        if ((ExMem.RegWrite)&& (ExMem.rd !=0) && ((ExMem.rd != IdEx.rb
&& ExMem.rd == IdEx.rb) || (ExMem.branch_flag == 0 && ExMem.rd ==
IdEx.rb && ExMem.rd == IdEx.ra)))
            ForwardingB = 2'b10;
        else if (MemWb.RegWrite && MemWb.rd !=0 && ExMem.rd != IdEx.rb
&& MemWb.rd == IdEx.rb)
           ForwardingB = 2'b01;
        else
```

```
            ForwardingB = 2'b00;

        case(ForwardingA)
            2'b00: alu1in_a = IdEx.da; // If there is no forwarding Alu
input1 from IdEx.da
            2'b01: alu1in_a = RF_WriteData;
            2'b10:  alu1in_a = ExMem.Alu1out; // If forwarding logic
set to 10, corresponding data at ExMem register
            default: alu1in_a = ExMem.Alu1out;
        endcase

        case(ForwardingB)
            2'b00: alu1in_b = alu1in_b_mux;
            2'b01: alu1in_b = RF_WriteData;
            2'b10: alu1in_b = ExMem.Alu1out;
            default: alu1in_b = ExMem.Alu1out;
        endcase

        if (MemWb.RegWrite && MemWb.rd !=0 && ExMem.rd != IdEx.rc &&
MemWb.rd == ExMem.rc)
            ForwardingC = 1;
        else
            ForwardingC = 0;

        if (MemWb.rd !=0 && MemWb.rd == ExMem.rd)
            ForwardingD = 1;
        else
            ForwardingD = 0;
    end

    always_comb begin
        case(IdEx.ALUSrc)
            2'b00: alu1in_b_mux = IdEx.db;
            2'b01: alu1in_b_mux = IdEx.signextend;
            2'b10: alu1in_b_mux = {{(27){1'b0}},IdEx.shamt};
            default: alu1in_b_mux = IdEx.db; endcase
    end

    always_comb begin
        case(IdEx.ALUOp)
            4'b0000: Alu1out = alu1in_a + alu1in_b;
            4'b0001: Alu1out = alu1in_a - alu1in_b;
            4'b0010: Alu1out = alu1in_a * alu1in_b;
            4'b0011: Alu1out = alu1in_a ^ alu1in_b;
            4'b0100: Alu1out = alu1in_a | alu1in_b;
            4'b0101: Alu1out = alu1in_a & alu1in_b;
            4'b0110: Alu1out = alu1in_a << alu1in_b;
            4'b0111: Alu1out = alu1in_a >>> alu1in_b;
            4'b1000: Alu1out = alu1in_a >> alu1in_b;
            4'b1001: Alu1out = (alu1in_a <=  alu1in_b) ? 1:0;
            default: Alu1out = alu1in_a + alu1in_b;
        endcase

        zero_flag = (Alu1out == 0) ? 1:0;
    end
```

```
    struct packed{
        logic [11:0] PCincremented;
        logic [3:0] MemRead;
        logic [3:0] MemWrite;
        logic MemSignExtend;
        logic RegWrite;
        logic ALUOp2;
        logic [1:0] MemToReg;
        logic [31:0] Alu1out;
        logic [31:0] db;
        logic [31:0] dc;
        logic [4:0] rd;
        logic [4:0] rc;
        logic zero_flag;
        logic [1:0]branch_flag;
        logic [13:0] branch_addr;
        logic double_jump_flag;
    } ExMem;

    // Ex Mem Stage
    always @ (posedge clk) begin
        if(mem_stall_flag == 0)begin
            ExMem.PCincremented <= IdEx.PCincremented;
            ExMem.MemSignExtend <= IdEx.MemSignExtend;
            ExMem.MemRead <= IdEx.MemRead;
            ExMem.MemWrite <= IdEx.MemWrite;
            ExMem.RegWrite <= IdEx.RegWrite;
            ExMem.MemToReg <= IdEx.MemToReg;
            ExMem.Alu1out <= Alu1out;
            ExMem.ALUOp2 <= IdEx.ALUOp2;
            ExMem.db <= IdEx.db;
            ExMem.dc <= IdEx.dc;
            ExMem.rc <= IdEx.rc;
            ExMem.zero_flag <= zero_flag;
            ExMem.branch_flag <= IdEx.branch_flag;
            ExMem.branch_addr <= IdEx.branch_addr;
            ExMem.double_jump_flag <= IdEx.double_jump_flag;
            ExMem.rd <= IdEx.rd;
        end
        else

    end

    logic [31:0] alu2in_a;
    logic [31:0] alu2in_b;
    logic [31:0] Alu2out;
    logic branch_src;
    logic branch_ne;
    logic branch_eq;

    assign alu2in_a = ExMem.Alu1out;
    assign alu2in_b = (ForwardingC == 1)?  RF_WriteData:ExMem.dc;

    assign branch_eq = (ExMem.zero_flag == 1 && ExMem.branch_flag[0] ==
1 && ExMem.branch_flag[1] == 0) ? 1:0;
```

```
    assign branch_ne = (ExMem.zero_flag == 0 && ExMem.branch_flag[1] ==
1 && ExMem.branch_flag[0] == 0) ? 1:0;
    assign branch_src = (branch_ne || branch_eq) ? 1:0;
    assign double_jump = (ExMem.Alu1out == 1 && ExMem.double_jump_flag
== 1) ? 1:0;

    always_comb begin
        case(ExMem.ALUOp2)
            1'b0: Alu2out = alu2in_a;
            1'b1: Alu2out = alu2in_a + alu2in_b;
        endcase
    end

    //memwb
    struct packed{
        //control signals
        logic [11:0] PCincremented;
        logic RegWrite;
        logic [1:0]MemToReg;
        logic [31:0] datamem_data;
        logic [31:0] Alu2out;
        logic [4:0] rd;
        logic [3:0] MemRead;
    } MemWb;

    always @ (posedge clk) begin
        if(mem_stall_flag == 0)begin
            MemWb.PCincremented <= ExMem.PCincremented;
            MemWb.RegWrite <= ExMem.RegWrite;
            MemWb.MemToReg <= ExMem.MemToReg;
            MemWb.datamem_data <= datamem_data;
            MemWb.Alu2out <= Alu2out;
            MemWb.rd <= ExMem.rd;
            MemWb.MemRead <= ExMem.MemRead;
            MemWb.datamem_data <= datamem_data;
        end
    end

    // Data Memory Address
    assign datamem_address = ExMem.Alu1out[11:0];
   // assign datamem_write_data = (ForwardingD) ?
MemWb.Alu2out:ExMem.db;
    always_comb begin
        if(ForwardingD && MemWb.MemRead == 0)
            datamem_write_data = MemWb.Alu2out;
        else if(ForwardingD && MemWb.MemRead != 0 && ExMem.MemWrite !=
0)
            datamem_write_data = MemWb.datamem_data;
        else
            datamem_write_data = ExMem.db;
     end
    // Data Memory Write Logic
    assign dcache_byte_en = ExMem.MemWrite;

//  always @(posedge clk) begin
    assign dcache_writeData = datamem_write_data;
```

```
//      end


    // Data Memory Read Logic
    assign dcache_dataRequest = (ExMem.MemWrite != 0 || ExMem.MemRead
!=0) ? 1:0;
    assign dcache_address = datamem_address;
//   assign datamem_data = dcache_readData;
    assign dcache_rw = (ExMem.MemWrite != 0) ? 1:0;

    always_comb begin
            datamem_data[7:0]  =   (ExMem.MemRead[0])?
dcache_readData[7:0]:8'bx;
             if(ExMem.MemRead[1] == 0 && ExMem.MemSignExtend)
               datamem_data[15:8] = {(8){datamem_data[7]}};
             else if (ExMem.MemRead[1])
               datamem_data[15:8] = dcache_readData[15:8];
             else
               datamem_data[15:8] = 8'b0;

             if(ExMem.MemRead[2] == 0 && ExMem.MemSignExtend &&
ExMem.MemRead[1] == 0)
                datamem_data[23:16] = {(8){datamem_data[7]}};
             else if (ExMem.MemRead[2] == 0 && ExMem.MemSignExtend)
                datamem_data[23:16] = {(8){datamem_data[15]}};
             else if(ExMem.MemRead[2])
                datamem_data[23:16] = dcache_readData[23:16];
             else
                datamem_data[23:16] = 8'b0;

             if(ExMem.MemRead[3] == 0 && ExMem.MemSignExtend &&
ExMem.MemRead[1] == 0)
                datamem_data[31:24] = {(8){datamem_data[7]}};
             else if (ExMem.MemRead[3] == 0 && ExMem.MemSignExtend)
                datamem_data[31:24] = {(8){datamem_data[15]}};
             else if(ExMem.MemRead[3])
                datamem_data[31:24] = dcache_readData[31:24];
             else
                datamem_data[31:24] = 8'b0;

    end

    //PC logic
    always@ (posedge clk)begin
        if(reset)
            PC <= PCSTART;
        else if(PCenable)
            PC <= (branch_src) ? ExMem.branch_addr[11:0]: (PCSrc ?
JumpAddress[11:0]:(double_jump ? PC+12'b1100:PC+12'b100));
    end
endmodule
```

control.sv

