Introduction to Processor Architecture

5-stage Pipelined Y86-64 Processor

Team Members:

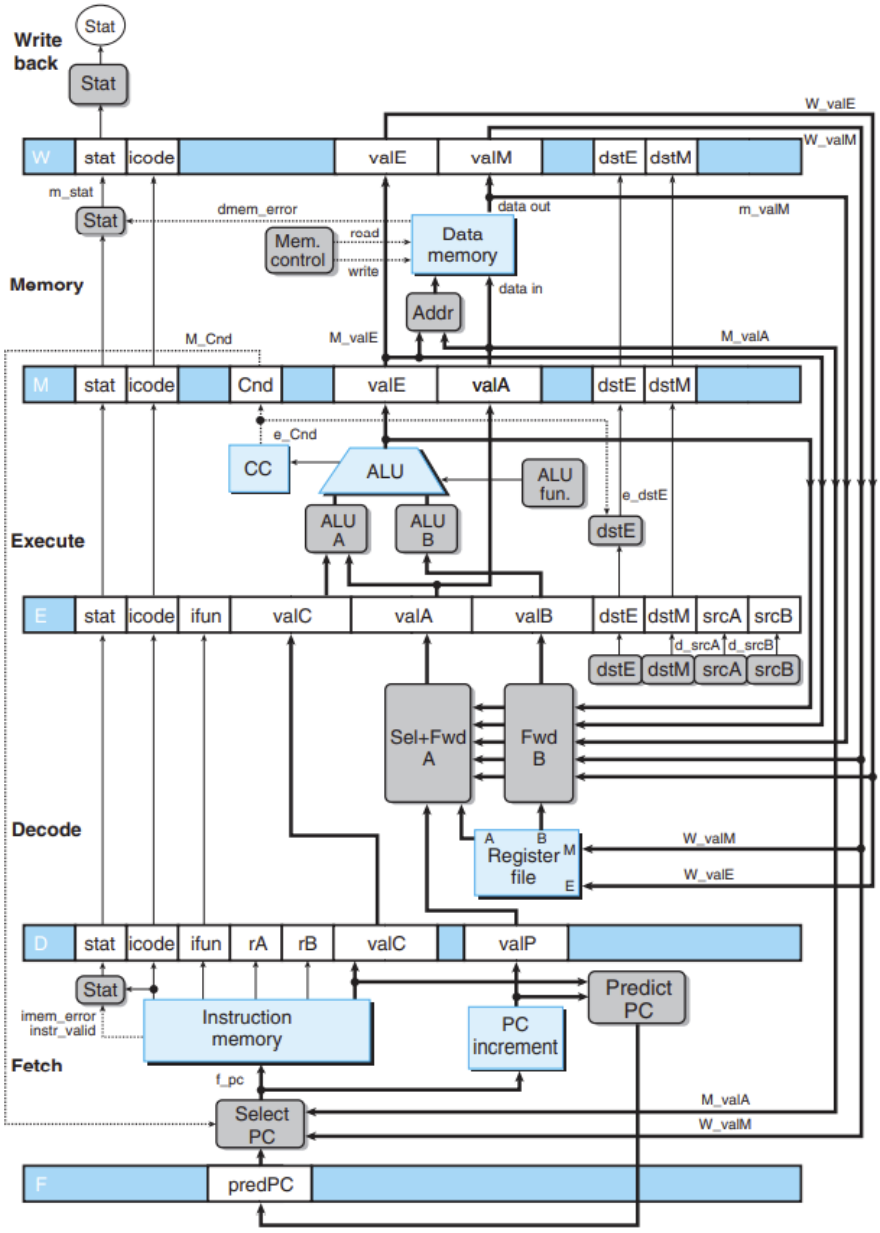
Atharva Sunil Gogate (2020112001)

Akshit Gureja (2020112004)

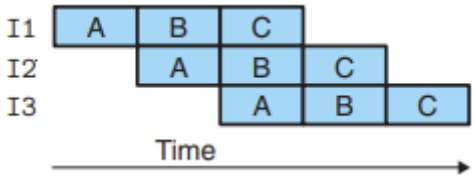
Content

Overview

The aim of the project is to implement a pipelined Y86-64 processor architecture with data forwarding and control logic in Verilog. The first part of the project involved making a sequential implementation of the Y86-64 architecture, whose details and report are included in the folder “sequential”.



