# Indian Institute of Technology Bombay

**EE309- Course Project**

**IITB- RISC**

Instructor: Prof. Virendra Singh

Team ID - 17

Aman Rishal CH – 22b3914

Aman Milind Moon – 22b1216

Chinmay Tripurwar – 22b3902

Swarup Dasharath Patil – 22b3953

## Preface

*This project has been made by the cumulative effort of Swarup Patil, Aman Rishal Ch, Aman Moon, and Chinmay Tripurwar. This project has been made using the knowledge gained through the course EE309: Microprocessors. In this project, we have implemented a 6-staged pipelined RISC microprocessor in VHDL language. The microprocessor is an 8-register, 16-bit computer system with 14 different instructions.*

## Acknowledgment

*We are really grateful for this project opportunity and would sincerely thank Prof. Virendra Singh for guiding us through the course and the project as well. We would also like to show our gratitude to the teaching assistants for helping through out the course*

## Abstract

*IITB-RISC is a 16-bit very simple computer developed for the 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.*

## Introduction

*IITB-RISC follows the standard 6-stage pipelines (Instruction fetch, instruction decode, register read, execute, memory access, and write back). The architecture is optimized for performance, by including hazard mitigation techniques. It has a forwarding mechanism.*

## Components

- Register File:
  The Register File comprises eight 16-bit registers labeled R0 through R7. R0 serves as both a general-purpose register and a program counter. Alongside, it features two data outputs, D1 and D2, and provides the program counter output from R0. Additionally, it includes a data input, D3.

- Instruction Memory:
  The Instruction Memory is a 512-bit memory specifically designed to store program instructions. It functions by outputting the instructions that need to be executed based on the input memory address provided to it.

- Data Memory:
  The Data Memory is a 512-bit storage unit designed to hold data. It has both read and write enable functionalities, allowing for data retrieval and storage operations. The memory is byte-addressable.

- ALU:
  The ALU performs arithmetic and logical operations, particularly associated with ADD, AND, and NAND instructions. Additionally, it updates the carry (c) and zero (z) flags to reflect the outcome of these operations.

- ALU2:
  This ALU serves the purpose of incrementing the program counter.

- Decoder:
  The Decoder component interprets the 16-bit instruction input by extracting opcodes to determine register addresses, immediate values, the computation bit, and the ALU select signal.

- SE6 and SE9:
  SE6 and SE9 are sign extenders utilized for extending the immediate values Imm6 and Imm9, respectively, to ensure proper sign preservation for further processing.

- Shifter:
  The SHIFTER component executes a left shift by one operation on the input vector and sets the least significant bit (LSB) of the output vector to '0'.

- Forwarding Unit:
  The Forwarding Unit architecture evaluates the current pipeline stage's instruction write signals and destination register addresses, comparing them with the source register address. Based on this comparison, it selects the appropriate forwarding data for the output and determines whether to hold or forward the data.
- Pipeline Registers:
  1. IFIDReg:
     The IFIDReg architecture manages the data flow between the pipeline's instruction fetch (IF) and instruction decode (ID) stages. It synchronously updates the output signals Iout and PCout based on the input signals Iin, PCin, reset, and WR_E, ensuring correct data propagation through the pipeline stages. Additionally, it initializes the outputs to default values when a reset signal is asserted.
  2. IDORReg:
     The IDORReg architecture manages the data flow between the instruction decode (ID) and operand read (OR) stages of the pipeline. It synchronously updates the output signals Iout, PCout, opcode, aRAout, aRBout, aRCout, Compbitout, SelAluout, Imm6out, and Imm9out based on the input signals and control inputs, ensuring correct data propagation through the pipeline stages. Additionally, it initializes the outputs to default values when a reset signal is asserted.

