EE 309 - Microprocessors

Pipelined RISC IITB

Roll Numbers	Name
210100021	Archit Gupta
210040138	Sarvadnya Desai
210040146	Rishabh Shetty
21011014	Akshat Taparia

INTRODUCTION:

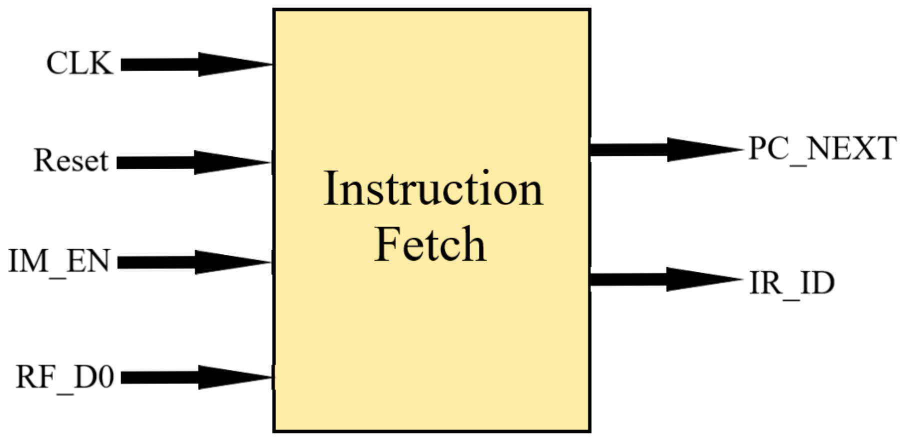
IITB RISC is a very basic yet powerful microprocessor which has 26 instructions in its ISA. It comprises 8 general purpose 16 bit registers. This pipelined version of IITB RISC follows these 6 stages for instruction execution:

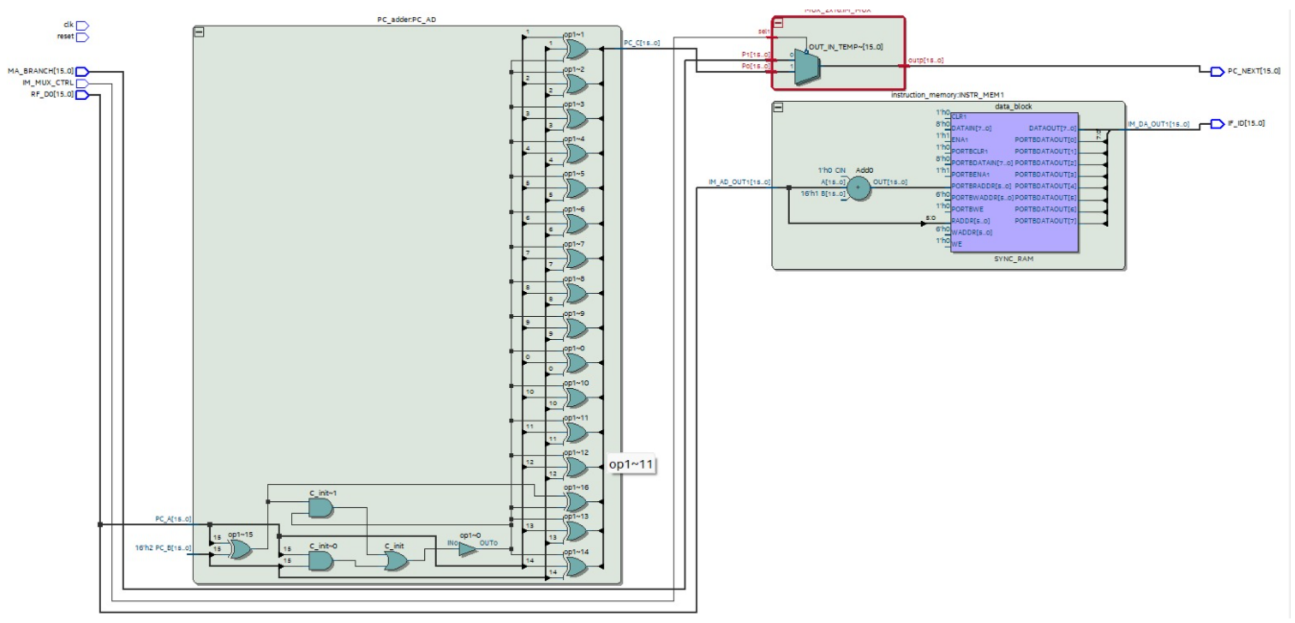
1. Instruction Fetch
2. Instruction Decode
3. Register Read
4. Execution
5. Memory Access
6. Write Back

Data Path Stages and Pipelining Registers:

The Data Path stages are listed below with their representative diagram showing inputs and outputs in the stage and their purpose

