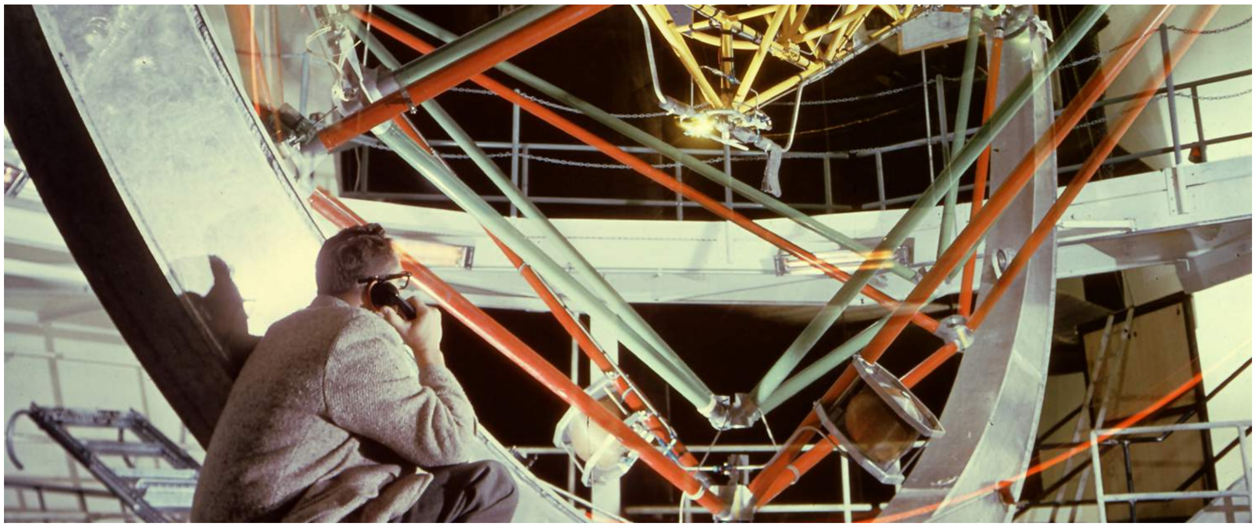


IPA Final Project Report

Autrio Das

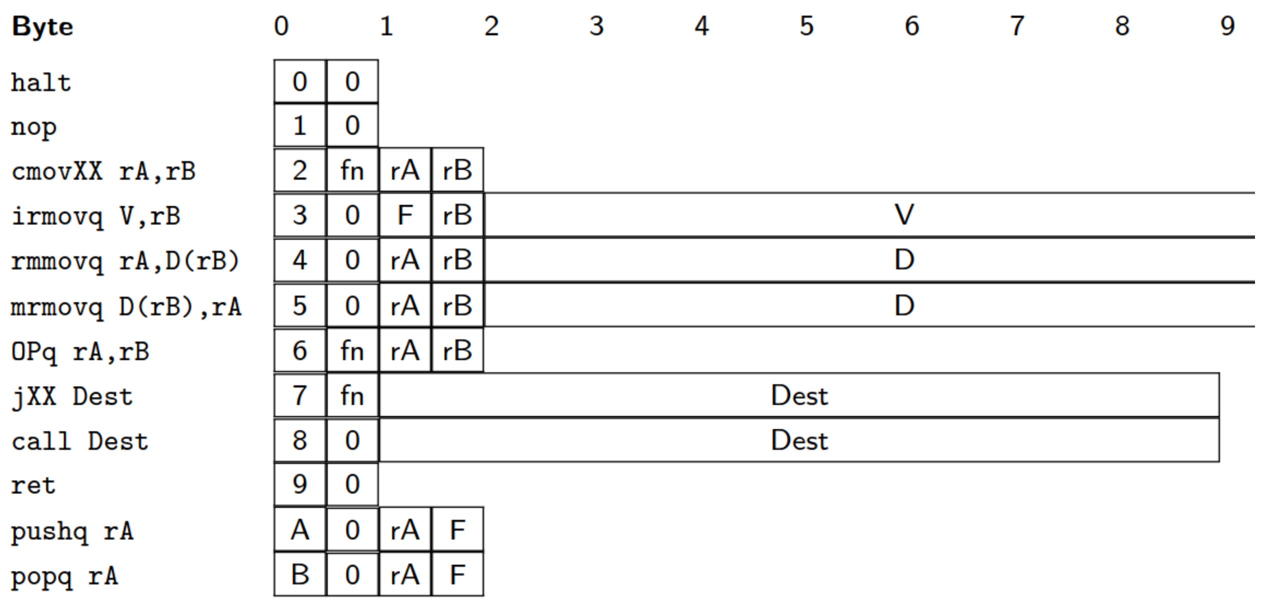
2022102017

Mohammed Ihsan Ali

2022102017

Sequential Implementation

We implement the Y86-64 Processor Model using Verilog HDL.This involves executing one instruction at a time, with each instruction going through a series of stages such as fetch, decode, execute, memory access, and write back. We are making a very simplified model of a processor, for ease of access.



Byte	0		1		2	3	4	5	6	7	8	9
`halt`	0	0										
`nop`	1	0										
`cmovXX rA,rB`	2	fn	rA	rB								
`irmovq V,rB`	3	0	F	rB	V							
`rmmovq rA,D(rB)`	4	0	rA	rB	D							
`mrmovq D(rB),rA`	5	0	rA	rB	D							
`OPq rA,rB`	6	fn	rA	rB								
`jXX Dest`	7	fn	Dest									
`call Dest`	8	0	Dest									
`ret`	9	0										
`pushq rA`	A	0	rA	F								
`popq rA`	B	0	rA	F								

Instructions that we will implement

There are five stages we will implement:- Fetch,Decode, Execute, Memory and Write Back

