A REPORT ON

IMPLEMENTATION OF RISC-V PROCESSOR

BY

SUKRUTH S

2020H1400236H

M.E. EMBEDDED SYSTEMS

Prepared in fulfilment of the

(MEL G624)

Advanced VLSI Architecture

BIRLA INSTITUTE OF TECHNOLOGY & SCIENCE, PILANI

(NOVEMBER 2021)

ABSTRACT

RISC-V is a new instruction set architecture used in various devices, from tiny microcontrollers to high-performance CPUs. RISC-V is an open standard instruction set architecture (ISA) based on the reduced instruction set computer (RISC). Unlike most previous ISA designs, the RISC-V ISA is free to use and is distributed under open-source licencing. RISC-V hardware is available from several firms, open-source operating systems with RISC-V support are available, and many major software toolchains support the instruction set.

A 32-bit 5-stage pipelined processor based on the RISC-V ISA is conceived and built in this project. Instruction Fetch, Instruction Decode, Execute, Memory, and Writeback are some of the pipelined steps. After each stage, some registers will save the current cycle's output and act as input for the next step in the following cycle (pipeline implementation). Verilog HDL is used in the implementation and synthesized using Xilinx vivado.

Contents

List of figures

Introduction

RISC-V (short for Reduced Instruction Set Computer) is an open standard instruction set architecture (ISA) based on recognized reduced instruction set computer (RISC) concepts. Unlike most previous ISA designs, the RISC-V ISA is free to use and is distributed under open-source licencing. RISC-V hardware is available from numerous firms, open-source operating systems with RISC-V support are available, and several powerful software toolchains support the instruction set.

The RISC-V ISA is a load-store architecture in terms of RISC architecture. IEEE 1554 floating-point instructions are used in its floating-point instructions. Bit patterns to simplify multiplexers in a CPU, an architecturally neutral architecture, and most-significant bits of immediate values stored at a fixed location to accelerate sign extension are all notable characteristics of the RISC-V ISA. The instruction set can be used for a variety of purposes. The standard instruction set has a fixed length of 72-bit naturally aligned instructions, but the ISA enables variable-length extensions that allow each instruction to be any number of 72-bit parcels long. Small embedded systems, personal computers, supercomputers with vector processors, and warehouse-scale 19-inch rack-mounted parallel computers are all supported by subsets.

Designing a CPU necessitates knowledge of multiple disciplines, including electronic digital logic, compilers, and operating systems. Commercial computer design providers, such as Arm Ltd. and MIPS Technologies, collect royalties for the use of their designs, patents, and copyrights to support the costs of such a team. They also frequently demand non-disclosure agreements before producing materials that detail the evident benefits of their ideas. In many situations, they never explain why they chose certain design elements.

RISC-V

Figure 1. RISC V logo and association

Electronic digital logic, compilers, and operating systems all demand design skills in CPU design. Commercial computer design providers like Arm Ltd. and MIPS Technologies collect royalties for the use of their designs, patents, and copyrights in order to fund the costs of such a team. They frequently demand non-disclosure agreements before revealing materials that outline the evident advantages of their ideas. They seldom explain why they chose certain design elements.

The RISC-V ISA designers purposefully support a wide range of practical use cases: compact, performance, and low-power real-world implementations without over-architecting for a particular microarchitecture to build a large, ongoing community of users and thus accumulate designs and software. RISC-V was designed to handle a wide range of potential uses in order to meet the needs of a large number of contributors.

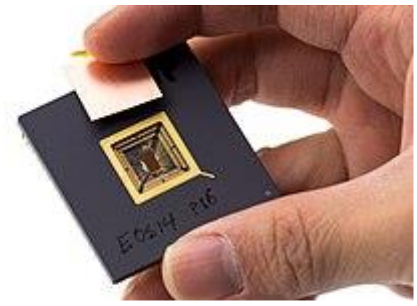


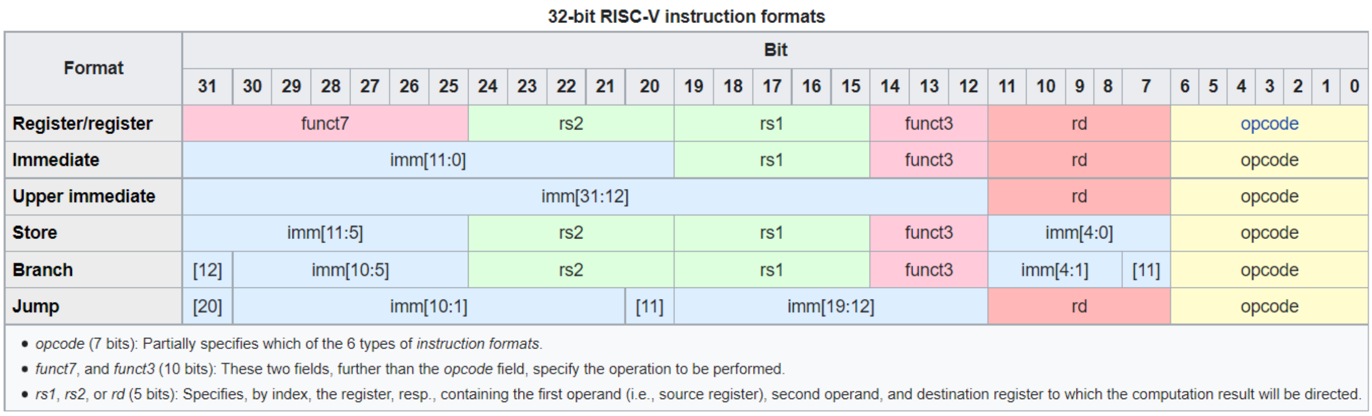
Figure 2. RISC V Prototype

The designers' central claim is that the instruction set is the critical interface since it is located at the hardware-software interface. By allowing significantly greater reuse, an appropriate instruction set that was open and available to anyone might drastically cut the cost of the software. It should also enhance competition among hardware manufacturers, focusing more resources on design and minor software maintenance.

According to the designers, new principles are becoming uncommon in instruction set design, as the most successful designs of the previous forty years have become more identical. The majority of those that failed did so because their sponsors were financially unsuccessful, not because the training sets were technically inadequate. As a result, many suppliers should be willing to embrace a well-designed open instruction set based on well-established concepts.

Academic use is also encouraged by RISC-V. The integer subset's simplicity allows for basic student activities and is a simple enough ISA for software to control research devices. The variable-length ISA allows for student exercises and study, while the distinct privileged instruction set allows for research into operating system support without rewriting compilers. The open intellectual property model of RISC-V enables the publication, reuse, and modification of derivative designs.

ISA



32-bit RISC-V instruction formats

Format	Bit																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- *opcode* (7 bits): Partially specifies which of the 6 types of *instruction formats*.
- *funct7*, and *funct3* (10 bits): These two fields, further than the *opcode* field, specify the operation to be performed.
- *rs1*, *rs2*, or *rd* (5 bits): Specifies, by index, the register, resp., containing the first operand (i.e., source register), second operand, and destination register to which the computation result will be directed.

Stages

INSTRUCTION FETCH STAGE

An Instruction Memory (for storing instructions) and an Address fetch unit make up the Instruction Fetch (IF) Stage. The IF stage is the pipeline's first step, and it serves three purposes:

1) Uses a separate adder to calculate the PC+4.

2) The PC register points to the Instruction memory, which retrieves the instruction and updates the IF/ID pipeline register on the next rising clock edge.

3) The PC value is updated by the fetch stage based on the PCSrc control signal:

PCSrc is 00 for normal programme flow, PC = PC + 4 for jump instruction; PCSrc is 01 for jump instruction, • PC = Jump Address in case of a jump.

PCSrc is 10 in the case of a conditional branch, and PC=PC+Offset in the case of a branch.

4) When a hazard occurs, the PC register is not changed; IF/ID is not updated if the hazard is caused by data/execution. (Stall=1)

• The IF/ID is flushed if a branch is actual or a jump command. (Flush=1)

The clock, reset, and next value of the Program Counter are all inputs to instruction fetch as a whole. It outputs the instruction code and the current value of the Program Counter.

Instruction Memory

The instructions in the instruction code are stored in separate memory locations by this unit (instruction address). It converts the input into an instruction address and outputs the relevant instruction. It also includes a reset input; when reset is triggered, the instructions are initialised in the Memory.

