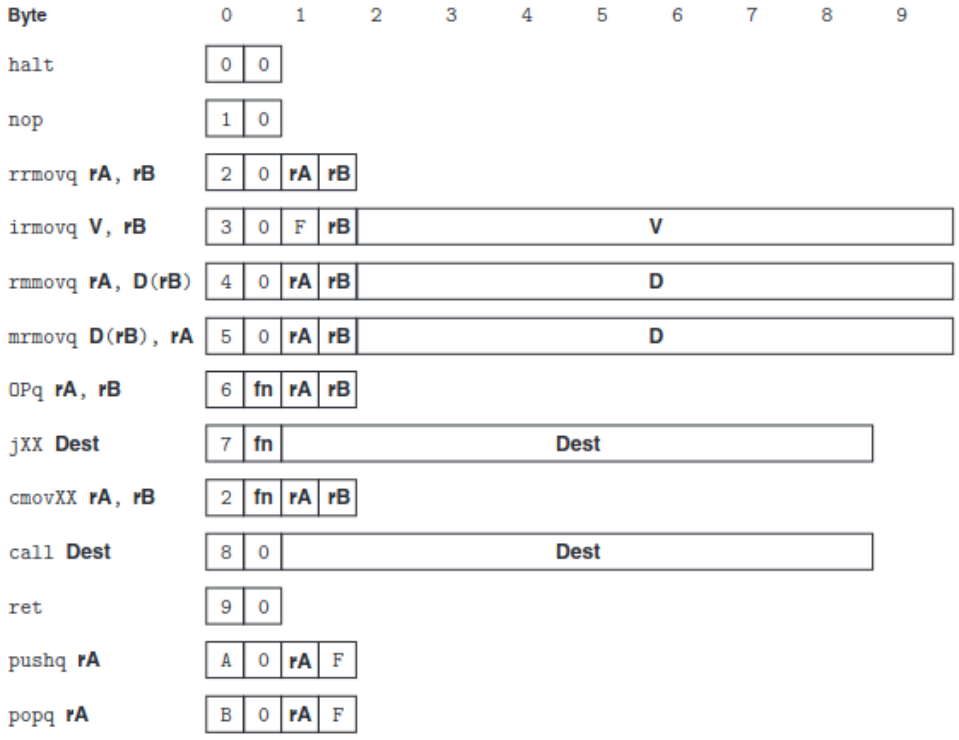
Team_19 Priya Mandot (2022102053) Pearl Shah (2022102073)

Goal : Develop a processor architecture design based on the Y86 ISA using Verilog through two implementations - Sequential and Pipeline.

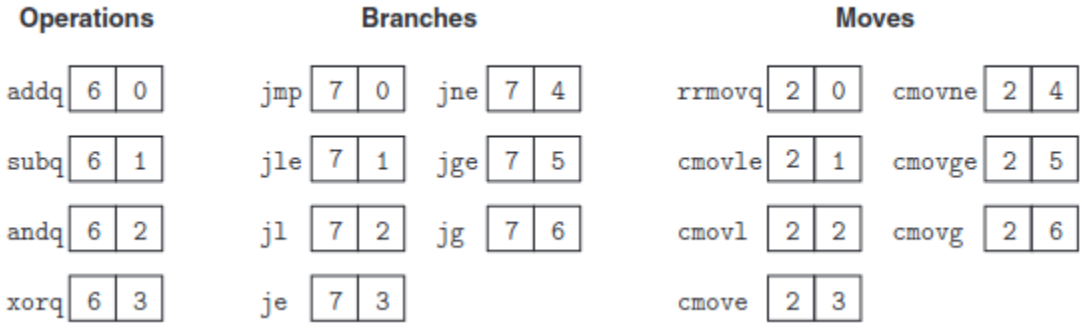
Y86- 64 instruction set:

Instruction encodings



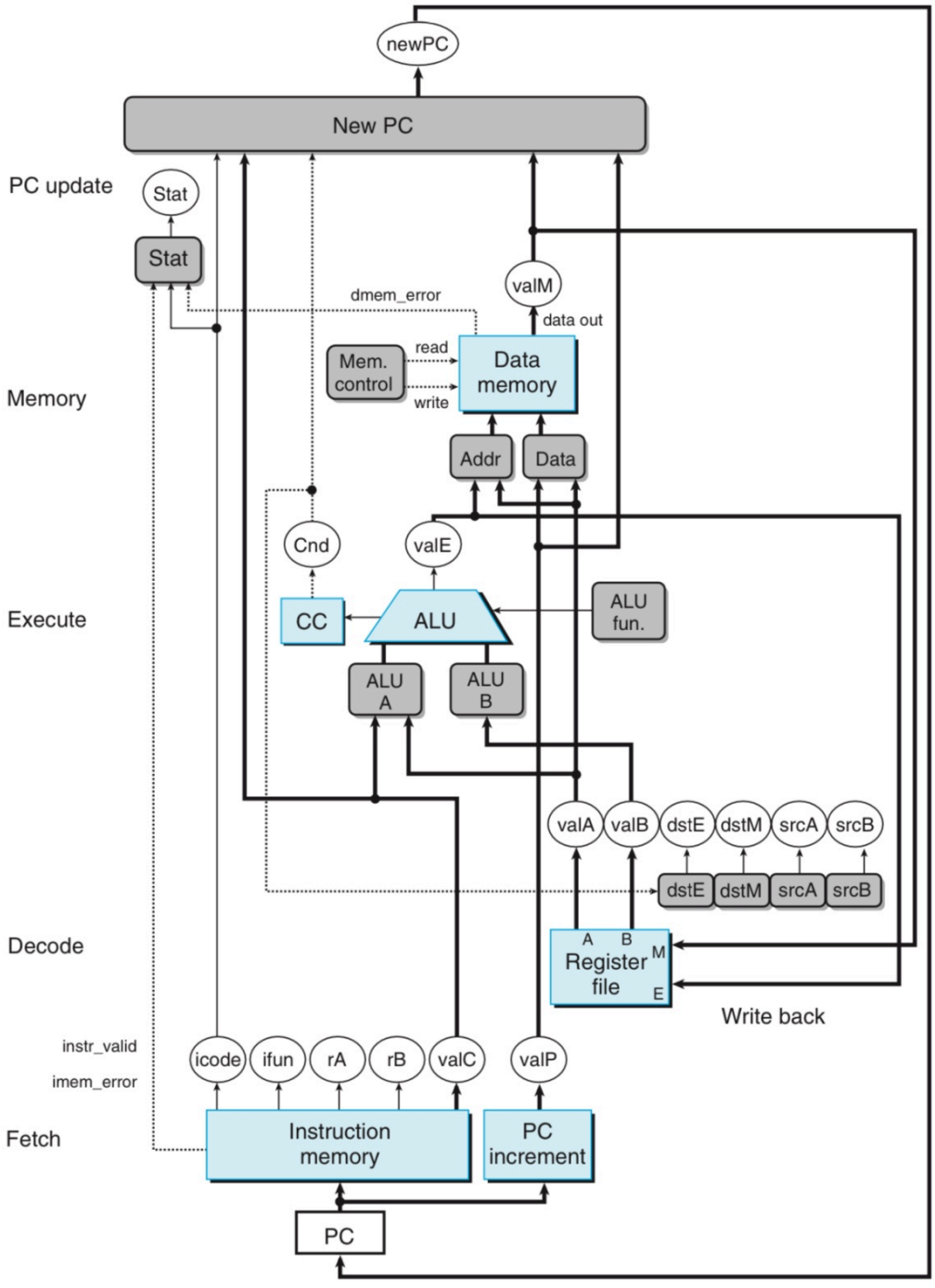
Instruction encodings range between 1 and 10 bytes. An instruction consists of a 1-byte instruction specifier, possibly a 1-byte register specifier, and possibly an 8-byte constant word. Field fn specifies a particular integer operation (OPq), data movement condition (cmovXX), or branch condition (jXX). All numeric values are shown in hexadecimal.

Function codes:



PART I: Sequential Implementation

Design



Here,

Diagram	Representation
White Rectangles	Clocked registers
Light blue boxes	Hardware units
Gray rounded rectangles	Control Logic blocks
White circles	Wire names
Medium lines	Word-wide data
Thin lines	Byte and narrower data connections
Dotted lines	Single-bit connections

The processor loops indefinitely, performing these stages. In our simplified implementation, the processor will stop when any exception occurs—that is, when it executes a halt or invalid instruction, or it attempts to read or write an invalid address.