```
module control( input logic [7:0]Op,
        output logic [1:0] AluSrc,
        output logic [3:0] AluOp,
        output logic [1:0] branch_flag,
        output logic jump_flag,
        output logic MemSignExtend,
        output logic PCSrc,
        output logic [3:0] MemRead,
        output logic [3:0] MemWrite,
        output logic AluOp2,
        output logic RbSelect,
        output logic [1:0] MemToReg,
        output logic double_jump_flag,
        output logic RegWrite );


    always_comb begin
        MemRead = 4'b0000;
        MemWrite = 4'b0000;
        MemToReg = 2'b00;
        MemSignExtend = 1'b0;
        jump_flag = 1'b0;
        branch_flag = 2'b00;
        RegWrite = 1'b0;
        RbSelect = 1'b0;
        AluSrc = 2'b00;
        AluOp = 4'b0000;
        AluOp2 = 1'b0;
        PCSrc = 1'b0;
        double_jump_flag = 1'b0;
        case(Op)
            //NOP
            default: begin
                MemRead = 4'b0000;
                MemWrite = 4'b0000;
                MemToReg = 2'b00;
                MemSignExtend = 1'b0;
                RegWrite = 1'b0;
                AluSrc = 2'b00;
                AluOp = 4'b0000;
                PCSrc = 1'b0;
            end

            //ADD
            8'b00001000: begin
                MemToReg = 2'b01;
                RegWrite = 1'b1;
            end
            //MUL
            8'b00011000: begin
                MemToReg = 2'b01;
                RegWrite = 1'b1;
                AluOp = 4'b0010;
            end
```

```
//SUB
8'b00010000: begin
    MemToReg = 2'b01;
    RegWrite = 1'b1;
    AluOp = 4'b0001;
end

//AND
8'b00101000: begin
    MemToReg = 2'b01;
    RegWrite = 1'b1;
    AluOp = 4'b0101;
end

//OR
8'b00110000: begin
    MemToReg = 2'b01;
    RegWrite = 1'b1;
    AluOp = 4'b0100;
end

//XOR
8'b00111000: begin
    MemToReg = 2'b01;
    RegWrite = 1'b1;
    AluOp = 4'b0011;
end

//MULA
8'b00000111: begin
    MemToReg = 2'b01;
    RegWrite = 1'b1;
    AluOp = 4'b0010;
    AluOp2 = 1'b1;
end

//SLT
8'b01000000: begin
    MemToReg = 2'b01;
    RegWrite = 1'b1;
    AluOp = 4'b1001;
end

//ADDI
8'b00000011: begin
    MemToReg = 2'b01;
    RegWrite = 1'b1;
    AluSrc = 2'b01;
    AluOp = 4'b0000;
end

//SUBI
8'b00001011: begin
    MemToReg = 2'b01;
    RegWrite = 1'b1;
```

```
        AluSrc = 2'b01;
        AluOp = 4'b0001;
    end

    //MULI
    8'b00010011: begin
        MemToReg = 2'b01;
        RegWrite = 1'b1;
        AluSrc = 2'b01;
        AluOp = 4'b0010;
    end

    //ORI
    8'b00011011: begin
        MemToReg = 2'b01;
        RegWrite = 1'b1;
        AluSrc = 2'b01;
        AluOp = 4'b0100;
    end

    //XORI
    8'b00100011: begin
        MemToReg = 2'b01;
        RegWrite = 1'b1;
        AluSrc = 2'b01;
        AluOp = 4'b0011;
    end

    //ANDI
    8'b00101011: begin
        MemToReg = 2'b01;
        RegWrite = 1'b1;
        AluSrc = 2'b01;
        AluOp = 4'b0101;
    end

    //SW
    8'b00011001: begin
        MemWrite = 4'b1111;
        RbSelect = 1'b1;
        AluSrc = 2'b01;
    end

    //SH
    8'b00010001: begin
        MemWrite = 4'b0011;
        RbSelect = 1'b1;
        AluSrc = 2'b01;
    end

    //SB
    8'b00001001: begin
        MemWrite = 4'b0001;
        RbSelect = 1'b1;
        AluSrc = 2'b01;
    end
```

```
            //LW
            8'b00110001: begin
                MemRead = 4'b1111;
                MemToReg = 2'b00;
                RegWrite = 1'b1;
                RbSelect = 1'b1;
                AluSrc = 2'b01;
            end

            //LH
            8'b00101001: begin
                MemRead = 4'b0011;
                MemToReg = 2'b00;
                RegWrite = 1'b1;
                RbSelect = 1'b1;
                AluSrc = 2'b01;
            end

            //LB
            8'b00100001: begin
                MemRead = 4'b0001;
                MemToReg = 2'b00;
                RegWrite = 1'b1;
                RbSelect = 1'b1;
                AluSrc = 2'b01;
            end

            //LHS
            8'b01010001: begin
                MemSignExtend = 1'b1;
                MemRead = 4'b0011;
                MemToReg = 2'b00;
                RegWrite = 1'b1;
                RbSelect = 1'b1;
                AluSrc = 2'b01;
            end
//LBS
            8'b01001001: begin
                MemSignExtend = 1'b1;
                MemRead = 4'b0001;
                MemToReg = 2'b00;
                RegWrite = 1'b1;
                RbSelect = 1'b1;
                AluSrc = 2'b01;
            end

            //BEQ
            8'b00111001 : begin
                RbSelect = 1'b1;
                branch_flag = 2'b01;
                AluOp = 4'b0001;
            end

            //BNE
            8'b01000001 : begin
```

```
                RbSelect = 1'b1;
                branch_flag = 2'b10;
                AluOp = 4'b0001;
            end

            //JAL
            8'b00000100: begin
                jump_flag = 1'b1;
                MemToReg = 2'b10;
                RegWrite = 1'b1;
                PCSrc = 1'b1;
            end

            //SLTI
            8'b00110011: begin
                MemToReg = 2'b01;
                RegWrite = 1'b1;
                AluSrc = 2'b01;
                AluOp = 4'b1001;
            end

            //SLL
            8'b01001000: begin
                MemToReg = 2'b01;
                RegWrite = 1'b1;
                AluOp = 4'b0110;
                AluSrc = 2'b10;
            end

            //SRL
            8'b01010000: begin
                MemToReg = 2'b01;
                RegWrite = 1'b1;
                AluOp = 4'b1000;
                AluSrc = 2'b10;
            end

            //SRA
            8'b01011000: begin
                MemToReg = 2'b01;
                RegWrite = 1'b1;
                AluOp = 4'b0111;
                AluSrc = 2'b10;
            end

            //SSLD
            8'b00111011: begin
                double_jump_flag = 1'b1;
                RbSelect = 1'b1;
                AluOp = 4'b1001;
            end

        endcase
    end
endmodule
```

dram_controller.sv

```
// This is a DRAM controller / interface that supports non-blocking
requests
// in parallel from Instruction and Data Caches
// It is developed by Ali Muhtaroglu in support of education for
Computer
// Architecture / Organization courses at METU Northern Cyprus Campus.

`include "config.sv"
`include "constants.sv"

module dram_controller(
    input                              clock, reset,
    input                              dram_port1_request,
    input [`DRAM_ADDRESS_SIZE-1:0]     dram_port1_address,
    input                              dram_port1_we,
    output [`DRAM_WORD_SIZE-1:0]
dram_port1_read_data[`DRAM_BLOCK_SIZE-1:0],
    input [`DRAM_WORD_SIZE-1:0]
dram_port1_write_data[`DRAM_BLOCK_SIZE-1:0],
    output                             dram_port1_acknowledge,

    input                              dram_port2_request,
    input [`DRAM_ADDRESS_SIZE-1:0]     dram_port2_address,
    input                              dram_port2_we,
    output [`DRAM_WORD_SIZE-1:0]
dram_port2_read_data[`DRAM_BLOCK_SIZE-1:0],
    input [`DRAM_WORD_SIZE-1:0]
dram_port2_write_data[`DRAM_BLOCK_SIZE-1:0],
    output                             dram_port2_acknowledge,
    output                             dram_busy
    );

    // dram access in progress - these signals can be used to stall CPU
during cache misses
    logic               dram_port1_busy;
    logic               dram_port2_busy;

    // dram is busy if either port 1 or port 2 is busy
    assign dram_busy = dram_port1_busy || dram_port2_busy;

    /* verilator lint_off SYNCASYNCNET */
     // following line replaced for quartus
   always_ff @(posedge(clock) or posedge dram_port1_request) begin
     // with:
//   always_ff @(posedge(clock)) begin
      if (dram_port1_request)
     dram_port1_busy <= 1'b1;
      else if (reset || (dram_port1_busy && dram_port1_acknowledge))
         dram_port1_busy <= 1'b0;  //reset port1 busy state
      else
         dram_port1_busy <= dram_port1_busy & !dram_port1_acknowledge;
    end
     // following line replaced for quartus
    always_ff @(posedge(clock) or posedge dram_port2_request) begin
```

```
     // with:
//   always_ff @(posedge(clock)) begin
       if (dram_port2_request)
     dram_port2_busy <= 1'b1;
       else if (reset || (dram_port2_busy && dram_port2_acknowledge))
         dram_port2_busy <= 1'b0;  //reset port2 busy state
       else
         dram_port2_busy <= dram_port2_busy & !dram_port2_acknowledge;
    end
    /* verilator lint_on SYNCASYNCNET */

