Pipelined CPU Implementation

Rishabh C S, EE17B067 Om Shri Prasath, EE17B113

November 13, 2019

Aim

To implement a Pipelined Microprocessor CPU running on RISC-V ISA using Verilog implemented on the Xilinx Spartan 3E FPGA.

Abstract

The CPU is capable of running programs written in the RISC-V ISA through a pipelined architecture. The pipeline architecture of the CPU consists of 5 stages, the Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM) and the Write Back (WB). As with any pipelined architecture, we encountered structural, data as well as control hazards. These hazards have been handled using the concepts of stalling and register forwarding.
In all of the stages, the signals which need to be sent as output to other stages of the pipeline are first registered within the module, and then sent as the output.

Module Explanation

Pipeline IF

Port Definitions

Inputs :

- **clock** - This provides the clock for the module in order to write into registers.
- **reset** - The reset signal resets all the registers to their designated base values.
- **pc_new [31:0]** - In case of branch or jump instructions, the details of the new value which the PC must jump to is known only at the end of the Execute stage in the pipeline. Thus, this input comes from the EX stage, and gives the value of the PC that must be jumped to.
- **if_branch** - In case of branch or jump instructions, this signal comes from the EX stage and tells whether we need to take the value given by "pc_new" or not.

Outputs :

- **instruction [31:0]** - This passes the 32 bit instruction to the ID stage to decode.
- **stall_load** - In case the previous instruction is a load instruction, and there is a data hazard with the current instruction, then we need to stall the current instruction. This signal is propagated till end of the pipeline so that we can set all the write signals to 0.

- **pc [31:0]** - The 32 bit value of PC is propagated to the ID stage.

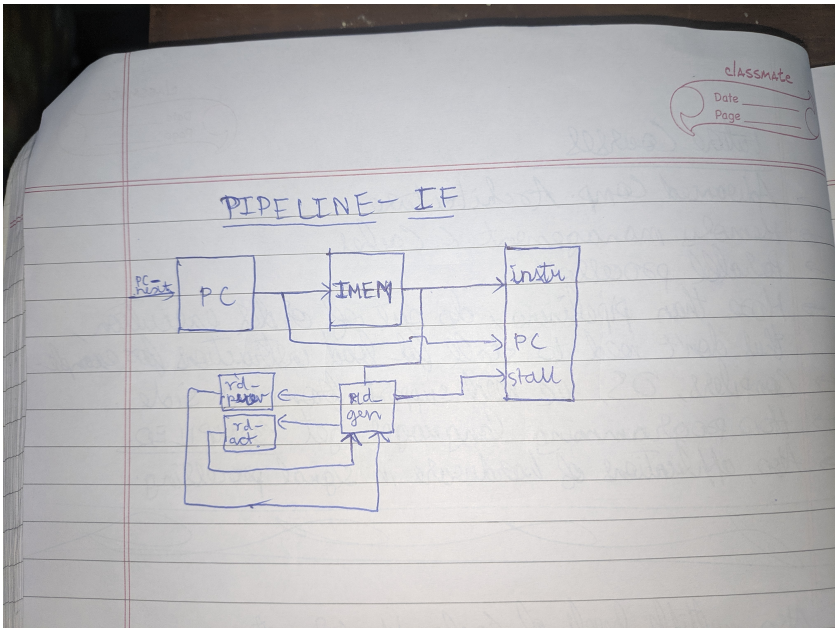
Module Function :

The Instruction Fetch module primarily fetches the 32-bit instructions from the IMEM. This module keeps track of the program counter (PC), and using the PC, fetches the desired instruction from the IMEM. The IF then passes on the instruction as well as the PC to the next stage of the pipeline, the Instruction Decode stage.

Some other specific functions of the IF stage :

- **Stalling due to Load** - When the previous instruction is of a Load instruction, and the source register of the current instruction is the same as the destination register of the previous Load instruction, then we need to stall the pipeline for one cycle. Thus we need to ensure two things:
  - The value of PC for the next clock cycle is set as the same value as the current PC.
  - We send a signal "stall_load" to the next stages of the pipeline to ensure all the write signals to the memory and the registers are set to 0.
- **Setting PC in the case of branches** - We get the inputs "pc_new" and "if_branch" from the EX stage. If the "if_branch" signal is high, then the value of PC for the next clock cycle is automatically set to "pc_new". Else, the value of PC for the next clock cycle is calculated in the normal way.

Module Diagram



Pipeline ID

Port Definitions

Inputs :

- **clock** - This provides the clock for the module in order to write into registers.
- **reset** - The reset signal resets all the registers to their designated base values.
- **instr** [**31:0**] - The 32 bit instruction is passed from the IF stage.
- **pc** [**31:0**] - The 32 bit value of the PC (Program Counter) is passed from the IF stage.
- **stall_load** - If this value is 1, we need to stall the current instruction due to data hazard with a previous Load instruction.