As compared to its sequential counterpart (SEQ), a pipelined implementation (PIPE) of a processor allows for an increased throughput. With SEQ, an instruction is executed only after that which is preceding it has finished executing. In other words, the instructions follow a sequence, hence the name. On the other hand, with PIPE, an instruction is divided into a fixed number of stages and is implemented as follows: -



(I1, I2, I3 are the instructions; A, B, C are the stages into which each of the instructions is divided). If we assume that each stage executes in one clock cycle, the pipelined implementation allows us to get 3 instructions done with in 5 clock cycles. Comparatively, a sequential implementation would take 9 clock cycles for the same.

Stage Registers

In PIPE, each stage of an instruction has associated registers which store the appropriate values required by the stage to which they pertain before they are needed by that stage. They prevent signals from freely passing between stages in order to prevent data hazards. The signals are allowed to pass only on the positive edge of the clock.

Fetch (F): F is inserted before the fetch stage and holds the predicted PC value to be fed to the fetch stage.

Decode (D): D is inserted between the fetch and decode stages and holds required information from the recently fetched instruction.

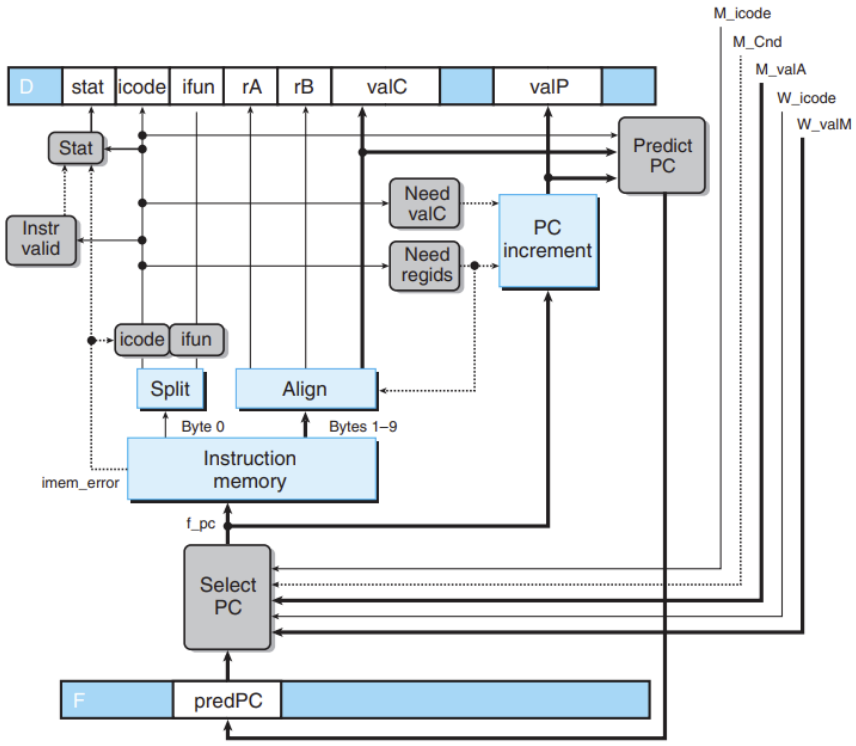
Execute (E): E is inserted between the decode and execute stages and holds required information from the recently decoded instruction.

Memory (M): M is inserted between the execute and memory stages and holds required information from the recently executed instruction.

Write back (W): W is inserted after the write back stage and has paths for writing to the required register(s) after an instruction is fully executed and also has provisions to provide the return address to fetch in case of the ret instruction.