    // dram interface / memory controller for I-CACHE
    dram_interface dram_interface_icache (
    .clock                      (clock),
    .reset                      (reset),
    .bus_read_enable            (!dram_port1_we && dram_port1_busy),
    .bus_write_enable           (dram_port1_we && dram_port1_busy),
    .bus_address_to_mem         (dram_port1_address),
    .bus_data_to_mem            (dram_port1_write_data),
    .bus_data_from_mem          (dram_port1_read_data),
    .acknowledge_from_mem       (dram_port1_acknowledge)
    );

    // dram interface / memory controller for D-CACHE
    dram_interface dram_interface_dcache (
    .clock                      (clock),
    .reset                      (reset),
    .bus_read_enable            (!dram_port2_we && dram_port2_busy),
    .bus_write_enable           (dram_port2_we && dram_port2_busy),
    .bus_address_to_mem         (dram_port2_address),
    .bus_data_to_mem            (dram_port2_write_data),
    .bus_data_from_mem          (dram_port2_read_data),
    .acknowledge_from_mem       (dram_port2_acknowledge)
    );

endmodule
```

config.sv

```
// This is a common configuration file for SystemVerilog macros to
support the development
// in  Computer Architecture / Organization courses at METU Northern
Cyprus Campus.
// Ali Muhtaroglu
//

////////////////////////////////////////
//              DRAM config           //
////////////////////////////////////////

// DRAM address and data bus/word size in bits
`define DRAM_ADDRESS_SIZE   12
`define DRAM_WORD_SIZE      32
```

```
// Definition of DRAM Block Size in terms of
// # of Data Words. Block size is assumed same
// for the full memory hierarchy.
`define DRAM_BLOCK_SIZE           8

// Definition of DRAM latencies in clock cycles
`define DRAM_READ_ACCESS_TIME    8
`define DRAM_WRITE_ACCESS_TIME   12
`define DRAM_CYCLE_TIME           4

// Definition of DRAM file
`define DRAM_HEX  "simple_test.hex"

//////////////////////////////////////////
//              I-CACHE config          //
//////////////////////////////////////////

// CACHE index and data bus sizes for direct-mapped cache
`define ICACHE_INDEX       3

// number of blocks in cache
`define ICACHE_SIZE        2**`ICACHE_INDEX

`define ITAGMSB            `DRAM_ADDRESS_SIZE-1  //tag msb
`define ITAGLSB            8  //tag lsb

//////////////////////////////////////////
//              D-CACHE config          //
//////////////////////////////////////////

// CACHE index and data bus sizes for direct-mapped cache
`define DCACHE_INDEX       4

// number of blocks in cache
`define DCACHE_SIZE        2**`DCACHE_INDEX

`define DTAGMSB            `DRAM_ADDRESS_SIZE-1  //tag msb
`define DTAGLSB            9  //tag lsb
```

APPENDIX C BENCMARKS

Assembler Python Code

```
# ////////////////////////////////////////////////////////
# ||||||||||||||||||||Welcome to optiRISC||||||||||||||||||||||
# Forked from optiMIPS
# Designers : Baris Guzel 2315935
#             Ilgar Sahin Kocak 2316024
#             Ege Ereren 2152387
# Developed at python3 version 3.8.5
# ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
# \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\


# importing numpy library functions for base conversion operations
from numpy import binary_repr

# Takes register and returns 5bit binary(str) or Error
def binaryRegisters(registerName):
    return {
        "r0": "00000",
        "r1": "00001",
        "r2": "00010",
        "r3": "00011",
        "r4": "00100",
        "r5": "00101",
        "r6": "00110",
        "r7": "00111",
        "r8": "01000",
        "r9": "01001",
        "r10": "01010",
        "r11": "01011",
        "r12": "01100",
        "r13": "01101",
        "r14": "01110",
        "r15": "01111",
        "r16": "10000",
        "r17": "10001",
        "r18": "10010",
        "r19": "10011",
        "r20": "10100",
        "r21": "10101",
        "r22": "10110",
        "r23": "10111",
        "r24": "11000",
        "r25": "11001",
        "r26": "11010",
        "r27": "11011",
        "r28": "11100",
        "r29": "11101",
        "r30": "11110",
        "ra": "11111",
    }.get(registerName, "Error")
```

```
# Takes opcode and returns 6bit binary or Error
def binaryOpcode(Opcode):
    return {
        "nop": "00000000",
        "add": "00001000",
        "sub": "00010000",
        "mul": "00011000",
        "div": "00100000",
        "and": "00101000",
        "or" : "00110000",
        "xor": "00111000",
        "slt": "01000000",
        "sll": "01001000",
        "srl": "01010000",
        "sra": "01011000",
        "mula": "00000111",
        "sb": "00001001",
        "sh": "00010001",
        "sw": "00011001",
        "lb": "00100001",
        "lbs": "01001001",
        "lh": "00101001",
        "lhs": "01010001",
        "lw": "00110001",
        "beq": "00111001",
        "bne": "01000001",
        "addi": "00000011",
        "subi": "00001011",
        "muli": "00010011",
        "ori": "00011011",
        "xori": "00100011",
        "andi": "00101011",
        "slti": "00110011",
        "ssld": "00111011",
        "jal": "00000100",
    }.get(Opcode, "Error")


# Takes op and returns type of the instr more elif statements can be
added to define new types
def insType(Opcode):
    if ((Opcode[5]+Opcode[6]+Opcode[7]) == "000") or
((Opcode[5]+Opcode[6]+Opcode[7]) == "111"):
        return "0"  # R type
    elif ((Opcode[5]+Opcode[6]+Opcode[7]) == "001") or
(Opcode[5]+Opcode[6]+Opcode[7]) == "011":
                # 001 load store 011 rest
        return "1"  # I type
    elif (Opcode[5]+Opcode[6]+Opcode[7]) == "100":
        return "2"  # J type

# Distinguises register order in the I type instruction
def ItypeSelect(op):
```

```
    return {"beq": 1, "bne": 1, "sw": 2, "lhs": 2, "lh": 2, "lb": 2,
"lbs":2, "sh":2, "sb":2, "lw": 2, "addi": 3, "slti": 3}.get(op, 0)


# Label dictionary: Default label is arbitrary data if there is a label
called Default its
# value will be changed to the label address
labelDict = {"Default": "Error"}

# At the start of batch mode reads program.src and records label names
and addresses to labelDict
def findLabelLine(programPath):
    f = open(programPath, "r")
    f1 = f.readlines()
    counter = 0  # starting address 0x80001000 in decimal
    # Search for labels and if it isnt a duplicate writes it to
labelDict
    for x in f1:
        if x.split(":").__len__() == 2:
            if x.split(":")[0] in labelDict.keys():
                print("Same label used twice or more")
                return "0"
            else:
                label = x.split(":")[0]
                labelDict[label] = counter
        counter += 4


# Checks required fields are created inside instr[] list This being
called before constructing each instruction
# For Rtype and Itype there are 4 spaces needed to be created for J it
is 2.
# Returns False when the case is true!!!!
def instrFormat(instrType, instr):
    if (instrType == "Rtype" and instr[0] != "mula" and instr[0] !=
"nop" ) or (instrType == "Itype" and instr[0] != "ssld"):
        if instr.__len__() == 4:
            return False
        else:
            return True
    elif instrType == "Jtype":
        if instr.__len__() == 2:
            return False
        else:
            return True
    elif (instrType == "Itype" and instr[0] == "ssld"):
        if instr.__len__() == 3:
            return False
        else:
            return True
    elif (instrType == "Rtype" and instr[0] == "mula"):
        if instr.__len__() == 5:
            return False
        else:
            return True
    elif (instrType == "Rtype" and instr[0] == "nop"):
```

```
        if instr.__len__() == 1:
            return False
        else:
            return True


# Common function to convert 32 bit binary input to hex and for each
mode prints
# or writes to file
def constructHex(binarycode, mode):
    output = "%0*X" % (8, int(binarycode, 2))
    outputSpace = output[0] + output[1] + " " + output[2] + output[3] +
" " + output[4] + output[5] + " " + output[6] + output[7]
    return outputSpace


# If instruction in R type doesnt match with the op rd rs rt shamt func
format
# Add fix algorithms here if needed
def RtypeFormatFix(instr, counter):
    if instr[0] == "sll":
        instr[4] = instr[3].strip()
        instr[3] = instr[2]
        instr[2] = "$zero"
        if instr[4][:2] == "0x" and int(instr[4][2:], 16) < 32:
            instr[4] = binary_repr(int(instr[4], 16), width=5)
        else:
            try:
                instr[4] = int(instr[4])
                if instr[4] < 32 and instr[4] >= 0:
                    instr[4] = binary_repr(instr[4], width=5)
                else:
                    print(
                        "Enter a shift value between 32 and 0 in line:"
+ str(counter)
                    )
                    return "0"
            except:
                print(
                    "Invalid shift amount. Enter hex or decimal in
line:" + str(counter)
                )
                return "0"
        return instr
    else:  # if not returns the same instr array
        return instr
# Handling Rtype Instruction
# Takes instr array converts each term to binary
# Starts by checking if its psudo instruction converts it to rtype then
checks if instruction taken corretly with instrFormat()
# Function converts special Rtype cases to default Rtype with
RtypeFormatFix() and sends it to constructHex function
# To add new pseudo instruction: add its conversion algorithm to else
case of pseudoCheck(instr[0])
# To add special case Rtype attach new algorithms to required fields
(Follow sll as example) inside RtypeFormatFix()
```