Address Fetch unit

This unit generates the instruction address, which is subsequently fed into an instruction memory, where the instruction code corresponding to the instruction address is retrieved.

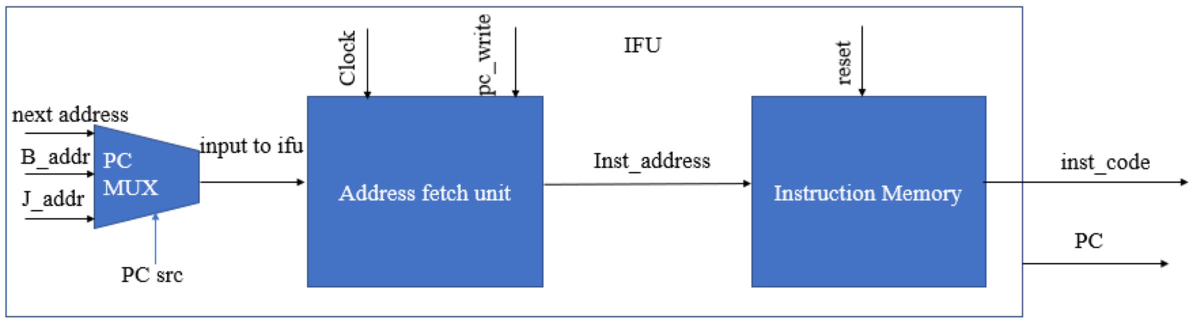


Figure 3.Instruction Fetch Unit

INSTRUCTION DECODE STAGE

The Decode stage is the second cycle of the pipeline, where the 32-bit Instruction code obtained from the fetch stage's instruction memory is decoded into various signals and flags, which are then passed to the execute stage for execution. An instruction decoder and a register file are combined in this step.

The register file also accepts clock and reset as inputs; if the reset is engaged, the register values are initialised to the predetermined values stored.

In this step, the register file writes to the write register, which receives the output from the execution unit, Data memory, or, in the event of branching instructions, the output from the addition of PC with the offset.

The contents of the registers are also read at this step and passed on to the next. In a nutshell, all fields of the instruction code are parsed, and based on the kind of instruction from the instruction fetch unit, certain operations are done.

In the instruction decode stage, it also computes the branch and jump addresses. By recognising the kind of instructions being decoded, the decoder can also provide suitable sign-extended values for use in subsequent stages.

PC src is the line that chooses between branch, jump, and following addresses (PC+4). Then, depending on PC src, a PC address choose mux is utilised to pick the needed address. Conditional leaps, or branches where PC jumps to PC+immediate Offset address, are possible. PC = Jump address for the Jump command. This computation is done before to the ID/EX step.

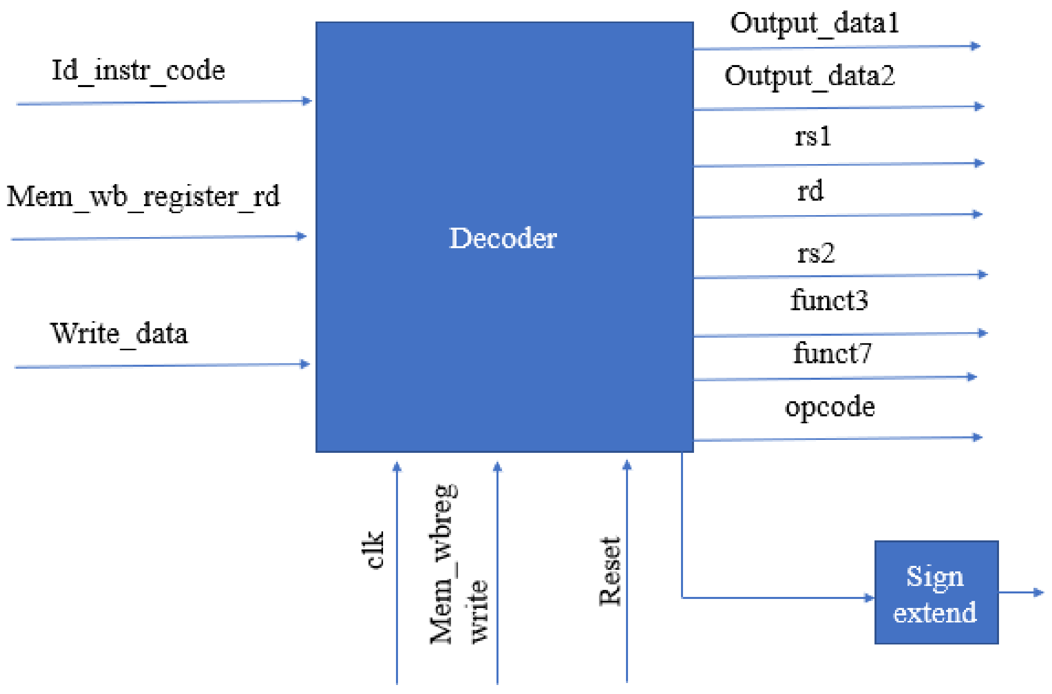


Figure 4. Decoder

EXECUTION STAGE

The execution stage gets the decode stage's output and executes the instruction's execution. The Arithmetic Logic Unit (ALU) and the Multipliers/Divide Unit (MUL/DIV) make up the execution stage. If if id enable = one, it conducts ALU operations; otherwise, it does multiply/divide operations based on the control signal from IF ID Stage. The decode stage provides two 32-bit data inputs to the execution unit. It executes operations in accordance with the kind of instruction. The execution's output is either directly written back to the registers or transmitted to the data memory (or the memory stage).

The operations that ALU can perform are:

1. Addition
2. Subtraction
3. Bitwise OR
4. Bitwise AND
5. Bitwise XOR
6. Set on less than
7. Set on greater than
8. Shift Left logical
9. Shift Right Logical

The operations that MUL/DIV can perform are:

1. Multiplication of two unsigned numbers.
2. Multiplication of two signed numbers.
3. Division of two unsigned numbers.
4. Division of two signed numbers.

In addition, a zero flag is set when the ALU output is zero. One of the signals that determines the PC source signal is the zero flag. As a result, ALU can assist in detecting when branch instructions will be executed. In addition, for jump instructions, ALU computes the effective address to which the leap is to be made.

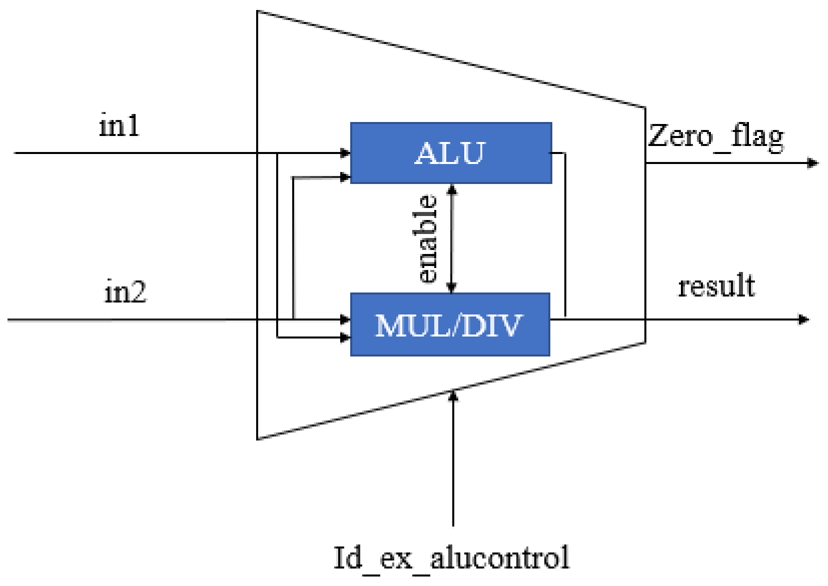


Figure 5. Execution Unit

MEMORY STAGE

The MEM stage is the pipeline's fourth stage. Its main job is to read and write data from and to memory. It is made up of a memory file that reads and writes data in response to memory read/write control signals (mem read/memWrite).

1. A Data memory having an Address In port, as well as Read Data and Write Data ports, is included in the memory unit. Data is read or written into memory based on the mem read/write signal.

2. Data is written into the memory for Store instructions, and data is retrieved from the memory to fill the registers for Load instructions.