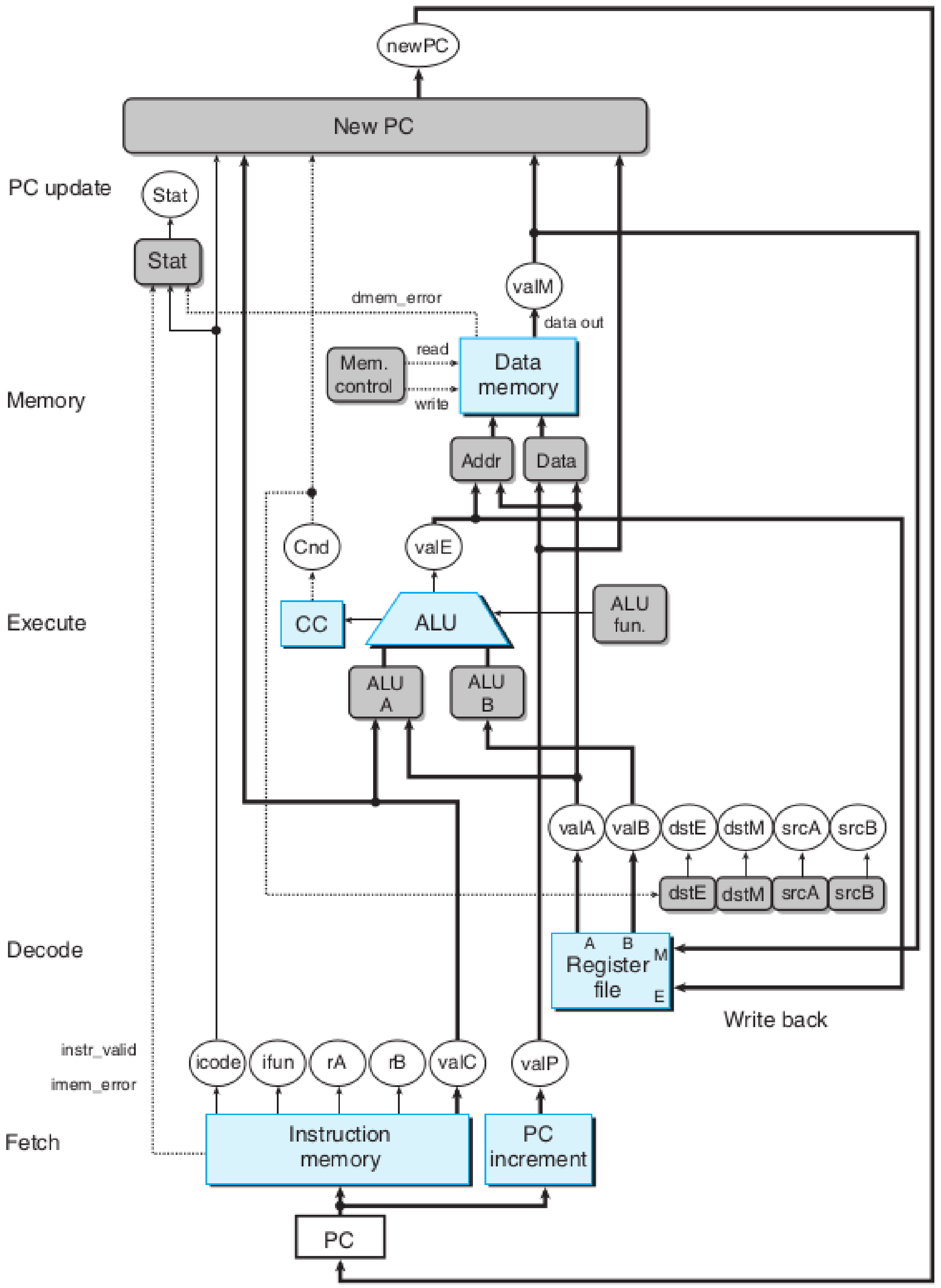
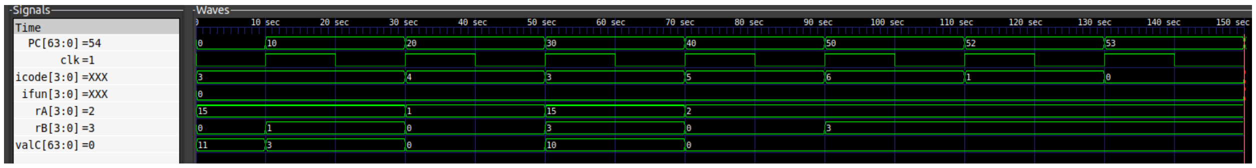


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Fetch Stage

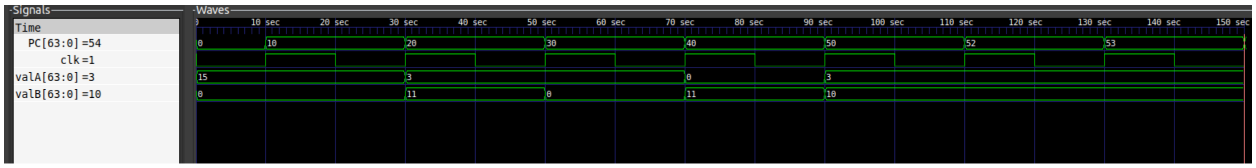
In the Fetch stage of the Y86-64 processor model, the processor retrieves the next instruction from memory using the Program Counter (PC). The PC holds the memory address of the next instruction to be executed.

We get the instruction code, function, register values, and other necessary variables in this stage



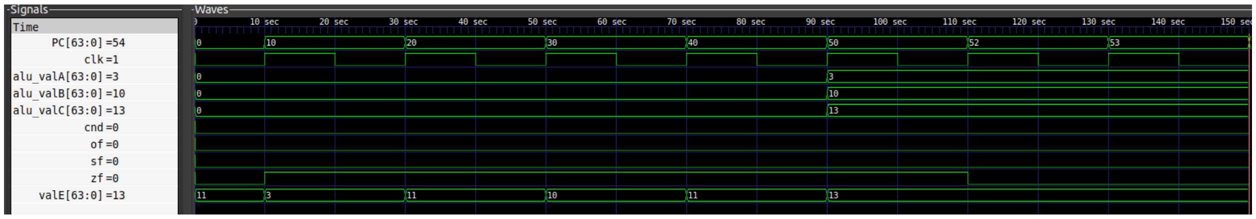
Decode Stage

In the Decode stage of the Y86-64 processor model, the fetched instruction is decoded to determine the type of operation, operands, and addressing modes. This is where the CPU understands what the instruction is supposed to do, and prepares for the next stages accordingly. It uses the outputs from the fetch stage to do this.



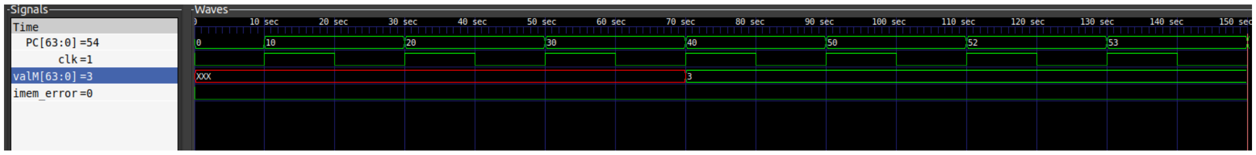
Execution Stage

In the Execution stage of the Y86-64 processor model, the operation specified by the instruction is performed. This could involve arithmetic operations, logical operations, or moving data from one register to another. The results of these operations are then stored for use in later stages. We also get the required addresses and values in this stage.



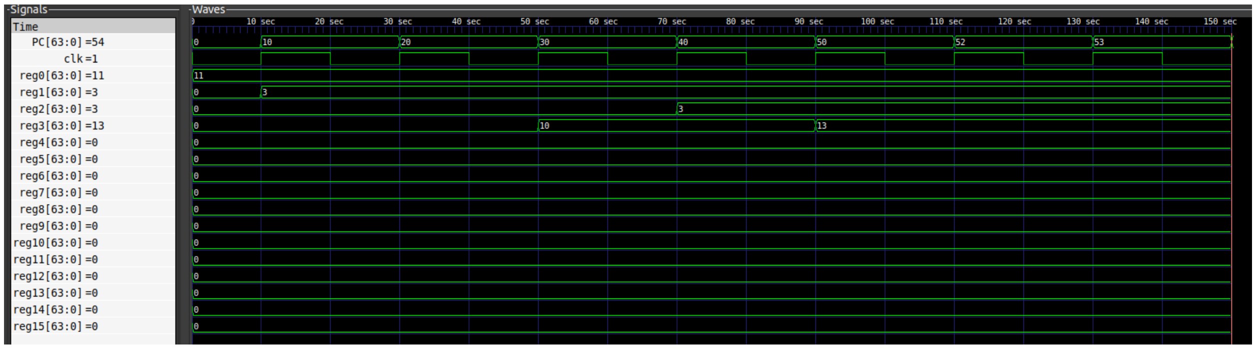
Memory Stage

In the Memory stage of the Y86-64 processor model, if the instruction requires it, data is either read from or written to memory. This could be for reading,writing into memory, updating stack pointer etc.