Instruction Fetch

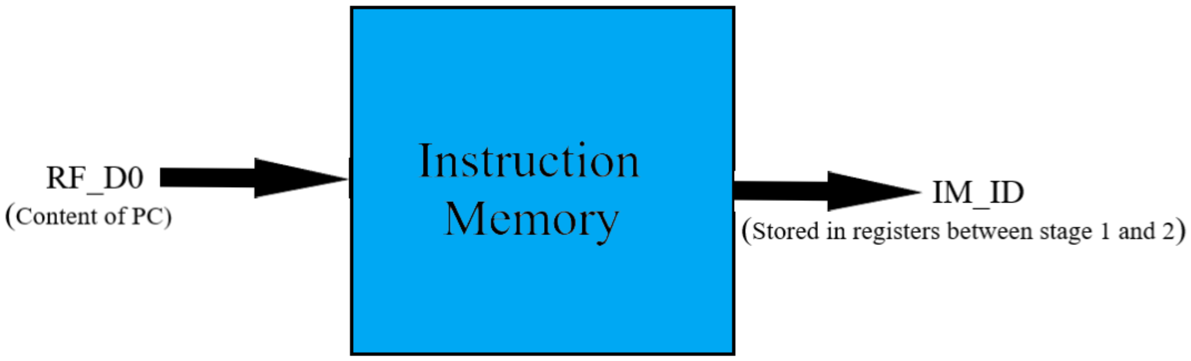




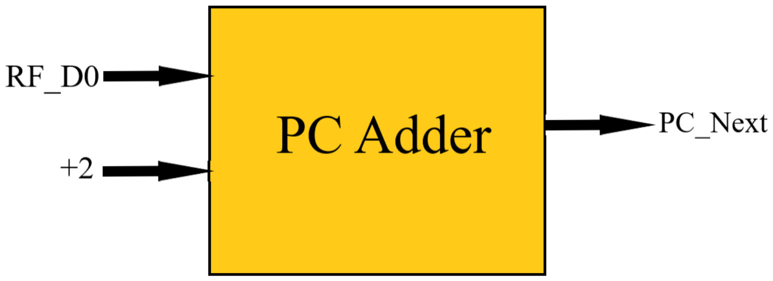
- Clock, Reset and IM_Enable(Instruction Memory enable bit) are 1 bit inputs to this stage
- RF_D0 is a 16 bit input which is the data stored in the PC
- PC_Next is the updated value of PC(PC + 2)
- IR_ID is the read instruction which is stored in the intermediate stage register, its decoded in the next stage

