


EE-309

# IITB-RISC-23

6-stage Pipelined Processor



Group no. 2

Pratik Yabaji (210070094)

Sumit Londhe (210070089)

Arya Agarwal (210070012)

Alok Kumar (210070006)



## Description

IITB-RISC is a 16-bit very simple computer developed for teaching that is based on the Little Computer Architecture. The IITB-RISC-23 is an 8-register, 16-bit computer system. It has 8 general purpose registers (R0 to R7). Register R0 always stores Program Counter. All addresses are byte addresses and instructions. Always it fetches two bytes for instruction and data. This architecture uses a condition code register which has two flags: Carry flag ( C ) and Zero flag (Z). The IITB-RISC-23 is very simple, but it is general enough to solve complex problems. The architecture allows predicated instruction execution and multiple load and store execution. There are three machine-code instruction formats (R, I, and J type) and a total of 14 instructions.

## Components :

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

- 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
- Sign-extender 7 and sign-extender 10
  - Converts the 7 bit and 10 bit inputs to 16 bit output
- Adder : Takes two input and add them
- PC Adder : updates PC after every cycle (PC+1)
- 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
- Complement : Outputs the complement of a 16 bit input
- 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	16
	op_ra	3
	op_rb	3
	op_rc	3
	complement_enable	1
	op_cz	2
	imm6	6
	imm9	9
	PC	16
P3 (Register read and write to Execution)	opcode	16
	op_ra	3
	op_rb	3
	op_rc	3
	rf_d1	16
	rf_d2	16
	complement_enable	1
	op_cz	2
	imm6	6

	imm9	9
	PC	16
P4 (Execution to Memory)	opcode	16
	op_ra	3
	op_rb	3
	op_rc	3
	alu_c	16
	Carry flag	1
	Zero flag	1
	op_cz	2
	imm6	6
	PC	16
P5 (Memory to write back)	opcode	16
	op_rc	3
	reg_d3	16
	PC	16
	Carry flag	1
	Zero flag	1
	op_cz	2



## 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



## 1. DATA HAZARDS :

### 1. Immediate length hazards:

Detection: immediate dependency will occur :

- When either of the address of input to alu matches with the destination address of last immediate instruction:

Ex. I1: ADA R1 R2 **R3**

I2: NDU **R3** R4 R5

Solution : forwarding: connect the alu input to the data calculated in previous instruction which is now stored in P4.

I1 is in P4 and I2 is in P3 now:

- If (P3\_op\_ra == P4\_op\_rc) then  
alu\_a\_input = P4\_alu\_c\_out  
Else  
alu\_a\_input = P3\_rf\_d1
- If (P3\_op\_rb == P4\_op\_rc) then  
alu\_b\_input = P4\_alu\_c\_out  
Else  
alu\_b\_input = P3\_rf\_d2

- When memory write address matches with immediate before instruction's destination address

Ex. I1: ADA R1 R2 **R3**

I2: SW **R3** R4 imm6

Solution : forwarding: connect the alu input to the data calculated in previous instruction which is now stored in P4.

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

## 2. 2 length hazards:

Detection: 2 length dependency will occur :

- When either of the address of input to alu matches with the address to destination of second last instruction:

Ex. 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 last second instruction which is now stored in P5.

I3 is in P3 and I1 is in P5 now:

- If (P3\_op\_ra == P5\_op\_rc) then  
alu\_a\_input = P5\_reg\_d3  
Else  
alu\_a\_input = P3\_rf\_d1
- If (P3\_op\_rb == P5\_op\_rc) then  
alu\_b\_input = P5\_reg\_d3  
Else  
alu\_b\_input = P3\_rf\_d2

### 3. 3 length hazards:

Detection: 3 length dependency will occur :

- When either of the address to select register matches with the address to destination of third last instruction:

Ex. 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 third last instruction which is now stored in P5.

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

## 2. CONTROL HAZARDS :

This is caused due to branch or jump instructions

Ex I:BEQ R1 R2 imm6

Detection : branch or jump instructions will be detected in IR decode stage , and for conditional branch we will get the flags in execution stage , after that we will take our decision based on that.

Problem : We will get to know weather we want to jump conditionally or not in execution stage and in IR decode stage for unconditional jump , till that 2 extra instructions have already been loaded in stage 1 and stage 2.

Solution: We have to disable all write signals for two instructions and update the PC accordingly.

Hardware solution :


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

## 3.LM-SM implementation :

Unlike other instructions LM and SM instructions cannot be completed in six stages with normal methods used for other instruction, because by nature the LM/SM instructions are like 8 separate LW/SW instructions.

Possible solutions :

1. Attach 8 ports to memory to simultaneously load and store 8 registers ( this solution is not very feasible )
2. Convert LM/SM into 8 equivalent LW/SW instructions and run it for 8 cycles , keeping other instructions at stall. ( This method will increase will the CPI , but its more feasible and easy to implement that 1st one).



Our solution: we implemented the second method , for that we have introduced a LM/SM block after P1 , which after detection of LM/SM instruction:

- a. Convert it into 8 different LW/SW instructions.
- b. Stall other instructions using the MUX introduced.
- c. Disable the general PC for 8 cycles.

## Setup and Testing :

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.

Test Results:

Inputs

```
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: 0000000000001110 | 14 | 00000000

Register 1 : 0000000000001100 | 12

Register 2 : 0000000000001000 | 8

Register 3 : 0000000000000100 | 4

Register 4 : 0000000000001011 | 11

Register 5 : 0000000000000010 | 2

Register 6 : 0000000000001011 | 11

Register 7 : 0000000000000011 | 3

Complete Outputfile:

[https://drive.google.com/file/d/1p5nt546loqX0YcSMHsE29v5DimPHoy98/view?usp=share\\_link](https://drive.google.com/file/d/1p5nt546loqX0YcSMHsE29v5DimPHoy98/view?usp=share_link)