IITB-RISC-23(PIPELINED)

03.05.2023

Soham Nivargi - 21D070074

Ananya Chinmaya - 210070008

Overview

The purpose of the experiment was to design a 16-Bit RISC capable of performing a set of tasks based on a given set of instructions.

This report consists of an overview of the different components that were used to construct the CPU as well as its working.

Working Principles and Components

The pipelined RISC has 6 pipeline stages, namely instruction fetch, instruction decode, register read, execute, memory access, and write back.

1. **Instruction Fetch** : The PC is fetched from the Register File and provided to the Instruction Memory which gives output in the instruction register. At the same time, the alu_PC computes the PC for the next instruction and writes it back into the register file.
2. **Instruction Decode**: The opcode is fetched from the instruction register and on the basis of the opcode, the control unit outputs the control signals of the instruction. We also have the sign extenders, 7 bit padder in this stage where we extend our immediate to 16 bits. For branch instructions, we are using a branch predictor, and the PC_branch calculated is given to the Instruction fetch stage alongwith a branch_id control signal.
3. **Register read:** In this stage, the read and write addresses are first updated, and then the appropriate forwarding logic is used to get the correct register value in case of any data hazards. In case of branch or jump instructions, the branch_rr control signal is set and the proper PC_branch is used to branch to the correct instruction, along with flushing any previous instructions running.
4. **Execute:** The alu proforms certain operations on the operands.

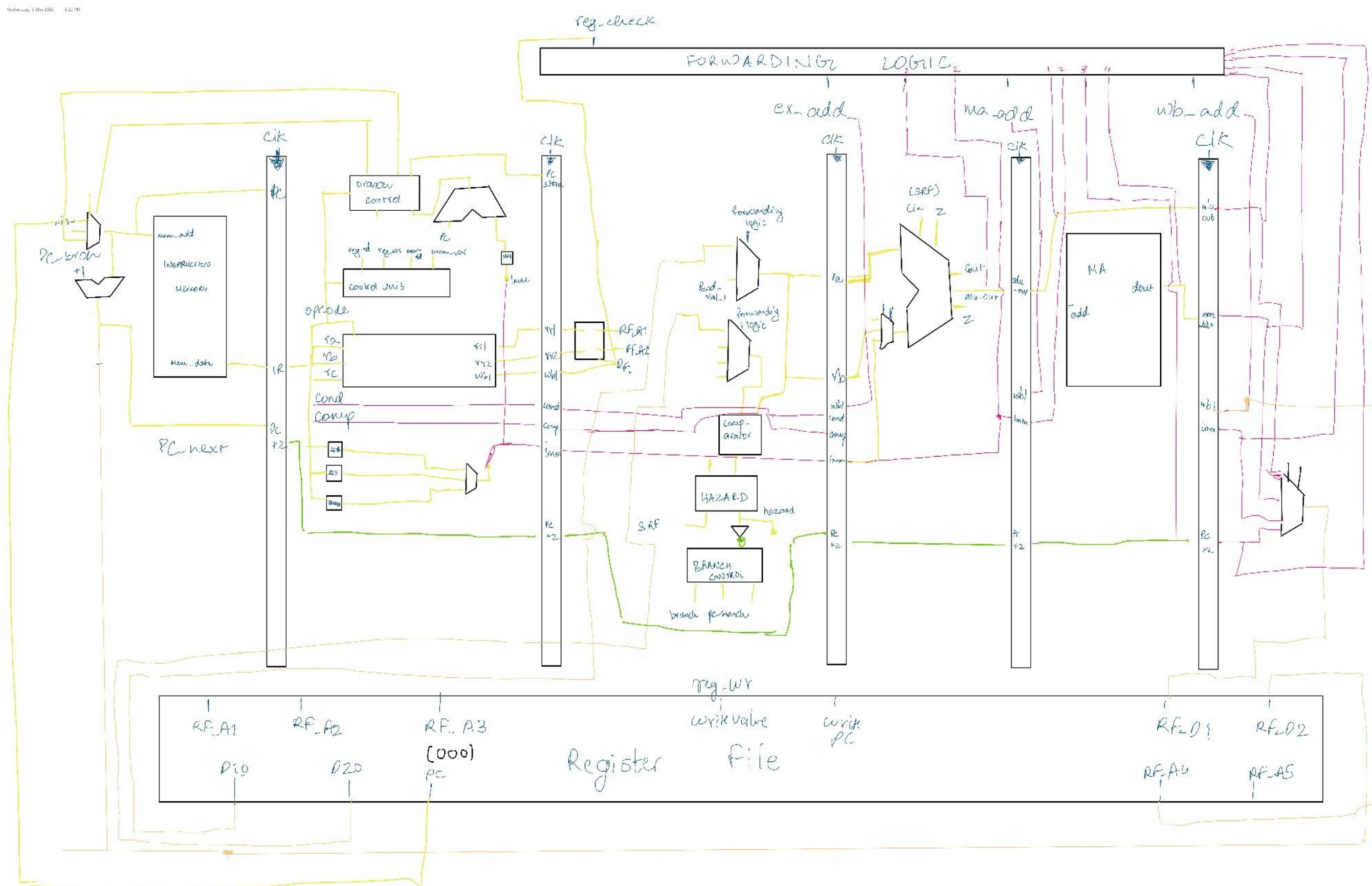
5. **Memory Access:** The memory address is used from the alu output and depending on the nature of instruction, i.e, load/store , the input is given or data is read from the memory.
6. **Writeback:** Depending on the nature of the instruction, the data is input into register file.