Outputs :

- **rs1** [**4:0**] - The value of the source register 1.
- **rs2** [**4:0**] - The value of the source register 2.
- **rd** [**4:0**] - The value of the destination register.
- **alu_op** [**3:0**] - The opcode for the ALU functioning.
- **imm** [**31:0**] - The immediate value for all the functions with sign extension and alignment.
- **main_opcode** [**5:0**] - This is a 6 bit opcode which defines any instruction in the RISC-V ISA uniquely, and is propagated till the end of the pipeline.
- **pc** [**31:0**] - The value of the PC from the IF stage is just passed on to the EX stage.
- **reg_forwarding_type** [**3:0**] - This 4 bit value defines the type of register forwarding that needs to be done.
- **stall_load** - The value of the "stall_load" from the IF stage is just passed on to the EX stage.

Module Function :

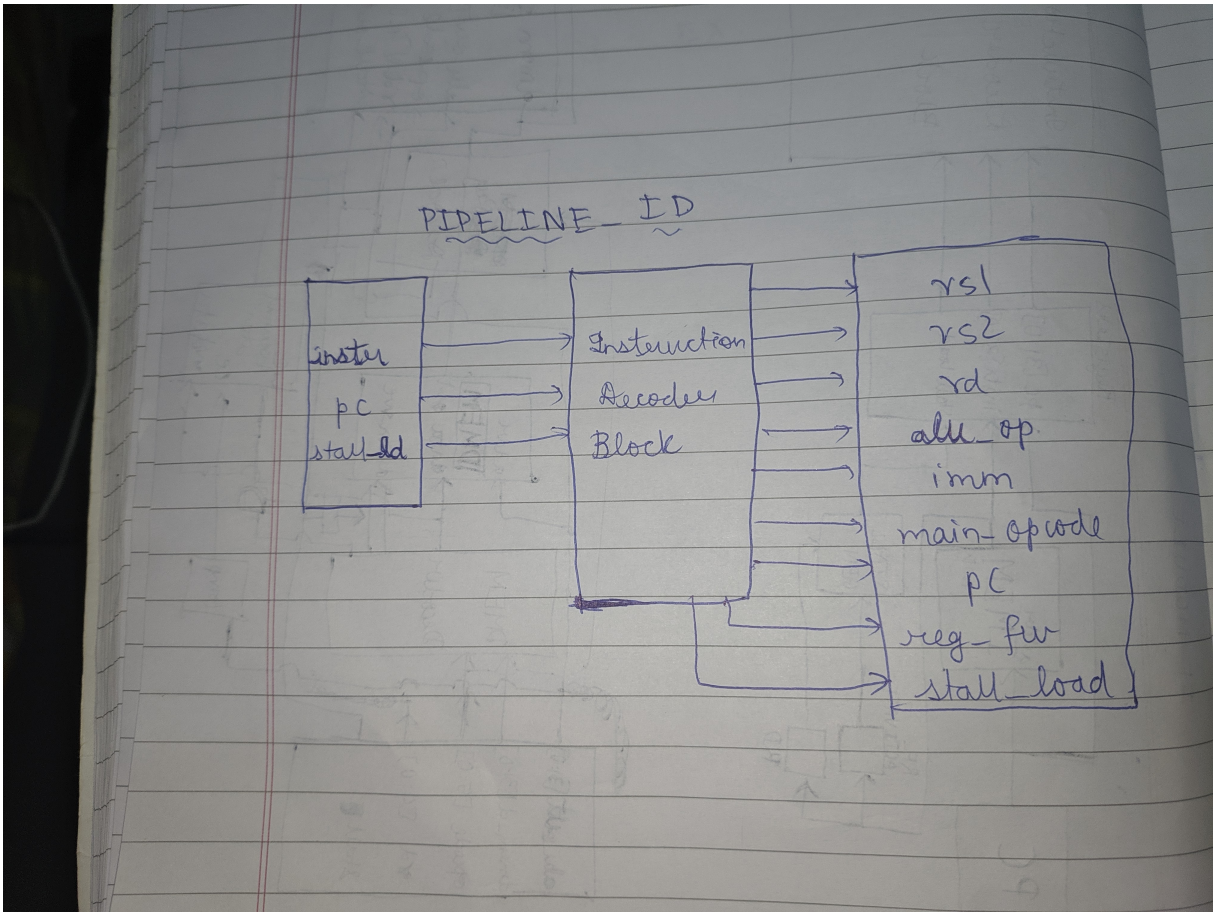
The Instruction Decode (ID) module gets the 32 bit instruction from the IF stage. Using this instruction, we set the control signals, required opcodes and register addresses for the subsequent stages of the pipeline.