```
# Default R type list instr[] consists these fields in order: opcode rd
rs rt shamt funct
def Rtype(instr, mode, counter):
    if instrFormat("Rtype", instr):
        return "Invalid R type instruction usage in line:" +
str(counter)


    if binaryOpcode(instr[0]) != "Error":
        opcode = binaryOpcode(instr[0])
    else:
        return "Invalid opcode definition in line:" + str(counter)

    if(instr[0] == "nop"):
        rd = "00000"
        ra = "00000"
        rb = "00000"
        rc = "00000"

    else:
        if binaryRegisters(instr[1]) != "Error":
            rd = binaryRegisters(instr[1])
        else:
            print(instr[1])
            return "Invalid rd register definition in line :" +
str(counter)

        if binaryRegisters(instr[2].strip()) != "Error":
            ra = binaryRegisters(instr[2].strip())
        else:
            return "Invalid ra register definition in line:" +
str(counter)

        if binaryRegisters(instr[3].strip()) != "Error":
            rb = binaryRegisters(instr[3].strip())
            rc = "00000"
        elif (instr[0] == "sll" or instr[0] == "srl" or instr[0] ==
"sra"):
            if instr[3][:2] == "0x":
                rc = binary_repr(int(instr[3], 10), width=5)
            else:
                rc = binary_repr(int(instr[3], 16), width=5)
            rb = "00000"
        else:
            return "Invalid rb register definition in line:" +
str(counter)



        if instr[0] == "mula":
            if binaryRegisters(instr[4].strip()) != "Error":
                rc = binaryRegisters(instr[4].strip())
            else:
                return "Invalid rc register definition in line:" +
str(counter)
```

```
    return constructHex(rd + ra + rb + rc + "0000" + opcode, mode)


# Handling J Type Instruction
# Takes instr array converts each term to binary
# Starts by checking if its psudo instruction converts it to jtype
format
# First takes opcode then checks jump address type: hex, decimal, label
# For label uses labelDict dictionary to find address of the label.
Check findLabelLine()
# To add new pseudo instruction: add its conversion algorithm to else
case of pseudoCheck(instr[0])
# To add special case Jtype attach new algorithms to required fields
# Default J type list instr[] consists these fields in order: opcode
jumpaddress
def Jtype(instr, mode, counter):
    if instrFormat("Jtype", instr):
        return "Invalid J type instruction usage" + counter

    if binaryOpcode(instr[0]) != "Error":
        opcode = binaryOpcode(instr[0])
    else:
        return "Invalid opcode definition"

    if instr[1][:2] == "0x":
        addr = binary_repr(int(instr[1], 16), width=19)
    else:
        try:
            instr[1] = int(instr[1])
            if instr[1] <= 33554431 and instr[1] >= -33554432:
                addr = binary_repr(instr[1], width=19)
            else:
                return "Value is out of reach"
        except ValueError:
            if mode == "2":
                return "In interactive mode enter hex or decimal"
            elif mode == "1":
                if instr[1].strip() in labelDict.keys():
                    print("Debug")

                    addr = binary_repr(labelDict[instr[1].strip()] ,
width=32)[13:32]
                    print("address jum:" + addr)
                else:
                    return "Jump location not defined in line:" +
counter

    return constructHex("11111" + addr + opcode, mode)


# For Lw and Sw instructions gets rid of ( and )
def memoryTypeFix(instr, counter):
    if ItypeSelect(instr[0]) == 2:
        try:
```

```
            lwSplit = instr[2].split("(")[:1] +
instr[2].split("(")[1].split(")")[:1]
            instr = instr[0:2] + lwSplit
            instr[2], instr[3] = instr[3], instr[2]
            return instr
        except IndexError:
            return "0"
    else:
        return instr


# Handling I Type Instruction
# Takes instr array converts each term to binary
# Starts by checking if its psudo instruction and converts it to Itype
format
# For label uses labelDict dictionary to find address of the label
# To add new pseudo instruction: add its conversion algorithm to else
case of pseudoCheck(instr[0])
# There are 3 types setted inside ItypeSelect() function.
(lw,sw/aritmetic/branch)
# If wanted to use different order than default add more types and
instructions to ItypeSelect
# And define a function similar to memoryTypeFix() to convert it to
default I type construct
# Default I type list instr[] consists these fields in order: opcode rs
rt address/offset
def Itype(instr, mode, counter):
    # Additional function for Lw Sw dependencies
    instr = memoryTypeFix(instr, counter)

    if instrFormat("Itype", instr):
        return "Invalid I type instruction usage in Line: " +
str(counter)

    if binaryOpcode(instr[0]) != "Error":
        opcode = binaryOpcode(instr[0])
    else:
        return "Invalid opcode definition in Line: " + str(counter)

    if binaryRegisters(instr[1]) != "Error":
        rd = binaryRegisters(instr[1])
    else:
        if ItypeSelect(instr[0]) == 1:
            return "Invalid ra register definition in Line: " +
str(counter)
        else:
            return "Invalid rd register definition in Line: " +
str(counter)

    #print(instr[2])
    if binaryRegisters(instr[2].lstrip()) != "Error":
        ra = binaryRegisters(instr[2].lstrip())
    else:
        if ItypeSelect(instr[0]) == 1:
            return "Invalid rd register definition in Line: " +
str(counter)
```

```
        else:
            return "Invalid ra register definition in Line: " +
str(counter)

    # Checks if branch address is written as hex decimal or a label
    if(instr[0] != "ssld"):
        try:
            instr[3] = instr[3].strip()
            #print("demo " + instr[3])
            print(labelDict)
            if instr[3][:2] == "0x" and int(instr[3][2:], 16) <= 65534:
                addr = binary_repr(int(instr[3], 16), width=14)
                print("address branch:" + addr)
            else:
                try:
                    instr[3] = int(instr[3])
                    if instr[3] <= 32767 and instr[3] >= -32768:
                        addr = binary_repr(instr[3], width=14)
                    else:
                        return "Out of reach"
                except:
                    if mode == "2":
                        return "In interactive mode enter hex or
decimal"
                    elif mode == "1":
                        print("yes: " + instr[3].strip())
                        #print("guys: " + labelDict['lbl1'])
                        if instr[3].strip() in labelDict.keys():
                            print("its in")
                            instrLocation = counter * 4
                            branchAddr =
int(labelDict[instr[3].strip()])
                            branchDistance = branchAddr - instrLocation
                            addr = binary_repr(int(branchAddr),
width=32)[18:32]
                            #addr = binary_repr(int(branchDistance),
width=32)[14:30]
                        #       print("instr loc: " + instrLocation)
                        #       print("branch addr: " + branchAddr)
                        #       print("address branchh:" + addr)
                            print(addr)
                            return constructHex(rd + ra + addr +
opcode, mode)
                        else:
                            return "Branch location not defined in
line:" + str(counter)
        except ValueError:
            return "I type format is instr reg, reg, addr in line:" +
str(counter)
    else:
        addr = "00000000000000"

    return constructHex(rd + ra + addr + opcode, mode)
    #if ItypeSelect(instr[0]) == 1:  # branch instructions
    #    return constructHex(opcode + rt + rs + addr, mode)
    #elif ItypeSelect(instr[0]) == 3 or ItypeSelect(instr[0]) == 2:
```

```
    #    return constructHex(opcode + rs + rt + addr, mode)
    #else:
    #    return "There were an internal Error!"
# Common builder
# Clears whitespaces gets rid of comments and splits instruction to
each field then
# Sends the splittedInstruction array to corresponding type's function
# There are 3 types defined: R type, I type, J type if another type
needed to be added:
# Add new instructions opcodes to binarycode(), define the new type
inside insType() and thoes instructions to insType()
def builder(instr, mode, counter):
    try:
        instruction = instr.strip().lower()
        instruction = instruction.split("#")[0]
        if instruction.split(":").__len__() == 2:
            instruction = instruction.split(":")[1]
        elif instruction.split(":").__len__() > 2:
            return "More than one ':' in line:" + str(counter)
        splittedInstruction = (
            instruction.split(",")[0].split() +
instruction.split(",")[1:]
        )
        # uncomment below line to print each elements send to
construction functions
        # print(splittedInstruction)
        print(splittedInstruction[0])
        if insType(binaryOpcode(splittedInstruction[0])) == "0":
            return Rtype(splittedInstruction, mode, counter)
        elif insType(binaryOpcode(splittedInstruction[0])) == "1":
            return Itype(splittedInstruction, mode, counter)
        elif insType(binaryOpcode(splittedInstruction[0])) == "2":
            return Jtype(splittedInstruction, mode, counter)
        else:
            return "Invalid or not defined instruction!"
    except IndexError:
        return ("Code should be written without empty lines In line:",
+str(counter))


# This will be initialized when program started. Asks batch mode or
interactive mode
if __name__ == "__main__":
    a = "default"
    print("Disclaimer: for batch mode create a file called program.src
in the same dir")

    while a != "q":
        mode = input("Enter 1 for batch 2 for interact1ive mode \n")
        if mode == "1":
            programPath = "iaxpy.src"
            f = open(programPath, "r")
            programFile = f.readlines()
            results = open("output.obj", "w")
            counter = 0  # line counter
            try:
```