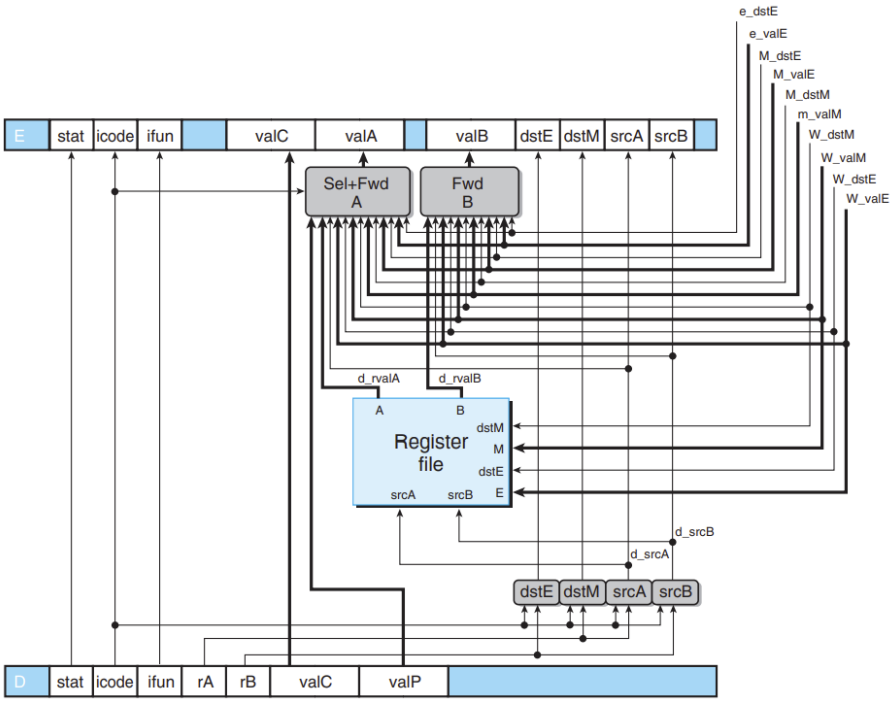
Module Design

Fetch



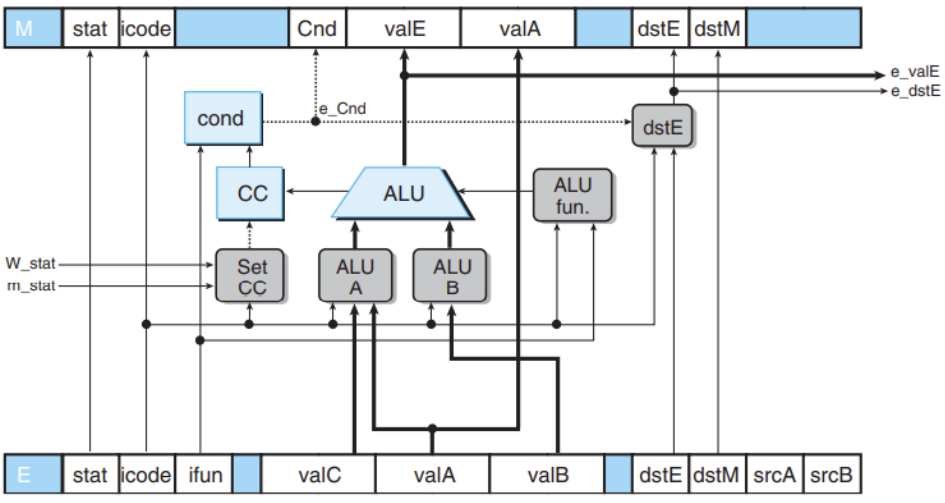
In the fetch stage, the predicted PC from F is sent to the select PC block on the positive edge of the clock, which also has several inputs from the later stages (of an earlier instruction) to correctly calculate the correct PC value for an instruction to be fetched. If the predicted PC value is correct, it is passed to the fetch stage, or it is calculated from these later inputs. The required instruction is fetched and the appropriate values needed by the decode stage are sent to be stored in D.