This simple processor design is divided into 6 stages:

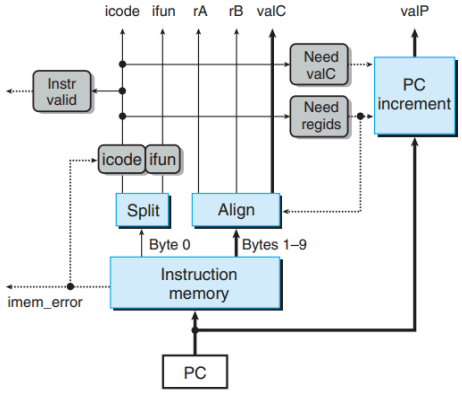
1) **Fetch**
   - **Task :** The main goal of the fetch stage in this process is to retrieve the instruction from memory using the program counter (PC) as the memory address. From this instruction, it extracts important components such as the instruction code (icode), the instruction function (ifun), and potentially register operand specifiers (rA and rB) as well as an 8-byte constant word (valC). Additionally, it computes the address of the next instruction (valP) by adding the length of the fetched instruction to the current PC value. In essence, the fetch stage aims to gather all necessary information about the current instruction and prepare for the execution stage of the processor pipeline.
2) **Decode**
   - **Task:** The decode stage retrieves up to two operands from the register file, obtaining values valA and/or valB. It usually reads registers specified by instruction fields rA and rB, occasionally accessing register %rsp for some instructions.
3) **Execute**
   - **Task:** In the execute stage, the arithmetic/logic unit (ALU) carries out operations specified by the instruction's ifun value, computes effective memory addresses, or adjusts the stack pointer. The resulting value is

termed valE. Condition codes may be set based on the operation. For conditional move instructions, the stage evaluates conditions and updates destination registers only if conditions are met. Likewise, for jump instructions, it decides whether to take the branch or not.

4) **Memory**
   - **Task:** The memory stage may write data to memory, or it may read data from memory (valM).
5) **Write Back**
   - **Task:** Writes up to two results to the register file to dstE and/or dstM.
6) **PC Update**
   - **Task:** Set PC to the address of next instruction.

<u>Stage-wise implementations</u>

FETCH : This is the first stage in sequential implementation.



It contains an instruction memory hardware unit which reads 10 bytes from memory using the PC given by the pc update in the previous cycle or is initialized to 0 in case of first instruction. PC is the address of the first byte.The 'Split' block splits this into 4-bit each icode and ifun.Using this icode we get 3 values -

instr_valid. Does this byte correspond to a legal Y86-64 instruction? This signal is used to detect an illegal instruction.

need_regids. Does this instruction include a register specifier byte?

need_valC. Does this instruction include a constant word?

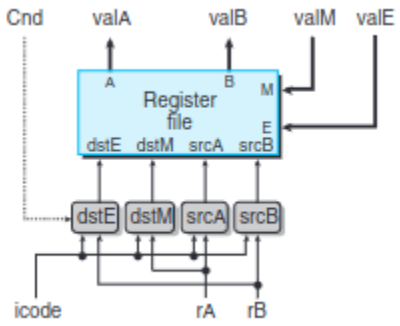
If PC is out of bounds that imem_error occurs , if icode is not valid hex value between 0 to B then instr_valid goes to 0.Example for an HCL code to determine need_regids is-

```
bool need_regids =
        icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                   IIRMOVQ, IRMMOVQ, IMRMOVQ };
```

If icode belongs to irmovq,rmmovq,mrmovq,jXX and call then bool need_valC is true.

According to these signals ,‘Align’ block splits bytes1-9 into rA,rB,valC where rA,rB are 1 byte each and valC is 8 bytes.If need_regids is 0- rA,rB are set to 0xF. The increment in PC or valP is generated based on need_regids and need_valC. If both are 0 valP=PC+1 , if need_regids is 1 and need_valC is 0 valP=PC+2,if need_valC is 1 and need_regids is 0 valP=PC+9 and if both are 1 valP=PC+10.

DECODE & WRITE-BACK : The two stages are combined as they both access the register files. However, we allow for Write Back only at the positive edge of the clock while decode happens for any change in the inputs.



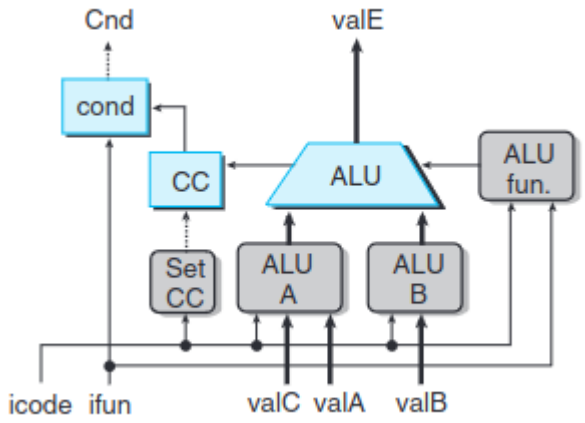
From the figure right above, it is seen that the register file has 4 ports- 2 read (A and B) and 2 write (E and M). Each port has an address - register ID and a data - 64 bit word connection. For the read ports the addresses are indicated by srcA and srcB and data as valA and valB respectively. Similarly, for write ports addresses are dstE and dstM and data is valE and valM. The 4 registers IDs are generated based on the icode, rA, rB and for cmov instruction condition signal Cnd from the execute block.

The register file consists of 15 registers capable of storing 64-bit integers initialized to random values. The %rsp register is given a higher value so that during push operation to the stack a negative value does not result as that would lead to a dmem_error. Description for the 4 register IDs:

```
word srcA = [
        icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
        icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];

 word srcB = [
         icode in { IOPQ, IRMMOVQ, IMRMOVQ  } : rB;
         icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
         1 : RNONE;  # Don't need register
 ];
```

```
word dstE = [
        icode in { IRRMOVQ } && Cnd : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];

word dstM = [
        icode in { IMRMOVQ, IPOPQ } : rA;
        1 : RNONE;  # Don't write any register
];
```

EXECUTE : This stage includes the ALU arithmetic logic unit which performs ADD,SUBTRACT,AND,EX-OR on aluA,aluB .



The operation is decided by ALU-fun which takes in icode and ifun .

```
word alufun = [
        icode == IOPQ : ifun;
        1 : ALUADD;
];
```

The blocks ‘ALU A’ and ‘ALU B’ decide the values of aluA,aluB.The value aluA is decide by-

```
word aluA = [
        icode in { IRRMOVQ, IOPQ } : valA;
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
```

```
        icode in { ICALL, IPUSHQ } : -8;
        icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];

word aluB = [
        icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                   IPUSHQ, IRET, IPOPQ } : valB;
        icode in { IRRMOVQ, IIRMOVQ } : 0;
        # Other instructions don't need ALU
];
```

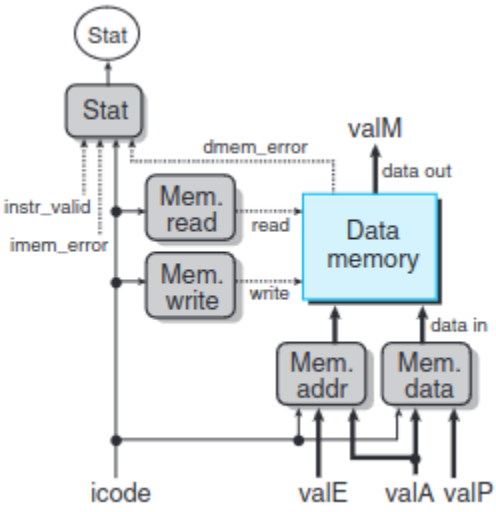
This stage also includes the condition code registers.ALU generates 3 signals based on which condition codes are based - zero flag,sign flag,overflow flag.
These flag are set only when set_cc = 1 which occurs during opq instruction.

ZF	(t == 0)	Zero
SF	(t < 0)	Negative
OF	(a < 0 == b < 0) && (t < 0 != a < 0)	Signed overflow

Cnd is set for cmovq and jump instructions otherwise it is kept 0.

```
if(ifun==4'h0)begin // jmp operand
    Cnd=1;
end
else if (ifun==4'h1)begin//jle
    Cnd=(SF^OF)|ZF;
end
 else if(ifun==4'h2)begin//jl
    Cnd=SF^OF;

 end
else if(ifun==4'h3)begin//je
    Cnd=ZF;

end
else if(ifun==4'h4)begin//jne
    Cnd=~ZF;

end
else if(ifun==4'h5)begin//jge
    Cnd=~(SF^OF);

end
else if(ifun==4'h6)begin//jg
    Cnd=~(SF^OF) & ~ZF;

end
```

MEMORY : Reads or writes program data to memory.



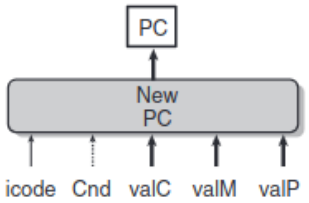
The data memory is defined in our code as a block of 1024 64-bit registers, initialized to zero.
Two control blocks generate mem_addr and mem_data (for write) and two other blocks generate if read or write should be performed. When data read is performed valM is generated. We also generate the dmem_error when 1) both read and write are enabled at the same time 2) memory address is outside the defined memory range. This block also generates the program Status (Stat) stating 1) normal operation- SAOK, 2) memory error- SADR, 3) invalid instruction - SINS and 4) halt - SHLT. Except for normal operations, everything else will terminate the program. For Stat, imem_error, instr_valid and icode signals are obtained from the Fetch stage.