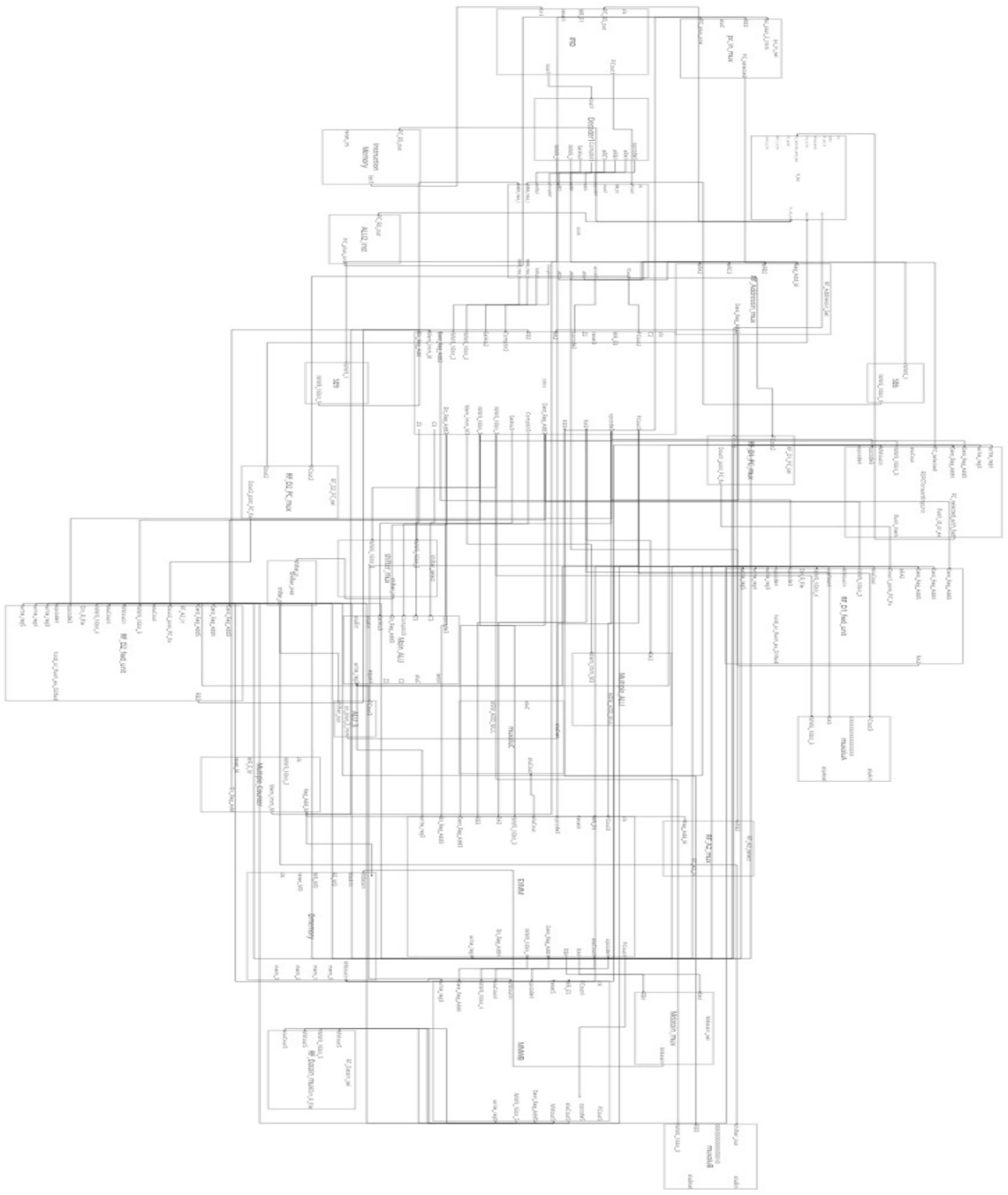
3. OREXReg:
   The OREXReg entity defines a register that stores data between the operand read (OR) and execute (EX) stages of the pipeline. It has inputs for various signals such as the instruction (Iin), program counter (PCin), control signals (WR_E, reset), and various operands and flags. The output ports represent the stored values for the next pipeline stage. The process inside the architecture updates the output signals based on the inputs and control signals, ensuring correct data flow through the pipeline stages.
4. EXMMReg:
   The EXMMReg entity defines a register that stores data between the execute (EX) and memory (MM) stages of the pipeline. It has inputs for various signals such as the instruction (Iin), program counter (PCin), control signals (WR_E, reset), and various operands, flags, and control signals from the execute stage. The output ports represent the stored values for the next pipeline stage. The process inside the architecture updates the output signals based on the inputs and control signals, ensuring correct data flow through the pipeline stages.
5. MMWBreg:
   The MMWBReg entity represents a register between the pipeline's memory (MM) and write-back (WB) stages. It contains inputs for various signals such as the instruction (Iin), program counter (PCin), control signals (WR_E, reset), and various operands, flags, and control signals from the memory stage. The output ports represent the stored values for the next pipeline stage. The process inside the architecture updates the output signals based on the inputs and control signals, ensuring correct data flow through the pipeline stages.