Decode



In the decode stage, the instruction is decoded and the required information is sent from D to E. This stage is the one where data forwarding is implemented, which often helps with prevention of loss of data. Since Verilog doesn't allow us to pass 2-dimensional arrays through function calls, all registers are sent separately.

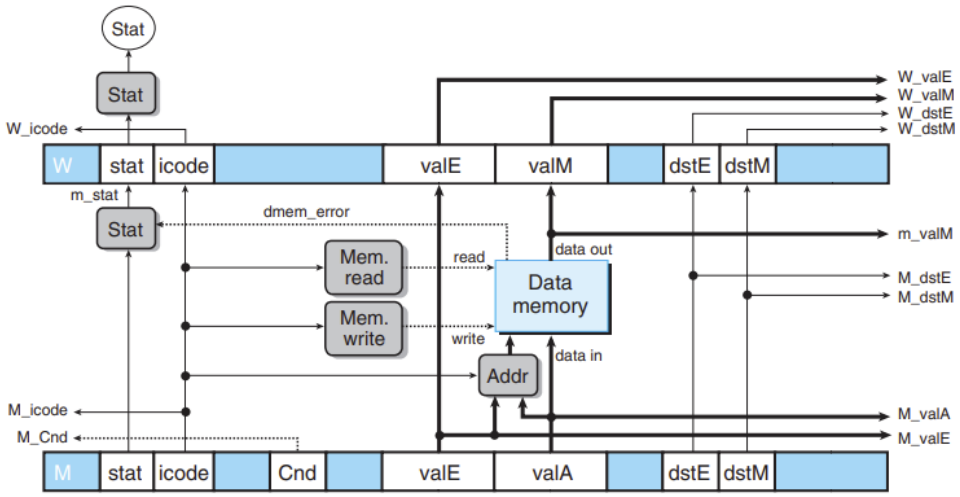
Execute



The execute stage contains the ALU. The instruction that was decoded is executed as required. The appropriate information is sent from E to M. The

condition codes are set, which lets us know whether to update them or not and condition is forwarded to M accordingly. The implementation, for the larger part, is very similar to that in SEQ.

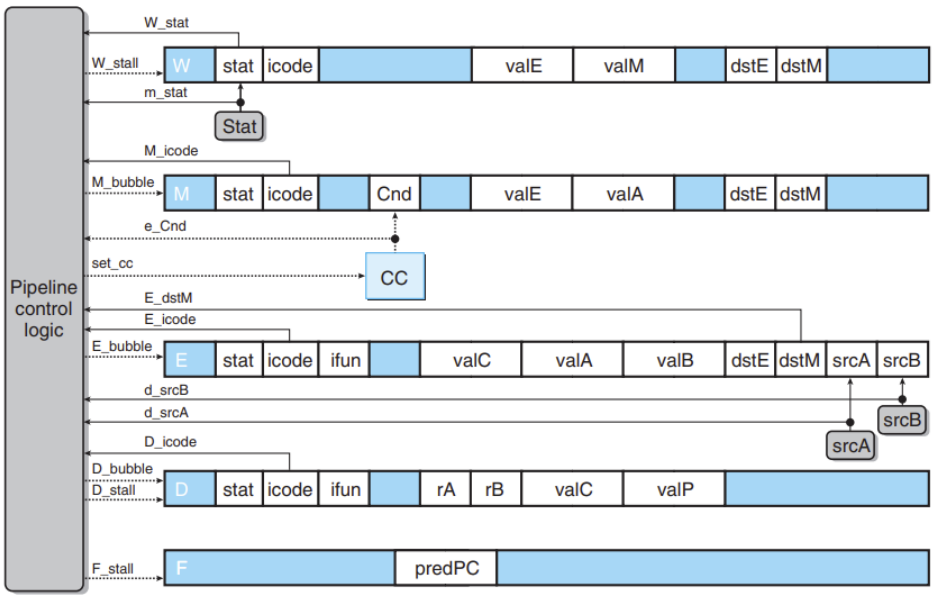
Memory and Write back



The memory stage is where the data is read from or written to the memory. The appropriate information is sent from M to W. The memory is declared separate from the instruction memory and does not see use in the other stages as it is accessed only in this stage. A striking feature of this stage is the large number of signals that are sent to the earlier instructions (for potential data problems of later instructions to take care of) from M, the stage itself and W.