```
                findLabelLine(programPath)
            except:
                print("Error occured from label definitions" +
str(counter))
            try:
                # Uncomment below line to print collected labels
                # print(labelDict)
                for x in programFile:
                    counter += 1
                    results.writelines(str(builder(x, mode, counter)) +
"\n")
                print("File is written")
                a = "q"
            except:
                print("Error occured program terminated")
                a = input("Press 'q' to quit or press 's' to select
mode again \n")
        elif mode == "2":
            a = "default"
            while a != "q" and a != "s":
                try:
                    instruction = input("Enter one line of instruction
\n")
                    print(builder(instruction, mode, "0") + "\n")
                    a = input("Press 'q' to quit or press 's' to select
mode again \n")
                except TypeError:
                    print("AAAError occured program terminated")
        else:
            print("Invalid  mode number!!\n", "Try 1 or 2")
```

IAXPY Assembly Code

```
        addi r1, r0, 2
        addi r3, r0, 0xA0
        mul r12, r1, r1
        muli r12, r12, 4
        add r4, r12, r3
        addi r2, r0, 3
load:   beq r5, r12, exit
        add r6, r3, r5
        add r9, r4, r5
        lw r7, 0(r6)
        mul r8, r7, r2
        addi r5, r5, 4
        lw r10, 0(r9)
        add r11, r10, r8
        sw r11, 0(r9)
        jal load
        nop
exit:   jal exit
```

IAXPY HEX

```
08 00 04 03
18 00 A0 03
60 42 00 18
63 00 04 13
23 06 00 08
10 00 03 03
2B 00 44 39
30 CA 00 08
49 0A 00 08
39 80 00 31
41 C4 00 18
29 40 04 03
52 40 00 31
5A 90 00 08
5A 40 00 19
F8 00 18 04
00 00 00 00
F8 00 44 04
```

Text Parse Assembly Code

```
        addi r9, r0, 0xA0
        addi r1, r0, 32
        addi r2, r0, 8
        addi r3, r0, 10
        addi r4, r0, 9
        addi r5, r0, 11
        addi r6, r0, 12
        addi r7, r0, 13
loop:   lb r8, 0(r9)
        beq r8, r0, exit
        addi r9, r9, 1
        beq r8, r1, compare
        beq r8, r2, compare
        beq r8, r3, compare
        beq r8, r4, compare
        beq r8, r5, compare
        beq r8, r6, compare
        beq r8, r7, compare
        lb r8, 0(r9)
        jal loop
compare:lb r10, 0(r9)
        nop
        beq r10, r8, conc
        jal loop
conc:   sb r1, -1(r9)
        addi r11, r0, 0
        addi r11, r9, 0
cloop:  lb r12, 0(r11)
        addi r11, r11, 1
        beq r12, r0, loop
        lb r13, 0(r11)
        sb r13, -1(r11)
        jal cloop
exit:   jal exit
```

Text Parse HEX

```
48 00 A0 03
08 00 20 03
10 00 08 03
18 00 0A 03
20 00 09 03
28 00 0B 03
30 00 0C 03
38 00 0D 03
42 40 00 21
40 00 90 39
4A 40 01 03
40 40 54 39
40 80 54 39
40 C0 54 39
41 00 54 39
41 40 54 39
41 80 54 39
41 C0 54 39
42 40 00 21
F8 00 20 04
52 40 00 21
00 00 00 00
52 00 64 39
F8 00 20 04
0A 7F FF 09
58 00 00 03
5A 40 00 03
62 C0 00 21
5A C0 01 03
60 00 20 39
6A C0 00 21
00 00 00 00
00 00 00 00
6A FF FF 09
F8 00 70 04
F8 00 90 04
```

Merge Sorter Assembly Code

```
        addi r1,  r0,  0x320    #array start address
        addi r20, r0,  0x960    #L[i]
        addi r21, r0,  0xFA0    #R[i]
        addi r5,  r0,  1        #less than compare =1
sizel:  lw   r3,  0(r1)
        addi r1,  r1,  4
        addi r2,  r2,  1
        bne  r3,  r0,  sizel
        addi r2,  r2,  -1
        addi r1,  r0,  0x320    #reset start address
        addi r3,  r0,  1        #curr_size = 1
        addi r7,  r2,  -1       #n-1
for1:   slt  r8,  r7,  r3
        nop
        beq  r8,  r5,  exit     #curr_size<=n-1
        addi r4,  r0,  0
for2:   slt  r8,  r4,  r7
        nop
        bne  r8,  r5,  bfor1
        add  r28, r4,  r3
        muli r25, r3,  2        #2*curr_size
        addi r6,  r28, -1       #r6 = left+curr-1
        slt  r8,  r6,  r7
        nop
        beq  r8,  r5,  less
        add  r9,  r7,  r0       #mid = r7 = n-1
proc:   add  r27, r25, r4
        addi r27, r27, -1       #left+2*curr-1
        slt  r8,  r27, r7
        nop
        beq  r8,  r5,  less2
        add  r12, r0,  r7
        jal  merge
less:   nop
        add  r9,  r6,  r0       #mid = r6 =left+curr-1
        jal  proc
bfor1:  muli r3,  r3,  2
        jal for1
bfor2:  add  r4,  r4,  r25      #for2 increment logic
        jal  for2
less2:  add  r12, r0,  r27
merge:  muli r15, r4,  4        #l*4
        add  r10, r9,  r0
        sub  r10, r10, r4       #n1 without 4 times
        muli r16, r9,  4        #m*4
        muli r10, r10, 4        #? n1*4 ok
        sub  r11, r12, r9       #n2 without 4 times
        addi r11, r11, -1
        muli r11, r11, 4        #n2*4
        add  r16, r16,  r1
        add  r15, r15, r1       #l+startaddr arr
        addi r16, r16, 4        #m+1+startadrs arr
for3:   slt  r8,  r13, r10
        nop
```

```
        bne   r8,   r5,   for4
        add   r18,  r13,  r20       #r18+i L staradr
        add   r15,  r15,  r13       #startadrs+l
        lw    r29,  0(r15)
        addi  r13,  r13,  4
        nop
        sw    r29,  0(r18)
        jal   for3
for4:   nop
        slt   r8,   r14,  r11
        nop
        bne   r8,   r5,   whil1
        add   r19,  r14,  r21
        add   r16,  r16,  r14
        lw    r29,  0(r16)
        addi  r14,  r14,  4
        nop
        sw    r29,  0(r19)
        jal   for4
whil1:  addi  r14,  r0,   0
        addi  r13,  r0,   0
        add   r17,  r4,   r1        #k+startadr = prev r15
        addi  r17,  r17,  -4
contw1: slt   r8,   r13,  r10
        nop
        bne   r8,   r5,   whil3      # i< n1
        slt   r8,   r14,  r11
        nop
        bne   r8,   r5,   whil2      # j< n2
        add   r15,  r13,  r20       # i+Lstartaddr
        add   r16,  r14,  r21       # k+Rstartaddr
        addi  r17,  r17,  4         #k++
        lw    r29,  0(r15)          #L[i]
        lw    r28,  0(r16)          #R[j]
        slt   r8,   r29,  r28
        nop
        bne   r8,   r5,   else
        sw    r29,  0(r17)
        addi  r13,  r13,  4         #i++
        jal   contw1
else:   sw    r28,  0(r17)
        addi  r14,  r14,  4
        jal   contw1
whil2:  nop
        slt   r8,   r13,  r10
        nop
        bne   r8,   r5,   bfor2
exit
        addi  r17,  r17,  4
        add   r15,  r15,  r13       #new L[i addres]
        lw    r29,  0(r15)          #L[i]
        nop
        addi  r13,  r13,  4
        sw    r29,  0(r17)
        jal   whil2
whil3:  nop
```

```
        slt  r8, r14, r11
        nop
        bne  r8, r5,  bfor2
        addi r17, r17, 4
        add  r16, r16, r14      #new R[j addres]
        nop
        lw   r28, 0(r16)        #R[j]
        addi r14, r14, 4
        sw   r28, 0(r17)
        jal  whil3
exit:   nop
        jal  exit
```

Merge Sorter HEX

```
08 03 20 03
A0 09 60 03
A8 0F A0 03
28 00 01 03
18 40 00 31
08 40 04 03
10 80 01 03
18 00 10 41
10 BF FF 03
08 03 20 03
18 00 01 03
38 BF FF 03
41 C6 00 40
00 00 00 00
41 41 DC 39
20 00 00 03
41 0E 00 40
00 00 00 00
41 40 90 41
E1 06 00 08
C8 C0 02 13
37 3F FF 03
41 8E 00 40
00 00 00 00
41 40 84 39
49 C0 00 08
DE 48 00 08
DE FF FF 03
46 CE 00 40
00 00 00 00
41 40 A0 39
60 0E 00 08
F8 00 A4 04
00 00 00 00
49 80 00 08
F8 00 68 04
18 C0 02 13
F8 00 30 04
21 32 00 08
```