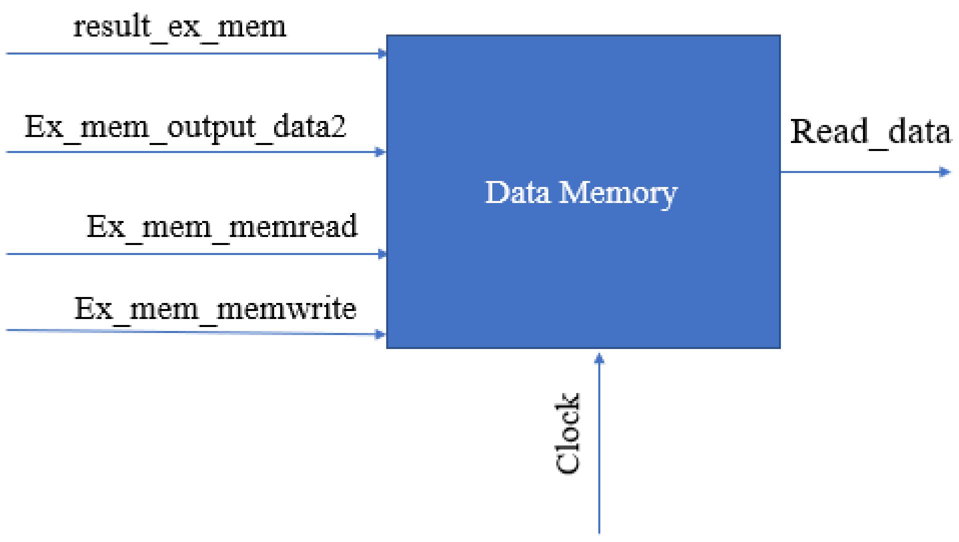


Figure 6. Data Memory

WRITE BACKSTAGE

Write backstage is the pipeline's fifth step. Its principal duty is to write the calculated value from the execution unit or data read from memory to the target register (Load). If the MemtoReg control signal is high, the data read from memory is written back to the Registers. The data from the execution unit output is written directly into the registers if the MemtoReg control signal is low.

Only a multiplexer in the write-back stage chooses the data to be written back, either from the execution unit is output or from Data Memory. The memtoreg control signal is used to make the decision.

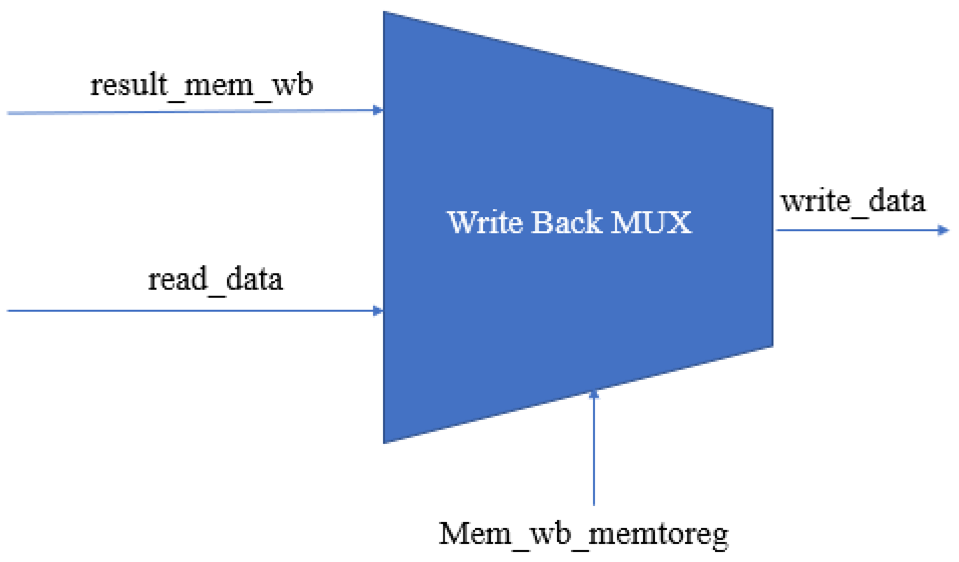


Figure 7.Write Back

CONTROL UNIT

The control unit generates a variety of control signals, ensuring that the needed operations are carried out correctly. The control unit is signals are effectively utilised as select lines for the multiplexers in the data path.

Funct3, Funct7, and Opcode, which are fields of the instruction code, are the inputs to the control unit. Control signals are created from these inputs, depending on the type of instruction being performed. The control unit for the RISC V core looks after the following control signals:

1. ALU Control: The ALU Control is responsible for selecting which operation is to be performed on the data entering the execution unit.
2. ALU source: The ALU source signal is a select line to a multiplexer responsible for choosing which Data (either data from RS2 or Immediate Data) is to be transmitted ahead in the execution stage.
    a) 0: data from RS2 is selected
    b) 1: immediate Data is selected
3. RegWrite: The RegWrite signal has to be high if Data is written back to a register.
    a) 0: Register File cannot be written
    b) 1: Register File can be written
4. MemWrite: The MemWrite signal, if high, allows data to be written into the Memory.
5. MemRead: The MemRead signal, if high, allows data to be read from Memory.
6. MemtoReg: The MemtoReg signal acts as a select line to mux to decide whether data to be written back to the register is from ALU result Or Memory.

7. Enable: To differentiate between I extension and M extension instructions, we use to enable. It is responsible for selecting the execution unit between ALU and MUL/DIV.
    a) 0: MUL/DIV unit executes performing M extension instructions
    b) 1: ALU unit executes performing I extension instructions

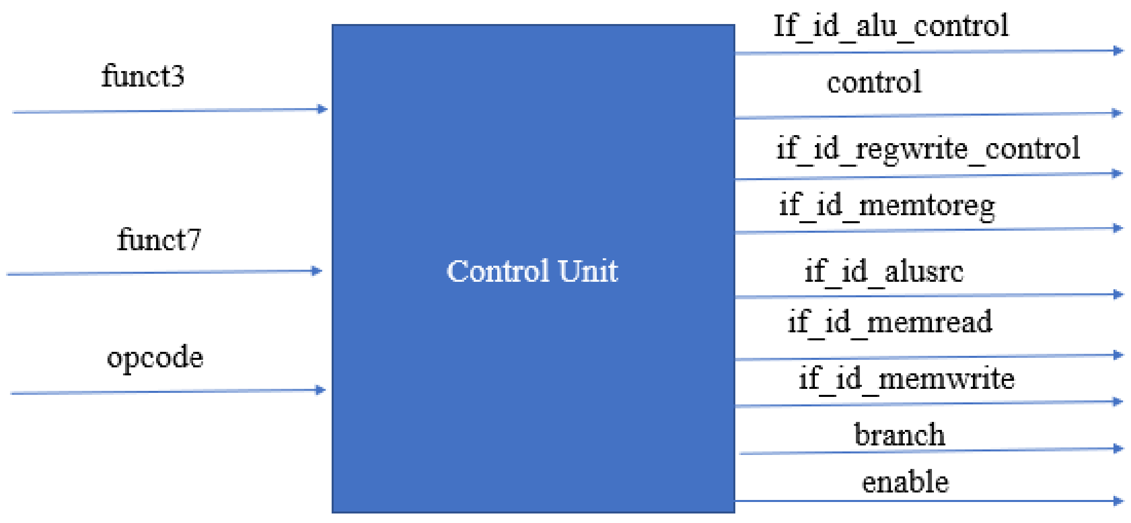


Figure 8. Control unit

PIPELINE REGISTERS

The pipelined technique divides instructions into many pieces that are performed one at a time. Between stages, we employ storage elements to keep track of the first instruction code while the second is retrieved. Pipeline registers are introduced, which aid in the execution of many instructions at different stages. This means that while a portion of one instruction is executed at one hardware stage, another portion of the same instruction is performed at a separate hardware level.

We created four pipeline registers since the core we implemented includes five stages: IF/ID, ID/EX, EX/MEM, and MEM/WB.

