



EE-309

IITB-RISC-23

Presented by

TEAM ID:13

Devtanu Barman(22B3904)

Sanjay Kumar Meena(22B3978)

Archisman Bhattacharjee(22B2405)

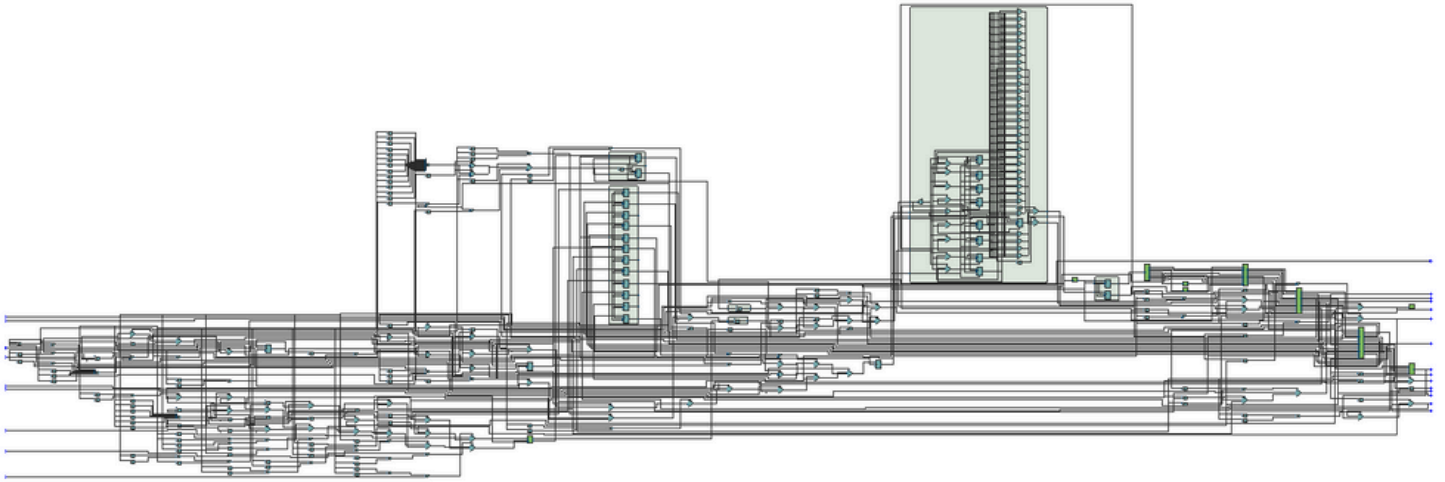
Shreyash Wanjari(22B3926)



Description:-

Based on the Little Computer Architecture, IITB-RISC-23 is a 16-bit extremely basic computer designed for education. An 8-register, 16-bit computer system is the IITB-RISC-23. R0 through R7 are its eight general purpose registers. Program Counter is always stored in Register R0. Every address consists of instructions and bytes. It always retrieves two bytes for the data and instructions. The two flags in the condition code register used by this architecture are the Carry flag (C) and the Zero flag (Z). Despite its extreme simplicity, the IITB-RISC-23 has sufficient generality to tackle challenging issues. Multiple load and store operations as well as predicted instruction execution are supported by the architecture. There are a total of 14 instructions in the three machine-code instruction forms (R, I, and J type).

Datapath



Netlist Viewer

Components:

1. **IR Memory** : Contains all the instruction

- PC In : Takes input the current PC
- IR Out : Gives the corresponding instruction from memory.

2. **IR Decoder** : Decodes the instruction

- IR In : Takes the corresponding instruction as input
- Its Output are :
 - Opcode
 - Ra (First operand)
 - Rb (Second operand)
 - Rc (Third operand)
 - Imm6 (6 bit immediate data)
 - Imm9 (9 bit immediate data)

3. **Register File** : Contain 8 registers

- Its inputs are:
 - RF_A1 : To select register to read
 - RF_A2 : To select register to read
 - RF_A3 : To select register to write
 - RF_D3 : Data to write
 - PC in
 - Clock
 - Reg_en : Write enable signal
 - Reset : Reset signal
- Its Output are :
 - RF_D1 : Data of Register selected
 - RF_D2 : Data of Register selected
 - PC out