Some other specific functions of the ID stage :

- **Setting the main opcode** - In the subsequent stages of the pipeline, we need various information from the 32 bit instruction. However, 32 bits is a huge size to pass on till the last stage of the pipeline. Thus, we compress the 32 bit value to a 6 bit opcode such that:
  - Each instruction in the RISC-V ISA is defined uniquely.
  - Instead of using a large number of 32-bit comparators to assign the 6-bit opcode, we reroute certain bits of the original instruction to generate the 6 bit opcode. This ensures that the least amount of hardware possible is used to generate the 6-bit opcode.
- **Setting the ALU opcode** - The ALU is capable of doing 16 operations. While the first 10 operations are arithmetic or logical operations, the other 6 operations are used in branch instructions to check whether the branch condition is met. Thus, for choosing from these 16 operations, we need a 4-bit opcode. The 4-bit opcode is assigned such that:
  - Instead of using a large number of 32-bit comparators to assign the 4 bit opcode, we reroute certain bits of the original instruction to generate the 4-bit opcode. This ensures that the least amount of hardware possible is used to generate the 4-bit opcode.
  - For all ALU instructions, the 4-bit opcode is just the LSB 4 bits of the 6-bit main opcode of the instruction. This reduces the hardware used in generating the ALU opcode.
- **Setting the Immediate value** - The immediate values need to be aligned and sign extended differently for different instructions. Thus, by rerouting specific bits from the original instruction, and a case statement to find the type of instruction, the 32-bit Immediate value is assigned. This is passed on to the subsequent stages of the pipeline.
- **The different types of register forwarding** - In every pipelined design, data hazard is one of the biggest problems. This happens when when the current instruction tries to read from a source register which is yet to be updated by a previous instruction. To minimize the number of stalls required, we do three types of register forwarding:
  - **Register Forwarding from the EX stage of previous instruction**
  - **Register Forwarding from the EX stage of instruction from 2 cycles before**
  - **Register Forwarding from the MEM stage of Load instruction from 2 cycles before**

These forwarded values will then replace either the value of RF[rs1] or RF[rs2] or both depending which source register is dependent on the destination register of some previous instruction.
Thus, to decide which type of register forwarding must be done, we set a 4-bit opcode. This is set by comparing the destination register values of the previous instructions to the source register values of the current register using case and if statements.

Module Diagram



Pipeline EX

Port Definitions

Inputs :

- **clock** - This provides the clock for the module in order to write into registers.
- **reset** - The reset signal resets all the registers to their designated base values.
- **rs1** [**4:0**] - The value of the source register 1.
- **rs2** [**4:0**] - The value of the source register 2.
- **rd** [**4:0**] - The value of the destination register, from the ID stage.
- **alu_op** [**3:0**] - The opcode for the ALU functioning.
- **imm** [**31:0**] - The immediate value for all the functions with sign extension and alignment.
- **main_opcode** [**5:0**] - This is a 6 bit opcode which defines any instruction in the RISC-V ISA uniquely, and is propagated till the end of the pipeline.
- **pc** [**31:0**] - The value of the PC from the IF stage is just passed on to the EX stage.
- **reg_forwarding_type** [**3:0**] - This 4 bit value defines the type of register forwarding that needs to be done.
- **stall_load** - The value of the "stall_load" from the IF stage is just passed on to the EX stage.
- **rd_wb** [**4:0**] - This comes from the WB stage. Since the RF module is present in the EX stage, WB sends the address "rd" to write into.
- **indata_wb** [**31:0**] - This comes from the WB stage. Since the RF module is present in the EX stage, WB sends the value of the data to be written into RF.
- **we_wb** - This comes from the WB stage. Since the RF module is present in the EX stage, WB sends the write enable signal of the RF.
- **reg_forwarding_mem** [**31:0**] - This is the 32-bit value from the MEM stage of the pipeline used in register forwarding in the case of Load instructions.

Outputs :

- **alu_out_o [31:0]** - The 32-bit output from the ALU. In the case of Memory instructions, it holds the value of the address to be read from/written into. In the case of JAL, JALR, LUI and AUIPC instructions, it holds the 32-bit value to be written into the Register File (RF).
- **wdata_o [31:0]** - The 32-bit value to be written into the DMEM in case of Store instructions.
- **rd_o [4:0]** - The value of the destination register in the RF, this will be propagated till the end of the pipeline.
- **main_opcode_o [5:0]** - The 6-bit opcode generated in the ID stage, and is propagated till the end of the pipeline.
- **pc_new_o_wires [31:0]** - In case of a branch or jump instruction, EX calculates the new 32-bit value of the PC that the program must jump to, and sends to the IF stage.
- **if_branch_o_wires** - This signal becomes 1 if the branch (or jump) is taken, and is sent to the IF stage.
- **main_stall_o_register** - This signal becomes 1 if the current instruction needs to be stalled, ie, the write signals to both the RF and the DMEM need to be set to 0.

Module Function :

The Execute (EX) module uses the control and data signals from the ID stage to read from the RF and then use the ALU for certain instructions. The EX module also gets signals from the WB stage, and writes the required values into the RF. Most of the ALU operations, address calculations for the LD and ST instructions, calculation of the new PC for branch and jump instructions and deciding whether a certain branch is taken or not is all done in the EX stage.