```
F8 00 40 04
60 36 00 08
79 00 04 13
52 40 00 08
52 88 00 10
82 40 04 13
52 80 04 13
5B 12 00 10
5A FF FF 03
5A C0 04 13
84 02 00 08
7B C2 00 08
84 00 04 03
43 54 00 40
00 00 00 00
41 40 F8 41
93 68 00 08
7B DA 00 08
EB C0 00 31
6B 40 04 03
00 00 00 00
EC 80 00 19
F8 00 D0 04
00 00 00 00
43 96 00 40
00 00 00 00
41 41 24 41
9B AA 00 08
84 1C 00 08
EC 00 00 31
73 80 04 03
00 00 00 00
EC C0 00 19
F8 00 F8 04
70 00 00 03
68 00 00 03
89 02 00 08
8C 7F FC 03
43 54 00 40
00 00 00 00
41 41 B0 41
43 96 00 40
00 00 00 00
41 41 84 41
7B 68 00 08
83 AA 00 08
8C 40 04 03
EB C0 00 31
E4 00 00 31
47 78 00 40
00 00 00 00
41 41 78 41
EC 40 00 19
6B 40 04 03
F8 01 34 04
E4 40 00 19
```

```
73 80 04 03
F8 01 34 04
00 00 00 00
43 54 00 40
00 00 00 00
41 40 98 41
8C 40 04 03
7B DA 00 08
EB C0 00 31
00 00 00 00
6B 40 04 03
EC 40 00 19
F8 01 84 04
00 00 00 00
43 96 00 40
00 00 00 00
41 40 98 41
8C 40 04 03
84 1C 00 08
00 00 00 00
E4 00 00 31
73 80 04 03
E4 40 00 19
F8 01 B0 04
00 00 00 00
F8 01 DC 04
```

APPENDIX D VERILATOR TESTS

Sim_main.cpp

```
// Booth Mulitplier Top level verilator simulation file:
// EEE 446 Spring 2021
// Ali Muhtaroglu, Middle East Technical University - Northern Cyprus Campus

#include <stdio.h>
#include <verilated.h>
#include <verilated_vcd_c.h>
#include "testbench.h"
#include "Vtop.h"
#include "Vtop___024root.h"
// Top level interface signals defined here:
#define Op          Op
// Internal signals defined here:
// Note systemverilog design hierarchy can be traced by appending __DOT__ at
every level:
#define MemRead         top__DOT__MemRead
#define MemWrite        top__DOT__MemWrite
#define PCSrc           top__DOT__PCSrc
#define MemToReg        top__DOT__MemToReg
#define RegWrite        top__DOT__RegWrite
#define ALUOp           top__DOT__ALUOp
#define ALUSrc          top__DOT__ALUSrc
#define RbSelect        top__DOT__RbSelect
#define IdEx            top__DOT__datapath__DOT__IdEx
#define RF              top__DOT__datapath__DOT__RF
#define Alu1out         top__DOT__datapath__DOT__Alu1out
#define PC              top__DOT__datapath__DOT__PC
#define mem_stall_flag  top__DOT__datapath__DOT__mem_stall_flag
#define dcache_sram_write
top__DOT__memory__DOT__dcache_controller__DOT__dcache_sram_write
#define dcache_array
top__DOT__memory__DOT__dcache_controller__DOT__data_cache_sram__DOT__dcache_sra
m
#define dcache_write
top__DOT__memory__DOT__dcache_controller__DOT__dcache_dirty_write
#define dram_port2_write
top__DOT__memory__DOT____Vcellinp__dram_controller__dram_port2_write_data
#define RF              top__DOT__datapath__DOT__RF
// In case you would like he simulator to do operations conditional to DEBUG
mode:
#define DEBUG       1

// Note the use of top level design name here after 'V' as class type:
class   TOPLEVEL_TB : public TESTBENCH<Vtop> {

  long m_tickcount;

public:

  TOPLEVEL_TB(void) {
  }

  void tick(void) {

    TESTBENCH<Vtop>::tick();
    m_tickcount++;
```

```
};

TOPLEVEL_TB *tb = new TOPLEVEL_TB;
void printer(int counter, int type);
int main(int argc, char** argv, char** env){


  long clock_count = 0;
  long instr_count = 0;
  long read_count = 0;
  long write_count = 0;
  int rw_count = 0;
  int rw_miss_count = 0;
  float rw_miss_percentage = 0;
  float CPI = 0;
  float exec_t = 0;
  float enj = 0;

  // Initialize Verilators variables
  Verilated::commandArgs(argc, argv);
  // Create an instance of our module under test

  // Message to standard output that test is starting:
  printf("Executing the test ...\n");

  // Data will be dumped to trace file in gtkwave format to look at waveforms
later:
  tb->opentrace("lab2_waveforms.vcd");

  // Note this message will only be output if we are in DEBUG mode:
  if (DEBUG) printf("Giving the system 1 cycle to initialize with reset...\n");
  int PC = 0;
  int error_count = 0;
  int dcache_pass = 0;
  int stall_dummy = 0;
  int check_next_write = 0;
  int dcache_counter = 0;
  int mem_stall_count = 0;
  tb->reset();
  clock_count++;
  // Hit that reset button for one clock cycle:
  for(int x=0; tb->m_topsim->rootp->PC != 72; x++){
    rw_count = ((tb->m_topsim->rootp->Op == 49) //lw
            || (tb->m_topsim->rootp->Op == 33) || (tb->m_topsim->rootp->Op ==
41) //lb lh
            || (tb->m_topsim->rootp->Op == 25) || (tb->m_topsim->rootp->Op ==
17) //sw sh
            || (tb->m_topsim->rootp->Op == 9)) ? rw_count+1:rw_count;// sb
    //printf(" OP: %02x ", tb->m_topsim->rootp->Op);
    //printf(" PC: %02x ", tb->m_topsim->rootp->PC);
    if(tb->m_topsim->rootp->dcache_write == 1){
        check_next_write = 1;
    }
    //printf("write: %d %d \n",tb->m_topsim->rootp->dcache_write,
check_next_write);

    mem_stall_count = (tb->m_topsim->rootp->mem_stall_flag == 1) ?
mem_stall_count+1:mem_stall_count;

    if(tb->m_topsim->rootp->mem_stall_flag == 1 && stall_dummy == 0){
        rw_miss_count += 1;
```

```
        stall_dummy = 1;
    }else if(tb->m_topsim->rootp->mem_stall_flag == 0){
        stall_dummy = 0;
    }
    if(tb->m_topsim->rootp->Op != 0 && tb->m_topsim->rootp->mem_stall_flag ==
0)
        instr_count++;

    tb->tick();
    clock_count++;

    if(check_next_write){
        check_next_write = 0;
        dcache_counter ++;
        printer(dcache_counter, 1);
    }
  }
  /*
  // Used in Text Parse
    for(int i=0;i<16;i++){
        if(tb->m_topsim->rootp->dcache_array[i][0] != 0){
            if( tb->m_topsim->rootp->dcache_array[i][0] != 544366946)
                error_count =1;
            if( tb->m_topsim->rootp->dcache_array[i][1] != 1931503883)
                error_count =1;
            if( tb->m_topsim->rootp->dcache_array[i][2] != 1963616032)
                error_count =1;
            if( tb->m_topsim->rootp->dcache_array[i][3] != 1701607968)
                error_count =1;
            if( tb->m_topsim->rootp->dcache_array[i][4] != 0)
                error_count =1;
            if( tb->m_topsim->rootp->dcache_array[i][5] != 0)
                error_count =1;
            if( tb->m_topsim->rootp->dcache_array[i][6] != 0)
                error_count =1;
            if( tb->m_topsim->rootp->dcache_array[i][7] != 0)
                error_count =1;
            if(error_count != 0)
                break;
            }
        }
        */
  if(error_count != 0)
      printf(" Failed #: %d\n",error_count);
  else{
      printer(0, 0);
      printf(" \n Execution completed successfully (simulation waveforms in
.vcd file) ... !\n");
      printf(" Elapsed Clock Cycles: %ld\n",clock_count);
      printf(" Total Number of Read Write Access: %d\n",rw_count);
      printf(" Total Number of Memory Stall Clock Cycles:
%d\n",mem_stall_count);
      rw_miss_percentage = ((float)rw_miss_count/rw_count)*100;
      printf(" Read Write Miss #: %d Read Write Miss Percentage: %f
%%\n",rw_miss_count, rw_miss_percentage);
      CPI = (float)clock_count/instr_count;
      printf(" Total # of instr: %ld CPI: %.3f\n",instr_count, CPI);
      exec_t = clock_count*1.92*(0.001);
      printf(" Execution Time: %f (ns) \n",exec_t);
      enj = exec_t*0.876;
      printf(" Energy: %f(uJ)\n",enj);
  }
```

```
  exit(EXIT_SUCCESS);
}
void printer(int counter, int type){
    if(type == 0){
        printf("RF Status: %dth Call\n", counter);
        for(int i = 0 ; i<32; i +=4){
            int j = 0;
            for(j =0; j<8; j++){
                if(j+i > 31)
                    break;
                printf("r%d = %07x  \t",j+i,tb->m_topsim->rootp->RF[j+i]);
                }
            printf("\n");
            if(j+i > 31)
                break;
        }
    }else if(type == 1){
        printf("Non Zero DCache Blocks : %dth Call\n", counter);

        int k=0;
        for(int i = 0; i < 16; i++){
            if(tb->m_topsim->rootp->dcache_array[i][k] != 0){
            printf("DCache Block[%d]: ",i);
                for(k =0; k < 8; k++){
                    printf("%d= %d\t ",k,tb->m_topsim->rootp-
>dcache_array[i][k] );
                }
                k= 0;
            printf("\n");
            }
        }
    }

printf("_____________________________________________________________________________
__");

printf("_____________________________________________________________________________
__\n");
}
```

IAXPY Verilator Test