- Multiple Register:
  The Multiple_Reg entity handles register management, associating each register with an immediate value. It features inputs for clock, a 16-bit immediate value, write enable, and reset. Outputs include the register address, the immediate value at that address, and a memory immediate value. The architecture decrements the register address and updates the immediate value based on the write enable. The process reg_to_imm_map maps register addresses to immediate values. Outputs are assigned based on updated values.

- R0PCForwardingUnit:
  The **R0PCForwardingUnit** entity forwards the program counter (PC) value to different pipeline stages based on instruction execution conditions. It determines whether the PC should be forwarded to the next stage or kept unchanged. Additionally, it signals when pipeline stages need to be flushed. Inputs include destination register addresses, ALU and memory outputs, instruction opcodes, and signals for instruction register writes. Outputs provide the updated PC value and signals for flushing pipeline stages.
- ALU3
  "ALU3" is an entity representing an arithmetic logic unit (ALU) that calculates the sum or difference of two 16-bit inputs, PC and Imm, and outputs the result through PCImm2. The architecture features a function, add_sub, which performs the addition or subtraction operation using bitwise operations. The process within ALU3 invokes this function to compute the result, which is then assigned to PCImm2 after removing the carry bit.
- Multiplexers:
  1. The entity "mux_4_1_3bit" represents a 4-to-1 multiplexer with 3-bit inputs and a 2-bit selection signal.
  2. The "mux_2_1_3bit" entity implements a 2-to-1 multiplexer with 3-bit inputs. The selection signal "S0" determines whether the output "mux_out" will be driven by input "I0" or "I1".
  3. The "mux_2_1" entity implements a 2-to-1 multiplexer with 16-bit inputs. The selection signal "S0" determines whether the output "mux_out" will be driven by input "I0" or "I1". If the selection signal is neither '0' nor '1', the output is set to all 'X's.
  4. The "mux_4_1" entity is a 4-to-1 multiplexer with 16-bit inputs. The output "output" is determined by the selection signal "sel". Depending on the value of "sel", the output is selected from inputs "zero", "one", "two", or "three". If "sel" is not '00', '01', '10', or '11', the output is set to all 'X's.

- Control signals:
  - ✓ The control signals **pc_in_sel(1)** and **pc_in_sel(0)** are determined based on various conditions including opcode bits, jump conditions (**jump_in_beq**, **jump_in_blt**, **jump_in_ble**), and the system reset. These signals contribute to selecting the next program counter value based on specific instructions or control flow conditions.
  - ✓ The control signal **WR_E1** enables writing to IFID under conditions where the system reset, jump, or certain hold/flush conditions are inactive.
  - ✓ The control signal **WR_E2** permits writing to a IDOR unless the system reset, jump, or specific hold/flush conditions are active.
  - ✓ **Reset2** also considers the flush and jump conditions that aries due to pipelining.
  - ✓ The control signal RF_Addressin_Sel for RF_Addressin_mux selects between two address inputs based on certain opcode conditions.
  - ✓ The signal `hold_or_flush_ex` is determined by whether there is data forwarding from D1 or D2 stage.
  - ✓ The **reset_M** signal is activated when there is no write enable signal for the memory stage. This helps to prevent memory hazard.
  - ✓ **WR_E_M** is enabled when there's no system reset or stop signal, and when specific conditions related to the opcode and register addresses are met.
  - ✓ **WR_E3** and **Reset3** does the same work as **WR_E2** and **reset2** for **OREX** pipleline register.
  - ✓ All the **AlU select** signals are a combination of opcodes derived by a truth table
  - ✓ The signal **reset4** is set high when either a system reset or a memory flush signal is active. **WR_E4** is enabled when there's no system reset, memory flush, or stop signal. These signals are used for the EXMM pipeline register.
  - ✓ The register file write signal **RO_write_RF** is asserted when there's a write enable signal for the register file and the destination register address is not being held. **PC_write** is active unless there's a system reset, instruction holding, register file write, pipeline flushing, or stop signal.
  - ✓ All the control signals take reset as an input condition for reset operation.