Some other specific functions of the EX stage :

- **The RF Module** - This has 32 registers of 32-bits each, with the first (zeroth) register always having the value of 0. The module can read values of 2 registers instantaneously and simultaneously, and write into 1 register at clock edges (synchronous write). The RF module has the following ports:
  Inputs :
  - **clock** - This provides the clock for the module in order to write into registers.

- **rs1** [**4:0**] - The 5-bit address of the first register to read from, out of 32 registers.
- **rs2** [**4:0**] - The 5-bit address of the second register to read from, out of 32 registers.
- **rd** [**4:0**] - The 5-bit address of the register to write into, out of 32 registers.
- **we** - The signal needs to be set to 1 if we need to write into the RF.
- **indata** [**31:0**] - In case of writing into the RF, the 32-bit value that needs to be written.

Outputs :

- **rv1** [**31:0**] - The 32-bit value read out from the register given by address rs1.
- **rv2** [**31:0**] - The 32-bit value read out from the register given by address rs2.

- **The ALU Module** - The ALU is capable of doing 16 operations. While the first 10 operations are arithmetic or logical operations, the other 6 operations are used in branch instructions to check whether the branch condition is met. The op_code used to select the required ALU operation is determined in the ID stage, and passed to the EX stage. The ALU module has the following ports:
Inputs :

  - **in1** [**31:0**] - The 32-bit first input to be operated on by the ALU.
  - **in2** [**31:0**] - The 32-bit second input to be operated on by the ALU.
  - **op** [**3:0**] - The 4-bit opcode used to select 1 operation out of the possible 16 operations.

  Outputs :

  - **alu_out** [**31:0**] - The 32-bit result of the ALU operations. In case of the 10 arithmetic or logical operations, the output will be the result of the operation on the 2 inputs. In case of the 6 operations used for branch instructions, the output will be a single bit, 1 if the branch should be taken, and 0 if the branch should not be taken.

- **Selecting appropriate data from Register Forwarding** - In case of data hazards with the previous instructions, register forwarding is done to avoid stalls. The values of the source registers cannot be taken from the RF module since they are not updated. The 4-bit reg_forwarding_type signal from the ID stage gives us information about the type of register forwarding that needs to be done. Hence, using this, the values of the 2 inputs to the ALU is either taken from the RF module or from the appropriate register forwarded values.

- The values of the Main ALU Output of the previous two instructions are stored in registers. These can be used for register forwarding from the EX stage.
- In case of the value that needs to be register forwarded in LD instructions, the value comes as input to the EX module from the MEM module.

The 2 inputs to the ALU are either one of the register forwarded values, or the values from the RF module, and this is decided by using the 4-bit reg_forwarding_type signal.

- **Setting the new value of PC** - In the case of branches or jumps, the PC must jump either to (PC+offset) in case of branch or JAL instruction or to (RF[rs1]+offset) in case of JALR instruction. Thus, using the 6-bit main opcode to determine the type of instruction, the value of the new PC is set.