- IF/ID: Input from the Fetch and Decode Stages - clk, PC, Instruction code from the Fetch Stage, if id write from the stall unit, and IF Flush from the branch comparator.
- In the Decode stage, output includes IF ID PC, IF ID Instruction code, if id register rs1, and if id register rs2.
- ID/EX: clk, RS1, RS2, RD, Data1, Data2, sign extended data from the Decode stage and Control signals from the control unit.
- RS1,RS2,RD, Data1,Data2,sign extended data to the Execution stage and individual control signals are the output signals.
- Between the Execute and Memory stages is the EX/MEM stage.
- The following signals are used as input: clk, RD, Data2, ALU result, zero flag from the Execution stage, and remaining control signals.
- RD, Data2, ALU result, memory stage zero flag, and remaining control signals are all output.
- MEM/WB: clk, RD, ALU result, data read from the Memory stage, and residual control signals between the Memory and WriteBack stages.
- clk, RD, ALU result, data read to the Write Backstage, and remaining control signals are all output signals.

HAZARDS IN PIPELINE

Data hazards:

They happen when data-dependent instructions affect data at multiple stages of a pipeline. Read after write (RAW), a real dependence, is a condition where data hazard might arise.

Forwarding Unit:

When data-dependent instructions affect data at separate stages of a pipeline, they cause them to fail. A genuine dependence, read after write (RAW), is a circumstance where data hazard can arise.

Inputs:

- ID_EX_RS1
- ID_EX_RS2
- EX_MEM_RD
- MEM_WB_RD
- EX_MEM_Regwrite
- MEM_WB_Regwrite

Outputs:

- FWD_RS1 (select signal for input1)
- FWD_RS2 (select signal for input2)

Hazard Detection Condition:

- EX_MEM_RegWrite == 1 && EX_MEM_RD != 0 && EX_MEM_RD == ID_EX_RS1
- MEM_WB_RegWrite == 1 && MEM_WB_RD != 0 && MEM_WB_RD == ID_EX_RS1
- EX_MEM_RegWrite == 1 && EX_MEM_RD != 0 && EX_MEM_RD == ID_EX_RS2
- MEM_WB_RegWrite == 1 && MEM_WB_RD != 0 && MEM_WB_RD == ID_EX_RS2

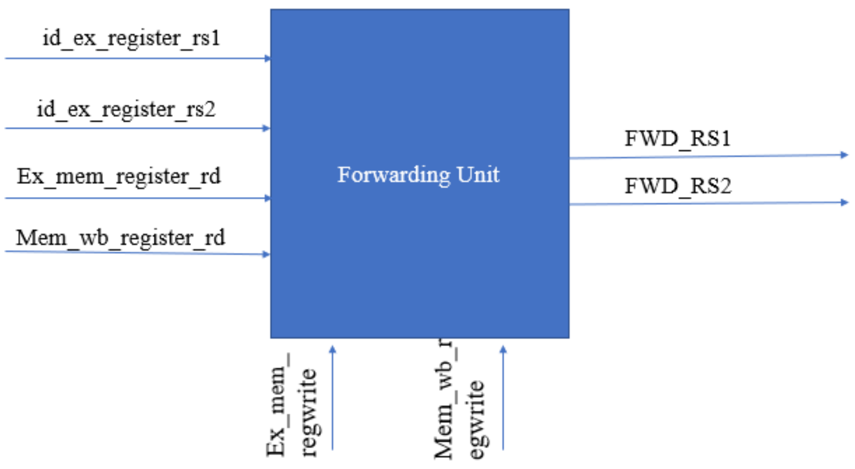


Figure 9. Forwarding Unit

Stalling Unit:

Forwarding the unit will not solve data dangers in the case of load and store instructions. As a result, we require a stalling unit.

During the ID stage, a stall is placed between the load and the following instruction that is dependent on it. The danger detecting unit is in the following situation after checking for load instructions:

If the pipeline is stalled by (ID/EX.MemRead and ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs2))

After that, a one-cycle stall should be followed by forwarding logic that can manage data dependencies, and execution should continue.

We delay the instructions in the ID and IF stages by not allowing the PC register and the IF/ID pipeline register to change; it executes nops commands.

Stalling the pipeline after detecting the danger in the ID stage is to put a bubble into the pipeline, so we halt the clock to PC and IF/ID register, setting all of the ID/EX register's control signals to 0.

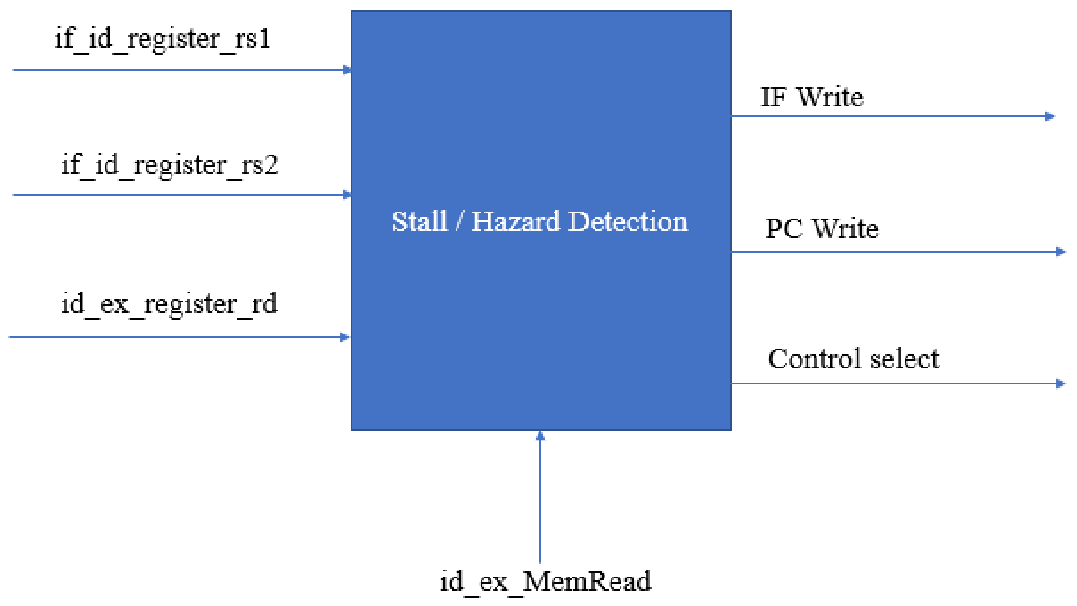


Figure 10. Stalling Unit

Control hazards:

When the processor retrieves a new instruction into the pipeline in the instruction Fetch stage before confirming the branch condition in the ID stage, branch and jump instructions occur. If the jump/branch is correct in this scenario, we must flush the previously obtained instruction from the pipeline and jump to the new one. Because the branch decision is made in the EX stage, the FlushIF, FlushID, and FlushEX functions flood the IF, ID, and EX stages that follow the branch instructions. NOP is introduced by turning off the pipeline's control signals.

Optimization to reduce the delay of branches:

We may enhance conditional branch speed by relocating branch execution earlier in the pipeline, which entails moving up the block that computes the target address and evaluates the branch choice. This involves flushing fewer instructions.

Computing the branch address and assessing the branch choice must happen earlier in the pipeline in order to move the branch execution up the pipeline. We require PC and immediate values in the IF/ID pipeline register to compute branch address at the ID stage rather than the EX stage. We would compare two register readings soon after the decoder stage and before loading into the ID/EX stage for branch if equal and branch if not equal.

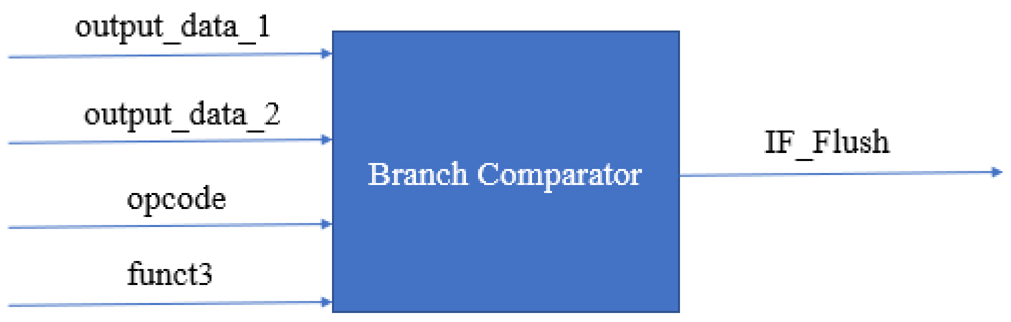


Figure 11. Branch Comparator

Structural Hazards:

It happens when many instructions in the pipeline use the same resource at the same time. The MEM stage, for example, writes data to a Memory address, while the IF stage attempts to acquire fresh instructions from the same place. Separate instruction and data memory are meant to avoid such dangers.