DataPath:



- Work Contribution
    1. Aman Moon: -
       ALUs, Memory, Muxes, Shifter, Register_File, Decoder, Stop instruction, creating test cases
    2. Aman Rishal: -
       Control Signals and its truth tables, debugging LM&SM(Multiple_Register)
    3. Chinmay:-
       Sign Extender, Report, Forwarding unit & Hazard Control logic , R0_fix , Output of Pipeline
    4. Swarup:- Pipeline Registers and connections, R0_Forwarding unit, Holding mechanism, debugging
- LM and SM instruction:

The LM (Load Multiple) instruction loads data from memory into multiple registers specified by the immediate field, with each bit indicating a register. Registers are loaded in reverse order, from R7 to R0, if the corresponding bit is set.

The SM (Store Multiple) instruction stores data from multiple registers specified by the immediate field into memory. Similarly, each bit in the immediate field corresponds to a register to be stored, with registers stored in reverse order, from R7 to R0, if the corresponding bit is set.

The 'WR_E_M' is a signal used to control write operations to memory based on the opcode, reset, stop and the register addresses. The execution of instructions before the LM or SM in the pipeline is kept on hold until the LM and SM instructions are flushed out of the OR stage. This is achieved by controlling the WR_E_M signal. The WR_E_M signal ensures proper synchronization and sequencing of instructions, allowing for correct execution of load and store operations without conflicts.

- Multiply 65*1013
- Assembly code: -

```
LLI R4, 0
LW R7, R4, 0
LW R6, R4, 1
LLI R6,1
BEQ R2, R4, 5
ADA R3, R1, R3
AWC R5, R4, R5
ACW R2, R6, R2
```

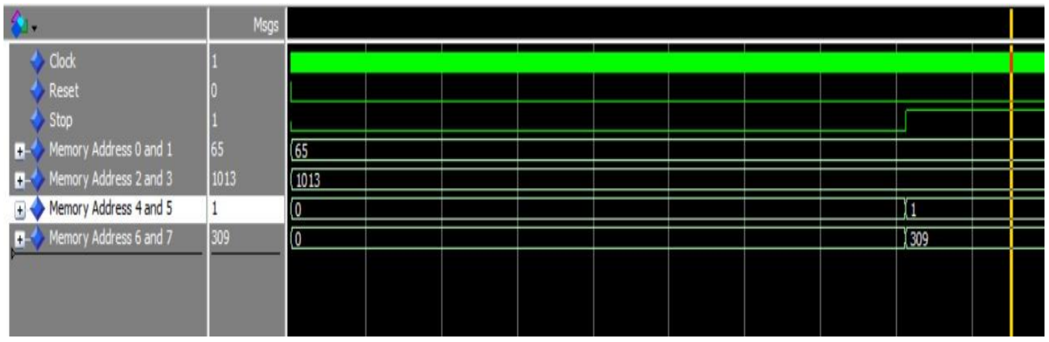
```
JAL R7, -4
SW R5, R4, 4
SW R3, R4, 6
STOP
```

## Data Memory:

```
memarr := (
0=>"00000000",
1=>"01000001",
2=>"00000011",
3=>"11110101",
4=>"00000000",
5=>"00000000",
6=>"00000000",
7=>"00000000",
8=>"00000000",
9=>"00000000",
others => "10110001");
```

In the data memory:

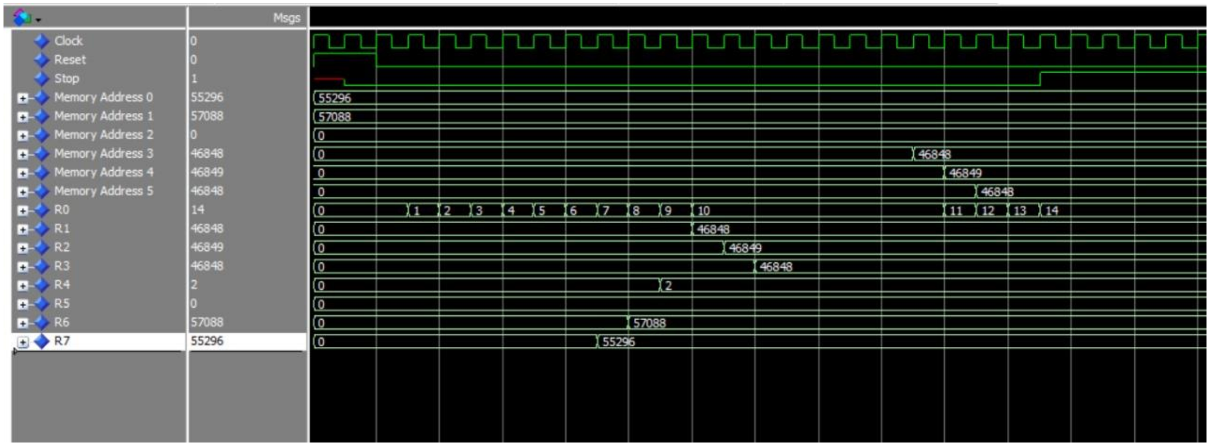
- **Location 0-1**: Contains the value 65.
- **Location 2-3**: Holds the value 1013.
- **Location 4-5**: Represents a carry flag with a value of 1.
- **Location 6-7**: Stores the result value, which is 309.