4. **Arithmetic Logic Unit (ALU)** : Performs ADD and NAND

- Input :
 - Ir_in : selects operation
 - Alu_in_a : First input
 - Alu_in_b : Second input
- Output :
 - Alu_out_c : Output

5. **Sign-extender 7 and sign-extender 10**

- Converts the 7 bit and 10 bit inputs to 16 bit output

6. **Adder** : Takes two input and add them

7. **PC Adder** : updates PC after every cycle (PC+1)

8. **Comparator** : Compares 2 input

- Input:
 - Input 1
 - Input 2
- Output:
 - El signal : is set if both inputs are equal
 - Lte signal : is set if input 1 is less than input 2

9. **Complement** : Outputs the complement of a 16 bit input

10. **Data memory** : Contains data in which all the operations are performed

- Read:
 - Memory-read-address : input address to the memory to read the data
 - Memory-data-output : data corresponding to the input address
- Write:
 - Memory-write-address : input address to the memory to write the data
 - Memory-write-data :input data to write at the input address

Pipeline Register Content

Pipeline register	content	No. of bits
P1 (Instruction fetch to Instruction decode)	IR	16
	PC	16
P2 (Instruction Decode to execution)	opcode	4
	op_ra	3
	op_rb	3
	op_rc	3
	cmp	1
	op_cz	2
	se7	16
	se10	16
	PC	16
	opcode	4
	op_ra	3
P3 (Register read and write to Execution)	op_rb	3
	op_rc	3
	cmp	1
	op_cz	2
	se7	16
	se10	16
	PC	16
	rf_d1	16
	rf_d2	16

P4 (Execution to Memory)	opcode	4
	alu_c	16
	op_rc	3
	cmp	1
	op_cz	2
	carry flag	1
	zero flag	1
	se7	16
	se10	16
	rf_d2	16
	op_ra	3
	op_rb	3
	<_ flag	1
	=_flag	1
P5 (Memory to write back)	opcode	4
	alu_c	16
	op_rc	3
	cmp	1
	op_cz	2
	carry flag	1
	zero flag	1
	se7	16
	se10	16
	<_ flag	1
	=_flag	1
	mem_data	16

Stages of Pipeline :

Instruction Fetch (IF): In this stage the instruction is fetched by the processor and the program counter(PC) is incremented ($PC++$).

Instruction Decode (ID): In this stage the fetched instruction is decoded into different signals such as opcode, operand ra, operand rb, operand rc, imm6, imm9.

Register Read (RR): In this stage the decoded instruction is used to read the necessary registers from the register file (RF).

Instruction Execute (IE): This stage mainly deals with execution of the instruction with the help of ALU, adder and also modifying the value of carry and zero flag wherever required.

Memory Access (MA): In this stage memory-read and memory-write operations are performed.

Write Back (WB): In this stage all the final results obtained after 5th stage are written back to the register.

Hazards:-

- **IMMEDIATE LENGTH HAZARD**

Immediate Dependency Detection:

1. When either of the addresses of the inputs to the ALU matches with the destination address of the last immediate instruction:

- Example:

I1: ADA R1 R2 R3

I2: NDU R3 R4 R5

- *Solution* (Forwarding):

Connect the ALU input to the data calculated in the previous instruction, which is now stored in P

- Current arrangement: I1 is in P4 and I2 is in P3 now.

a. If (P3_op_ra == P4_op_rc), then alu_a_input = P4_alu_c_out.

Else, alu_a_input = P3_rf_d1.

b. If (P3_op_rb == P4_op_rc), then alu_b_input = P4_alu_c_out.

Else, alu_b_input = P3_rf_d2.

2. When the memory write address matches with the immediate before instruction's destination address:

- Example:

I1: ADA R1 R2 R3

I2: SW R3 R4 imm6

- *Solution* (Forwarding):

Connect the ALU input to the data calculated in the previous instruction, which is now stored in P4.

- Current arrangement: I1 is in P5 and I2 is in P4 now.

- If (P4_alu_c == P5_op_rc), then memory_wr_data = P5_reg_d3.

Else, memory_wr_data = P4_rf_d1.

Two-Length Hazard Detection:

1. *When either of the addresses of the inputs to the ALU matches with the address of the destination of the second last instruction:*

- Example:

I1: ADA R1 R2 R3

I2: NDU R6 R7 R1

I3: ADA R3 R4 R5

- *Solution* (Forwarding):

Connect the ALU input to the data calculated in the last-second instruction, which is now stored in P5.

