

IITB-RISC : EE309 Course Project

May 6, 2023

Group Members:

1. Tamojeet Roychowdhury - 21d070079
2. Joel Anto Paul - 210070037
3. Krisha Shah - 21d070039
4. Mrunal Chourey - 21d070045

1 Introduction

The aim of the project is to design an 8 - register, 16 bit computer system which follows the standard 6 stage pipelines (Instruction fetch, instruction decode, register read, execute, memory access, and write back) and implements VHDL as the High Definition Language. The architecture includes hazard mitigation techniques. The IITB-RISC-23 is general enough to solve complex problems. This architecture allows predicated instruction execution and multiple load and store execution.

The IITB-RISC-23 has 8 general purpose registers R0 - R7. Registers R1 - R7 is used as storage whereas Register R0 is used as the Program Counter which points to the next Instruction. It has a 16-bit memory and an Arithmetic and Logical Unit, also known as ALU.

There are three machine-code instruction formats: R, I, and J type.

R type instruction format

| | | | | | |
|--------------------|----------------------------|----------------------------|----------------------------|-----------------------|--------------------------|
| Op code (4 bit) | Register A (RA) (3 bit) | Register B (RB) (3 bit) | Register C (RC) (3 bit) | Complement (1 bit) | Condition(CZ) (2 bit) |
|--------------------|----------------------------|----------------------------|----------------------------|-----------------------|--------------------------|

I type instruction format

| | | | |
|--------------------|----------------------------|----------------------------|------------------------------|
| Op code (4 bit) | Register A (RA) (3 bit) | Register C (RC) (3 bit) | Immediate (6 bits signed) |
|--------------------|----------------------------|----------------------------|------------------------------|

J type instruction format

| | | |
|--------------------|----------------------------|------------------------------|
| Op code (4 bit) | Register A (RA) (3 bit) | Immediate (9 bits signed) |
|--------------------|----------------------------|------------------------------|

It can process 26 Instructions each of whose encoding and description is given below:
Instruction Encoding

| | | | | | | |
|------|-------|----|--|-----------------|---|----|
| ADA: | 00_01 | RA | RB | RC | 0 | 00 |
| ADC: | 00_01 | RA | RB | RC | 0 | 10 |
| ADZ: | 00_01 | RA | RB | RC | 0 | 01 |
| AWC: | 00_01 | RA | RB | RC | 0 | 11 |
| ACA: | 00_01 | RA | RB | RC | 1 | 00 |
| ACC: | 00_01 | RA | RB | RC | 1 | 10 |
| ACZ: | 00_01 | RA | RB | RC | 1 | 01 |
| ACW: | 00_01 | RA | RB | RC | 1 | 11 |
| ADI: | 00_00 | RA | RB | 6 bit Immediate | | |
| NDU: | 00_10 | RA | RB | RC | 0 | 00 |
| NDC: | 00_10 | RA | RB | RC | 0 | 10 |
| NDZ: | 00_10 | RA | RB | RC | 0 | 01 |
| NCU: | 00_10 | RA | RB | RC | 1 | 00 |
| NCC: | 00_10 | RA | RB | RC | 1 | 10 |
| NCZ: | 00_10 | RA | RB | RC | 1 | 01 |
| LLI: | 00_11 | RA | 9 bit Immediate | | | |
| LW: | 01_00 | RA | RB | 6 bit Immediate | | |
| SW: | 01_01 | RA | RB | 6 bit Immediate | | |
| LM: | 01_10 | RA | 0 + 8 bits corresponding to Reg R0 to R7 (left to right) | | | |
| SM: | 01_11 | RA | 0 + 8 bits corresponding to Reg R0 to R7 (left to right) | | | |
| BEQ: | 10_00 | RA | RB | 6 bit Immediate | | |
| BLT | 10_01 | RA | RB | 6 bit Immediate | | |
| BLE | 10_01 | RA | RB | 6 bit Immediate | | |

| | | | | |
|------|-------|----|------------------------|---------|
| JAL: | 11_00 | RA | 9 bit Immediate offset | |
| JLR: | 11_01 | RA | RB | 000_000 |
| JRI | 11_11 | RA | 9 bit Immediate offset | |

RA: Register A

RB: Register B

RC: Register C

Instruction Description

| Mnemonic | Name & Format | Assembly | Action |
|----------|-------------------------|-----------------------|--|
| ADA | ADD (R) | <i>ada rc, ra, rb</i> | Add content of regB to regA and store result in regC. <i>It modifies C and Z flags</i> |
| ADC | Add if carry set (R) | <i>adc rc, ra, rb</i> | Add content of regB to regA and store result in regC, if carry flag is set. <i>It modifies C & Z flags</i> |
| ADZ | Add if zero set (R) | <i>adz rc, ra, rb</i> | Add content of regB to regA and store result in regC, if zero flag is set. <i>It modifies C & Z flags</i> |
| AWC | Add with carry (R) | <i>awc rc,ra,rb</i> | Add content of regA to regB and Carry and store result in regC regC = regA + regB + Carry <i>It modifies C & Z flags</i> |
| ACA | ADD (R) | <i>aca rc, ra, rb</i> | Add content of regA to complement of regB and store result in regC. <i>It modifies C and Z flags</i> |
| ACC | Add if carry set (R) | <i>acc rc, ra, rb</i> | Add content of regA to Complement of regB and store result in regC, if carry flag is set. <i>It modifies C & Z flags</i> |