Components

- Instruction Memory

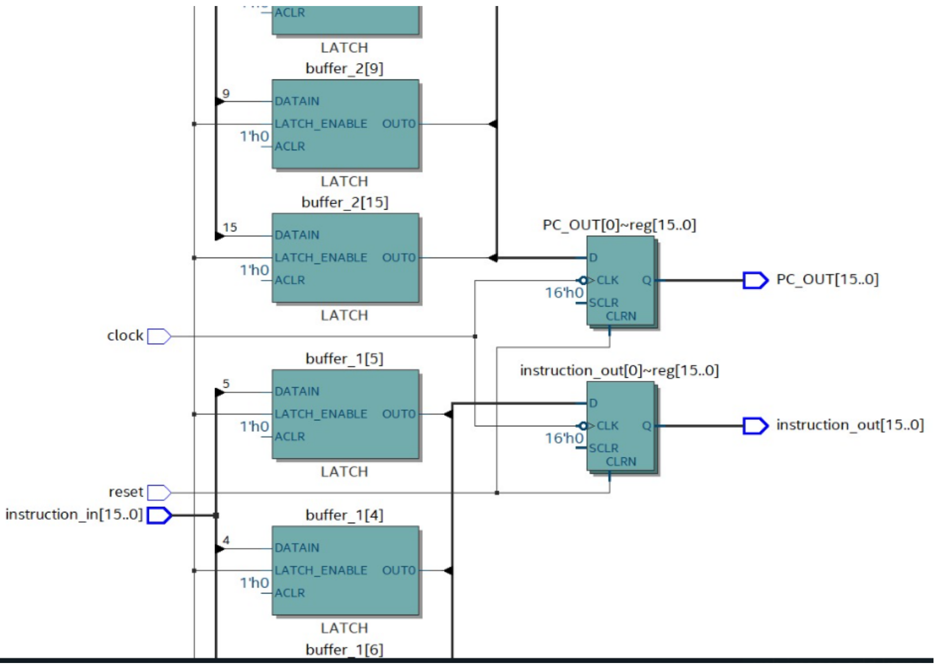


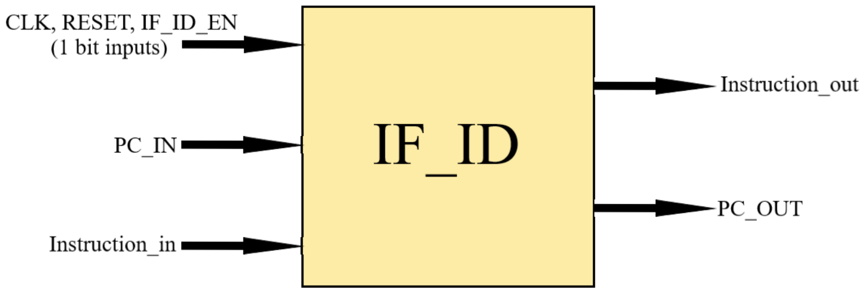
- PC Adder



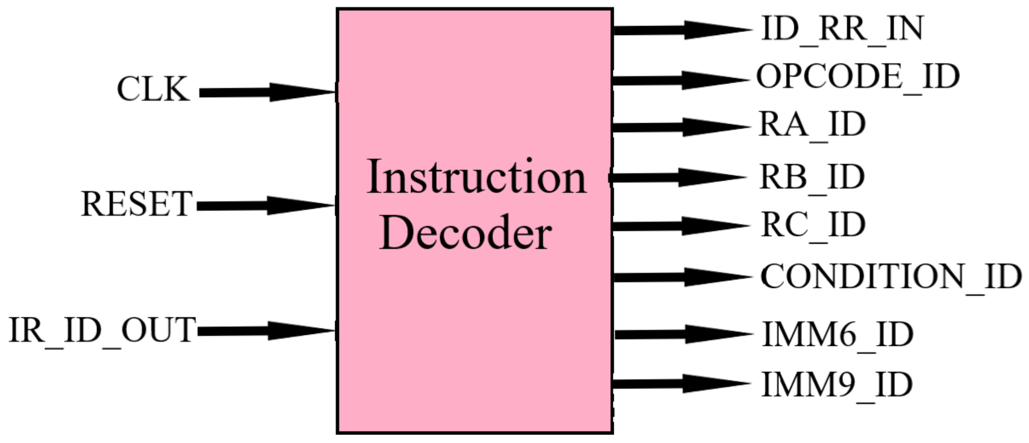
Pipeline Register(IF_ID):

- This register contains the instruction and the current instruction location(PC).

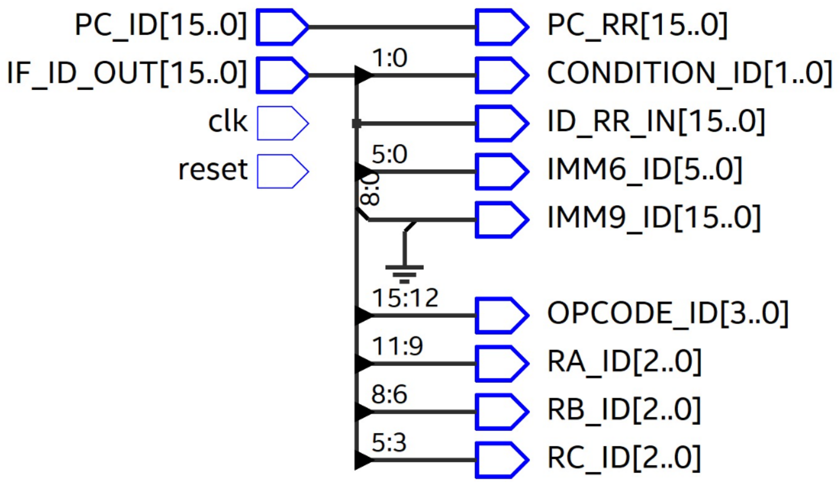




Instruction Decoder

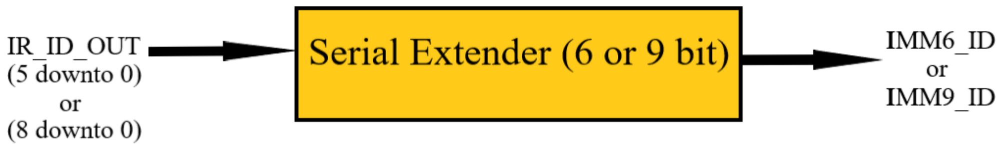


- Clock and Reset signals are 1 bit inputs (common to every stage)
- IR_ID_OUT is the 16 bit instruction fetched in the previous stage.
- ID_RR_IN is simply an output which carries forward the instruction read IR_ID_OUT to the next stage.
- The other outputs are decoded information from our instruction like OPCODE, addresses of reg A,B,C conditional flags and immediate values after being serially extended to 16 bits.