Description for the control blocks:

```
word mem_addr = [
        icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
        icode in { IPOPQ, IRET } : valA;
        # Other instructions don't need address
];

word mem_data = [
        # Value from register
        icode in { IRMMOVQ, IPUSHQ } : valA;
        # Return PC
        icode == ICALL : valP;
        # Default: Don't write anything
];
```

```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };

bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };

## Determine instruction status
word Stat = [
        imem_error || dmem_error : SADR;
        !instr_valid: SINS;
        icode == IHALT : SHLT;
        1 : SAOK;
];
```

PC UPDATE : Generates a new value for Program Counter.



The selection for the new PC is done according to:

```
word new_pc = [
        # Call.  Use instruction constant
        icode == ICALL : valC;
        # Taken branch.  Use instruction constant
        icode == IJXX && Cnd : valC;
        # Completion of RET instruction.  Use value from stack
        icode == IRET : valM;
        # Default: Use incremented PC
        1 : valP;
];
```

The PC is incremented for valP by 1, 2, 9 or 10 based on the instruction length as defined in the Y86-64 instruction set.

SEQ Timing

- An entire instruction is performed every clock cycle.
- Combinational logic does not require any sequencing or control from the clock and the values propagate through whenever the input changes. In Verilog, denoted by always@(*). Also, the instruction memory can be treated as a combinational logic unit as it is only used to read instructions.
- PC, Conditional control register, Data memory and Register file are controlled via a single clock cycle that triggers loading and writing of new values.
- The PC is loaded with a new instruction every clock cycle. In Verilog : always@(posedge(clk)) => PC = updated_PC

- Conditional control register is loaded only if an integer operation is executed.
- Data memory is written only for rmmovq, pushq, and call instructions.
- By the “No reading back” principle new state values are calculated every clock cycle but only updated at the positive edge of the clock transition.

Test case

This test case encompasses all the instructions and we can explain the outputs of each stage through gtkwave plots. The test values are in a hexadecimal system. Each line is numbered according to how it is stored in the instruction memory. # explains the instruction.

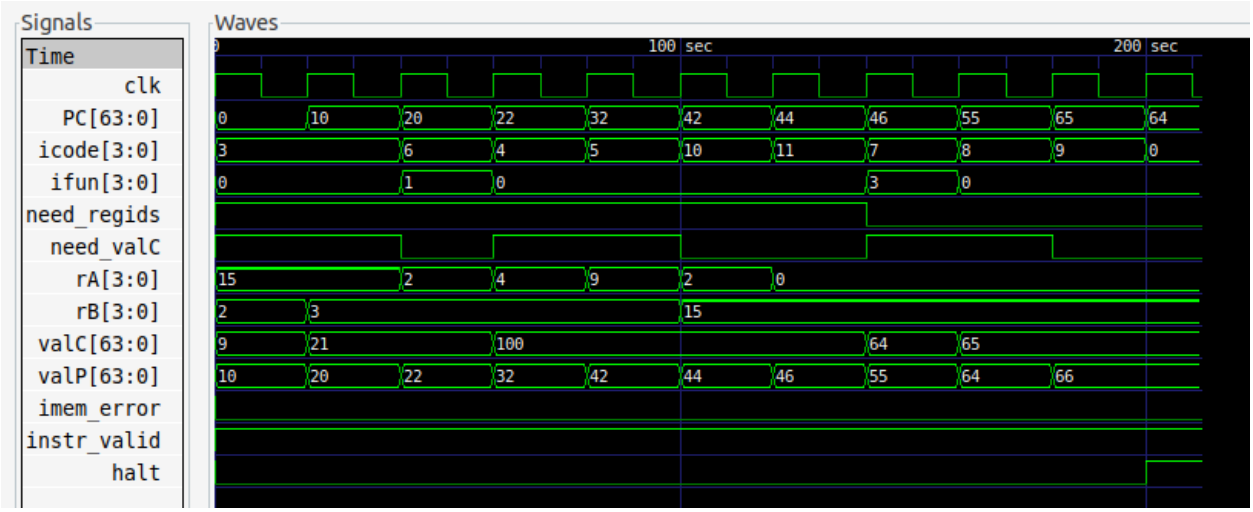
Explanation of the test case:

0. 30 # irmovq F2 09 00 00 00 00 00 00 00 : move $9 to the second register(%rdx)
1. F2 # F indicates no register is accessed. dstE = register_file[2] = rdx
2. 09 # valC = 9
3. 00
4. 00
5. 00
6. 00
7. 00
8. 00
9. 00 # valP incremented by 10 => updated PC = 0+ 10 = 10
10. 30 # similar to the above, move $15 to %rbx
11. F3 # dstE = register_file[3] = rbx
12. 15 # valC = 15 (21 in decimal)
13. 00
14. 00
15. 00
16. 00
17. 00
18. 00
19. 00 #valP incremented by 10, updated PC = 10+10 = 20
20. 61 #OPq: 61 indicates subtraction, CC register updates,valP increments by 2: 22
21. 23 #subtract value of register_file[2] from register_file[3] and store in latter
22. 40 #rmmovq : copy value from register_file[8] to memory
23. 83 # memory location = Value(register_file[3]) + valC = dstM
24. 64 #valC = 64 (100 in decimal), mem_write enabled
25. 00
26. 00
27. 00
28. 00
29. 00
30. 00

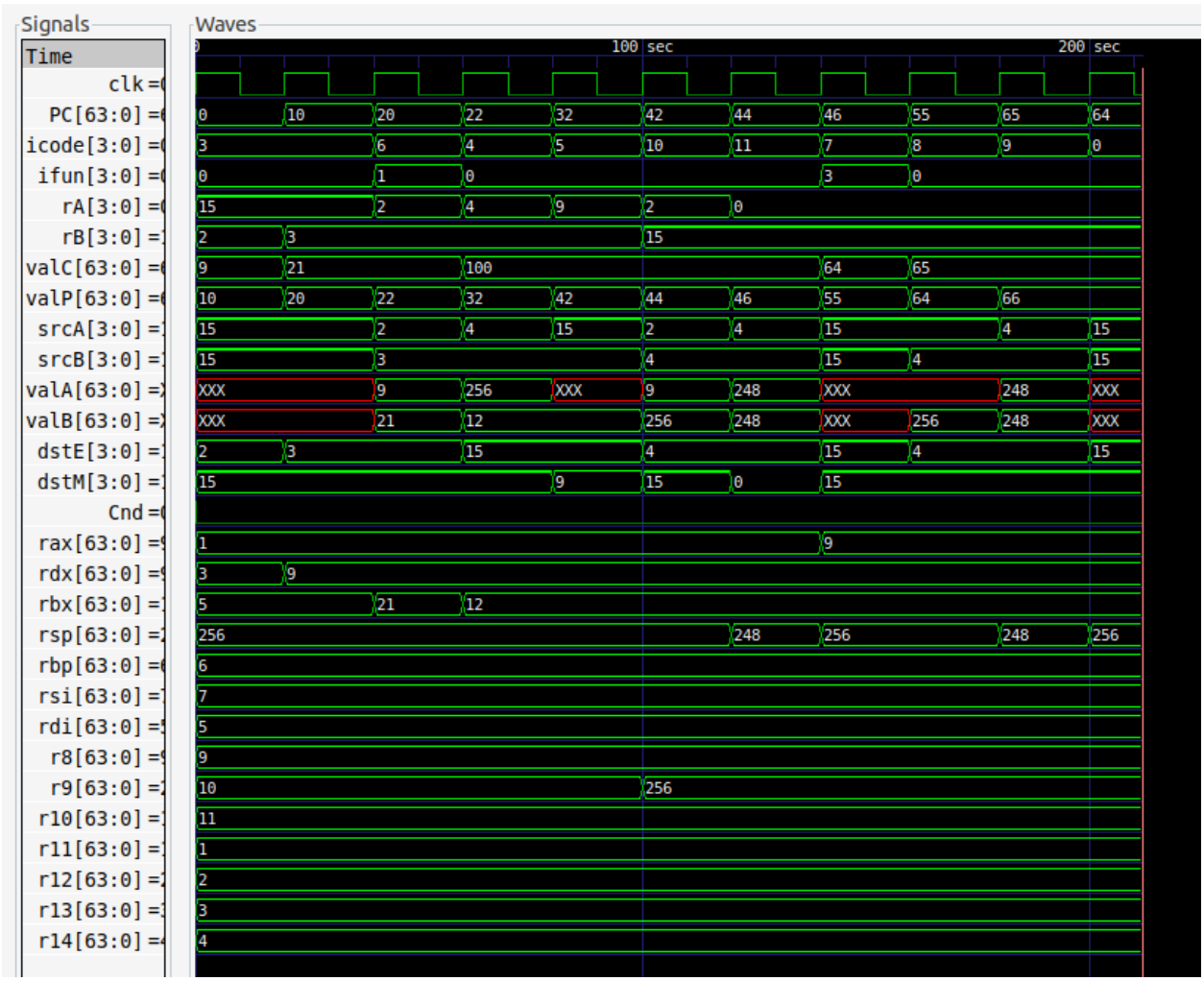
```
00  #valP increments by 10 : 22 + 10 = 32
50  # mrmovq: copies value from memory to register_file[9]
93  #valM read from dstM, mem_read enabled
64  # valC = 64(100 in decimal) => dstM = value(register_file[3]) + valC
00
00
00
00
00
00
00  #valP incremented by 10 : 32 + 10 = 42
A0 #pushq , valP increments by 2: 42 + 2= 44
2f  #push value of rdx to stack in memory by decrementing %rsp by 8
B0  #popq, valP increments by 2:  44 + 2 = 46
0f   #pops value from stack, then increments %rsp by 8. Stores in o index reg:
    rax
73  #jle: jumps if ZF = 1 as set earlier by integer operation
40  #in our case branch not taken, valC = 40 (64 in decimal)
00
00
00
00
00
00
00  #valP increments by 9 : 46 + 9 = 55,( if jmp taken updated PC = valC)
80  #call: store valP in stack and go to valC,%rsp decremented by 8
41  #valC = 41 (65 in decimal) => goes to line 65
00
00
00
00
00
00
00  #valP increments by 9 : 55+ 9 = 64, updated PC = valC
00  #halt, executed after ret. program terminated with Stat = 4
90  #ret  : returns to line 64 (address stored in stack), increments %rsp by 8. This
    is executed right after encountering call instruction.
```