| | | | |
|-----|------------------------|-----------------------|--|
| ACZ | Add if zero set (R) | <i>acz rc, ra, rb</i> | Add content of regA to Complement of regB and store result in regC, if zero flag is set. <i>It modifies C & Z flags</i> |
| ACW | Add with carry (R) | <i>acw rc,ra,rb</i> | Add content of regA to Complement of regB and Carry and store result in regC $\text{regC} = \text{regA} + \text{complement of regB} + \text{Carry}$ <i>It modifies C & Z flags</i> |

| | | | |
|-----|----------------------------|------------------------|--|
| | | | <i>It modifies zero flag.</i> |
| SW | Store (I) | <i>sw ra, rb, Imm</i> | Store value from reg A into memory. Memory address is formed by adding immediate 6 bits (signed) with content of reg B. |
| LM | Load multiple (J) | <i>lw ra, Imm</i> | Load multiple registers whose address is given in the immediate field (one bit per register, R0 to R7 from left to right) in reverse order from right to left, i.e, registers from R7 to R0 if corresponding bit is set. Memory address is given in reg A. Registers which are expected to be loaded from consecutive memory addresses. |
| SM | Store multiple (J) | <i>sm, ra, Imm</i> | Store multiple registers whose address is given in the immediate field (one bit per register, R0 to R7 from left to right) in reverse order from right to left, i.e, registers from R7 to R0 if corresponding bit is set. Memory address is given in reg A. Registers which are expected to store must be stored to consecutive addresses. |
| BEQ | Branch on Equality (I) | <i>beq ra, rb, Imm</i> | If content of reg A and regB are the same, branch to $\text{PC} + \text{Imm} * 2$, where PC is the address of beq instruction |
| BLT | Branch on Less Than (I) | <i>blt ra, rb, Imm</i> | If content of reg A is less than content of regB, then it branches to $\text{PC} + \text{Imm} * 2$, where PC is the address of beq instruction |

| | | | |
|-----|--------------------------------|-----------------|--|
| BLE | Branch on Less or Equal (I) | ble ra, rb, Imm | If content of reg A is less than or equal to the content of regB, then it branches to PC+Imm*2, where PC is the address of beq instruction |
| JAL | Jump and Link (J) | jalr ra, Imm | Branch to the address PC+ Imm*2. Store PC+2 into regA, where PC is the address of the jalr instruction |

| | | | |
|-----|----------------------------------|-------------|--|
| JLR | Jump and Link to Register (I) | jlra ra, rb | Branch to the address in regB. Store PC+2 into regA, where PC is the address of the jlr instruction |
| JRI | Jump to register (J) | jri ra, Imm | Branch to memory location given by the RA + Imm*2 |

2 Components

Now coming onto the components used in our CPU, We have used 8 registers clubbed into a Register File, a memory unit and one ALU. Additionally, we have used combinational logic in each of these components to decide the inputs and outputs of these components. Each of the components used in our VHDL code is described below:

2.1 ALU

Input Ports: ALU_A, ALU_B, ALU_load_A, ALU_load_B, r_dest, C, Z, ir3, ir5.

Output Ports: Output, C_out and Z_out.

Functioning - The correct inputs are provided to the ALU in the register read stage. C and Z flags are updated based on whether C and Z are to be written onto or not for that particular instruction. The ir3 register that acts as an input can be used to decide this.

In order to facilitate data forwarding, the output of the add or nand operation is fed to the output of the ALU only if write is to be done (for example, in ADC the destination register is to be written onto only if carry flag in the previous operation is set). Otherwise the ALU output takes the previous value of the destination register so that the data forwarding cases do not get wrong inputs from the ALU outputs.

For second order load dependency i.e. for cases where a load instruction is immediately followed by an arithmetic or logical instruction on the register in which the value is loaded, we use the output of the memory read stage directly into the ALU (this works because for every load instruction we introduce a delay of 1 cycle/ 1 instruction, by the end of which the memory output is available to us but not written onto the reg file yet). For this we need to do extra checking of the previous instruction

3.2 Instruction Decode

The instruction fetched from memory is used to assign control signals (reg write, mem write, C write and Z write). Signal `is_it_load` = 1 whether the decoded instruction corresponds to load or not. PC will be updated for jump instruction.

Based on whether the fetched instruction is load or jump, the `ctr1` signal is updated (whether the fetched instruction asks us to write into memory or update the registers with different values) as follows -

if `is_it_load` = 0 and `pc_change_wait_counter` = 0,

then `ctr1` = `xxx01` if register has to be updated

or `ctr1` = `xxx10` if memory has to be updated

and `ctr1` = `xxx00` for all the other cases.