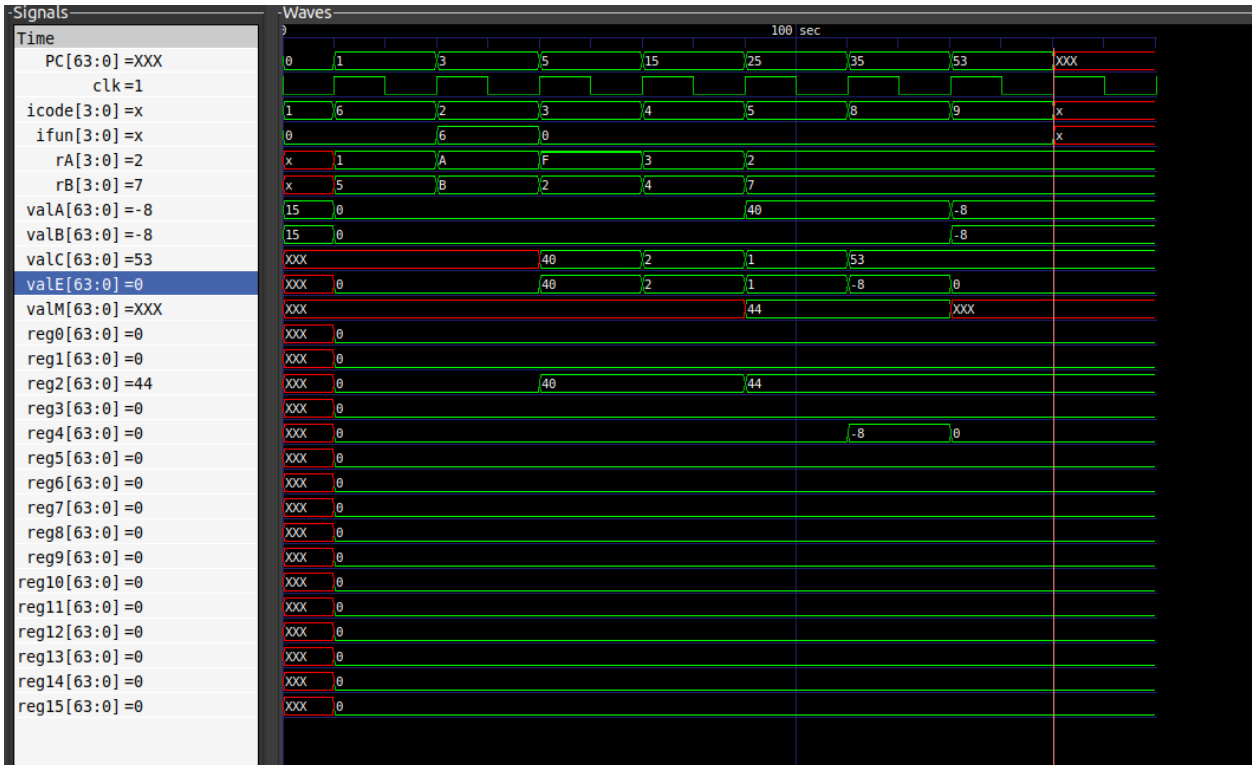
Writeback Stage

In the Writeback stage of the Y86-64 processor model, the results of the operation are written back to the processor's registers. This is the final stage of the instruction cycle, completing the execution of the instruction.

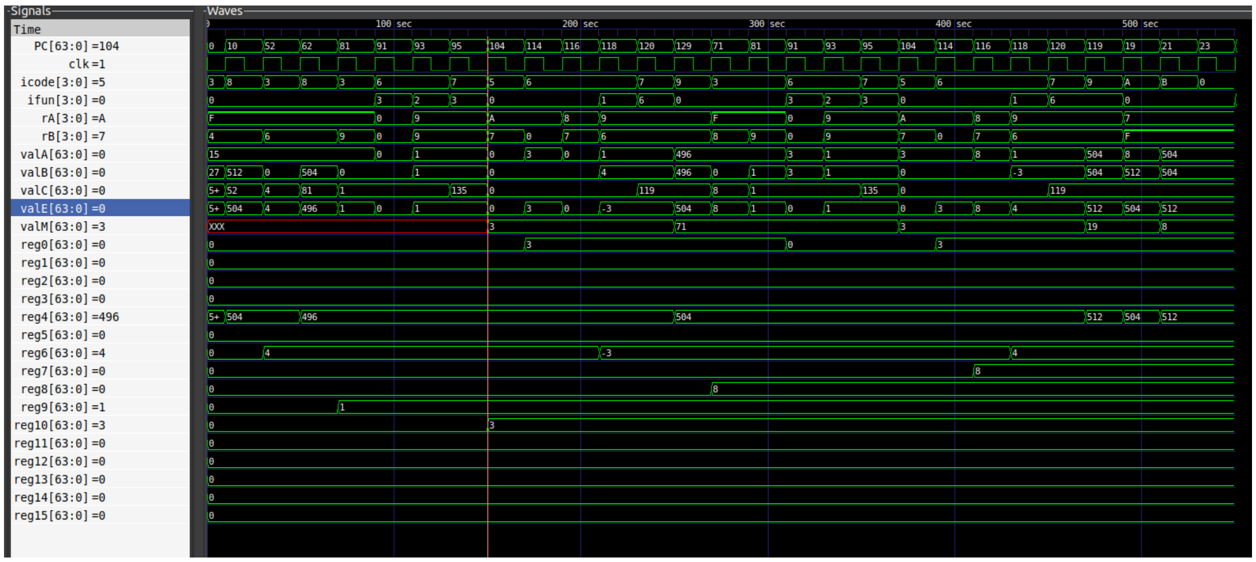


There is also a PC update stage which updates the program counter, which the fetch stage then uses to choose the next instruction.

We now run a sample testcase using the processor.



We see the functionalities of the majority of the instructions in the processor in this testcase.



We use another testcase which has majority of the functionalities here.

Difficulties Faced

→ Syncing up the timing for all the stages

→ Finding ways to store registers/ instruction memory/ data memory

→Using stack pointer/data memory interchangeably

→ Program takes too long to load

Pipelined Implementation

A pipelined implementation for the Y86-64 architecture divides the processor into distinct stages, each handling a specific part of executing instructions. This setup enables different stages to work simultaneously, improving overall performance and instruction throughput.. We optimise a lot in this kind of implementation, which leads to a lot of new modules.

There is a lot of advantages of pipeline compared to sequential including increased throughput, better utilization of resources and efficiency.