The stages are combinational units taking inputs from the pipeline registers placed before them whereas the pipeline registers, register file and data memory are all sequential components which get written only at the rising edge clock.

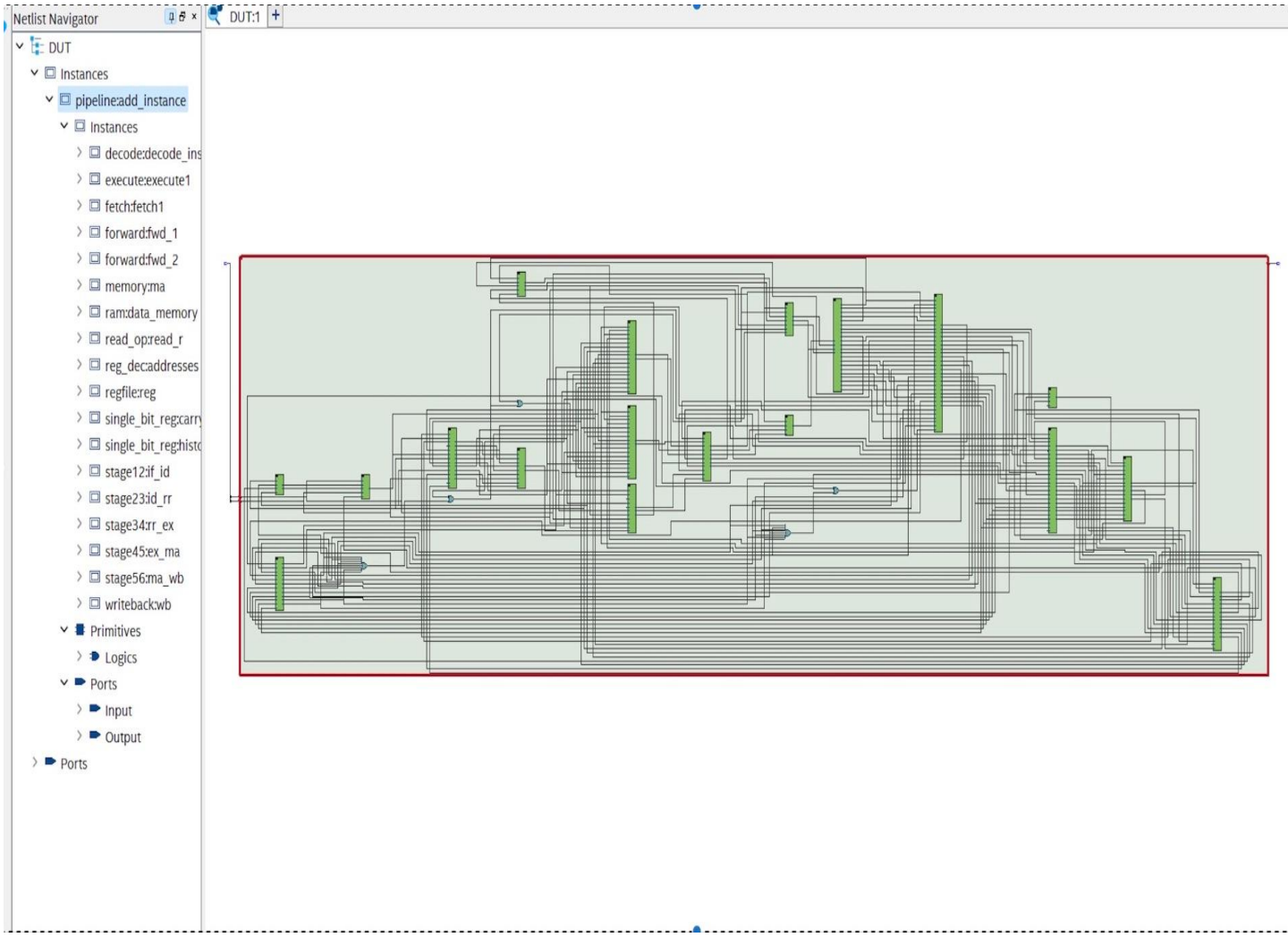
The components used in the making of the datapath are:

- Register File
- 2 Single bit registers
- ALU
- Instruction Memory
- Data Memory
- Control unit
- Forwarding unit
- Branch control
- Control hazard unit
- Comparator
- Complementor
- Left_shifter
- Sign extenders/ Padder