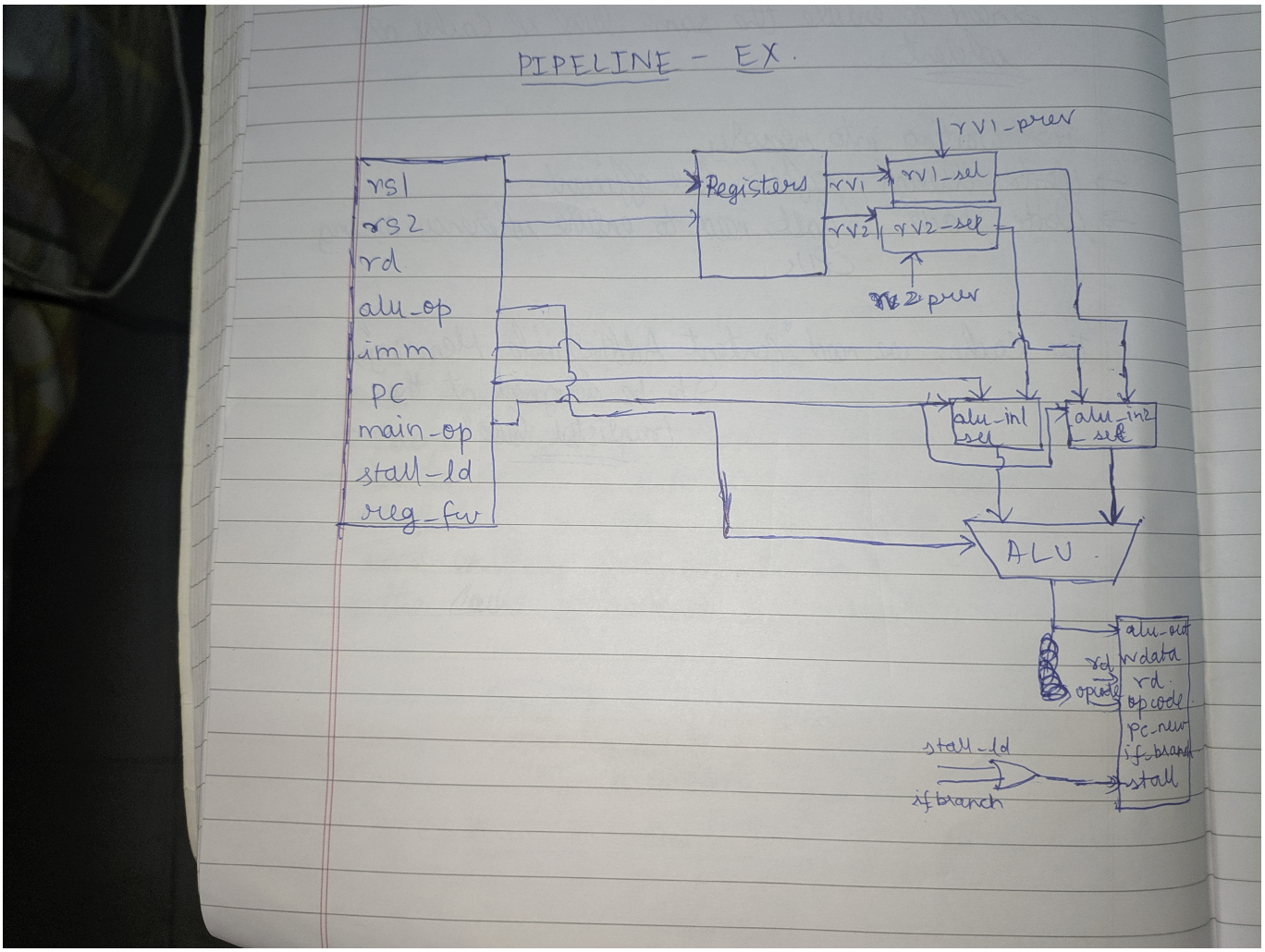
- **Determining whether a branch should be taken** - The PC must jump if it is a jump instruction or if it is a branch instruction, and the branch is taken. Thus, using the 6-bit main opcode to determine the type of instruction, we set this signal to 1 if the PC needs to jump. In case of branch instruction, whether the branch should be taken or not can be known by seeing the ALU output.
  An Interesting Catch
  We also need to be careful when any of the previous two instructions have been successful branch instructions/jump instructions. In that case, the current instruction needs to be stalled. Hence, even if the current instruction is a successful branch/jump instruction, this signal must still be set to 0!

- **Setting the Main ALU output** - This 32-bit Main ALU output is propagated till the end of the pipeline, the WB stage. This contains the 32-bit value that must be written into the RF in case of ALU, LUI, AUIPC, JAL and JALR instructions, and contains the 32-bit address for the DMEM in case of the LD and ST instructions.
  - In case of ALU, LD and ST instructions, the Main ALU output is the same as the output of the ALU module.
  - In case of JAL, JALR, LUI and AUIPC instructions, the values PC+4, PC+4, Immediate and PC+Immediate values respectively are assigned to the 32-bit Main ALU Output signal.

- **Determining whether the current instruction should be stalled** - The current instruction should be stalled in 3 cases.
  - If the "stall_load" signal is 1, propagated from the IF stage.
  - If the previous instruction was a successful branch instruction/jump instruction.
  - If the instruction 2 cycles before was a successful branch instruction/jump instruction.

If any of the above condition is satisfied, then the "main_stall" signal is set to 1, and is propagated to the subsequent stages of the pipeline.

Module Diagram



Pipeline MEM

Port Definitions

Inputs :

- **clock** - This provides the clock for the module in order to write into registers and to submodules
- **reset** - The reset signal resets all the registers and submodules to their designated base values.
- **alu_out [31:0]** - Output of the ALU Module
- **dmem_write_data [31:0]** - Raw data to be written into DMEM
- **instr_opcode [5:0]** - 6-bit Opcode for the instructions
- **reg_write_address [4:0]** - Address to which the Register Write Data should be written to
- **stall** - Signal to check whether stalling should be implemented

Outputs :

- **dmem_out_data [31:0]** - Formatted output of DMEM
- **instr_opcode [5:0]** - 6-bit Opcode for the instructions
- **alu_out [31:0]** - Output of ALU Module
- **stall** - Signal to check whether stalling should be implemented
- **reg_write_address [4:0]** - Address to which the Register Write Data should be written to