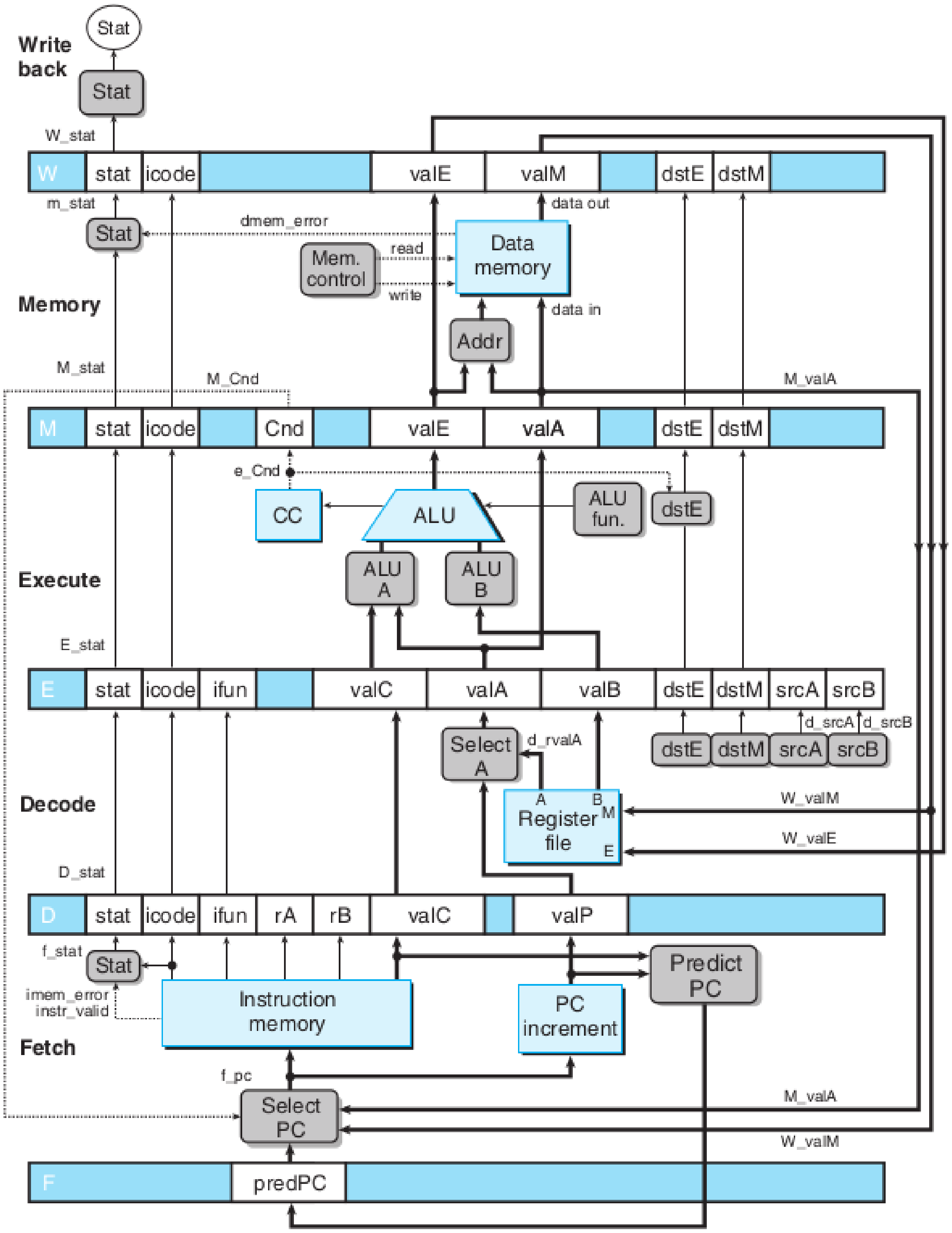


Figure 4.41 **Hardware structure of PIPE−, an initial pipelined implementation.** By inserting pipeline registers into SEQ+ (Figure 4.40), we create a five-stage pipeline. There are several shortcomings of this version that we will deal with shortly.

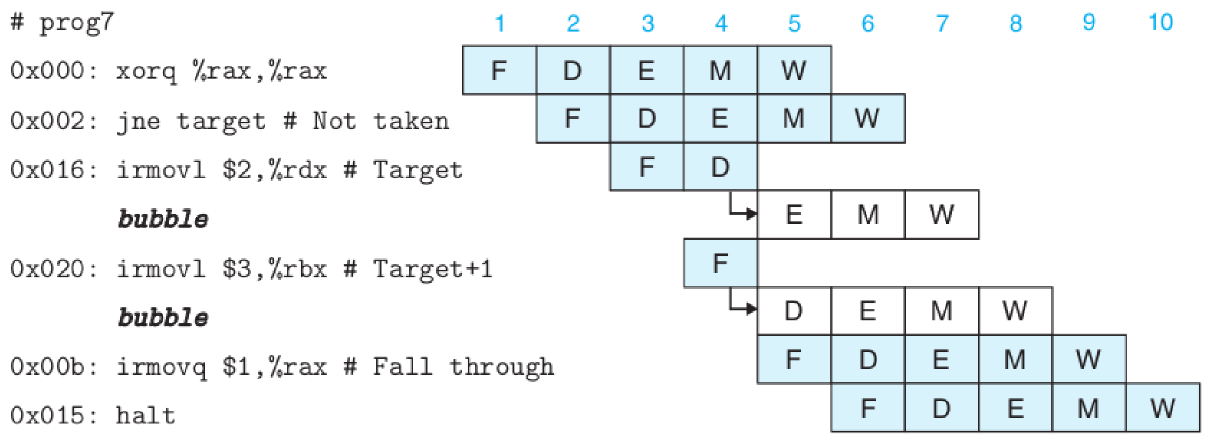
In pipe lined implementation we have to predict which instruction to do next. i.e we have predict pc and select pc. This block chooses the address depending on whether it is a return instruction, jump/call instruction and conditional jumps, and everything else.

We also have a control block which sets the conditions and decides on how to do several tasks, such as data forwarding, misdirected jump handling.