Outputs:

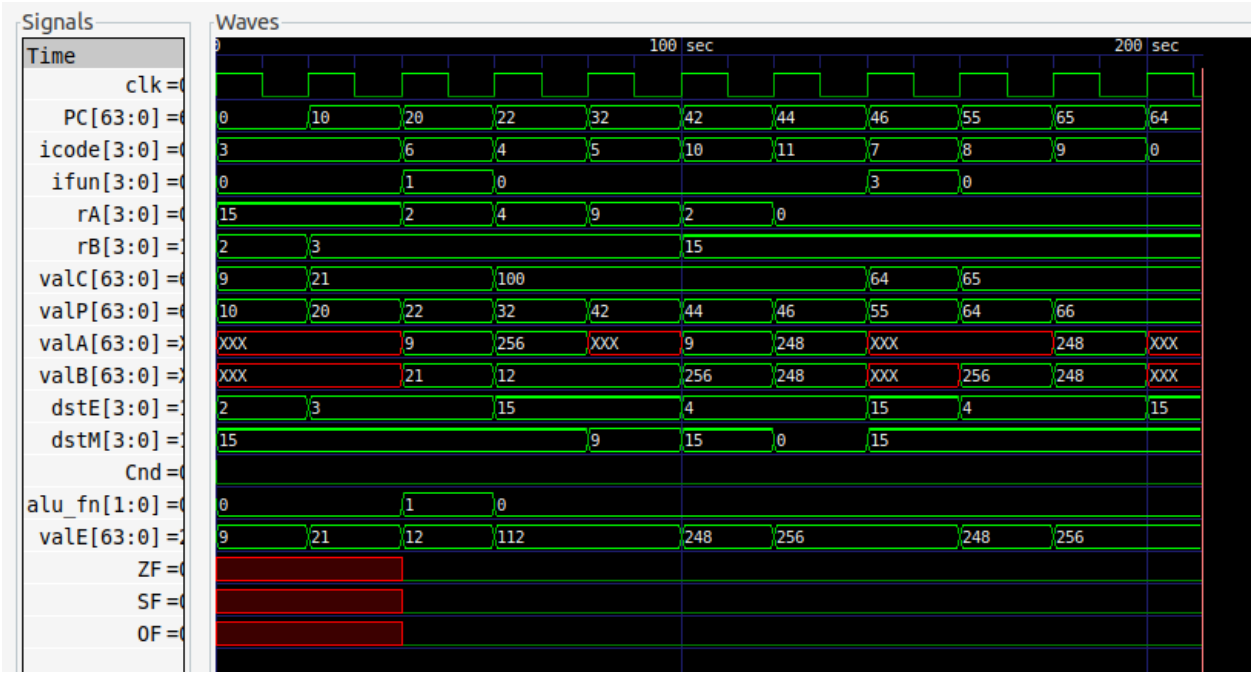
Fetch



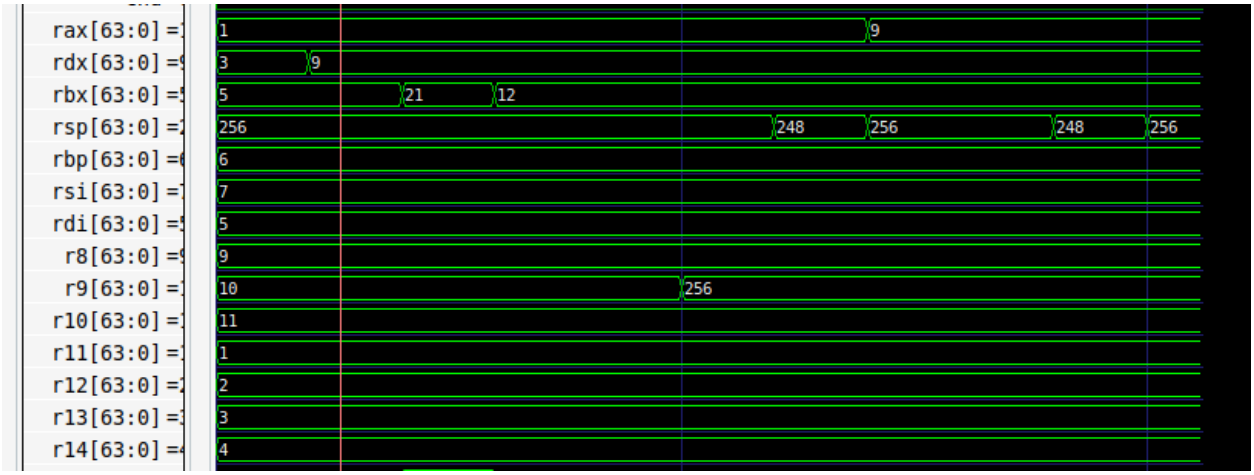
Decode:


Signals
Waves
Time
clk
PC[63:0]
icode[3:0]
ifun[3:0]
rA[3:0]
rB[3:0]
valC[63:0]
valP[63:0]
srcA[3:0]
srcB[3:0]
valA[63:0]
valB[63:0]
dstE[3:0]
dstM[3:0]
Cnd
rax[63:0]
rdx[63:0]
rbx[63:0]
rsp[63:0]
rbp[63:0]
rsi[63:0]
rdi[63:0]
r8[63:0]
r9[63:0]
r10[63:0]
r11[63:0]
r12[63:0]
r13[63:0]
r14[63:0]
100 sec
200 sec

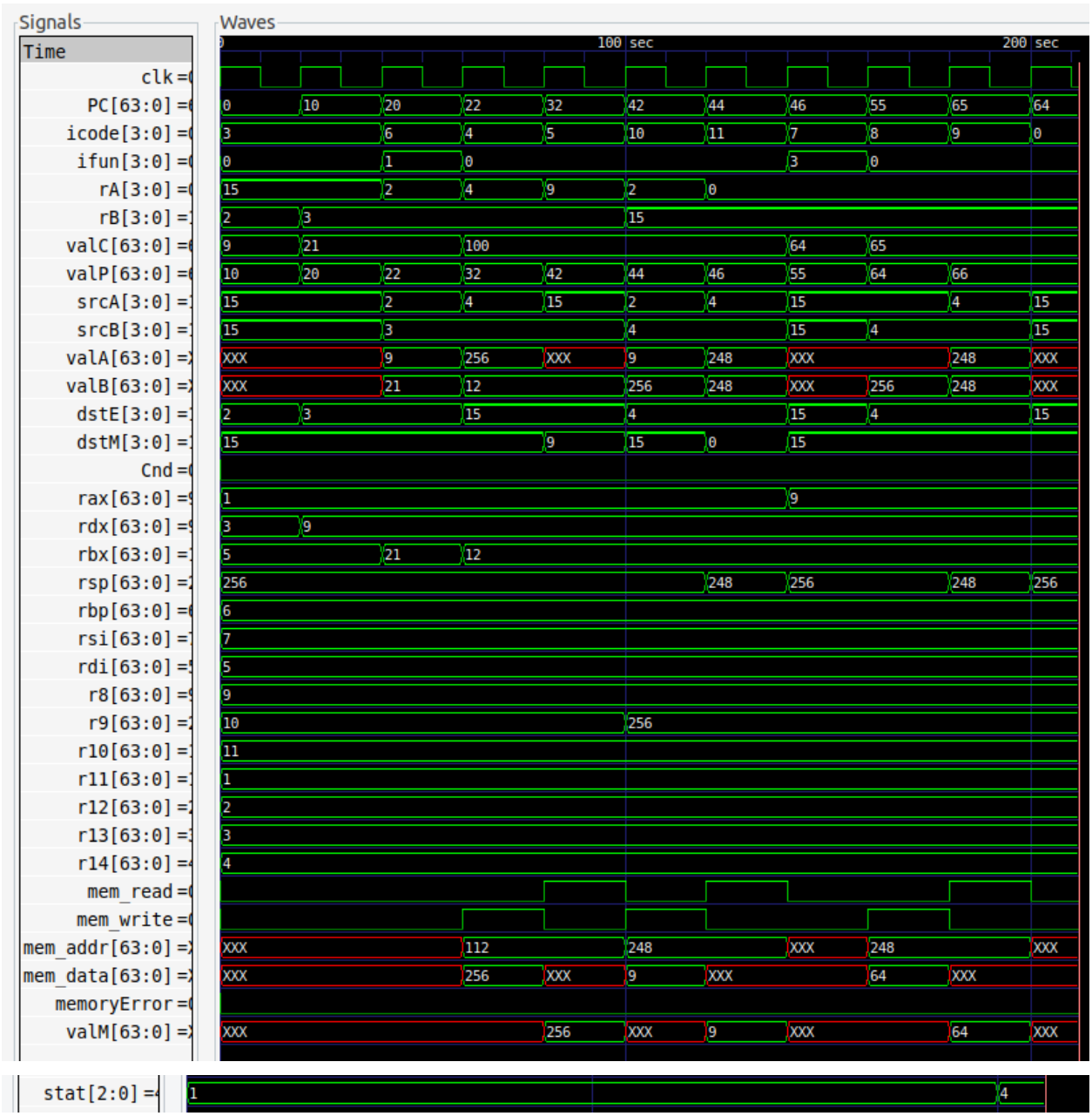
Execute:



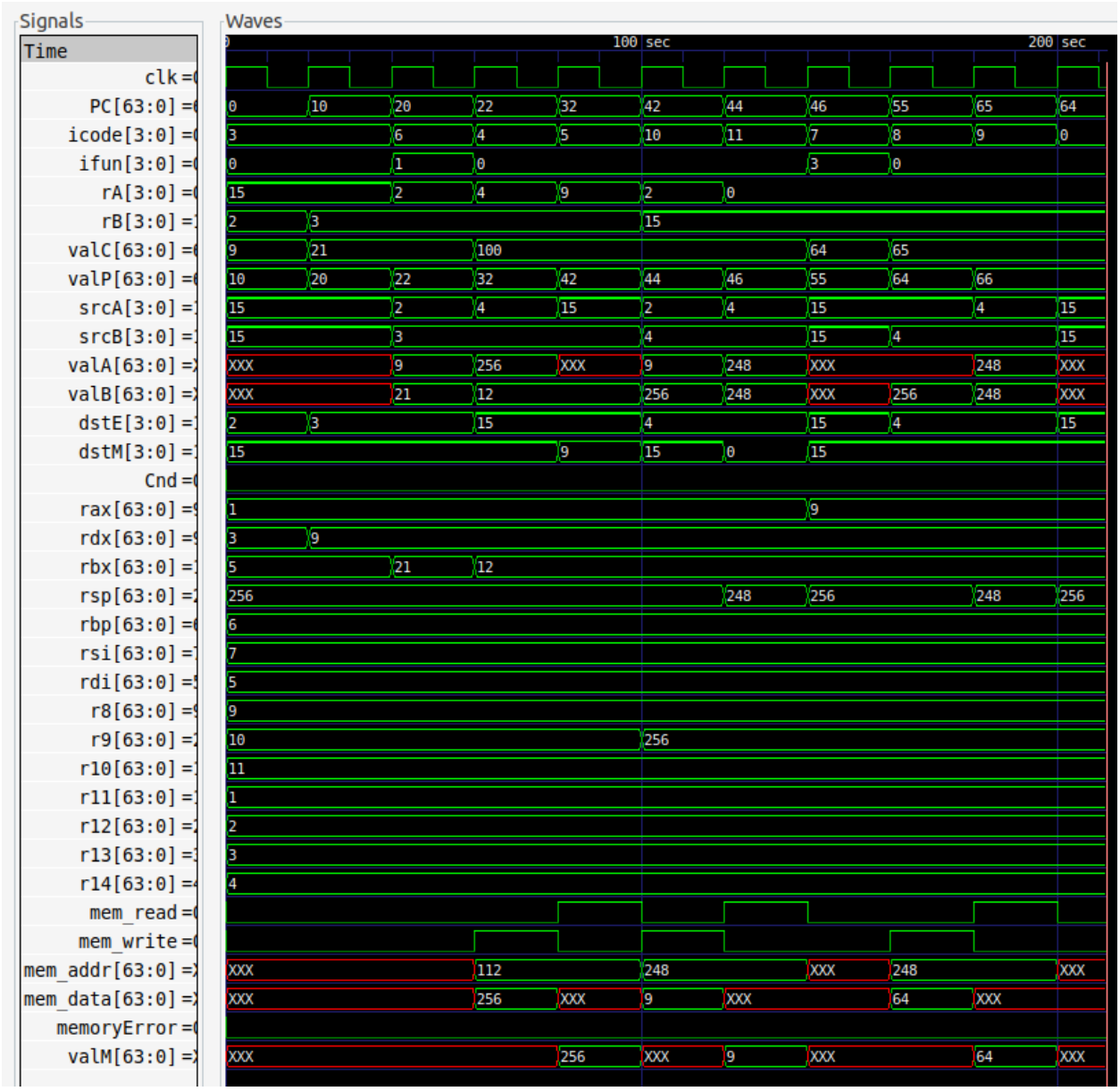
Register file



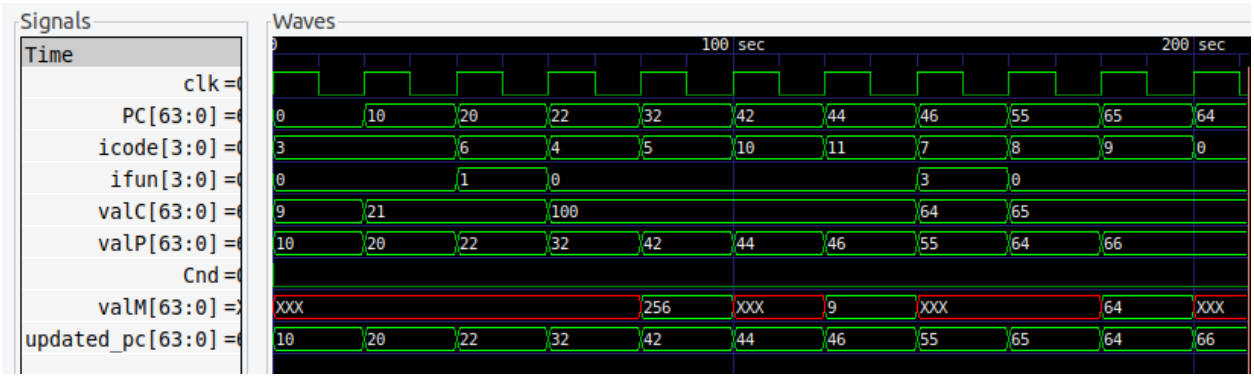
Memory:


Signals
Waves
Time
clk
PC[63:0]
icode[3:0]
ifun[3:0]
rA[3:0]
rB[3:0]
valC[63:0]
valP[63:0]
srcA[3:0]
srcB[3:0]
valA[63:0]
valB[63:0]
dstE[3:0]
dstM[3:0]
Cnd
rax[63:0]
rdx[63:0]
rbx[63:0]
rsp[63:0]
rbp[63:0]
rsi[63:0]
rdi[63:0]
r8[63:0]
r9[63:0]
r10[63:0]
r11[63:0]
r12[63:0]
r13[63:0]
r14[63:0]
mem_read
mem_write
mem_addr[63:0]
mem_data[63:0]
memoryError
valM[63:0]
stat[2:0]
100 sec
200 sec

Write Back:


Signals
Waves
Time
clk
PC[63:0]
icode[3:0]
ifun[3:0]
rA[3:0]
rB[3:0]
valC[63:0]
valP[63:0]
srcA[3:0]
srcB[3:0]
valA[63:0]
valB[63:0]
dstE[3:0]
dstM[3:0]
Cnd
rax[63:0]
rdx[63:0]
rbx[63:0]
rsp[63:0]
rbp[63:0]
rsi[63:0]
rdi[63:0]
r8[63:0]
r9[63:0]
r10[63:0]
r11[63:0]
r12[63:0]
r13[63:0]
r14[63:0]
mem_read
mem_write
mem_addr[63:0]
mem_data[63:0]
memoryError
valM[63:0]
100 sec
200 sec

PC update:


Signals
Waves
Time
clk
PC[63:0]
icode[3:0]
ifun[3:0]
valC[63:0]
valP[63:0]
Cnd
valM[63:0]
updated_pc[63:0]
100 sec
200 sec

Challenges

Problems with SEQ implementation-

1. Limited Hardware Utilization - Hardware units are active only for a fraction of total clock cycle . This is because each instruction performs sequentially within a single clock cycle leading to underutilization of resources.
2. Slow clock speed- The clock speed is kept slow so as to allow a complete instruction to run within a single clock cycle.
3. Instruction Processing Bottlenecks- Complex instructions such as ‘ret’ require multiple stages within a single clock cycle. This creates bottlenecks in instruction execution since each stage must be completed within a time frame.
4. Inefficient Resource Allocation- A major part of the processor remains idle for a significant period of time at any instant since only one stage is working at a time.This reduces overall system throughput or the number of instructions executed.
5. Highly dependent on clock speed- We cannot increase the speed of clock due to limitations of signal propagation and hardware constraints and slow clock speed would make the overall processor slow.