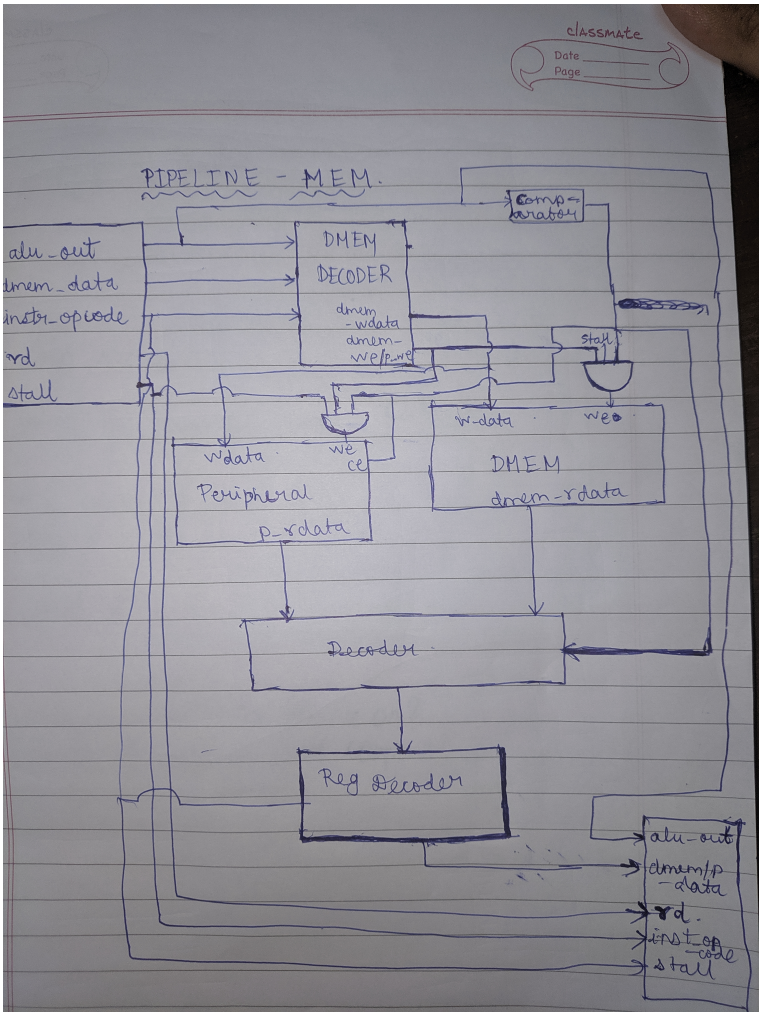
Module Function :

The Memory Write Module (MEM) takes the instruction and data to be written into the DMEM. It then sets the data to be written into the DMEM into the proper format based on the instruction, otherwise it reads the data from DMEM based on the instruction and sends it to the next module (WB).

Some other specific functions of the MEM stage

- **Decoding the data for DMEM** - The module contains a decoder which decodes the incoming data based on the instruction opcode and sets the data to be written into the DMEM. This decoder also generates the write signal for DMEM.
- **Decoding the data for Registers** - The module contains another decoder which decodes the data which is being outputted by the DMEM module based on the instruction, to be written into Registers. (It is done here itself instead of the WB module to enable register forwarding)
- ***Decoding for Peripheral** - For the peripheral case, the module also generates the Chip Enable signal based on the incoming address, and sets the write signals to DMEM and Peripheral accordingly

Module Diagram



Pipeline WB

Port Definitions

Inputs :

- **clock** - This provides the clock for the module in order to write into registers and to submodules
- **reset** - The reset signal resets all the registers and submodules to their designated base values.
- **reg_write_address** [**4:0**] - Address to which the Register Write Data should be written to
- **dmem_out_data** [**31:0**] - Formatted output of DMEM
- **instr_opcode** [**5:0**] - 6-bit Opcode for the instructions
- **alu_out** [**31:0**] - Address to which the Register Write Data should be written to
- **stall** - Signal to check whether stalling should be implemented

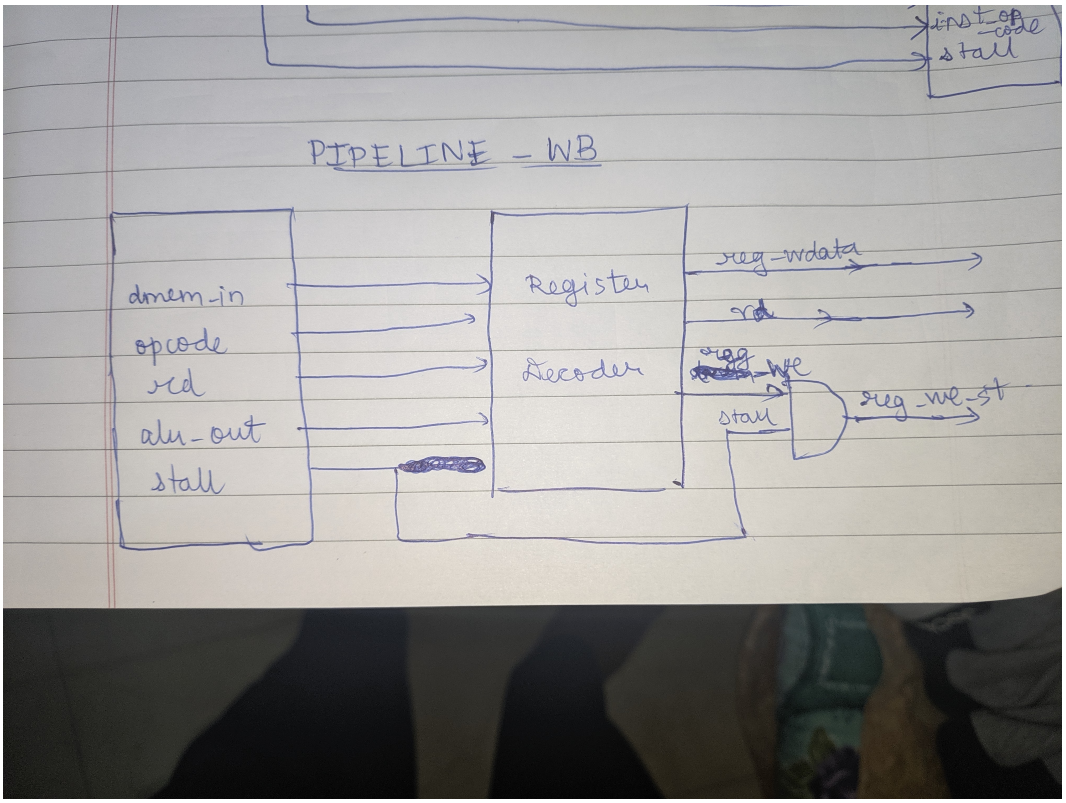
Outputs :