Similarly C and Z are updated based on whether a carry or zero is generated or not. The `ctr1` signal is updated as follows -

if `is_it_load` = 0 and `pc_change_wait_counter` = 0,

then `ctr1` = `x11xx` if both C and Z has to be updated

or `ctr1` = `x01xx` if only Z has to be updated

and `ctr1` = `x00xx` if none of them has to be updated.

3.3 Register Read

ALU inputs are assigned to in this stage. Data forwarding is done from the next 3 stages based on whether the previous instruction was an arithmetic/logical one too (i.e. modified the reg file value). Carry and Z inputs are also assigned in this stage and have data forwarding too.

In case of branch or jump instructions, we send in PC and immediate as inputs to ALU to calculate the required jump address.

3.4 Execution

The ALU processes the instruction according to the input given to it. The output from ALU, C flag, and the controls bits of execution stage (`ir3` and `ctr2`) are forwarded to the next stage. In this stage data forwarding for load dependent instructions is also done, by mapping the memory read output to the ALU inputs.

It is important to note that the ALU output is equal to the destination register's old value if write doesn't occur. This prevents data forwarding errors in subsequent stages.

3.5 Memory Read

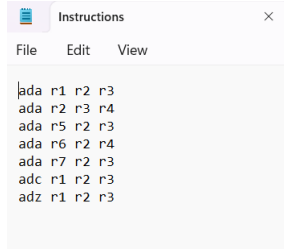
The data is read from memory address given by `alu_out`. The output from ALU, C flag, and the controls bits of memory read stage (`ir4` and `ctr3`) are forwarded to the next stage.

3.6 Write Back

If `mem_write` is 1, then the data in the register address given in the instruction is stored in memory address given by ALU output. Since `mem_write` is set to 1 only if it isn't during a load or branch waiting time, no additional signals need to be checked.

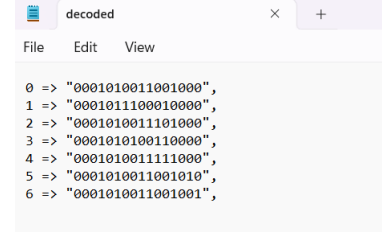
The changes mentioned next are done only if `pc_change_wait_counter` is 0 (its value is set in instruction fetch stage). If `is_it_load` is 1, it signifies a load instruction and we don't change either the PC or the reg file contents. Thus we send the same instruction again for 1 cycle and set `load_wait_over` to 1, which resets `is_it_load` to 0 and subsequent instructions continue normally.

If it is an instruction other than load then PC is set to PC+1 i.e. next instruction. The register files are updated according to the instruction. Also the carry and zero flag are set according to the forwarded data. We keep checking the instruction opcode to check if write actually should occur For



```
ada r1 r2 r3
ada r2 r3 r4
ada r5 r2 r3
ada r6 r2 r4
ada r7 r2 r3
adc r1 r2 r3
adz r1 r2 r3
```

(a) Input instructions in assembly language



```
0 => "0001010011001000",
1 => "0001011100010000",
2 => "0001010011101000",
3 => "0001010011000000",
4 => "0001010011111000",
5 => "0001010011001010",
6 => "0001010011001001",
```

(b) Decoded instructions in binary format

branch and jump instruction, PC_change_wait_counter goes from 0 to 4 for 4 cycles till the branch/jump instruction is executed, after which PC is set to the new value and normal execution resumes.

4 Hazards Control

4.1 Data Dependency for Arithmetic/Logical

We use data forwarding in the reg read stage, from three subsequent stages. This can be handled without having to stop the clock or PC update since the ALU output is available immediately after the inputs are modified according to the previous instruction. Whether the ALU output is actually written onto the reg file or not is taken care of by using the opcode of the instruction being executed in the ALU and the previous values of the C and Z flags.

4.2 Data Dependency for Load

In this case having a one cycle delay i.e. halt is essential since ALU first calculates the address from which data needs to be fetched and output is available from memory only after two cycles. By using a separate update block for signals ALU_load_A and ALU_load_B, which updates at the rising edge of clock (instead of the normal falling edge) we get the ALU output with just a delay of one cycle.

4.3 Branch/Jump and Load Instructions

We need to halt the PC update and send write signals as 0 for one cycle in case of load, and to halt until PC is updated for branch and jump (i.e. a delay of 4 cycles). This is done using the is_it_load, pc_change and pc_change_wait_counter signals as has been described above.

5 Testing

To simulate and test, we initially write the instructions using assembly language in the 'Instructions.txt' file located in the 'Testing' directory. The instructions are written without commas. Afterward, we execute the 'tester.py' script which generates the corresponding binary format instructions and saves them in a file called 'decoded.txt' in the same folder.

We copy the content of 'decoded.txt' and paste it in our VHDL file where memory is initialised.


```

20
21
22 type mem is array(1023 downto 0) of std_logic_vector(15 downto 0);
23
24
25
26
27
28
29
30
31
32
33

```

```

0 => "0001010011001000",
1 => "0001011100010000",
2 => "0001010011101000",
3 => "00010100110000",
4 => "0001010011111000",
5 => "0001010011001010",
6 => "0001010011001001",

```

← Copy paste here
from 'decoded.txt'

```

others=>("1110000000000000");

```