Single cycle 5 stage processor

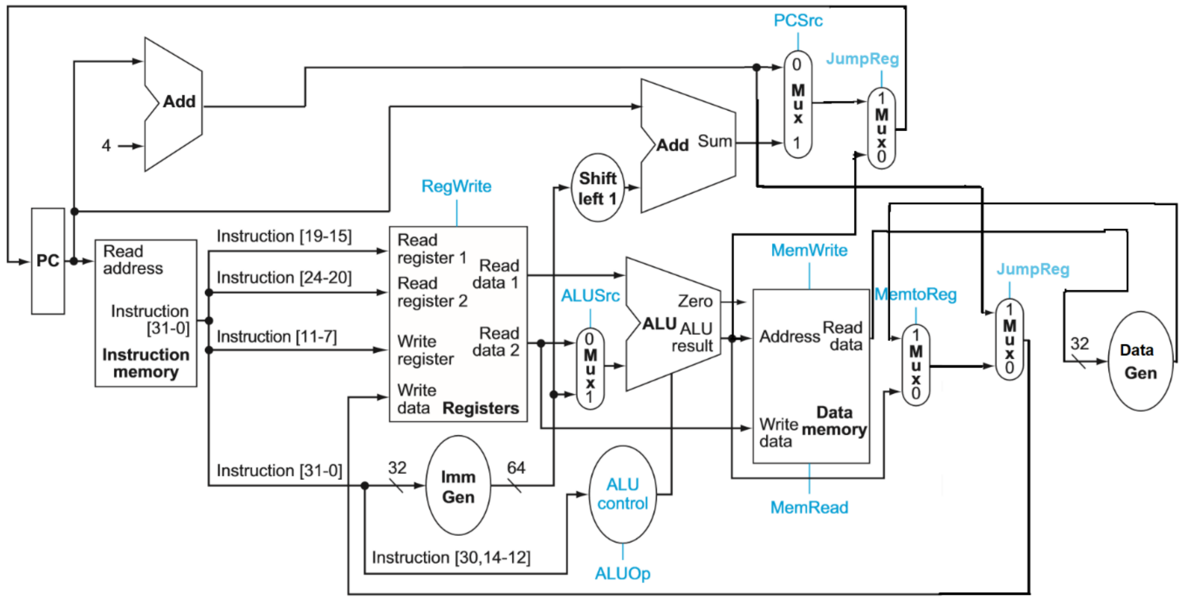


Figure 12.Single cycle processor

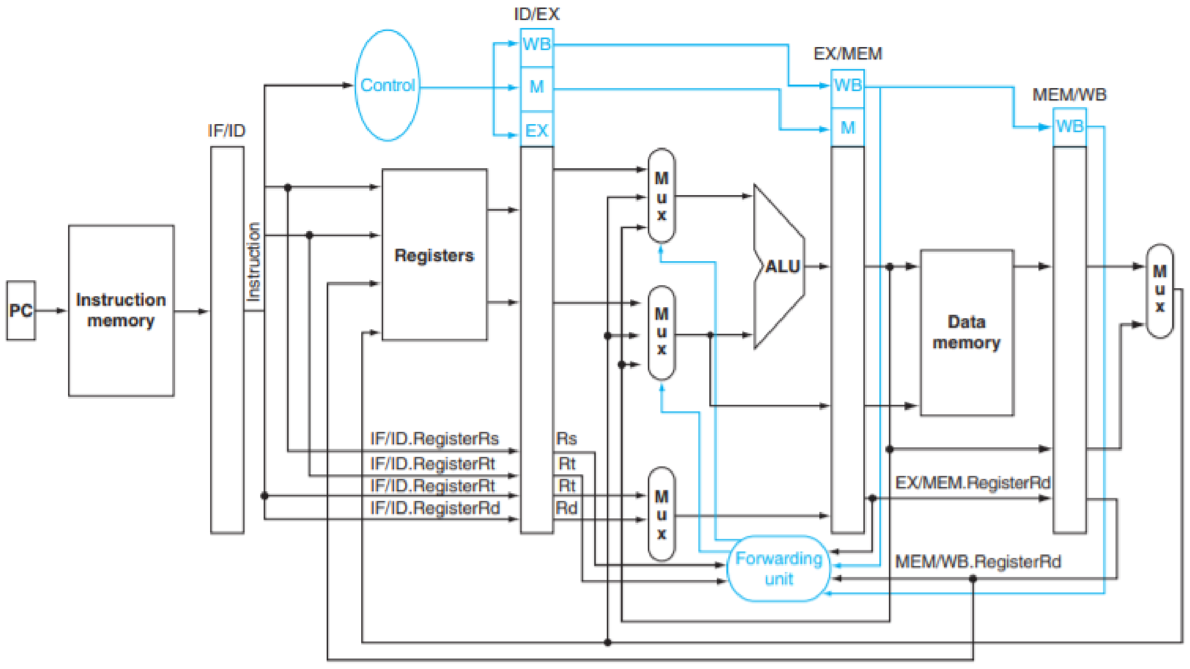


Figure 13. Single cycle processor with stalling and forwarding unit

Results

The RISC V design and implemented. The induvial module results are shown.

