EE309 Project
RISC design Microprocessor

Shubham Birange, 160070014
Pravesh Trivedi, 160070041
Parshva Shah, 160070009
Bhuyashi Deka, 16D070015

October 1, 2024

1 Overview

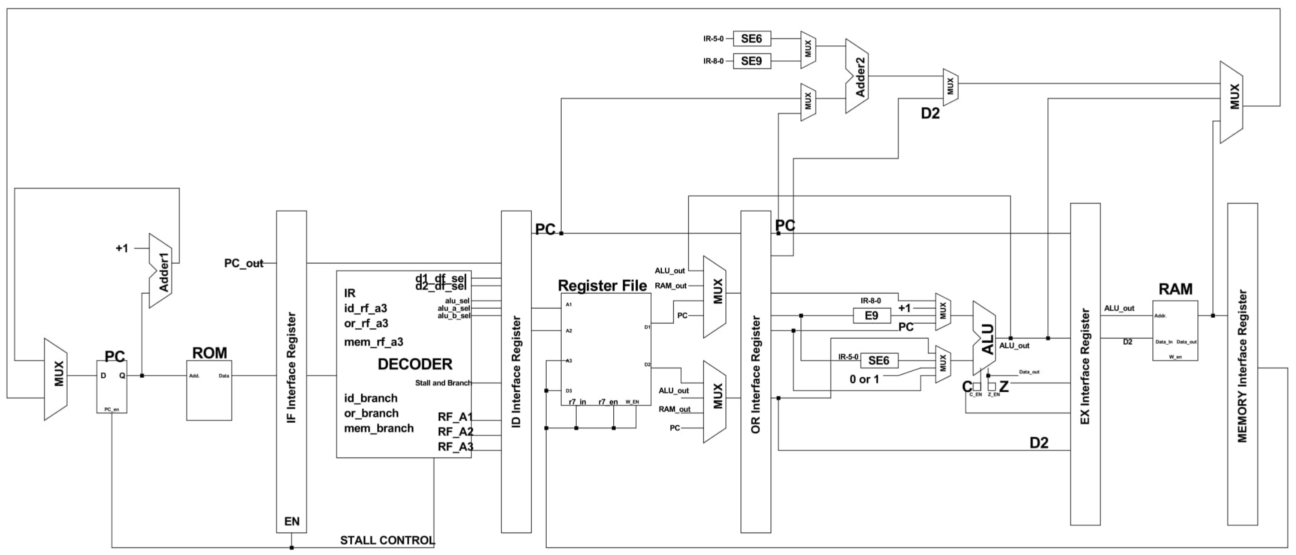
The aim of this project was to implement a fully functional RISC based computer design, IITB RISC whose ISA is provided below. IITB RISC is a 8 - register, 16-bit architecture using 6 stage pipelined implementation. These stages are as follows, Instruction fetch, instruction decode, register read, execute, memory access, and write back. It also includes hazard mitigation techniques like data forwarding.

2 Approach

Since the processor is based on 6 stage pipelining the first step in desiging it was to design a single cycle implementation. It wad then further divided into 6 stages.

- **IF stage:-** This stage consists of the code memory or the ROM and the PC which stores the address of the instruction.
- **ID stage:-** This stage consists of the decoder that decodes the control signals from the instruction including the ALU operation select, RF-A1, RF-A2, RF-A3, and the register file and RAM write enable signals. And also the data forwarding, stall and branch control signals.
- **OR stage:-** This stage contains the register file and the data forwarding multiplexers.
- **EX stage:-** This consists of the ALU of this processor.
- **MEM stage:-** This the Memory read stage and contains the RAM with ALU-out as the address and RF-D2 from the interface register as the inputs and the data-out going to the RF-D3.

- **RB stage:-** In this stage the result from the ALU or the RAM is writen to the register file and also the flags are updated.



--- C:\Users\Vaibhav Birange\Desktop\RISC Pipelining\single_cycle.asc ---

3 Hazards and Dependencies

1. Firstly every branch instruction is assumed to be not taken. So, to nullify the instructions after them we check in the decoder if the any instruction before that in the pipeline is a branch instruction and nullify their write enable signals.