In the write back stage, we take the outputs of the write back register(s) to write up to 2 results into the register file(s). W_dstE, W_dstM represent the registers to be potentially written into and the W_valE, W_valM are the corresponding results to be written into the registers.

Control Logic

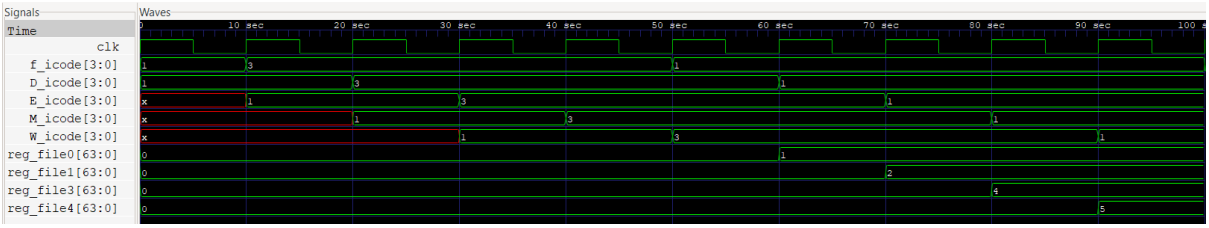


There are certain control cases which cannot completely be handled by data forwarding and branch prediction. These cases are listed below: -

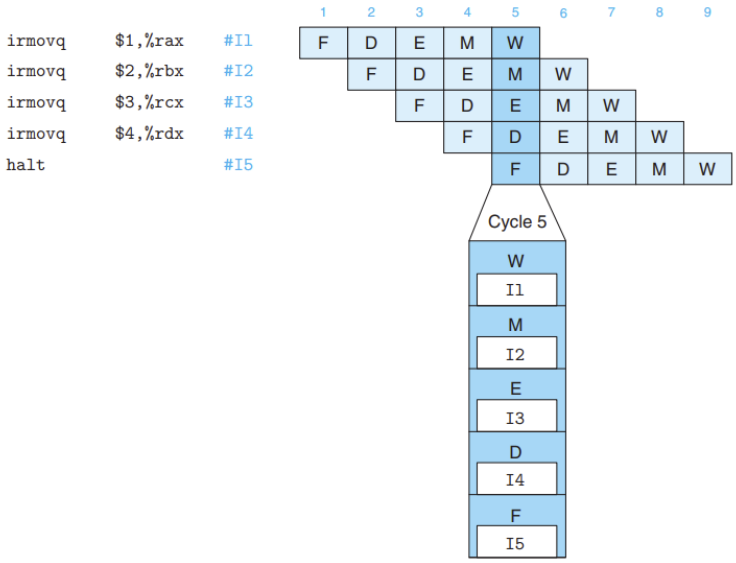
1. Load/use hazards: For two consecutive instructions where the first reads a value from memory and the second uses that value requires the pipeline to stall for a single cycle.
2. Processing ret: While the ret instruction is being run, the pipeline must be stalled until ret reaches write back.
3. Mispredicted branches: If a jump that was not supposed to happen occurs, the pipeline must cancel all the instructions that have already entered the pipeline and fetch the instruction just after the jump instruction.
4. Exceptions