- **reg_write_data** [**31:0**] - Formatted output of DMEM
- **reg_we** - 6-bit Opcode for the instructions
- **reg_write_address** [**4:0**] - Address to which the Register Write Data should be written to

Module Function :

The Write Back Module (WB) sets the value to be written into Registers based on the instruction opcode and also sets the write enable signal for the registers accordingly.

Module Diagram



Peripheral (Accumulator)

Inputs :

- **clock** - This provides the clock for the module in order to write into registers and to submodules
- **reset** - The reset signal resets all the registers and submodules to their designated base values.
- **ce** - Signal to enable the peripheral
- **we** - Signal to enable changing the peripheral
- **addr** - Address (actually to indicate the function to be executed in the peripheral)

Outputs :

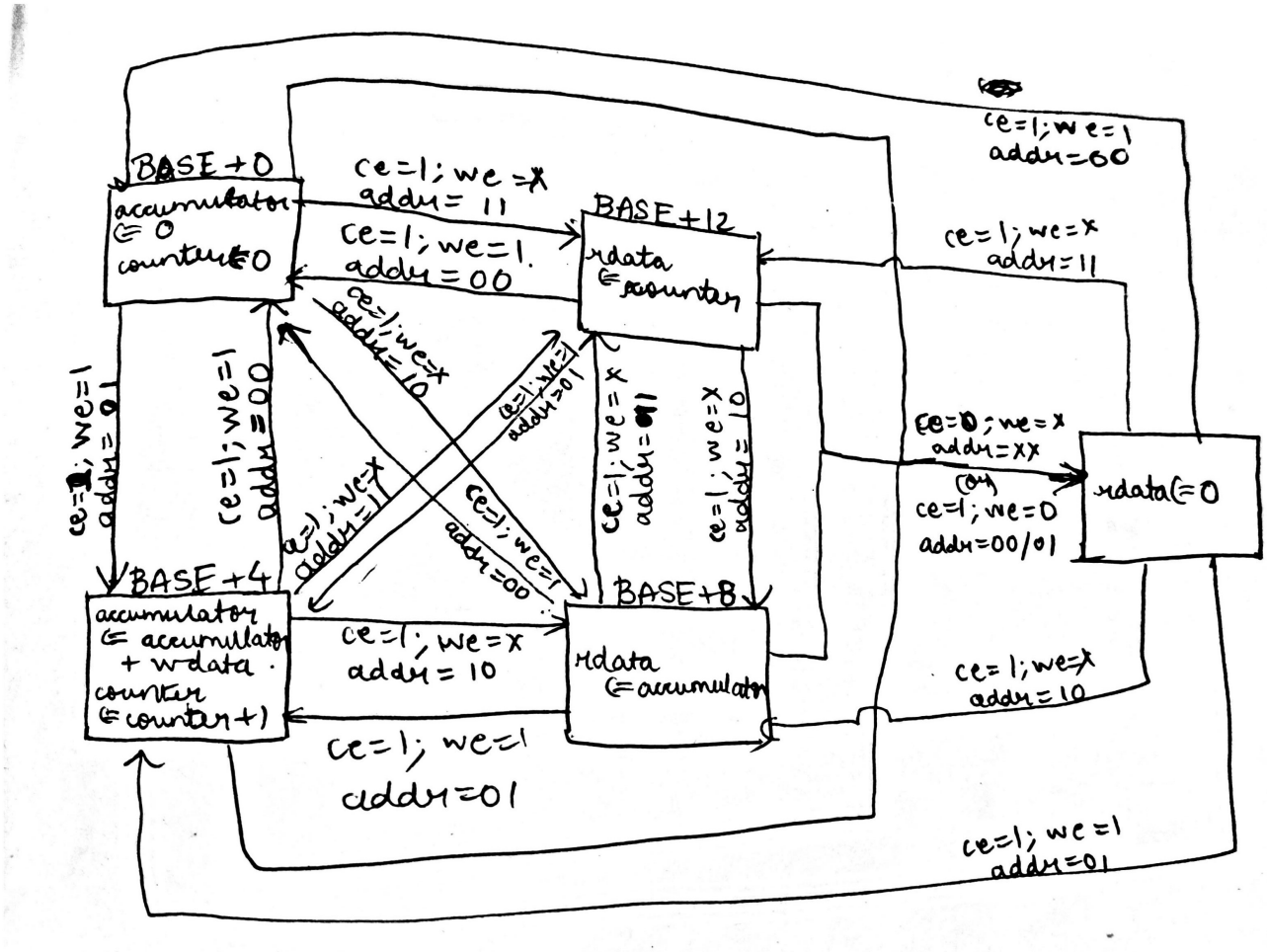
- **rdata** - Data to be read from the peripheral