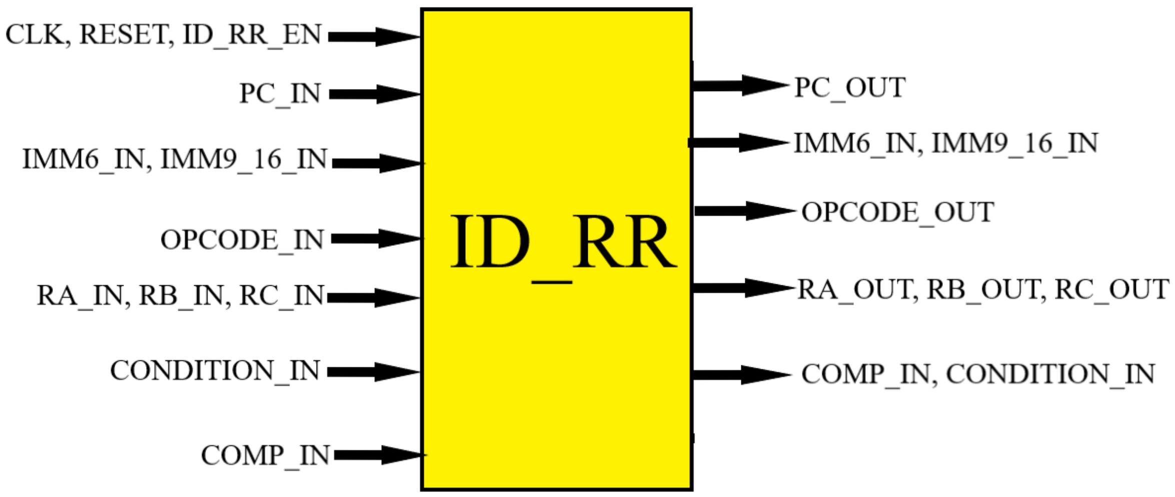


Components

- Serial Extender(6 and 9 bits)

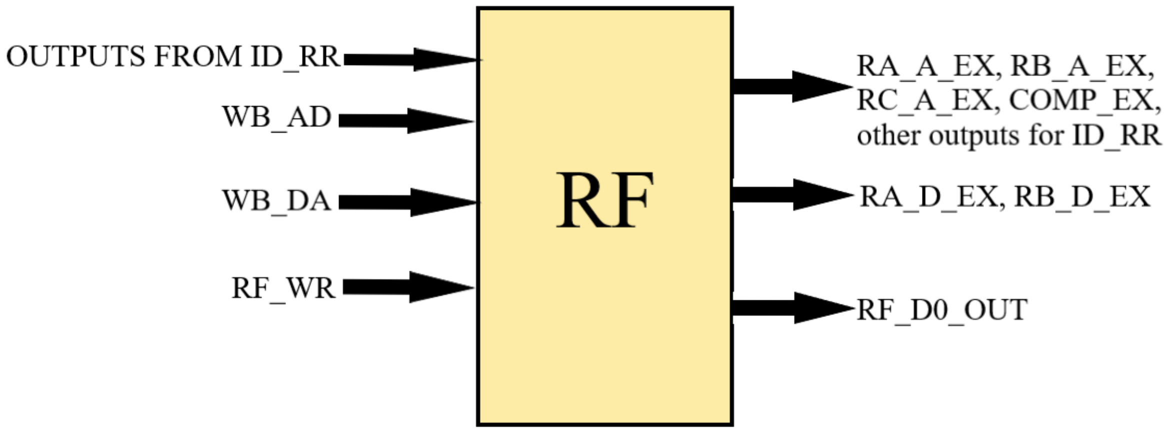


Pipeline Register(ID_RR)