GTKwave Results

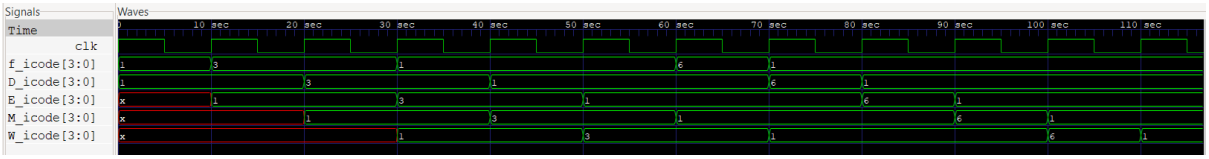
Program 0



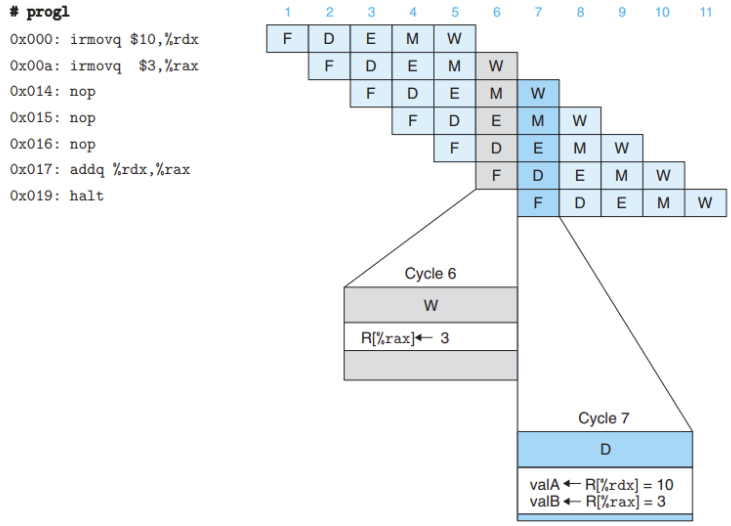
Program 0 displays the basic functioning of the pipeline.



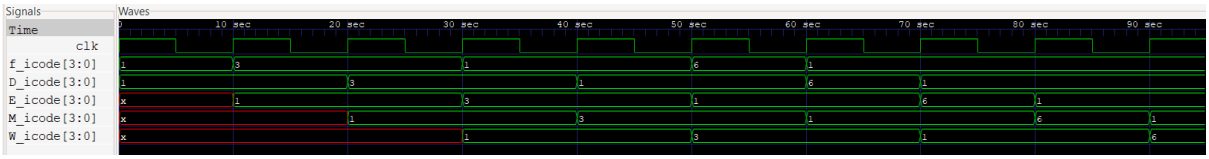
Program 1



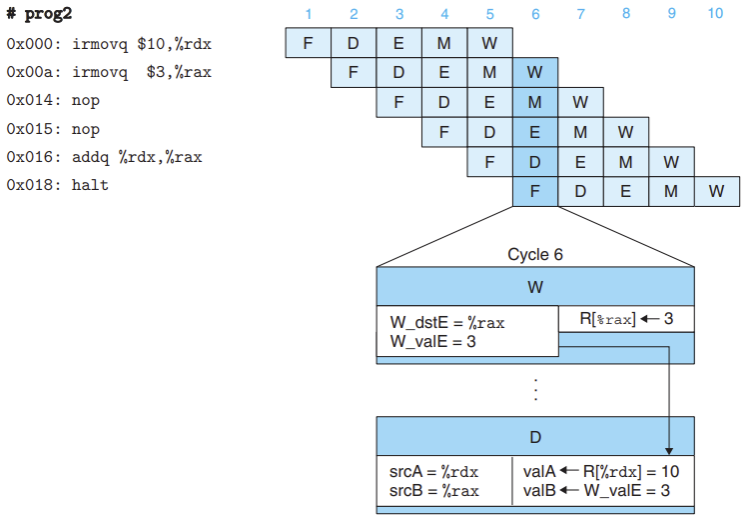
Program 1 displays the pipeline's functioning without special pipeline control. The use of 3 nop's negates any need for adding bubbles.