Datapath


reg_check
FORWARDING LOGIC
ex_add
ma_add
wb_add
clk
PC_brch
+1
branch control
control unit
opcode
PC_next
forwarding logic
HAZARD
BRANCH CONTROL
MA
reg_wr
RF_A1
RF_A2
RF_A3
(000)
Register File
write value
write PC
RF_D1
RF_D2
RF_A4
RF_A5

RTL Viewer


Netlist Navigator
DUT
Instances
pipeline:add_instance
Instances
decode:decode_ins
execute:execute1
fetch:fetch1
forward:fwd_1
forward:fwd_2
memory:ma
ram:data_memory
read_op:read_r
reg_dec:addresses
regfile:reg
single_bit_reg:carry
single_bit_reg:histo
stage12:if_id
stage23:id_rr
stage34:rr_ex
stage45:ex_ma
stage56:ma_wb
writeback:wb
Primitives
Logics
Ports
Input
Output
Ports
DUT:1

Handling Hazards

1. Data hazards

Data Forwarding:

- Data forwarding is an efficient way to get the correct value of register in register read stage, instead of stalling which leads to an increase in CPI.
- The correct value of the register may be in any of the execute, memory access or writeback stage.
- We first check if there's a match of any read register address from register read stage to any write register address from execute address, then memory access, then writeback stage, in that order as the latest instruction close to register stage will have the updated value.
- If there's a match for execute stage, the possible instructions before the instruction where the value is read, in the execute stage are
    1. ADA, ADC, ADZ, AWC, etc or ADI or NDU, NDC, NDZ, etc (take the output of the ALU)
    2. LLI (take the immediate value)
- Else If there's a match for memory access,
    1. Take the ALU output if instruction in memory stage is ADD, NAND
    2. Take immediate value if instruction in memory stage is LLI,
    3. Take the data read from memory if the instruction in memory stage is LW, LM
    4. Take the PC+2 value in the pipeline register if the instruction in memory stage is JAL, JLR
- Else if there's a match for writeback stage,
    1. Take the ALU output if instruction in memory stage is ADD, NAND
    2. Take immediate value if instruction in memory stage is LLI,
    3. Take the data read from memory if the instruction in memory stage is LW, LM
    4. Take the PC+2 value in the pipeline register if the instruction in memory stage is JAL, JLR
- Thus, it is ensured that you get the right value in your pipeline register at the end of the register read stage.

2. Control hazards

Branch Instructions(using a Branch Predictor):

- If the opcode is that of a branch instruction in stage 2,we set the branch_id flag to 1, we use a dynamic branch predictor in stage 2 to check if our branch is taken is PC+ Imm*2 OR PC+2.
- So, the branch controller takes the history bit as an input alongwith PC+2 as one input and PC+ Imm*2 as the other input, the branch controller selects PC+ Imm*2 if the history bit is one and selects PC+2 if the history bit is zero.
- It stores the other input into a temporary register.
- Now since we have set the branch flag, we use this flag as the control signal to the mux in instruction fetch stage and it takes the PC from the branch controller in stage 2.
- Now as the instruction moves to stage 3, we have the register values, so we now use a comparator which gives a single bit output as 1 if the branch should've been to PC + imm*2 and output as 0 if the branch should've been to PC+2.
- If the branch taken is to the correct PC, the instruction goes on as it is, else a hazard flag is set in the RR stage. If this flag is set we set our branch_rr flag to 1 and the PC to be branched to is taken from the temporary register which has the alternate PC.
- If the hazard flag is set, the instruction in current ID stage is flushed and the PC_branch from RR stage is given to the instruction memory in stage 1.

Jump Instructions (stall for 1 cycle):

- If the instruction in stage 2 turns out to be a jump instruction, we flush the instruction in if stage running right now,because we are jumping and we will not need it.
- We then move to stage 3, where it calculates the branch location using another ALU used in RR stage, and it puts this value into PC_branch_RR and sets the branch_rr flag which allows stage 1 to take the updated PC.

This is the logic I have used to make the components necessary for mitigating control hazards. The components involved are branch_controller(in stage 2), control_hazard_unit, comparator, and another branch_controller(in stage 3).

I believe this is the best possible solution, in lieu of this course, to minimize the CPI increase associated with control hazards.

If the branch predictor is correct, there are no loss of cycles

If the branch predictor is wrong, one cycle is lost.

Jump instructions lead to a loss of one cycle always.

Conclusion

That's about it. This was a fun-learning process to understand how the microprocessor works using a pipelined architecture, and how the different hazards are handled to give us the maximum performance for real-life applications.

We got an application-based experience and understanding of how processor is designed and what are the possible problems one could face while doing the same as well as how to overcome them.

Overall this project was a great way to understand computer architecture and design and I hope I can use the knowledge I've gained in the time to come.