Module Function :

The peripheral is capable of adding numbers written to it. It will have multiple register addresses as described below. Values that are written into the BASE+4 register will be accumulated in an internal 32-bit register, and each time a value is written, an internal 32-bit count register will be incremented. BASE+8 can be used to read back the total accumulated sum, and BASE+12 will return the number of values written so far.
CE and WE signals are inputs. Only when CE is 1 should the peripheral respond to read requests, and only when CE=WE=1 should it accept writes.
Memory map of the peripheral:

- BASE + 0: Write - reset: write any value here to reset internal sum and count to 0
- BASE + 4: Write - value: write a number here to add it to internal sum
- BASE + 8: Read - return the total accumulated sum so far (since last reset)
- BASE + 12: Read - return the number of values written so far (since last reset)

Module State Diagram



Synthesis Reports and Timing Analysis :

Assignment 4 :

- Best Clock Period Achievable - 4.800ns
- Best Clock Frequency Achievable - 208.333 MHz
- Worst case slack - 0.844ns
- Timing Errors - 0

Assignment 5 :

- Best Clock Period Achievable - 4.962ns
- Best Clock Frequency Achievable - 201.532 MHz
- Worst case slack - 0.847ns
- Timing Errors - 0

Appendix

The Main Opcode

Instead of propagating the whole 32-bit instruction through the stages of the pipeline, we compress into a 6-bit opcode to uniquely define every instruction in the RISC-V ISA.

ALU Non Immediate	[5:4] = 00
ALU Immediate	[5:4] = 01
Load	[5:3] = 100
Store	[5:3] = 101
Branch	[5:3] = 110
LUI	[5:0] = 111000
AUIPC	[5:0] = 111001
JAL	[5:0] = 111010
JALR	[5:0] = 111011

- The MSB 2-3 bits of the main_opcode are enough to determine the type of the instruction. This helps in setting many control signals where just the type of instruction is required. For example, by just comparing one bit, main_opcode[5], if =0, it is ALU type, if =1, it is not ALU type instruction.

- The main_opcode has been set in such a way that the most common case is fastest. Since most of the control signals just require to know whether it is an ALU instruction or not, it takes just comparison of 1 bit to determine it.
- The LSB bits of the main_opcode give details about what the instruction is exactly, such as whether it is Load Byte or Load Word.
- The LSB bits are assigned by using the least hardware by just rerouting bits from the main 32-bit instruction. For example, in the Load/Store/Branch type of instructions, the main_opcode[2:0] is just instruction[14:12].

Register Forwarding

The values used for register forwarding can be obtained from 3 locations, depending on the type of instruction:

- From the EX stage of previous instruction.
- From the EX stage of instruction 2 cycles before.
- From the MEM stage of instruction 2 cycles before.

Each of these values from register forwarding can replace either RF[rs1] or RF[rs2] or both.
Thus, we use the 4-bit signal 'reg_forwarding_type' to determine the specific type of register forwarding required:

0000	EX_ONE_RS1
0001	EX_ONE_RS2
0010	EX_TWO_RS1
0011	EX_TWO_RS2
0100	MEM_ONE_RS1
0101	MEM_ONE_RS2
0110	EX_ONE_TWO_RS1_RS2
0111	EX_ONE_TWO_RS2_RS1
1000	EX_ONE_ONE_RS1_RS2
1001	EX_TWO_TWO_RS1_RS2
1111	No register forwarding

- Here, the first column "EX" or "MEM" means whether we get the register forwarded value from the EX (EX) stage or the MEM (MEM) stage.
- The second column "ONE" or "TWO" means whether we get the register forwarded value from the instruction of one cycle ago (ONE) or the instruction of two cycles ago (TWO).
- The third column "RS1" or "RS2" means whether the register forwarded value replaces RF[RS1] (RS1) or RF[RS2] (RS2).

Splitting of Work :

Assignment 4 :

- IF, ID, EX - Rishabh
- MEM, WB - Om Shri
- Debugging - Rishabh and Om Shri

Assignment 5 :

- Peripheral and Arbiter - Om Shri
- Debugging - Rishabh and Om Shri