   - For JAL, we need to nullify only the next instruction.
   - For JLR, we need to nullify next 2 instructions.
   - For BEQ and any instruction that branches using the R7 register, we need to nullify next 3 instructions.

     For BEQ branch instruction the instruction are be nullified after the BEQ enters Execute stage.

2. For solving data dependencies we check if the destination of any instruction ahead in the pipeline is the same as the source in decode stage and not nullified.
3. For data dependency in which a destination of LW or LM is the same as the source of an instruction immediately after it we need to stall and put a bubble in the pipeline.
4. For implementing LM and SM we have a priority encoder-decoder in the decoder stage of the processor connected to a 8-bit register. If the instruction in the decoder is LM or SM the pipeline is stalled for as many cycles as the number of register we need to write. When the input to the 8-bit register is all 0 then the stall is lifted and next instruction is fetched. So, the number of cycled required for LM or SM is exactly equal to the number of registers we need to write to.

4 Simulation and testing

The testing and verification of the design was done using the following code:

```
    0 => "0001100100000111",--ADI R4 R4 111
1 => "0001101101000010",--ADI R5 R5 10
2 => "0111110000100000",--SM R6 00100000
3 => "0101100110000001",--SW R4 R6 01
4 => "0011100000000000",--LHI R4 0000
5 => "0011101000000000",--LHI R5 0000
6 => "0001011011000111",--ADI R3 R3 7
7 => "0110101000000110",--LM R5 00000110
8 => "1100001101000011",--BEQ R1 R5 3
9 => "0001001001111111",--ADI R1 R1 -1
10 => "1000110111111110",--JAL R6 -2
11 => "1100010101000100",--BEQ R2 R5 4
12=> "0001010010111111",--ADI R2 R2 -1
13=> "0100001101000001",--LW R1 R5 1
14=> "1000110111111010",--JAL R6 -6
15=> "0001000000000001",--ADI R0 R0 1
16=> "0011110000000000",--LHI R6 0
17=> "0001110110000011",--ADI R6 R6 3
18=> "1100000110000010",--BEQ R6 R0 +2
19=> "0000101011111000",--ADD R5 R3 R7
20=> "0011000000000000",--LHI R0 0
21=> "0000101011111000",--ADD R5 R3 R7
```

This is equivalent to:

```
    R0 = 0
    for(R1 = 7, R1 < 0, R1--)
        for(R2 j = 2, R2 < 0, R2--)
            if(R0 = 3)
        R0 = 0
    else
        R0++
```