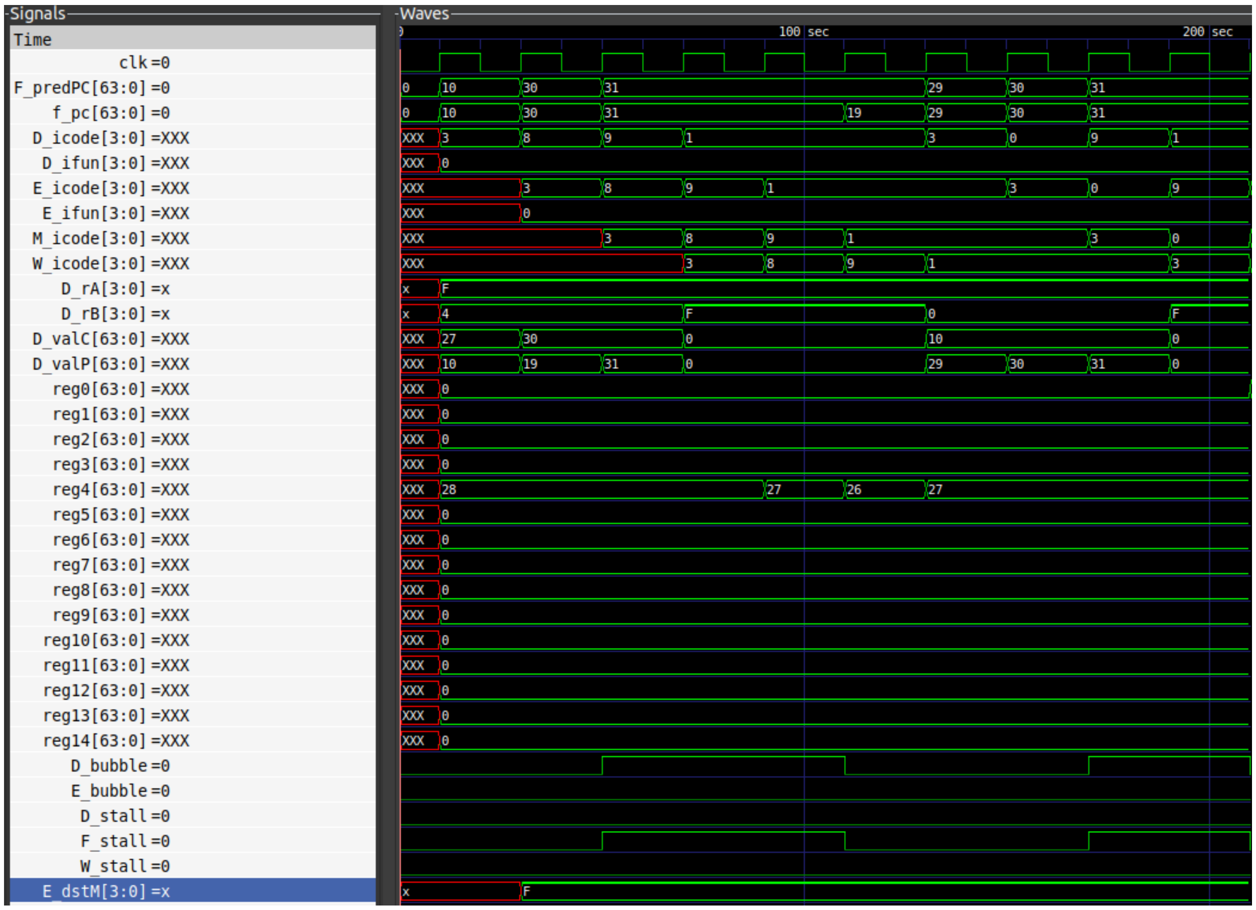
We now run a testcase to see how the pipelined implementation works. We also showcase different features and issues with pipelining and how we tackle them,

Return Instruction

We need to make sure that after the return instruction, we dont let any instruction after it get processed, hence we stall everything else until the return instruction is processed.