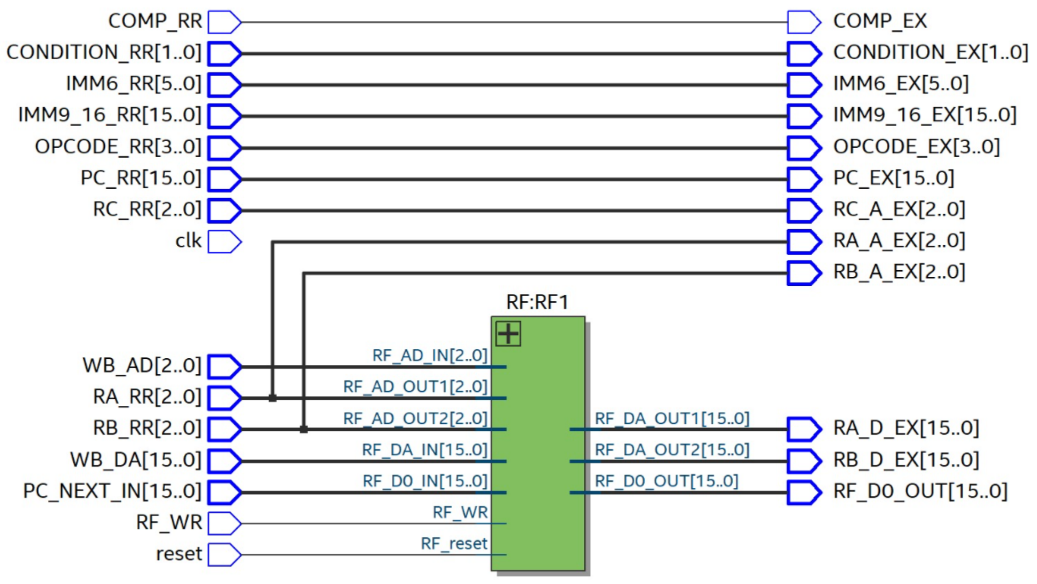


- The instruction Decode decodes the instruction in all possible formats i.e. R, I, J formats. We use the meaningful entries based on the OPCODE of the instruction in the future stages using multiplexing.

Register Read

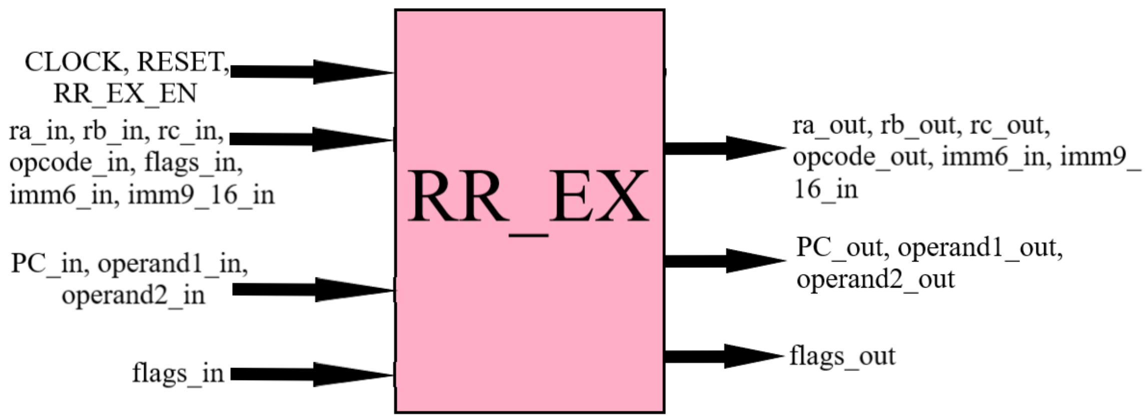


- In this stage of the pipeline we are trying to read the data contents in the register file which contains 8 registers, addresses corresponding to 3 bits given in the opcode.
- We obtain the decoded information from instruction as input from previous stage pipeline register ID_RR.
- WB_AD is the line which comes from the Write Back stage(stage 6) and it gives a 3 bit input corresponding to the address of the register.
- WB_DA is similar to WB_AD and it gives a 16 bit input which is the data corresponding to the register address provided by WB_AD, basically we write back the data in a register.
- RF_WR is the register file enable signal which allows us to write data into a register file.
- The RF_WR signal is under control of the hazard unit to avoid unnecessary writes on the Register Read.



The above RTL netlist should add proper clarity to the points mentioned previously.

Pipeline Register (RR_EX)


CLOCK, RESET, RR_EX_EN
ra_in, rb_in, rc_in, opcode_in, flags_in, imm6_in, imm9_16_in
PC_in, operand1_in, operand2_in
flags_in
RR_EX
ra_out, rb_out, rc_out, opcode_out, imm6_in, imm9_16_in
PC_out, operand1_out, operand2_out
flags_out