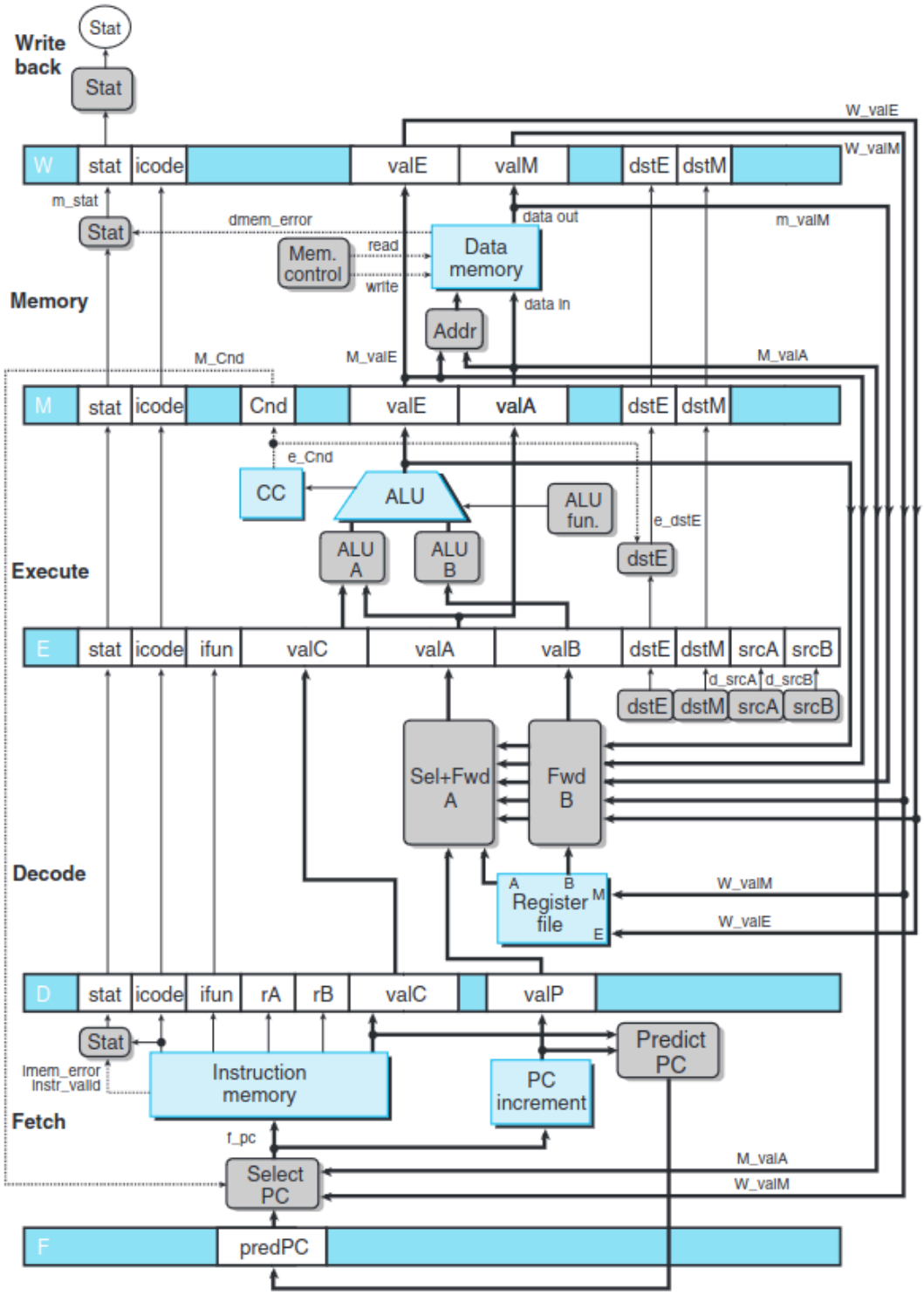
Problems faced

1. Making sure the “No Read back” principle was followed- we did not put Write Back in posedge first.
2. %rsp was initialized to a small value as a result of which on pushq instructions the memory became invalid thus terminating the program.
3. Also, before terminating we ensured to give an extra 5 seconds so that any previous write backs were completed. This saw to it that all instructions before an exception were completed and any after were not.

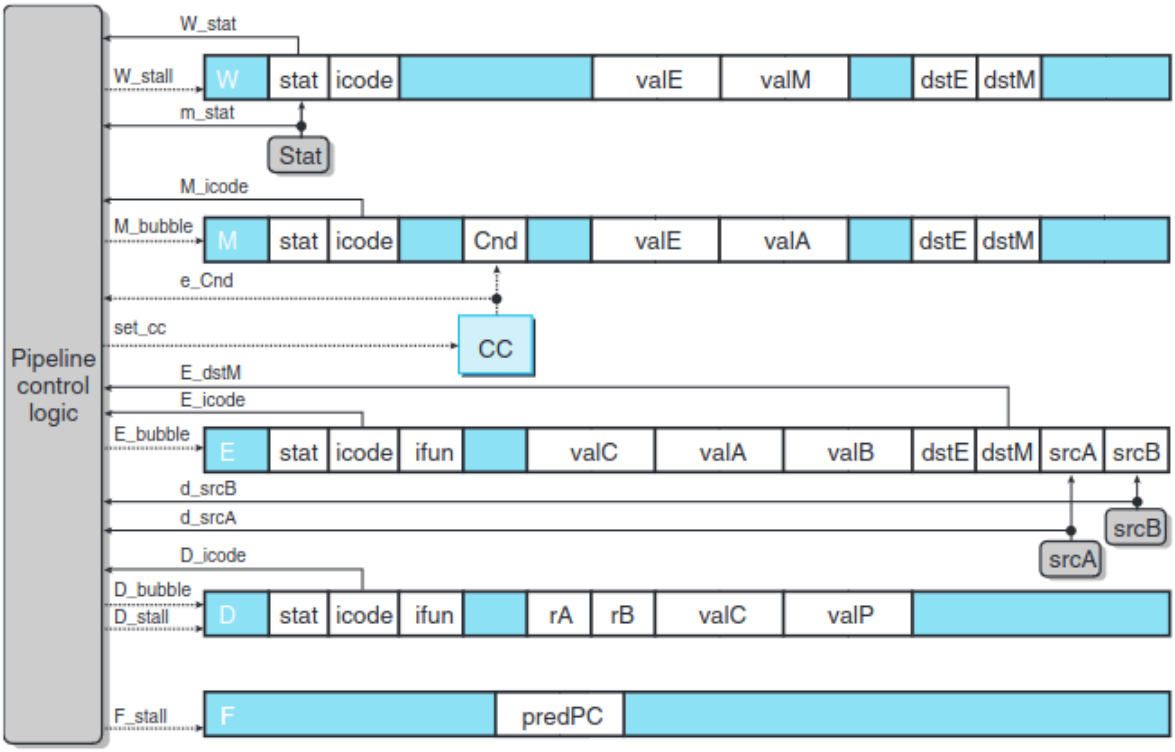
PART II: Pipeline Implementation

Design

Hardware structure:



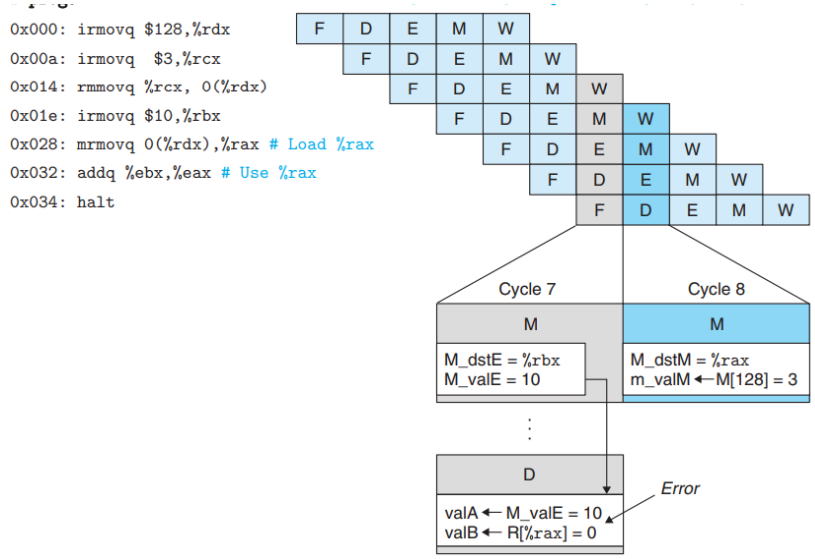
Pipeline control logic



This logic must handle the following four control cases for which other mechanisms, such as data forwarding and branch prediction, do not suffice:

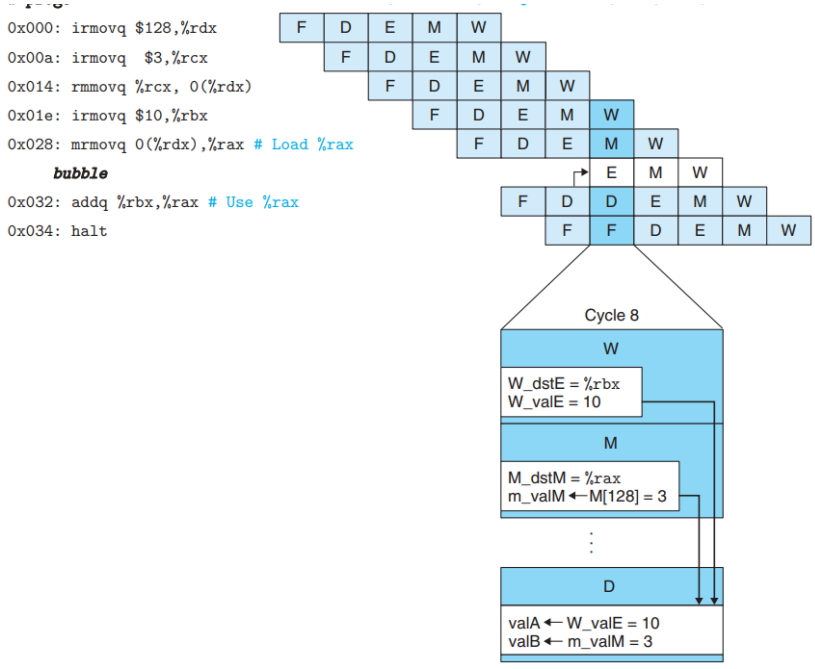
1. **Load/use hazards**-
   Problem-These hazards occur when a memory read instruction is followed by another instruction needing that memory value, causing timing conflicts.Forwarding alone can't resolve load/use hazards because memory reads happen late in the pipeline.The issue arises when an instruction needs

value from memory before it's available.



Here the addq instruction requires the value of register %rax during the decode stage in cycle 7. The preceding mrmovq reads a new value for this register during the memory stage in cycle 8, which is too late for the addq instruction.

Solution-Stalling and forwarding are combined to handle load/use hazards effectively.Control logic detects the need for a memory value in the decode stage and stalls the pipeline for one cycle.During this stall, a bubble is injected into the execute stage, allowing the memory value to be forwarded to the dependent instruction.Thus a load/use hazard is resolved effectively.



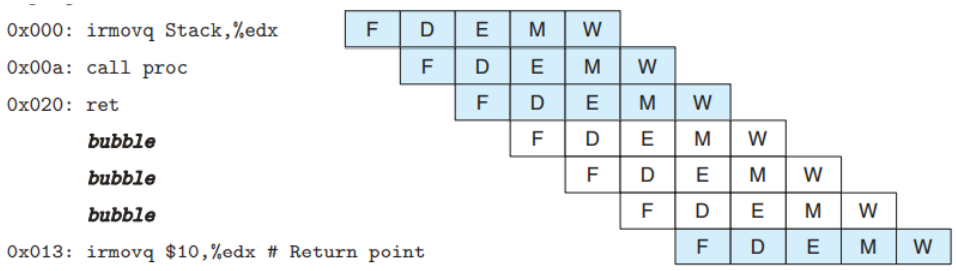
2.**Processing Return-**

Problem-

```
0x000:    irmovq stack,%rsp  #   Initialize stack pointer
0x00a:    call proc          #   Procedure call
0x013:    irmovq $10,%rdx    #   Return point
0x01d:    halt
0x020: .pos 0x20
0x020: proc:                 # proc:
0x020:    ret                #   Return immediately
0x021:    rrmovq %rdx,%rbx   #   Not executed
0x030: .pos 0x30
0x030: stack:                # stack: Stack pointer
```

Here we see that when the return instruction is given the pipeline should fetch the address on the top of stack instead of the next instruction after return . However when we use pipeline until the return is written back we cannot get the address on top of the stack.

Solution-



The ret instruction is fetched during cycle 3 and proceeds down the pipeline, reaching the write-back stage in cycle 7. While it passes through the decode, execute, and memory stages, the pipeline cannot do any useful activity. Instead, we want to inject three bubbles into the pipeline. Once the ret instruction reaches the write-back stage, the PC selection logic will set the program counter to the return address, and therefore the fetch stage will fetch the irmovq instruction at the return point.

3. **Mispredicted branch-**

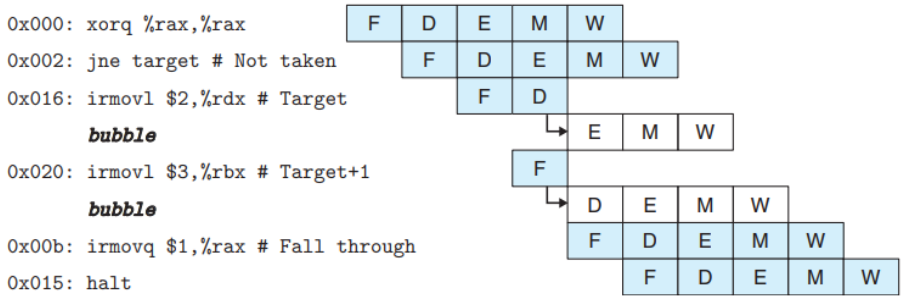
Problem- By the time the branch logic detects that a jump should not have been taken, several instructions at the branch target will have started down the pipeline.

```
0x000:    xorq %rax,%rax
0x002:    jne  target        # Not taken
0x00b:    irmovq $1, %rax    # Fall through
0x015:    halt
0x016: target:
0x016:    irmovq $2, %rdx    # Target
0x020:    irmovq $3, %rbx    # Target+1
0x02a:    halt
```

Since the jump instruction is predicted as being taken, the instruction at the jump target will be fetched in cycle 3, and the instruction following this one will be fetched in cycle 4. By the time the branch logic detects that the jump should not be taken during cycle 4, two instructions have been fetched that should not continue being executed.

Solution-

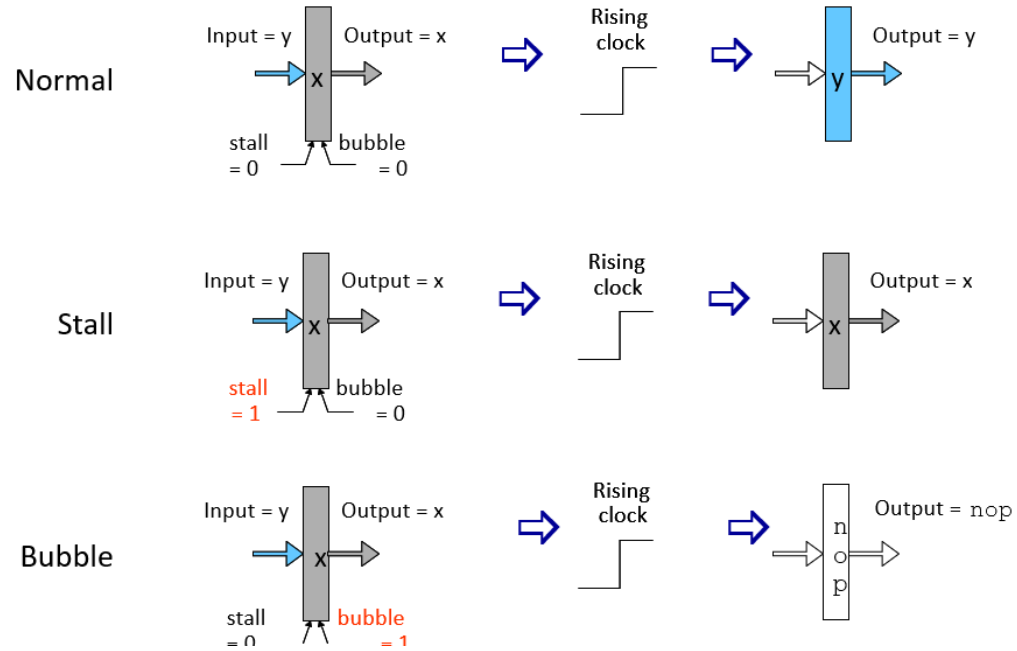
Fortunately, neither of these instructions has caused a change in the programmer-visible state. That can only occur when an instruction reaches the execute stage, where it can cause the condition codes to change. At this point, the pipeline can simply cancel (sometimes called instruction squashing) the two misfetched instructions by injecting bubbles into the decode and execute stages on the following cycle while also fetching the instruction following the jump instruction. The two misfetched instructions will then simply disappear from the pipeline and therefore not have any effect on the programmer-visible state.