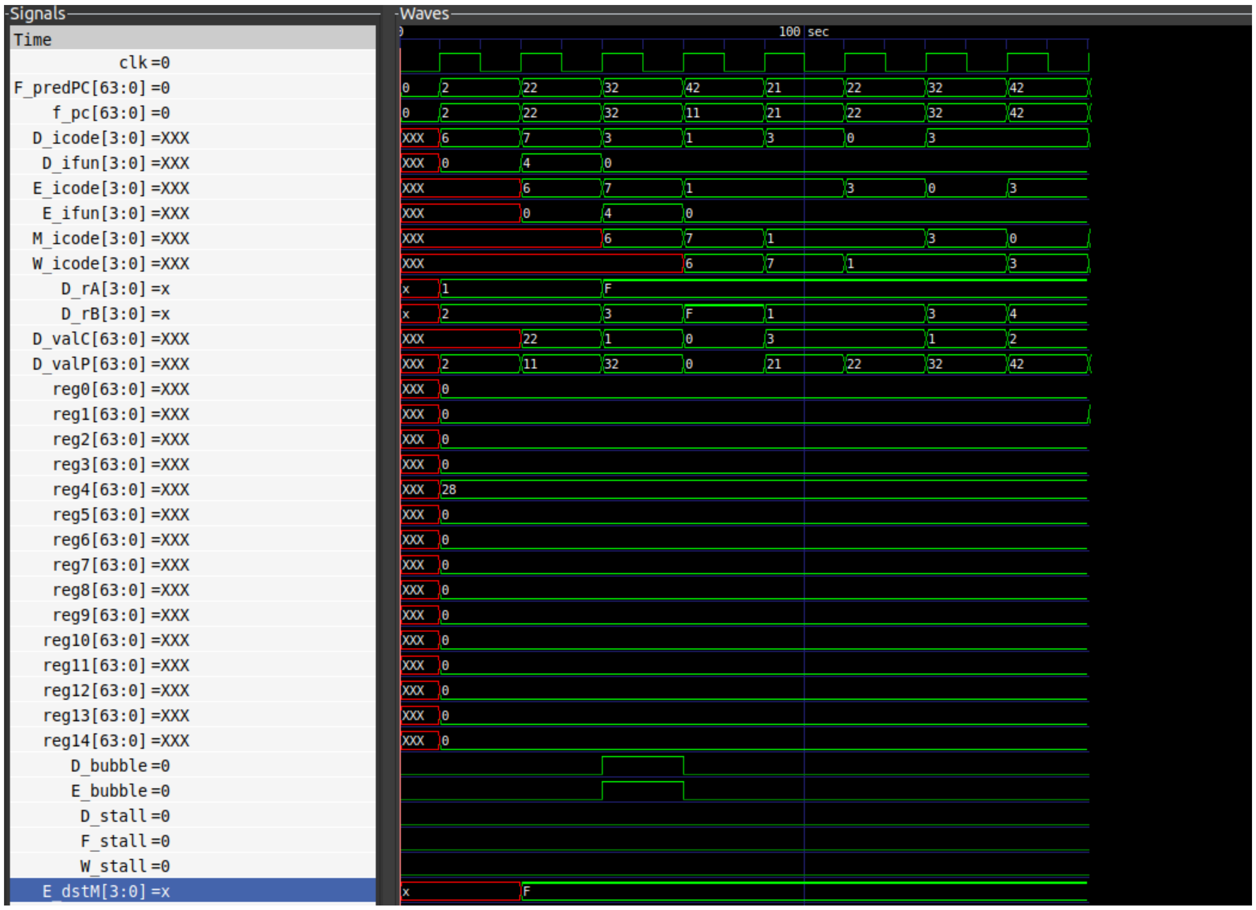
```
# prog7
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovl $2,%rdx # Target
       bubble
0x020: irmovl $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



We see that after every return instruction, we have a bubble (nop condition) that ensures that the return instruction finishes before starting the next instruction.

Mispredicted Branch

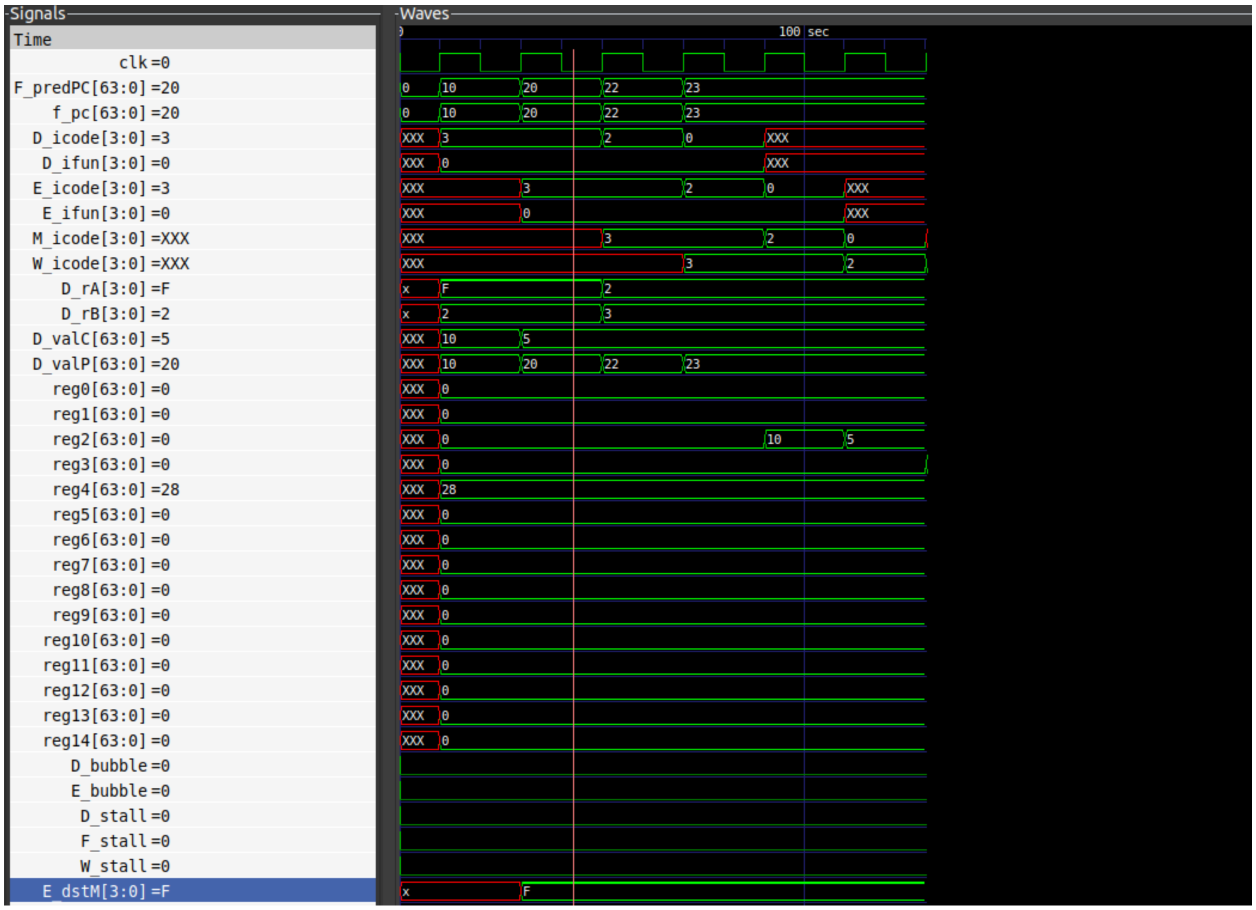
We take a jump regardless of whether the condition is fulfilled in our implementation. If we later find that our jump was wrong, we need to return to the PC after the jump.



Here we see that we take a wrong jump in the second clock cycle, after which we go back during the execute stage of the jump (after we see whether the conditions are right or wrong).

Forwarding instead of Stalling

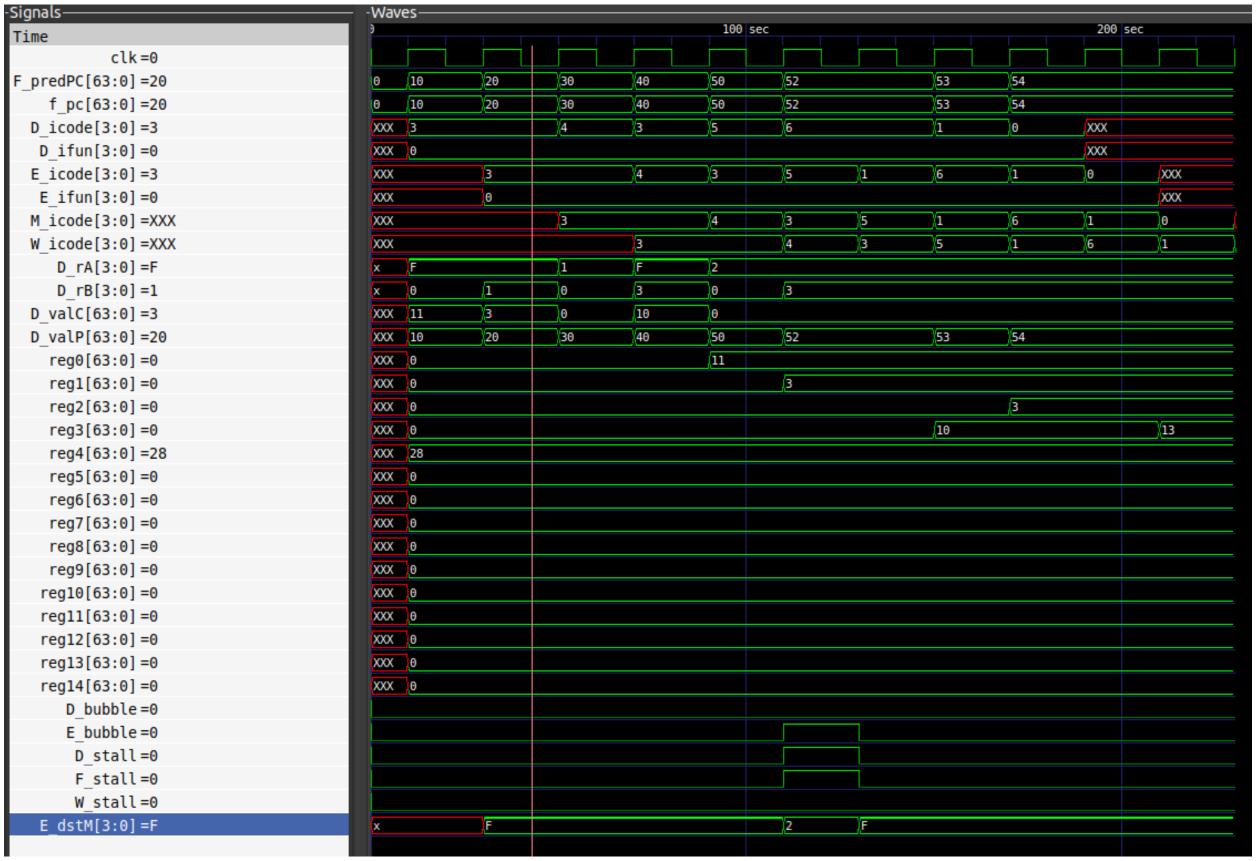
We use forwarding to optimize our instruction flow, instead of stalling and wasting clock cycles, we forward data across instructions. This is possible when two instructions use the same register/memory location