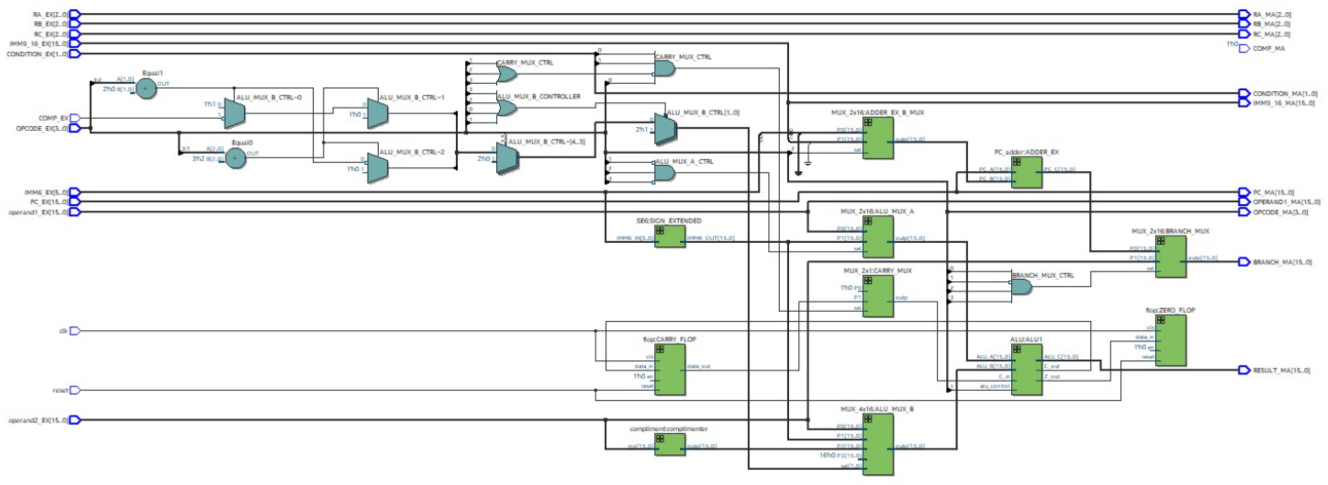
- Operand1, Operand2 are the data contents of RA and RB
- Flags are the C,Z Condition flags obtained from the instruction.
- Implemented a flop for the C and Z flag that is passed to next stages

Execution Stage

- In this stage, based on the flags, conditions and opcode received from the previous pipelining register RR_EX we perform instruction on the received operand(s). The following are the important component of the datapath for this stage:

Components:

- ALU
- ADDER_EX(An dedicated adder for branching instructions.)
- Flops to store the generated flags and control on generating flags.

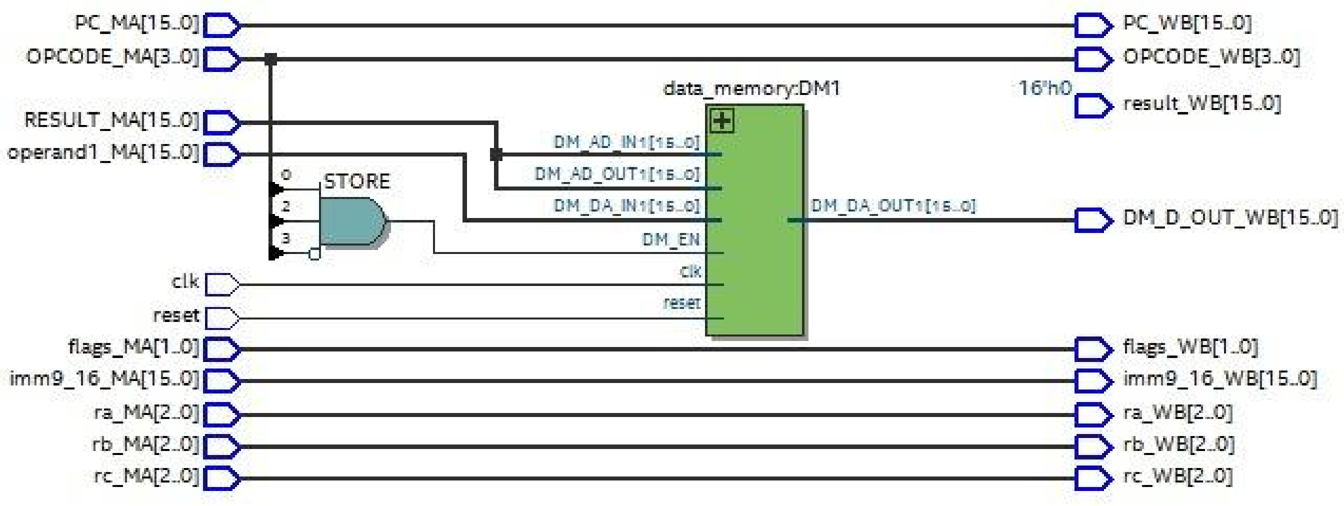


EX_MA Pipelining Register:

- This contains information like Branching data in case of a branch instruction, current instruction location in memory, immediate values to be used for memory access, result to be carried from the execution stage, register file addresses for the write back, values of flags and conditions for those flags and the operation code.

Memory Read:

- This Stage has the Data Memory in it. It is used to access the RAM of the device inorder for the program to run on the RISC processor.
- This stage could be prone to hazard if there is a conflict between the address to be written to and an execution instruction being manipulated at the same instance in the pipeline.