ALU

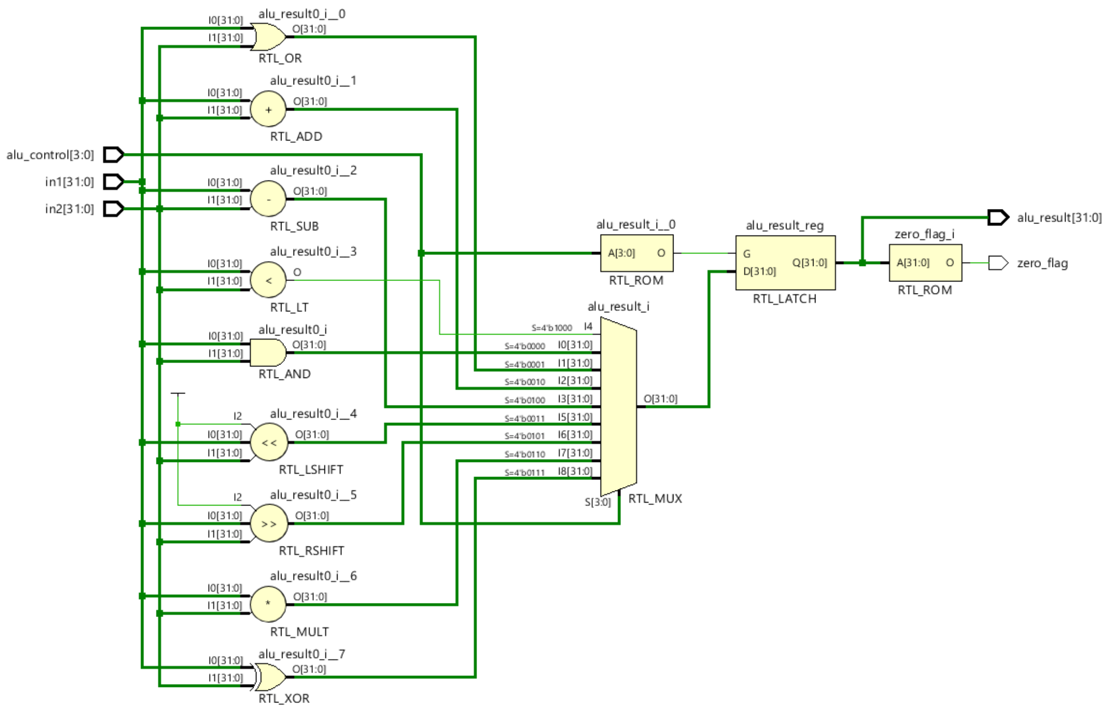


Figure 14. RTL of ALU

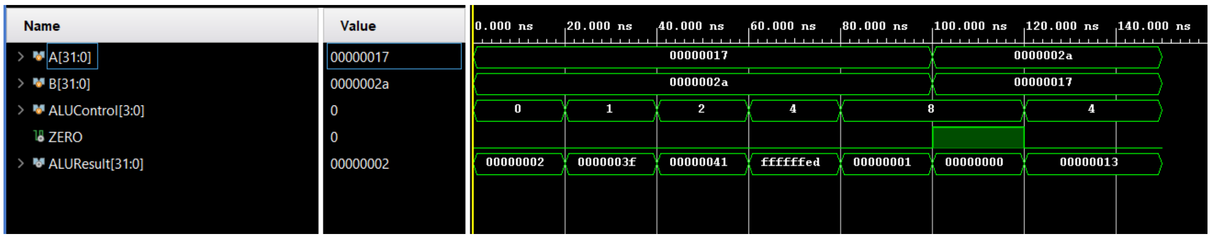


Figure 15. ALU Verification

Memory unit

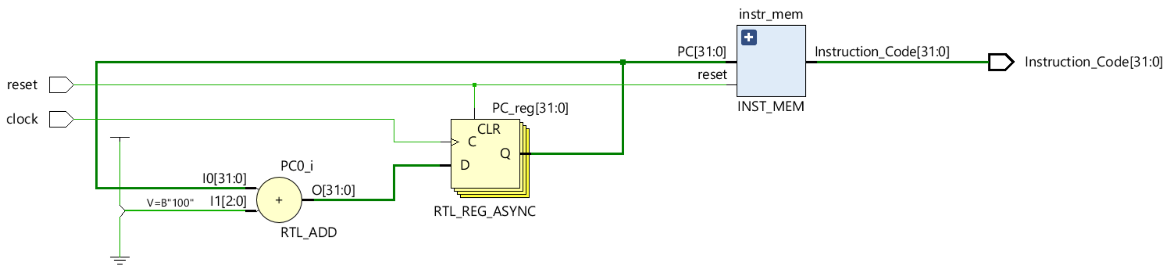
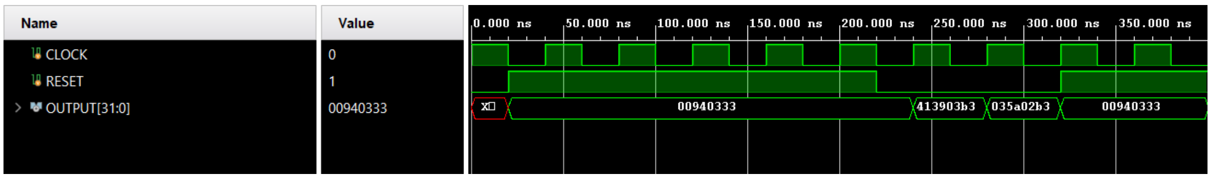
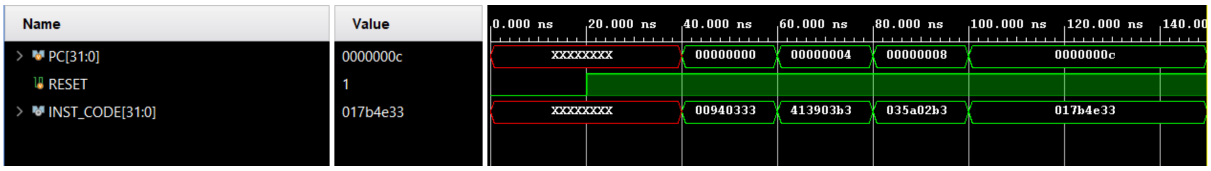


Figure 16. RTL of Memory unit



Name	Value
CLOCK	0
RESET	1
OUTPUT[31:0]	00940333

Figure 17. Memory unit verification



Name	Value
PC[31:0]	0000000c
RESET	1
INST_CODE[31:0]	017b4e33

Figure 18. Instruction unit verification

Register unit

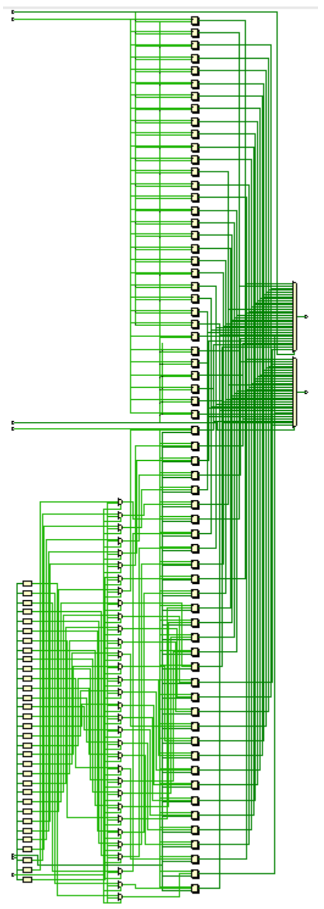
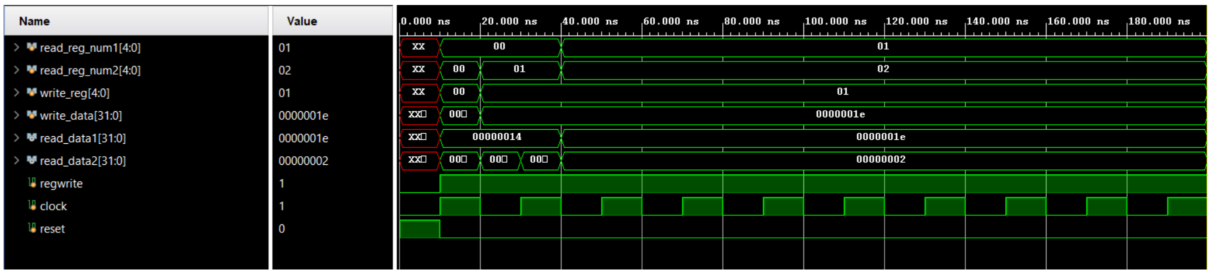


Figure 19. RTL of register unit



Name	Value
read_reg_num1[4:0]	01
read_reg_num2[4:0]	02
write_reg[4:0]	01
write_data[31:0]	0000001e
read_data1[31:0]	0000001e
read_data2[31:0]	00000002
regwrite	1
clock	1
reset	0