```
Time
clk=0
F_predPC[63:0]=20
f_pc[63:0]=20
D_icode[3:0]=3
D_ifun[3:0]=0
E_icode[3:0]=3
E_ifun[3:0]=0
M_icode[3:0]=XXX
W_icode[3:0]=XXX
D_rA[3:0]=F
D_rB[3:0]=2
D_valC[63:0]=5
D_valP[63:0]=20
reg0[63:0]=0
reg1[63:0]=0
reg2[63:0]=0
reg3[63:0]=0
reg4[63:0]=28
reg5[63:0]=0
reg6[63:0]=0
reg7[63:0]=0
reg8[63:0]=0
reg9[63:0]=0
reg10[63:0]=0
reg11[63:0]=0
reg12[63:0]=0
reg13[63:0]=0
reg14[63:0]=0
D_bubble=0
E_bubble=0
D_stall=0
F_stall=0
W_stall=0
E_dstM[3:0]=F
```

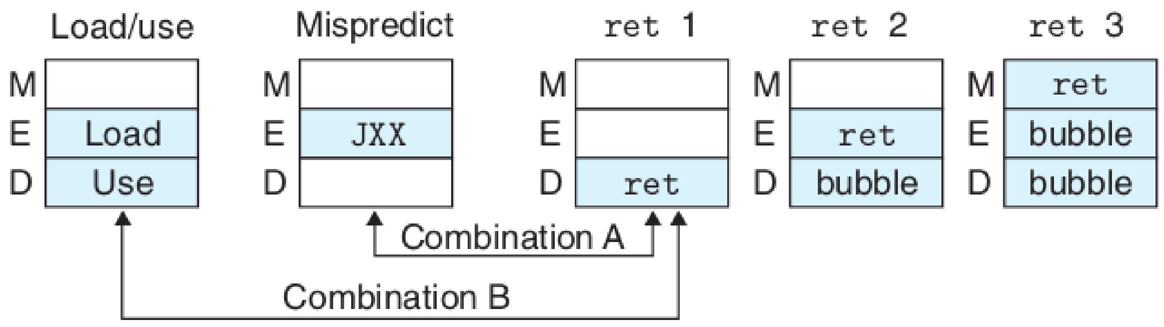
Load/Use Hazard

A load-use hazard, also known as a data hazard, occurs in pipelined processors when there is a dependency between a load instruction (which loads data from memory) and a subsequent instruction that uses the loaded data.

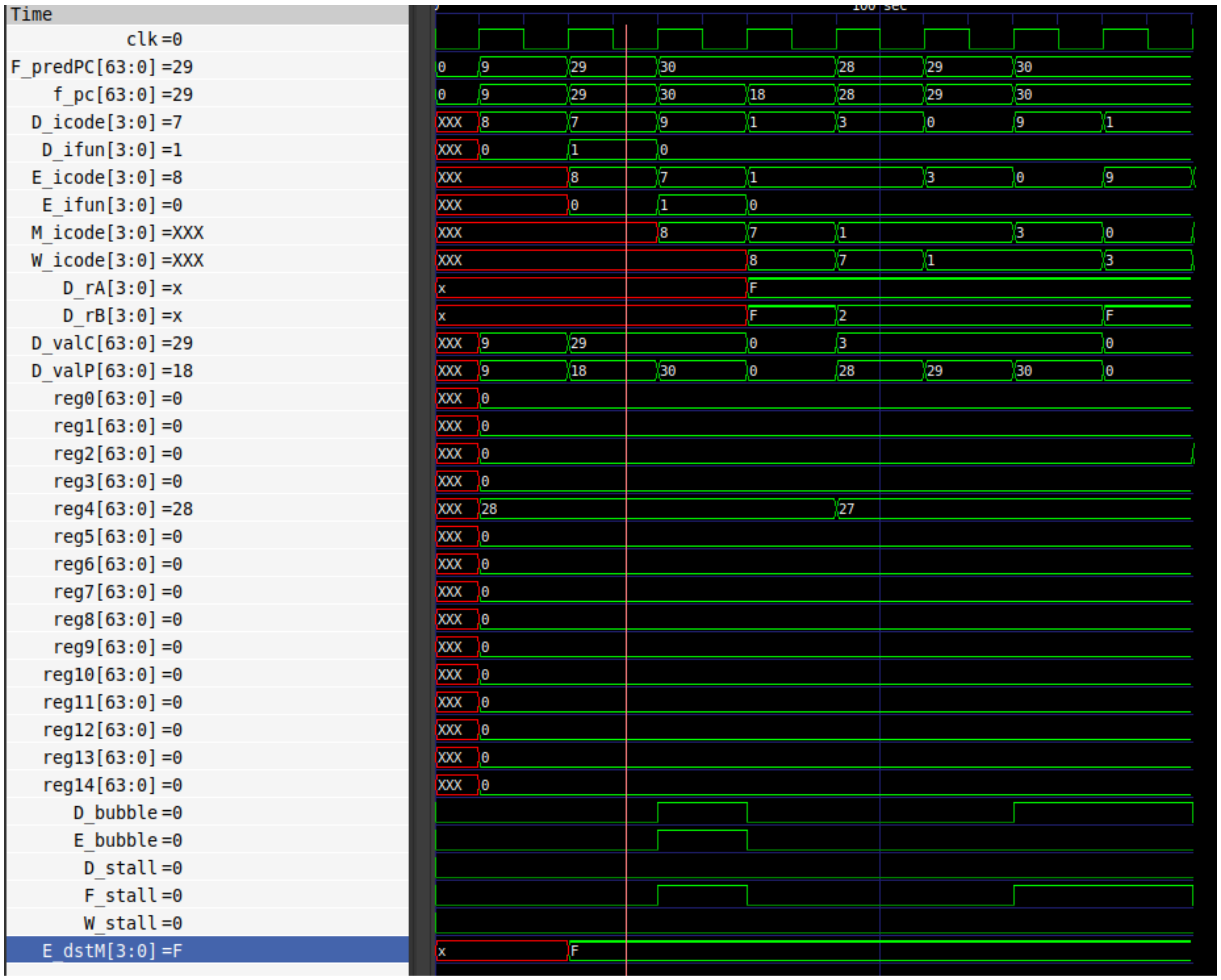