```
Non Zero DCache Blocks : 15th Call
DCache Block[5]: 0= 1    1= 2    2= 3    3= 4    4= 5    5= 6    6= 7    7= 8
DCache Block[6]: 0= 9    1= 10   2= 11   3= 12   4= 13   5= 14   6= 15   7= 16
DCache Block[7]: 0= 4    1= 8    2= 12   3= 16   4= 20   5= 24   6= 28   7= 32
DCache Block[8]: 0= 36   1= 40   2= 44   3= 48   4= 52   5= 56   6= 60   7= 16

Non Zero DCache Blocks : 16th Call
DCache Block[5]: 0= 1    1= 2    2= 3    3= 4    4= 5    5= 6    6= 7    7= 8
DCache Block[6]: 0= 9    1= 10   2= 11   3= 12   4= 13   5= 14   6= 15   7= 16
DCache Block[7]: 0= 4    1= 8    2= 12   3= 16   4= 20   5= 24   6= 28   7= 32
DCache Block[8]: 0= 36   1= 40   2= 44   3= 48   4= 52   5= 56   6= 60   7= 64

RF Status: 0th Call
r0 = 0000000    r1 = 0000004    r2 = 0000003    r3 = 00000a0    r4 = 00000e0    r5 = 0000040    r6 = 00000dc    r7 = 0000010
r4 = 00000e0    r5 = 0000040    r6 = 00000dc    r7 = 0000010    r8 = 0000030    r9 = 000011c    r10 = 0000010   r11 = 0000040
r8 = 0000030    r9 = 000011c    r10 = 0000010   r11 = 0000040   r12 = 0000040   r13 = 0000000   r14 = 0000000   r15 = 0000000
r12 = 0000040   r13 = 0000000   r14 = 0000000   r15 = 0000000   r16 = 0000000   r17 = 0000000   r18 = 0000000   r19 = 0000000
r16 = 0000000   r17 = 0000000   r18 = 0000000   r19 = 0000000   r20 = 0000000   r21 = 0000000   r22 = 0000000   r23 = 0000000
r20 = 0000000   r21 = 0000000   r22 = 0000000   r23 = 0000000   r24 = 0000000   r25 = 0000000   r26 = 0000000   r27 = 0000000
r24 = 0000000   r25 = 0000000   r26 = 0000000   r27 = 0000000   r28 = 0000000   r29 = 0000000   r30 = 0000000   r31 = 0000044

Execution completed successfully (simulation waveforms in .vcd file) ... !
Elapsed Clock Cycles: 544
Total Number of Read Write Access: 85
Total Number of Memory Stall Clock Cycles: 259
Read Write Miss #: 7 Read Write Miss Percentage: 8.235294 %
Total # of instr: 199 CPI: 2.734
Execution Time: 1.044480 (ns)
Energy: 0.914964(uJ)
```

Figure 2 IAXPY Benchmark Result Screen Capture

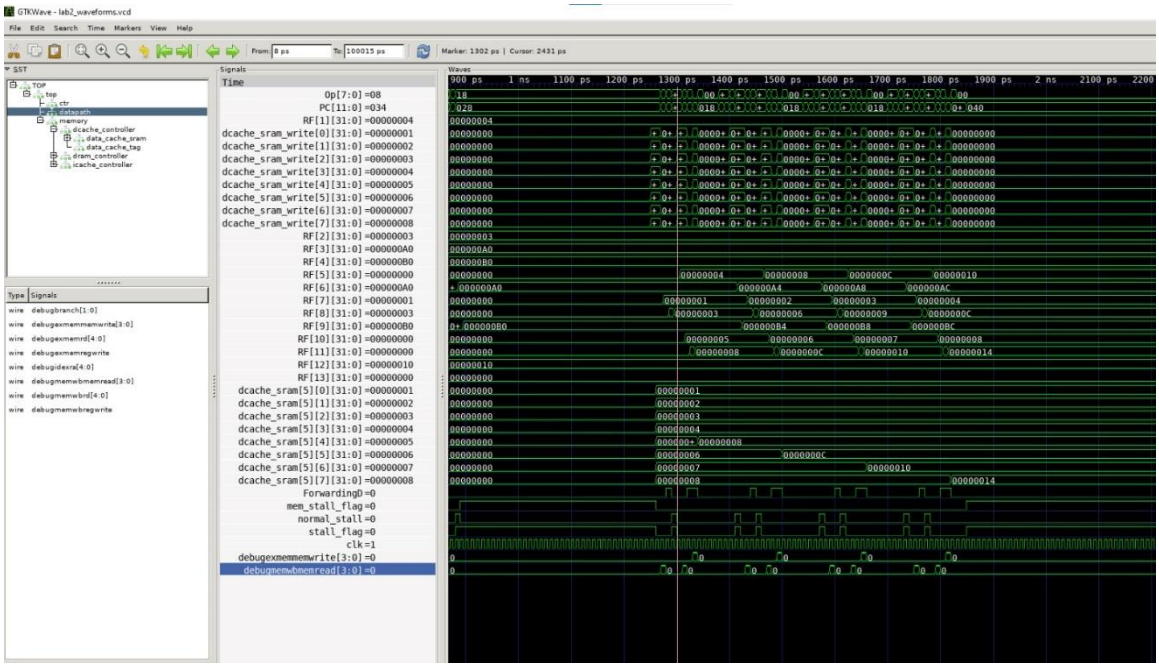


Figure 3 IAXPY GTKWave Waveform

Text Parse Verilator Test

```
Non Zero DCache Blocks : 24th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 7a0a0a0a     4= 000656c      5= 0000000      6= 0000000      7= 0000000

Non Zero DCache Blocks : 25th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 7a7a0a0a     4= 000656c      5= 0000000      6= 0000000      7= 0000000

Non Zero DCache Blocks : 26th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 6c7a0a0a     4= 000656c      5= 0000000      6= 0000000      7= 0000000

Non Zero DCache Blocks : 27th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 6c7a0a0a     4= 0006565      5= 0000000      6= 0000000      7= 0000000

Non Zero DCache Blocks : 28th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 6c7a0a0a     4= 0000065      5= 0000000      6= 0000000      7= 0000000

Non Zero DCache Blocks : 29th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 6c7a0a20     4= 0000065      5= 0000000      6= 0000000      7= 0000000

Non Zero DCache Blocks : 30th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 6c7a7a20     4= 0000065      5= 0000000      6= 0000000      7= 0000000

Non Zero DCache Blocks : 31th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 6c6c7a20     4= 0000065      5= 0000000      6= 0000000      7= 0000000

Non Zero DCache Blocks : 32th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 656c7a20     4= 0000065      5= 0000000      6= 0000000      7= 0000000

Non Zero DCache Blocks : 33th Call
DCache Block[5]: 0= 20726162     1= 7320690b     2= 750a6720     3= 656c7a20     4= 0000000      5= 0000000      6= 0000000      7= 0000000

RF Status: 0th Call
r0 = 0000000    r1 = 0000020    r2 = 0000008    r3 = 000000a    r4 = 0000009    r5 = 000000b    r6 = 000000c    r7 = 000000d
r4 = 0000009    r5 = 000000b    r6 = 000000c    r7 = 000000d    r8 = 0000000    r9 = 00000b0    r10 = 000000a   r11 = 00000b2
r8 = 0000000    r9 = 00000b0    r10 = 000000a   r11 = 00000b2   r12 = 0000000   r13 = 0000000   r14 = 0000000   r15 = 0000000
r12 = 0000000   r13 = 0000000   r14 = 0000000   r15 = 0000000   r16 = 0000000   r17 = 0000000   r18 = 0000000   r19 = 0000000
r16 = 0000000   r17 = 0000000   r18 = 0000000   r19 = 0000000   r20 = 0000000   r21 = 0000000   r22 = 0000000   r23 = 0000000
r20 = 0000000   r21 = 0000000   r22 = 0000000   r23 = 0000000   r24 = 0000000   r25 = 0000000   r26 = 0000000   r27 = 0000000
r24 = 0000000   r25 = 0000000   r26 = 0000000   r27 = 0000000   r28 = 0000000   r29 = 0000000   r30 = 0000000   r31 = 0000054