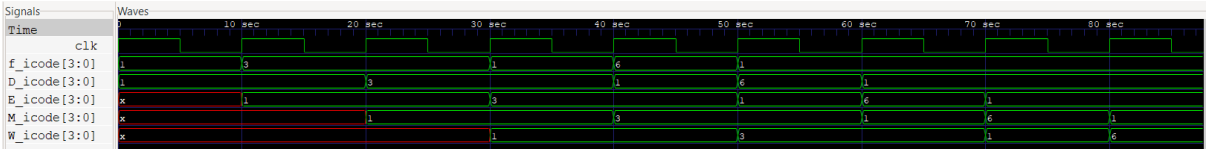
Program 2



Program 2 is Program 1 with 2 nop's instead of 3. Now, data forwarding takes care of this issue by detecting that in cycle 6, a write to register %rax is still not done in write back. Therefore, W_dstE is used for valB than the value in %rax itself.

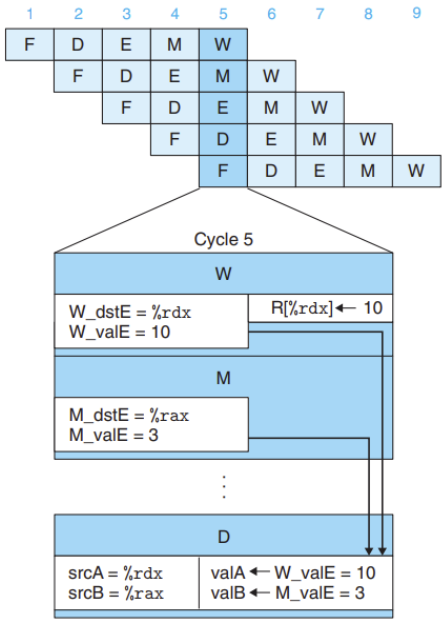


Program 3

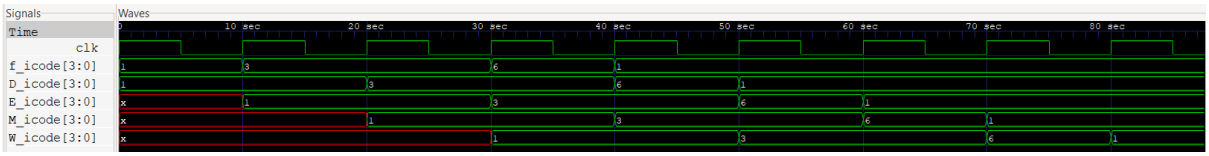


Program 3 is Program 1 but with just a single nop instead of 3. Again, data forwarding is implemented so that valA is written to from W_valE instead of from the value to be written to %rdx itself in write back; and valB is written into from M_valE instead of from the value to be written into %rax itself in memory. Attempting the later would have resulted in data hazards, which data forwarding is able to take care of successfully.

```
# prog3
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x005: addq %rdx,%rax
0x017: halt
```

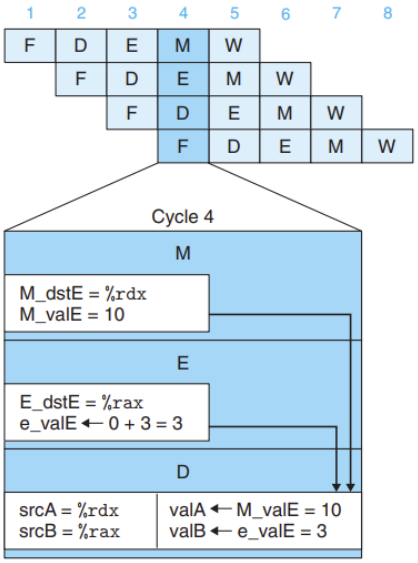


Program 4



Program 4 is Program 1 with zero nop's. Data forwarding is implemented such that valA is written into from M_valE rather than the value to be written to %rdx itself; and valB is written into from e_valE rather than the value to be computed for %rax itself. Doing the latter for both valA and valB would have resulted in data hazards but data forwarding, as explained, is able to take care of the issues.

```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



Challenges Encountered

1. Figuring out where to implement always@(*) and where to implement always@(posedge clk).
2. Differentiating between the various PC signals in the fetch stage proved to be confusing.
3. Because the pipelined implementation more than doubles the number of wires, keeping track of all of them in the various modules is difficult.
4. Keeping decode and write back in the pipelined implementation in separate files for clarity's sake proved to be difficult.