```
Time
clk=0
F_predPC[63:0]=20
f_pc[63:0]=20
D_icode[3:0]=3
D_ifun[3:0]=0
E_icode[3:0]=3
E_ifun[3:0]=0
M_icode[3:0]=XXX
W_icode[3:0]=XXX
D_rA[3:0]=F
D_rB[3:0]=1
D_valC[63:0]=3
D_valP[63:0]=20
reg0[63:0]=0
reg1[63:0]=0
reg2[63:0]=0
reg3[63:0]=0
reg4[63:0]=28
reg5[63:0]=0
reg6[63:0]=0
reg7[63:0]=0
reg8[63:0]=0
reg9[63:0]=0
reg10[63:0]=0
reg11[63:0]=0
reg12[63:0]=0
reg13[63:0]=0
reg14[63:0]=0
D_bubble=0
E_bubble=0
D_stall=0
F_stall=0
W_stall=0
E_dstM[3:0]=F
```

We solve a data hazard by using both forwarding and stalling.

We now look at combinations of data hazards, there are two such types of combinations

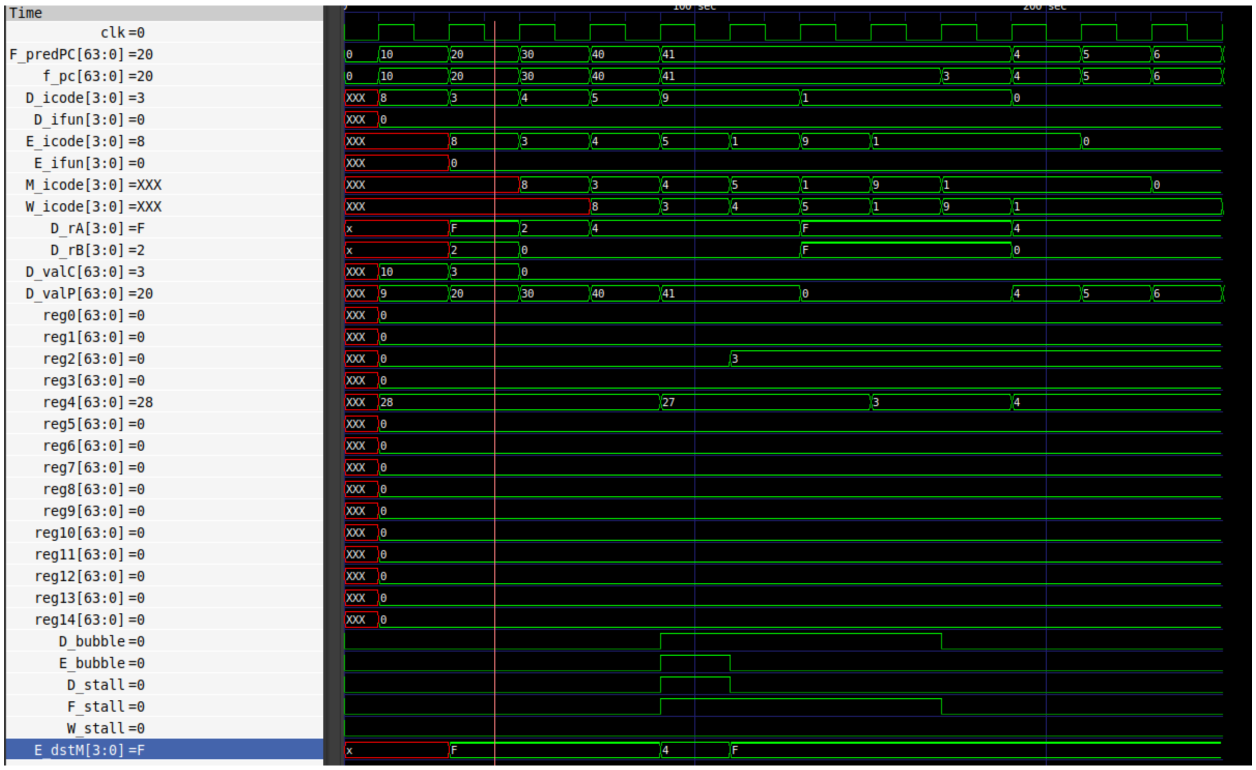


We first look at combination A.

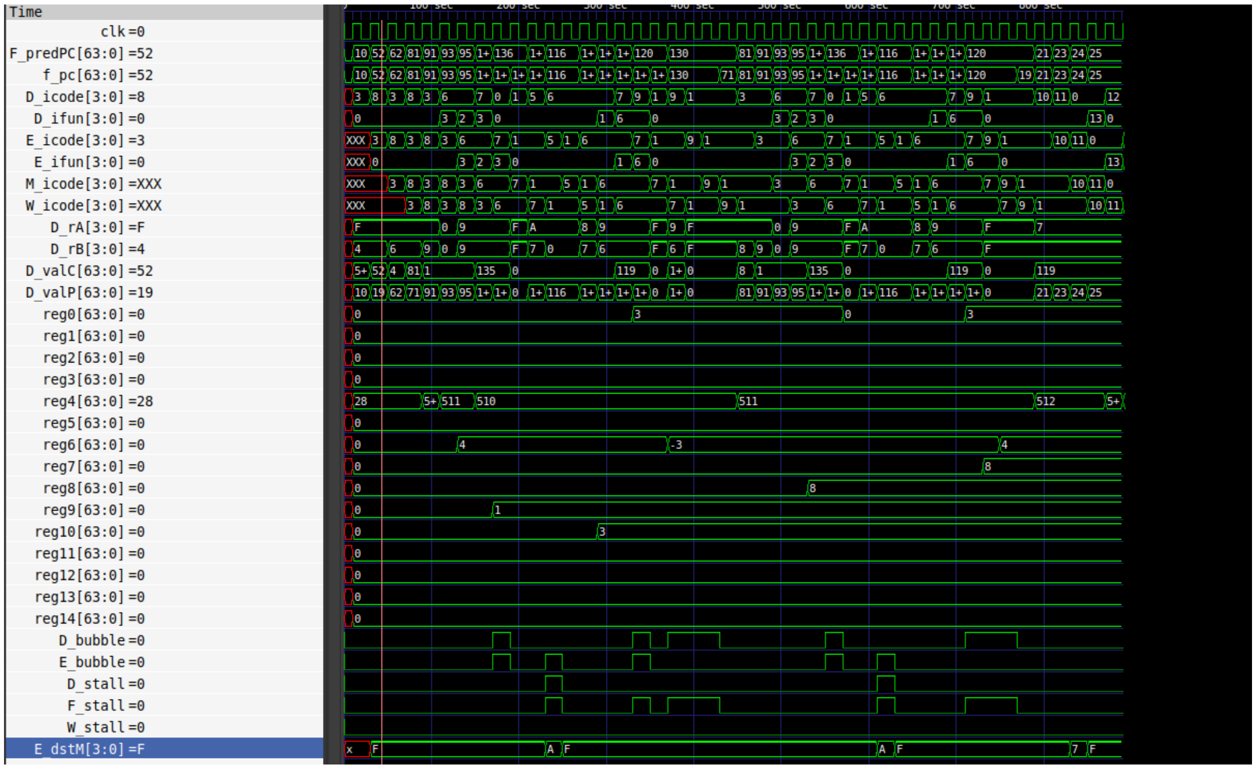


We do not have to make any further conditions for this

For combination B however, we need to write an extra condition.


```
Time
           clk=0
F_predPC[63:0]=20
    f_pc[63:0]=20
  D_icode[3:0]=3
   D_ifun[3:0]=0
  E_icode[3:0]=8
   E_ifun[3:0]=0
  M_icode[3:0]=XXX
  W_icode[3:0]=XXX
     D_rA[3:0]=F
     D_rB[3:0]=2
  D_valC[63:0]=3
  D_valP[63:0]=20
    reg0[63:0]=0
    reg1[63:0]=0
    reg2[63:0]=0
    reg3[63:0]=0
    reg4[63:0]=28
    reg5[63:0]=0
    reg6[63:0]=0
    reg7[63:0]=0
    reg8[63:0]=0
    reg9[63:0]=0
   reg10[63:0]=0
   reg11[63:0]=0
   reg12[63:0]=0
   reg13[63:0]=0
   reg14[63:0]=0
      D_bubble=0
      E_bubble=0
       D_stall=0
       F_stall=0
       W_stall=0
   E_dstM[3:0]=F
```

We now look at a testcase which has all required instructions.



Difficulties faced

→ Integrating pipeline registers between each stage

→ Taking care of program counter conditions and cases

→ Implementing bubbles and forwarding