Figure 20.Register unit verification

Control unit

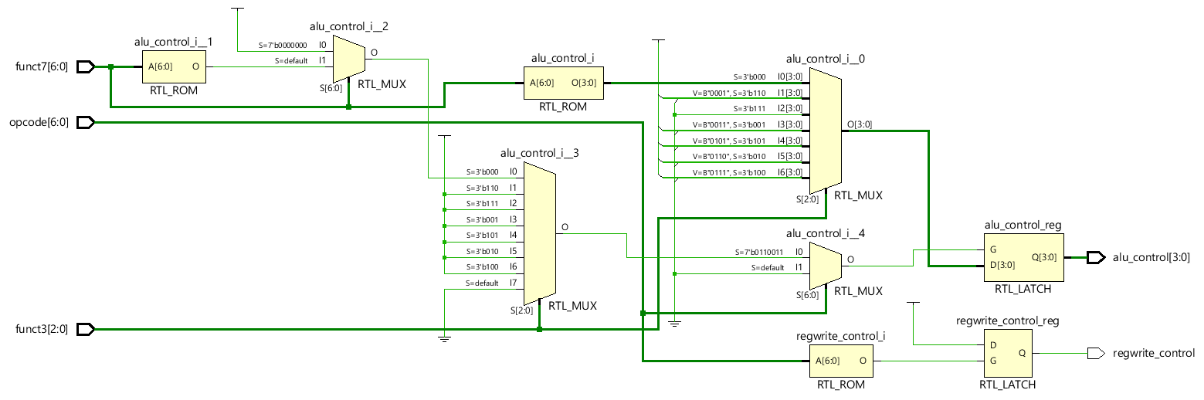


Figure 21.RTL of control unit

Data path

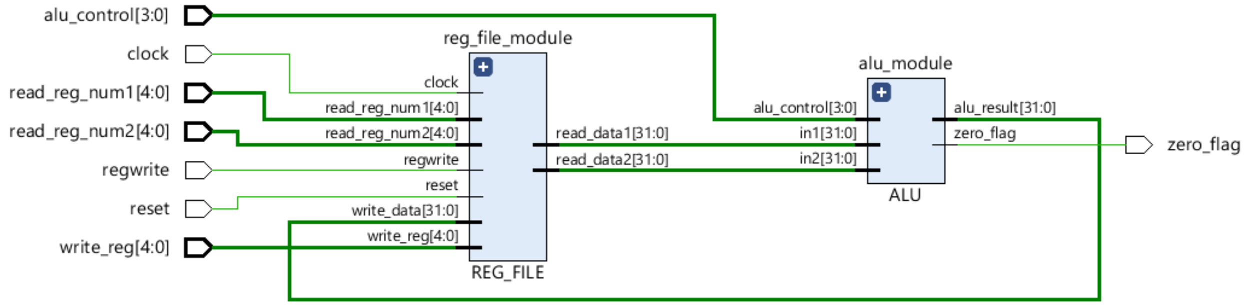
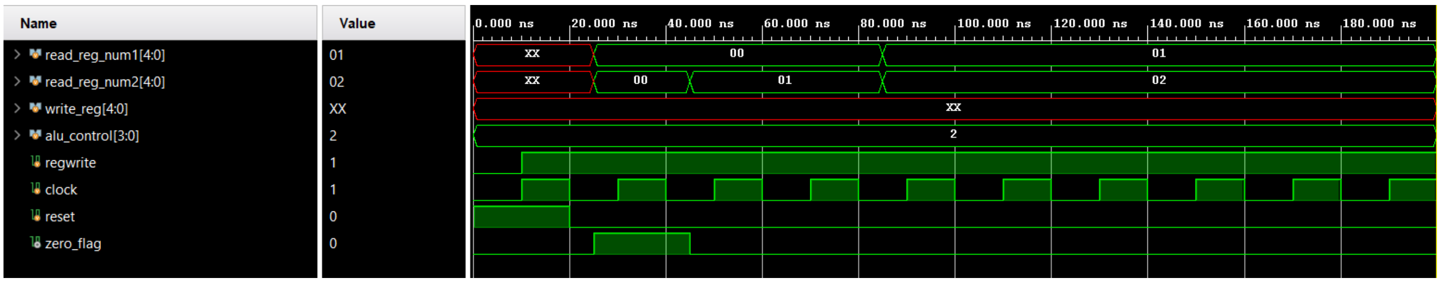


Figure 22.RTL of data path



Name	Value
read_reg_num1[4:0]	01
read_reg_num2[4:0]	02
write_reg[4:0]	XX
alu_control[3:0]	2
regwrite	1
clock	1
reset	0
zero_flag	0