Execution completed successfully (simulation waveforms in .vcd file) ... !
Elapsed Clock Cycles: 1149
Total Number of Read Write Access: 166
Total Number of Memory Stall Clock Cycles: 222
Read Write Miss #: 6 Read Write Miss Percentage: 3.614458 %
Total # of instr: 376 CPI: 3.056
Execution Time: 2.206080 (ns)
Energy: 1.932526(uJ)
```

Figure 4 Text Parse Benchmark Result Screen Capture

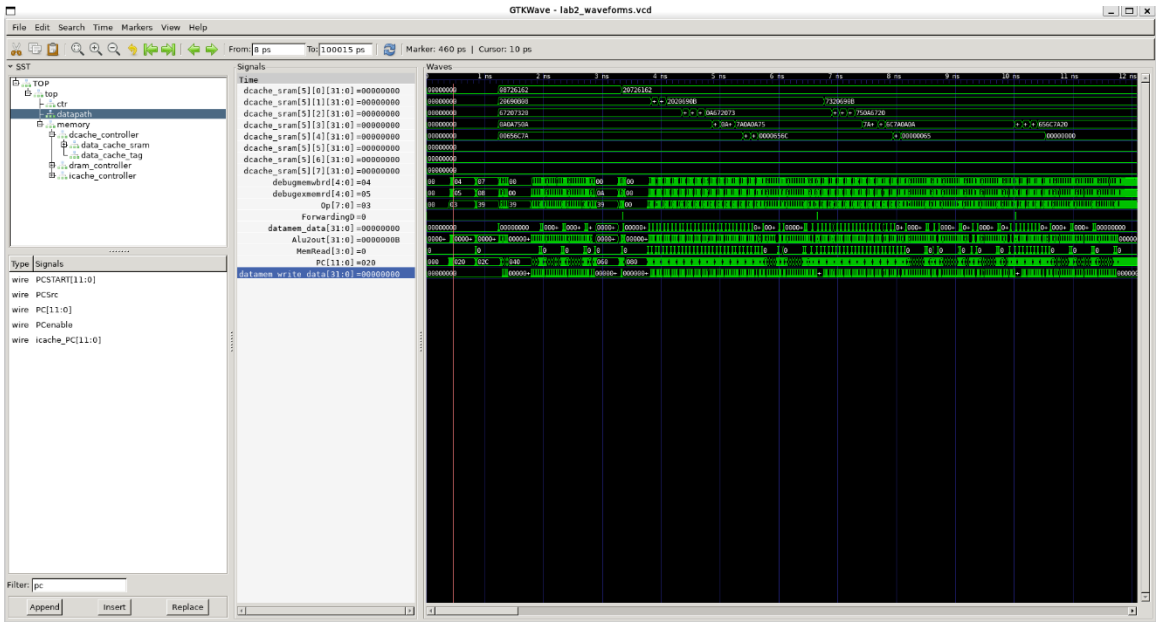


Figure 5 Text Parse GTKWave Waveform

Merge Sort+CRC Verilator Test

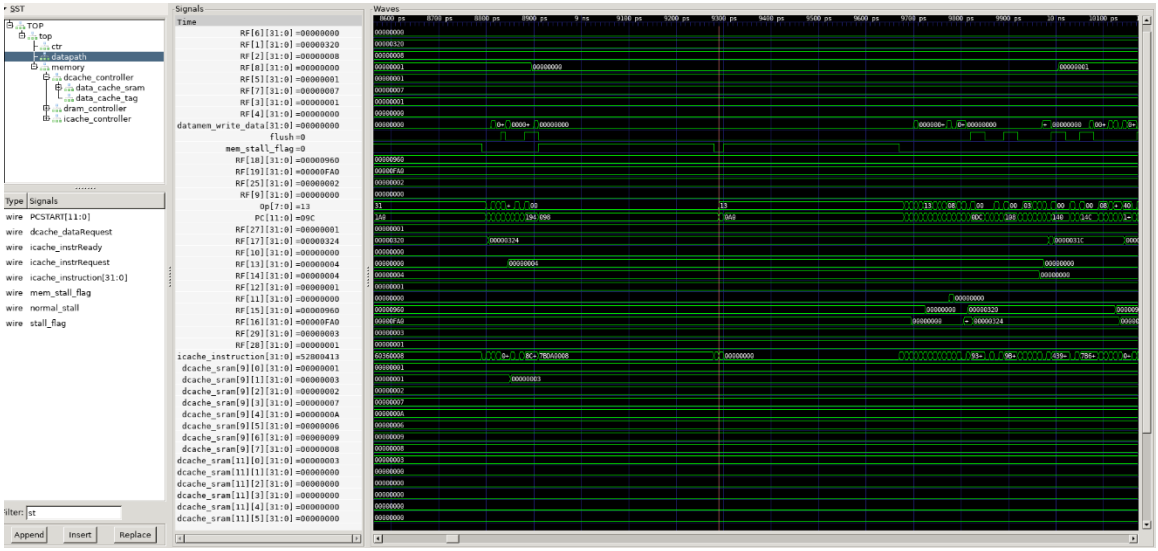
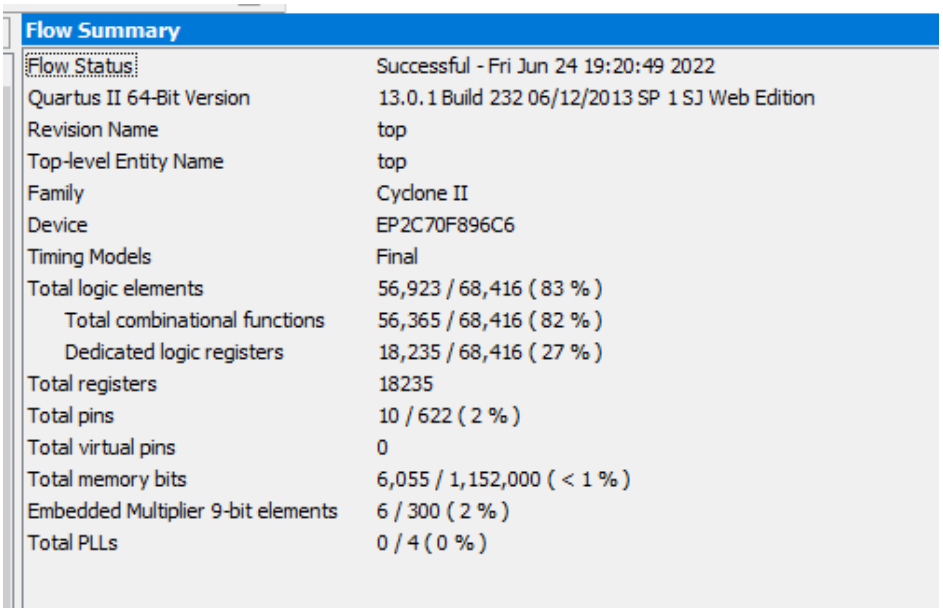


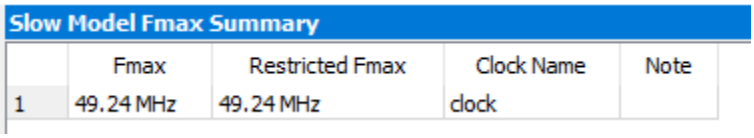
Figure 6 Merge Sort GTKWave Waveform

APPENDIX E QUARTUS II TESTS



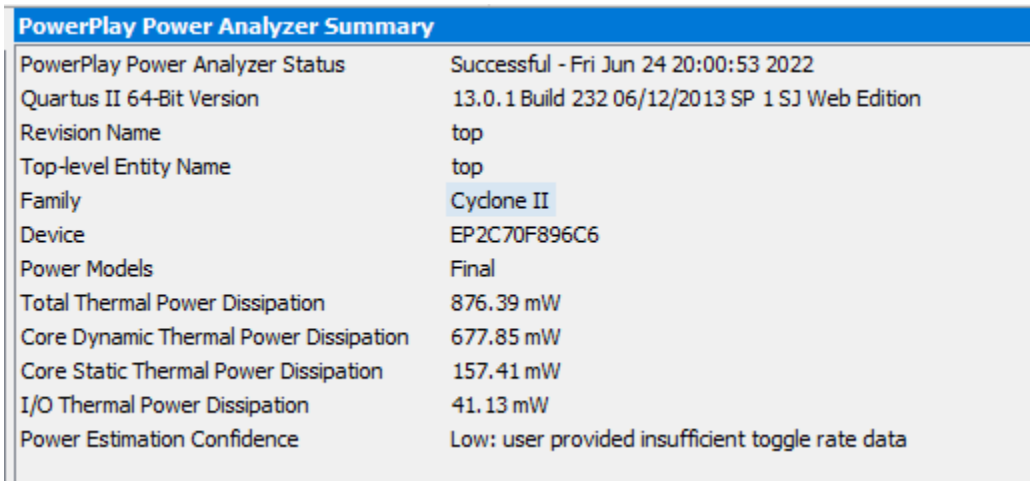
Flow Summary	
Flow Status	Successful - Fri Jun 24 19:20:49 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	top
Top-level Entity Name	top
Family	Cyclone II
Device	EP2C70F896C6
Timing Models	Final
Total logic elements	56,923 / 68,416 (83 %)
Total combinational functions	56,365 / 68,416 (82 %)
Dedicated logic registers	18,235 / 68,416 (27 %)
Total registers	18235
Total pins	10 / 622 (2 %)
Total virtual pins	0
Total memory bits	6,055 / 1,152,000 (< 1 %)
Embedded Multiplier 9-bit elements	6 / 300 (2 %)
Total PLLs	0 / 4 (0 %)

Figure 7 Quartus II Simulation Result Summary



Slow Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	49.24 MHz	49.24 MHz	clock	

Figure 8 Fmax



PowerPlay Power Analyzer Summary	
PowerPlay Power Analyzer Status	Successful - Fri Jun 24 20:00:53 2022
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	top
Top-level Entity Name	top
Family	Cyclone II
Device	EP2C70F896C6
Power Models	Final
Total Thermal Power Dissipation	876.39 mW
Core Dynamic Thermal Power Dissipation	677.85 mW
Core Static Thermal Power Dissipation	157.41 mW
I/O Thermal Power Dissipation	41.13 mW
Power Estimation Confidence	Low: user provided insufficient toggle rate data

Figure 9 PowerPlay Power Analyzer Summary