Msgs
Clock 1
Reset 0
Stop 1
Memory Address 0 and 1 65
Memory Address 2 and 3 1013
Memory Address 4 and 5 1
Memory Address 6 and 7 309
65
1013
0
1
0
309

- Assembly code:-

```
lli r4, 000000000
lw r7, r4, 000000
lw r6, r4, 000001
lli r4, 000000002
ada r1, r6, r7
awc r2, r6, r7
```

adc r3, r6, r7
adz r5, r6, r7
sm r4, 001110100



- Division 7636/37

  Instruction memory

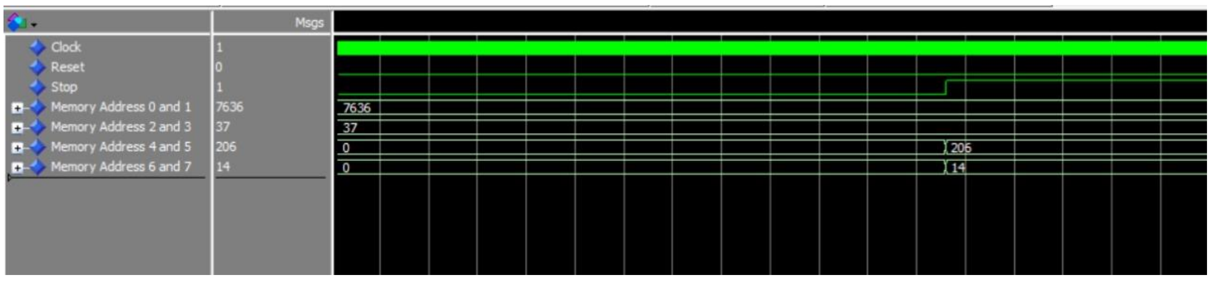
  type memarr is array(0 to 63) of std_logic_vector(7 downto 0);
  signal RAM : memarr := (
  0 =>"00110110",1 =>"00000000", --- LLI R3, 0
  2 =>"00111010",3 =>"00001010", --- LLI R5, 10
  4 =>"00111100",5 =>"00000000", --- LLI R6, 0
  6 =>"01000011",7 =>"10000000", --- LW R1, R6, 0
  8 =>"01000101",9 =>"10000010", --- LW R2, R6, 2
  10=>"10010010",11 =>"10000100", --- BLT R1, R2, 4
  12=>"00000110",13 =>"11000001", --- ADI R3, R3, 1
  14=>"00010010",15 =>"10001111", --- ACW R1, R2, R1
  16=>"11111010",17 =>"00000000", --- JRI R5, 0
  18=>"01010111",19 =>"10000100", --- SW R3, R6, 4
  20=>"01010011",21 =>"10000110", --- SW R1, R6, 6
  22=>"11100111",23 =>"00000011", -- STOP
  others => "10110000");

  Data Memory:

  signal RAM : memarr := (
  0=>"00011101",
  1=>"11010100",
  2=>"00000000",
  3=>"00100101",
  4=>"00000000",
  5=>"00000000",
  6=>"00000000",
  7=>"00000000",

```
        8=>"00000000",
        9=>"00000000",
        others => "10110001");
```

- Location 0-1: Stores the dividend with a value of 7636.
- Location 2-3: Contains the divisor, which is 37.
- Location 4-5: Holds the quotient, calculated as 206.
- Location 6-7: Stores the remainder, which is 14.



```
                          Msgs
Clock                     1
Reset                     0
Stop                      1
Memory Address 0 and 1    7636     7636
Memory Address 2 and 3    37       37
Memory Address 4 and 5    206      0          206
Memory Address 6 and 7    14       0          14
```