- Current arrangement: I3 is in P3 and I1 is in P5 now.

a. If (P3_op_ra == P5_op_rc), then alu_a_input = P5_reg_d3.

Else, alu_a_input = P3_rf_d1.

b. If (P3_op_rb == P5_op_rc), then alu_b_input = P5_reg_d3.

Else, alu_b_input = P3_rf_d2.

Three-Length Hazard Detection:

1. *When either of the addresses to select the register matches with the address of the destination of the third last instruction:*

- Example:

I1: ADA R1 R2 R3

I2: NDU R6 R7 R1

I3: LW R7 R4 imm6

I4: ADA R3 R1 R5

- *Solution* (Forwarding):

Connect the register write data to the data calculated in the third last instruction, which is now stored in P5.

- Current arrangement: I4 is in P2 and I1 is in P5 now.

a. If (P2_op_ra == P5_op_rc), then P3_rf_d1 = P5_reg_d3. Else, P3_rf_d1 = Rf_d1.

b. If (P3_op_rb == P5_op_rc), then P3_rf_d2 = P5_reg_d3. Else, P3_rf_d2 = Rf_d2.

• **CONTROL HAZARDS:**

Cause: Control hazards are caused by branch or jump instructions.

- ***Example:***

I: BEQ R1 R2 imm6

Detection: Branch or jump instructions are detected in the IR decode stage. For conditional branches, flags are obtained in the execution stage, and decisions are made based on that.

Problem: In the execution stage, we determine whether we need to jump conditionally or not. For unconditional jumps, this decision comes in the IR decode stage. However, by this time, two extra instructions have already been loaded into stage 1 and stage 2.

Solution:

To resolve this issue, we need to disable all write signals for two instructions and update the PC accordingly.

Hardware Solution:


1. Introduce a MUX before the PC to select either the calculated PC or update it normally.
2. Disable register enable, PC enable, ALU enable, etc., for two instructions.

Testing :

Complete Output file:-

<https://drive.google.com/file/d/1p5nt546loqX0YcSMHsE29v5DimPHoy98/view?usp=sharing>

Input:

```
boot >  input.txt
1    LLI R1 000000011
2    LLI R2 000000011
3    LLI R3 000000100
4    LLI R4 000001000
5    LLI R5 000000010
6    LLI R6 000001011
7    LLI R7 000000011
8    ADA R1 R4 R4
9    ADI R2 R2 000111
10   ADA R3 R3 R2
11   NDU R4 R1 R1
12   ADA R2 R1 R3
13   ADC R2 R3 R1
```

output:

Cycle : 19

Register PC: 00000000000001110 | 14 | 00000000

Register 1 : 00000000000001100 | 12

Register 2 : 00000000000001000 | 8

Register 3 : 00000000000000100 | 4

Register 4 : 00000000000001011 | 11

Register 5 : 00000000000000010 | 2

Register 6 : 00000000000001011 | 11

Register 7 : 00000000000000011 | 3

Procedure and Setup:

We have created a boot loader and compiler to test the instruction, also displayed all register content (in file registers.txt) at each cycle along with the PC values and flags (in file output.txt).

1. Extract all the files from
“IITB_RISC_23_Pipelined_Processor.zip”
2. Compile: By running the ‘compile.py’ file present in the boot folder.
3. Run the instructions: We write a program in the ‘input.txt’ file (path: boot/) and run that program on the IITB_RISC_23 CPU by running the ‘boot.py’ file (path: boot/).
4. All the results can be seen in the ‘register_values.txt’ (path: simulation/) and all the 3 memory contents can be viewed through the Model-Sim memory window.

Note: compile.py file compiles the whole VHDL code and the ‘boot.py’ each instruction in the ‘input.txt’ and loads it into the ir_memory

Contribution:-

Shreyash Wanjari :

Design datapath, Register File, Hazards, Pipeline Register 1

Archisman Bhattacharjee:

Design Pipeline Registers 3 & 5, Written the code for implementation of instructions in IITB_RISC_23

Sanjay Kumar Meena:

Design Comparator, ALU, Adder, sign extender, decoder, Report, Testing, Pipeline Register 2

Devtanu Barman:

Design Data Memory, Report, Complementer, Pipeline Register 4

*Thank
you!*