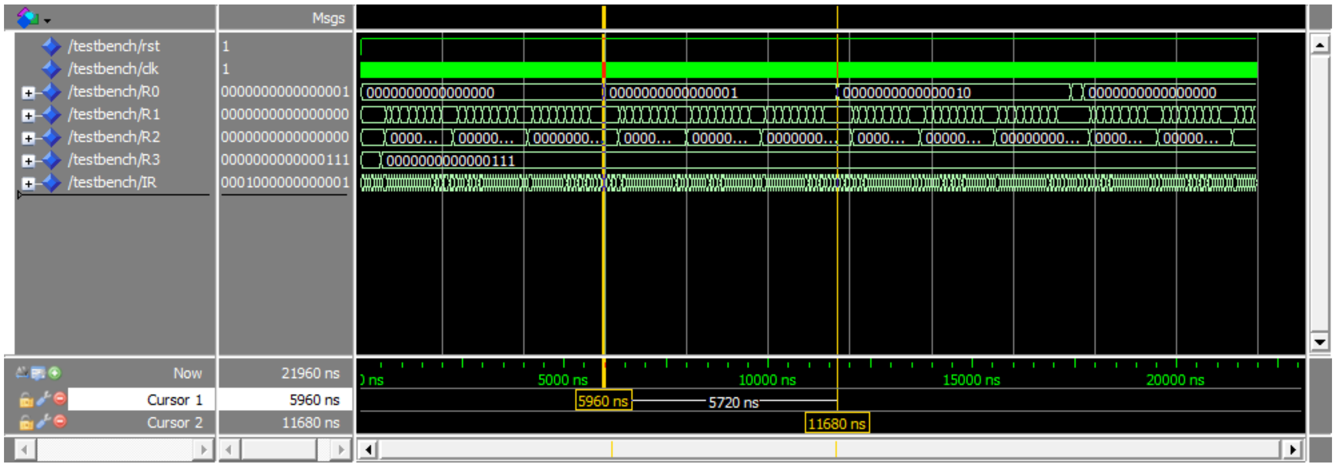

Msgs
/testbench/rst 1
/testbench/clk 1
/testbench/R0 0000000000000001
/testbench/R1 0000000000000000
/testbench/R2 0000000000000000
/testbench/R3 0000000000000111
/testbench/IR 0001000000000001
Now 21960 ns
Cursor 1 5960 ns
Cursor 2 11680 ns
5000 ns
10000 ns
15000 ns
20000 ns
5960 ns
5720 ns
11680 ns

Figure 1: RTL simulation for the above code on the multi-cycle implementation

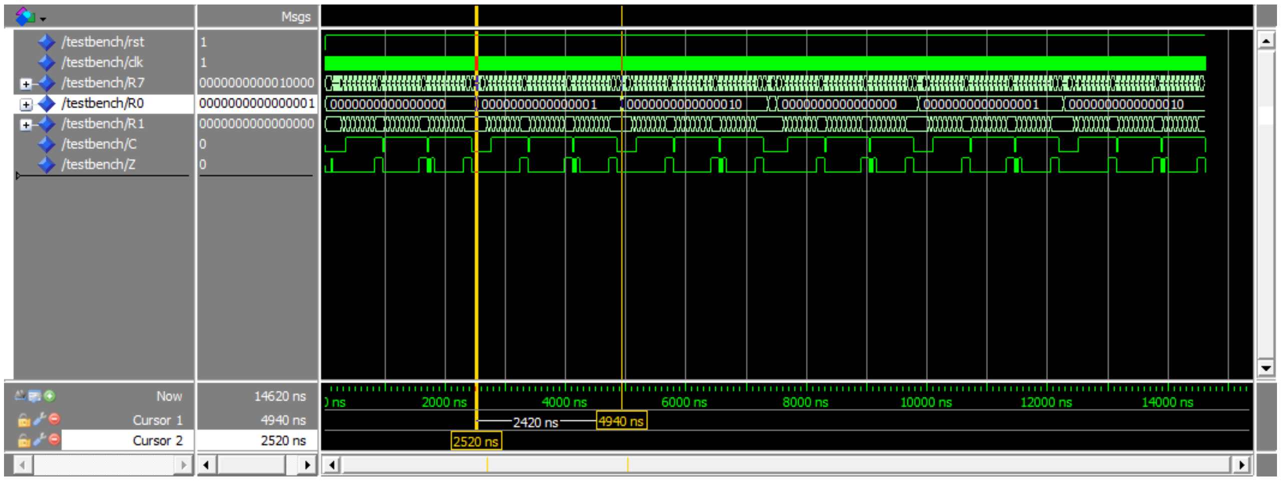


Figure 2: RTL simulation for the above code in a pipelined processor

From figure 1 we can see that the time for change in R0 is 5.72us V/s 2.42 us in Figure 2. So, the pipelined version is more than twice as quick as the multi-cycle version.

5 Code VHDL Files

1. **alu.vhd -** contains the code for the alu used in the design. It can perform addition, subtraction and nand operation on 16-bit numbers with the respective carry and zero flags.

2. **busmultiplexer.vhd -** code for a generic input 2^n multiplexer.
3. **decoder.vhd -** contains the code for decoding the instruction and gives the control signals as output including the reg file addresses and stall and nullify signals.
4. **Interface-Reg.vhd -** contains all the entities interface registers.
5. **processor.vhd -** this is the top-level entity that combines all the components of the datapath including the ALU, the decoder and the interface registers.
6. **RAM.vhd -** useed as the RAM in the processor containing 16-bit data-in, data-out and an address bus.
7. **reg-file.vhd -** contains the code for the 8 registers and the multiplexers for selecting them.
8. **ROM.vhd -** memory containing the instructions.
9. **register.vhd -** generic n-bit register