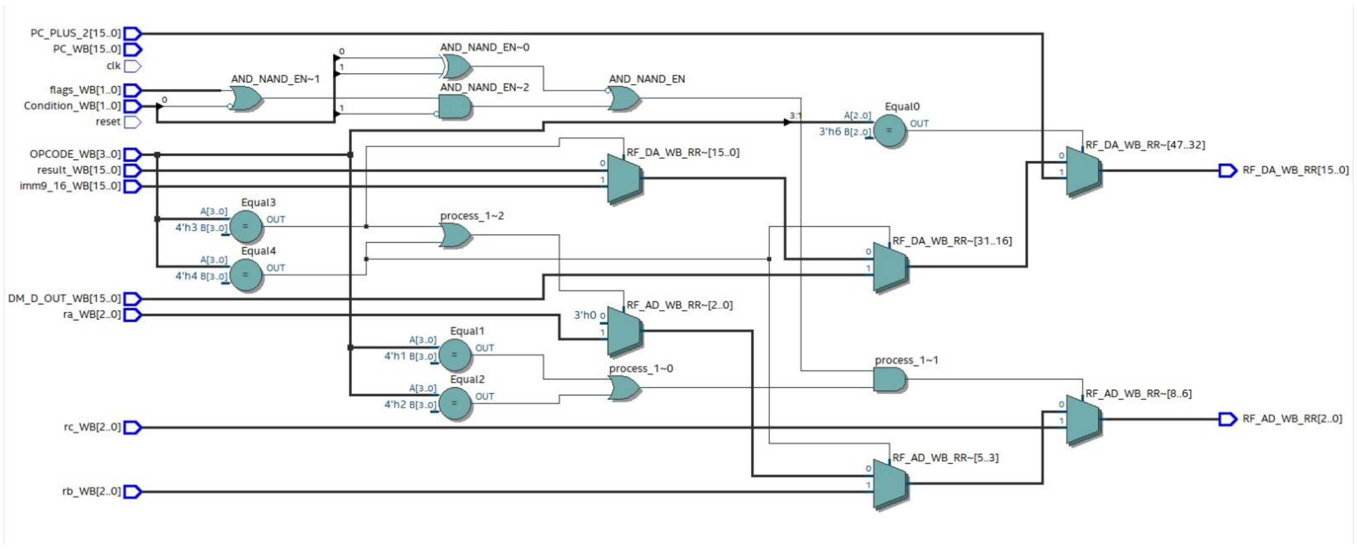
MA_WB Pipelining Register:

- This contains data like the result of the execution stage taken to be written back, the data memory that has to be written, the address where the write back should occur in the RF, the flags and the condition for those flags, the OPCODE, and the instruction location in instruction memory(IM).

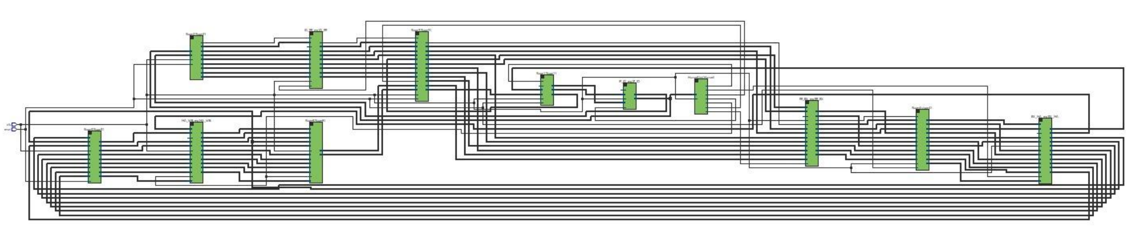
Write Back Stage:

- In layman terms, this is the stage required if any information is to be written onto the register file post computation.
- This stage consists of 2 multiplexers at the primary component.
- One of the multiplexers determines the write address in the register while the other determines the data that has to be written back.

- The first multiplexer multiplexes the addresses RA, RB, RC given in the ISA for the project.
- The Data is a multiplex between the Immediate, the ALU result, the next PC instruction and the data output.



A RTL View of the all Pipelining Registers:



* For demonstration purpose only

Hazard Control:

- Maximum hazards may occur up to 2 dependencies for the given ISA. For the LM SM instructions we have defined a special M_control_unit which will incrementally act on the rest of the architecture.
- Thus to construct this unit we require decoded instruction from 3 stages namely Instruction Decode(ID) Register Read(RR) and Execute(Ex).
- The output of this block should be the enable signal to the major data units of the CPU architecture namely the Data Memory and the Register Read.

- The logic inside this block must cover all possible hazard cases we might encounter and we shall attempt to solve this using behavioural nested logic statements available in VDHL.

To map all the possible cases of a discrepancy let us approach the problem with the following table:
1 - Stage is required in the table.
0 - Stage is not required in the table.