Figure 23.Data path verification

Single stage processor

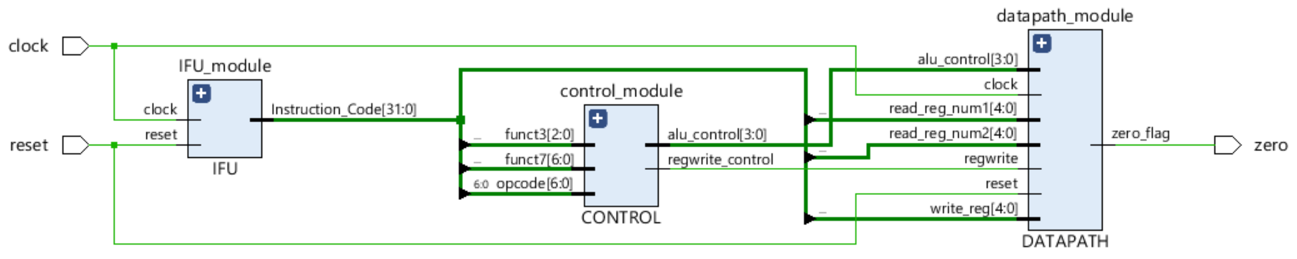


Figure 24.RTL of single stage processor

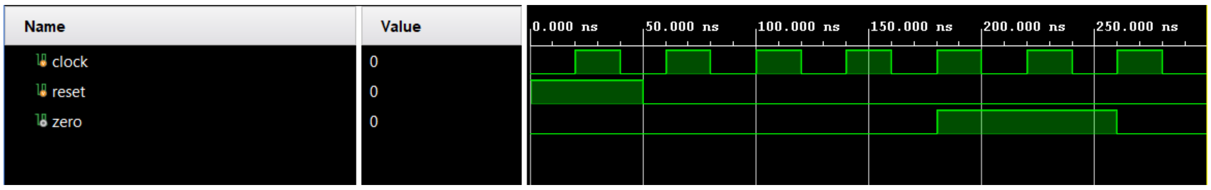


Figure 25.Processor verification

Forwarding unit

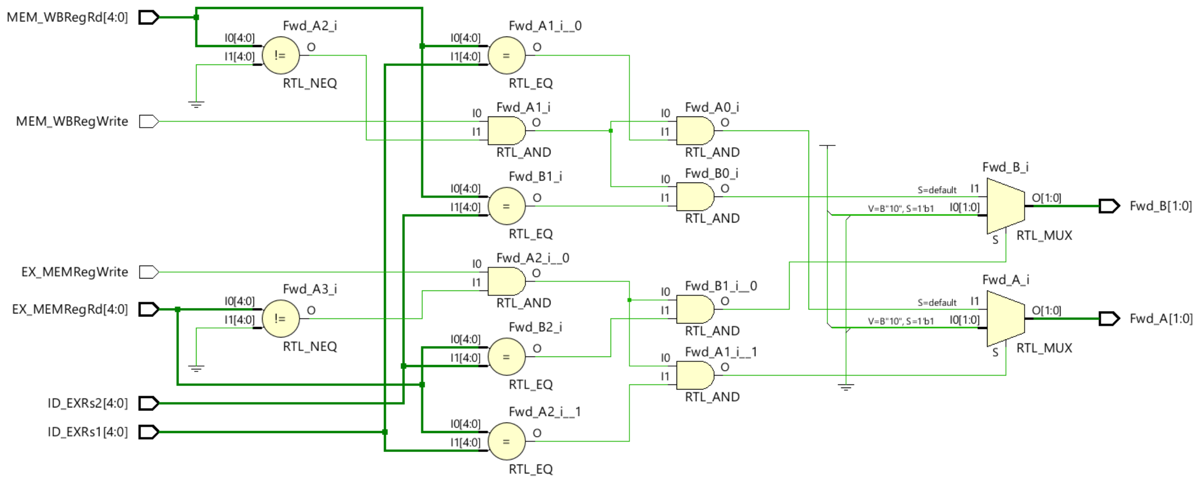


Figure 26.RTL of forwarding unit

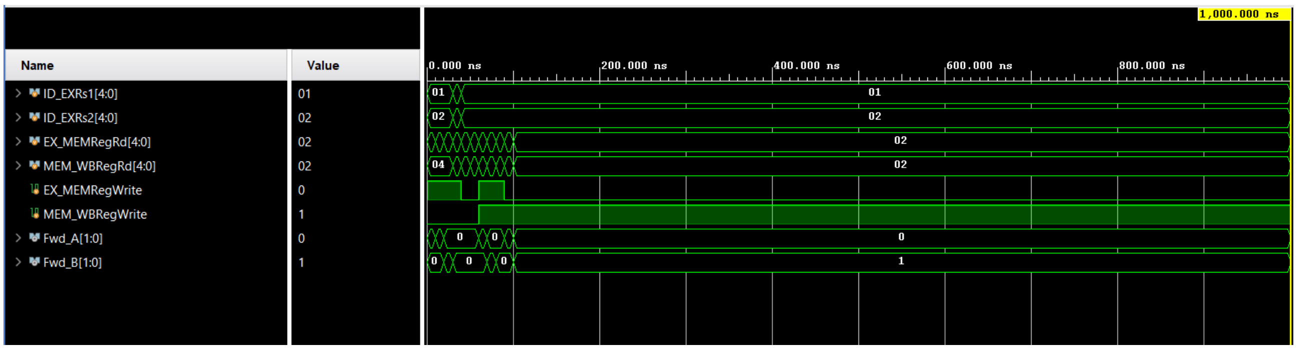


Figure 27. Forwarding unit verification

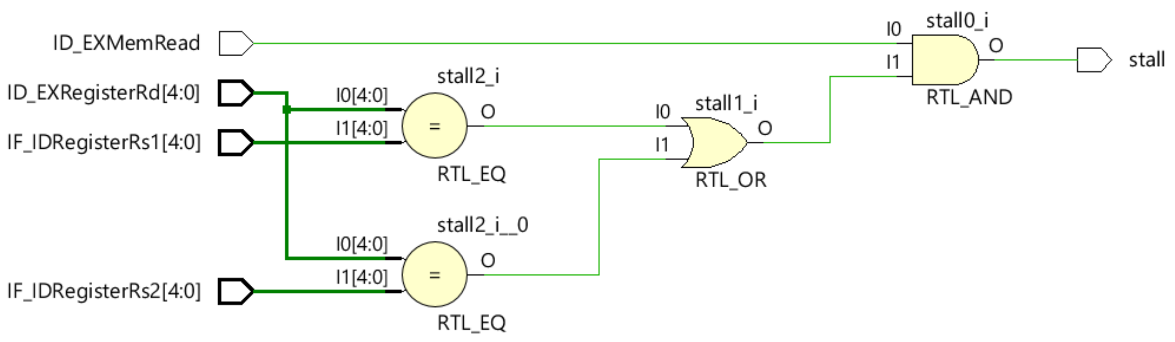


Figure 28. RTL of hazard unit

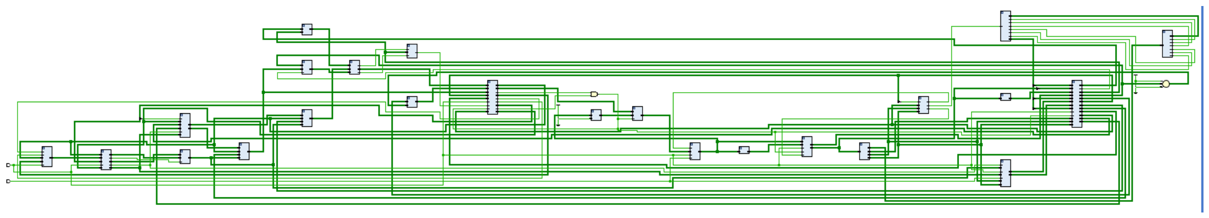


Figure 29. RTL of processor

INSTRUCTIONS TO BE IMPLEMENTED

R-Type Instructions ADD SRL SLL SLT XOR SRL OR AND SUB

S-Type Instructions SW

I-Type Instructions ADDI SLTI ORI ANDI XORI LW

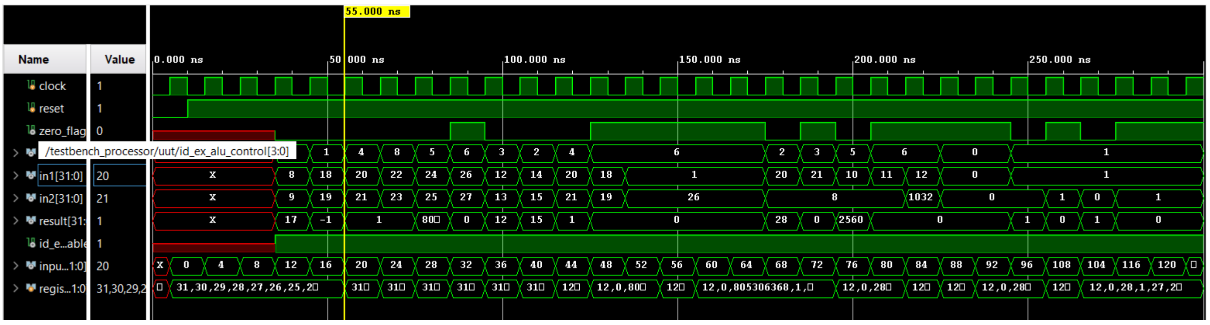
Branch Instructions BNEQZ BEQZ

Following are the output waveforms for the registers corresponding to the instructions:

Add, sub,slt, xor,sll, srl

Mul, mulh,mulhsu,mulhu, div, divu, rem, remu

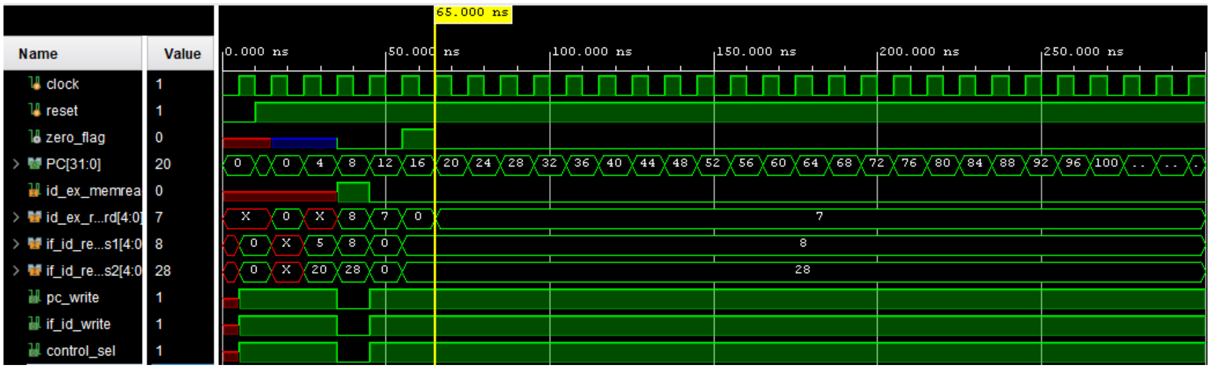
Ori, andi,slli,srli,addi



Considering the situation where data hazard is present

lw s0, 20(t0)

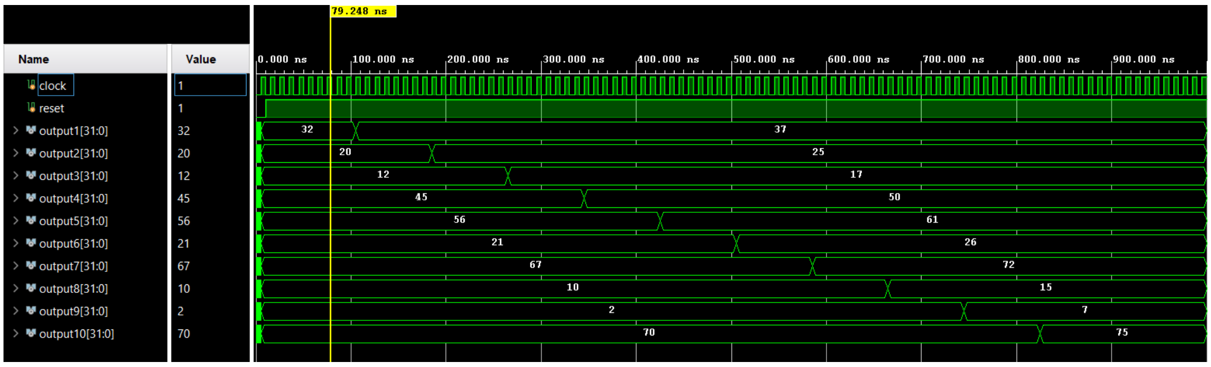
sub t2, s0, t3



Application

Storing N words in data memory and add a constant value 5 to all the N words.

Let N value be 10 stored in s0 register


79.248 ns
Name
Value
clock 1
reset 1
output1[31:0] 32
output2[31:0] 20
output3[31:0] 12
output4[31:0] 45
output5[31:0] 56
output6[31:0] 21
output7[31:0] 67
output8[31:0] 10
output9[31:0] 2
output10[31:0] 70
0.000 ns
100.000 ns
200.000 ns
300.000 ns
400.000 ns
500.000 ns
600.000 ns
700.000 ns
800.000 ns
900.000 ns
32
37
20
25
12
17
45
50
56
61
21
26
67
72
10
15
2
7
70
75

Conclusion

The RISC-V architecture is free source and uses a little-endian memory scheme. The RV32IM is supported by a 5-stage 32-bit pipelined RISC-V core that contains ALU handling R-type instructions, Memory instructions (load and store), control transfer instructions (both conditional and unconditional branch instructions), and MUL/DIV handling M extension instructions. It also has a data hazard handling unit and a stalling unit, as well as static branch prediction and flushing to avoid control hazards.

References

1. D. Patterson and J. Hennessy, “Computer Organization and Design: The Hardware/Software Interface—RISC-V Edition”, Amsterdam The Netherlands: Elsevier, 2018.

2. A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic. (Dec. 2019). “The RISC-V Instruction Set Manual”, Volume I: Unprivileged ISA, Document Version 20191213.

3. N. M. Qui et al. “Design and Implementation of a 256-Bit RISC-V-Based Dynamically Scheduled Very Long Instruction Word on FPGA” IEEE Journal, Vol. 8, October 2020

4. RV32I ISA Reference: https://riscv.org/technical/specifications/