Instruction Type	Instruction Fetch(IF)	Instruction Decode(ID)	Register Read(RR)	Execute (Ex)	Memory Access(MA)	Write Back(WB)
Arithmetic/ Logical	1	1	1	1	0	1
Load	1	1	0	1	1	1
Store	1	1	1	1	1	0
Conditional Branch	1	1	1	1	0	0
Unconditional Branch	1	1	1	1	0	0
Jump and Link	1	1	1	1	1	1

Possible Hazard Cases:

- Add instruction has a dependency of 1 instruction in case we add a forwarding unit to forward ALU_C result from Ex to WB stage.
- Load Instructions have a dependency of 2 instructions in case the register value has not loaded the instruction back in the register read and the same is accessed back for an operation earlier than it has been written.
- Branch instructions, add challenges with control flow. In such a case instructions might have to be flushed out and the corresponding stages be stalled to form a process bubble in between.

Data Hazards and Stalling:

1. **Branching Stalls:**

Implementation of a Stall until the branch is determined. Let the instruction pass entirely through the pipeline independently and then when the new branch is determined run that instruction. This stalling is accomplished by the use of a counter.

How is the stall implemented?

- The stall is implemented using a bubble creation signal which disables the updation of the PC at the first stage. All other enable signals are controlled accordingly.
- We have used signals called BUBBLE_CTRL and BUBBLE_BRANCH to stall/resume the pipeline.

Examples:

1. EX_MA_register(destination component) = RR_EX_register(any one of the source component)
2. MA_WB_register(destination component) = RR_EX_register(any one of the source components)

Forwarding unit:

- Solution to the Hazarding case is provided by this block.
- It should essentially eliminate the dependencies in the code for the pipeline to achieve smoothly. This is the complete opposite of stalling the pipeline for a cycle inorder to avoid dependencies.

Solutions:

1. if(EX_MA_register(destination component) = RR_EX_register(any one of the source components) then Forwarding Mux Control => source component is forwarded from ALU EX computation directly.
   Note: destination and source components are detected from the OPCODE as their location in the instruction is not consistent.

2. if(EX_MA_register(destination component) = RR_EX_register(any one of the source components) then Forwarding Mux Control => source component is forwarded from ALU EX computation directly.

Let us start with the following approach, we try to identify different registers in the OPCODE as either Source or destination, this will help us identify the conflicts as and when they arrive. We shall also try implementing the above solutions in case these conflicts occur.

OPCODE	**Source(Rs)**	**Source2(Rt)**	**Destination**
0000	RA	—---	RB

0001,0010	RA	RB	RC
0011	—--	—--	RA
0100(Load)	RB	——---	RA
0101(Store)	RA	RB	—---
0110	TBD	TBD	TBD
0111	TBD	TBD	TBD
1000	RA	RB	—----
1001	RA	RB	——----
1010	RA	RB	
1100	—---	—---	RA
1101	RB	—---	RA
1111	RA	—----	——----

Now to resolve the conflicts we use the appropriate control signals on forwarding muxes to send the correct operand. This is how the data forwarding unit will function.

Idea to implement LM and SM:

- These instructions are perhaps the most difficult to implement due to their looping nature in case they are implemented sequentially.
- We shall use the lower 8 bits of the instruction in a shifting fashion to provide enable to the register file inorder to implement the instruction in a sound manner.
- For the chosen, stalling based architecture we require a large forward dependency to perform this instruction with the current building blocks we are using.

Implementation of LM instruction:

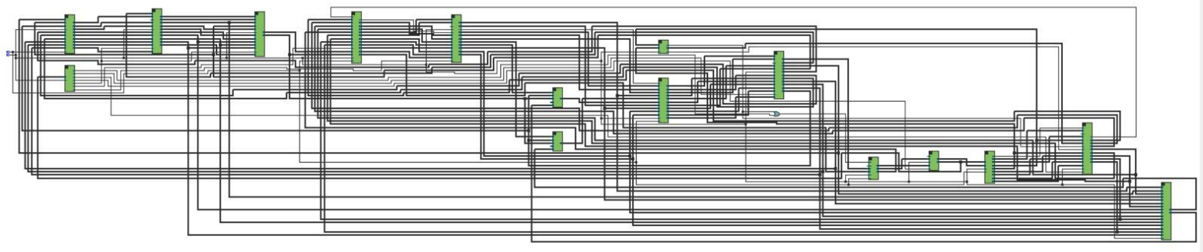
- We shall decode this instruction separately at the ID_RR pipelining register component. Generate one set of responses if this instruction OPCODE is seen.
- The trick here is to split one load instruction into multiple LW instructions. Keep register B as the base register and accordingly increment the immediate as per requirement to cater to all 8 stages.
- The activation bits of the register are stored in a register called LM_IMM for stage-wise use.

- The counter(named Multi-Flop) shall count 8 times to generate equivalent 8 LW instructions.

Implementation of LM instruction:

- The SM instruction is implemented in the same way. The same counter unit is used for this instruction too and the method of breaking 1 instruction to 8 SW instructions with increasing immediates is used.

A RTL view of the entire Pipelined RISC design:



* For demonstration purpose only