The pipeline predicts branches will be taken and so starts fetching instructions at the jump target. Two instructions are fetched before the misprediction is detected in cycle 4 when the jump instruction flows through the execute stage. In cycle 5, the pipeline cancels the two target instructions by injecting bubbles into the decode and execute stages, and it also fetches the instruction following the jump.

Pipeline Register Modes-

A register can be in three modes in a pipelined implementation - normal, stall or bubble.



Normal mode is where the input is given normally as the output at rising clock.

Stall-

A stall, also known as a pipeline stall, occurs when a stage in the pipeline cannot proceed due to a data hazard or a structural hazard.In a data hazard situation, the next instruction in the pipeline depends on the result of a previous instruction that hasn't yet completed its execution. As a result, the pipeline must stall until the required data becomes available.Stalls are generally introduced by inserting no-operation (NOP) instructions into the pipeline to delay execution until the required data is available.The purpose of a stall is to wait for the required data to become available before proceeding with the execution of the next instruction.

Bubble-

A bubble is a specific type of stall where a no-operation (NOP) instruction is inserted into the pipeline to maintain the pipeline's structural integrity, rather than to resolve a data hazard.By inserting a bubble, the pipeline ensures that instructions don't overwrite each other's results or interfere with each other's execution.The purpose of a bubble is to ensure that instructions do not interfere with each other's execution due to structural constraints.

Special Control Cases-

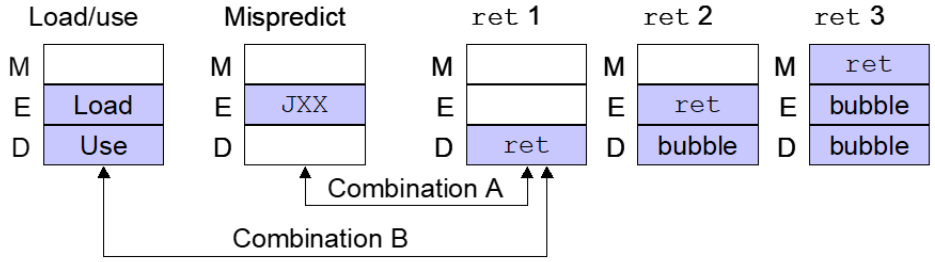
Detection

Condition	Trigger
Processing `ret`	IRET in { D_icode, E_icode, M_icode }
Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
Mispredicted Branch	E_icode = IJXX & !e_Cnd

Action (on next cycle)

Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Mispredicted Branch	normal	bubble	bubble	normal	normal

After considering the control combinations as given below-



A corrected pipeline logic is implemented.

Condition	F	D	E	M	W
Processing ret	stall	bubble	normal	normal	normal
Load/Use Hazard	stall	stall	bubble	normal	normal
Combination	*stall*	*stall*	*bubble*	*normal*	*normal*

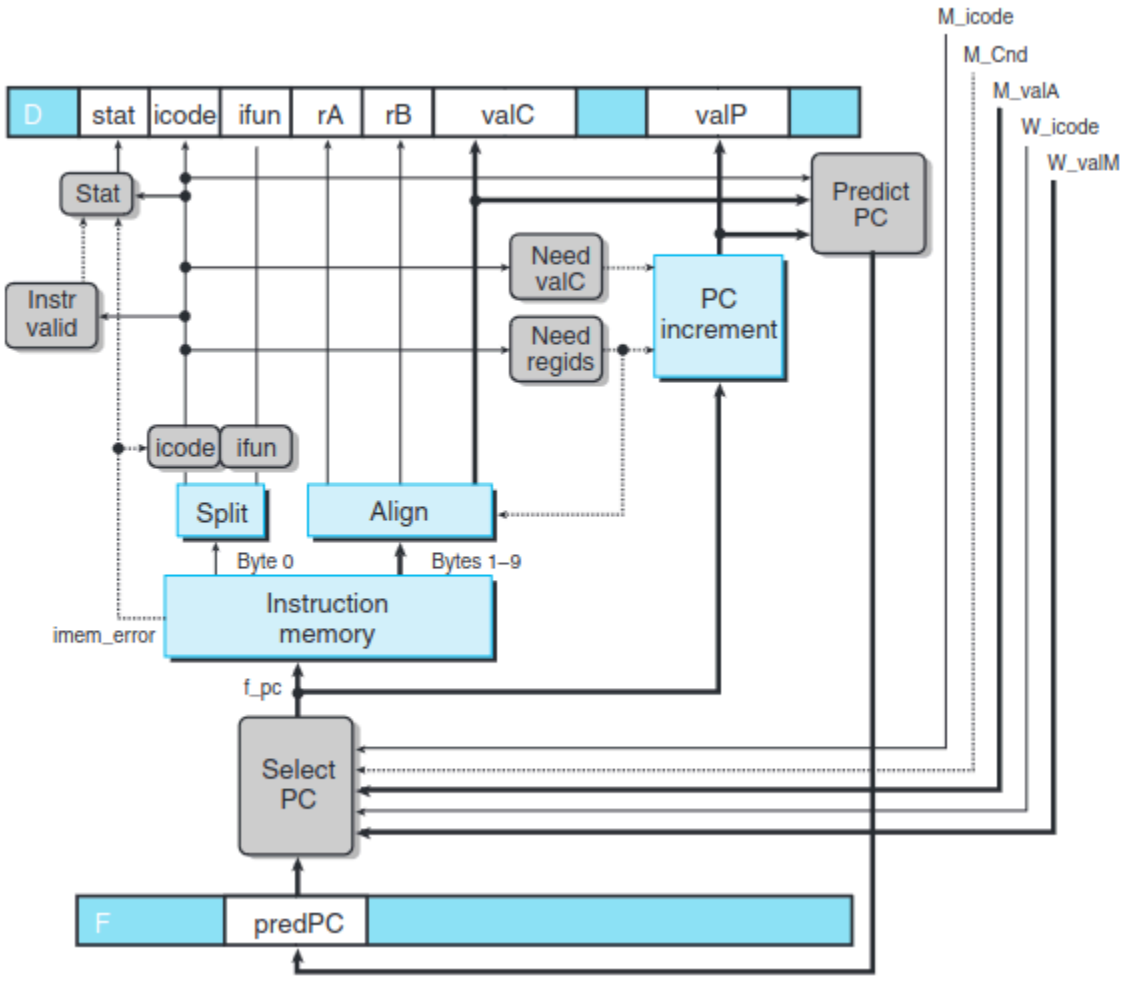
Handling all the cases we implement the following pipeline control logic-

```
//fstall = 1 for load/use hazard and ret ins
if((E_icode == 4'h5 || E_icode == 4'hB) &&(E_dstM == d_srcA || E_dstM == d_srcB)
|| (D_icode == 4'h9 || E_icode == 4'h9 ||M_icode == 4'h9))
F_stall = 1'b1;
else
F_stall = 1'b0;

//dstall = 1 for load/use
if((E_icode == 4'h5 || E_icode == 4'hB) &&(E_dstM == d_srcA || E_dstM == d_srcB))
D_stall= 1'b1;
else
D_stall = 1'b0;

//dbubble = 1 for mispredicted branch, ret but not load use
if (((E_icode == 4'h7) && !e_Cnd) || (!((E_icode == 4'h5 || E_icode == 4'hb)
&& (E_dstM == d_srcA || E_dstM == d_srcB)) && (D_icode == 4'h9 || E_icode == 4'h9 || M_icode == 4'h9)))
D_bubble =1'b1;
else
D_bubble = 1'b0;

//ebubble = 1 if mispredicted or load use hazard
if (((E_icode == 4'h7) && !e_Cnd) || ((E_icode == 4'h5 || E_icode == 4'hb)
&& (E_dstM == d_srcA || E_dstM == d_srcB)))
E_bubble =1'b1;
else
E_bubble = 1'b0;

//set cc = 1 if m_stat and W_stat = 1
if(E_icode == 4'h6 && !(m_stat==3'd2 || m_stat == 3'd3 || m_stat==3'd4)
&& !(W_stat==3'd2 || W_stat == 3'd3 || W_stat==3'd4))
set_cc = 1'b1;
else
set_cc=1'b0;

//Mbubble = 1 if m_stat or W_stat not 1
if(m_stat != 3'd1 || W_stat != 3'd1 )
M_bubble = 1'b1;
else
M_bubble=1'b0;

if(W_stat != 3'd1)
W_stall = 1'b1;
else
W_stall=1'b0;
```

Stage-wise implementation

PIPE uses the same set of hardware units as the earlier sequential designs, with the addition of pipeline registers, some reconfigured logic blocks, and additional pipeline control logic. There are 5 stages here. PC update is combined with fetch. Several logic blocks resemble those in SEQ, albeit requiring appropriate selection of signals from pipeline registers (prefixed with uppercase pipeline register names) or stage computations (prefixed with lowercase initials of stage names). Only the differences from SEQ are mentioned.

PC SELECTION & FETCH : This stage must select a current value for the program counter(Select PC block) and predict the next PC value (Predict PC block).



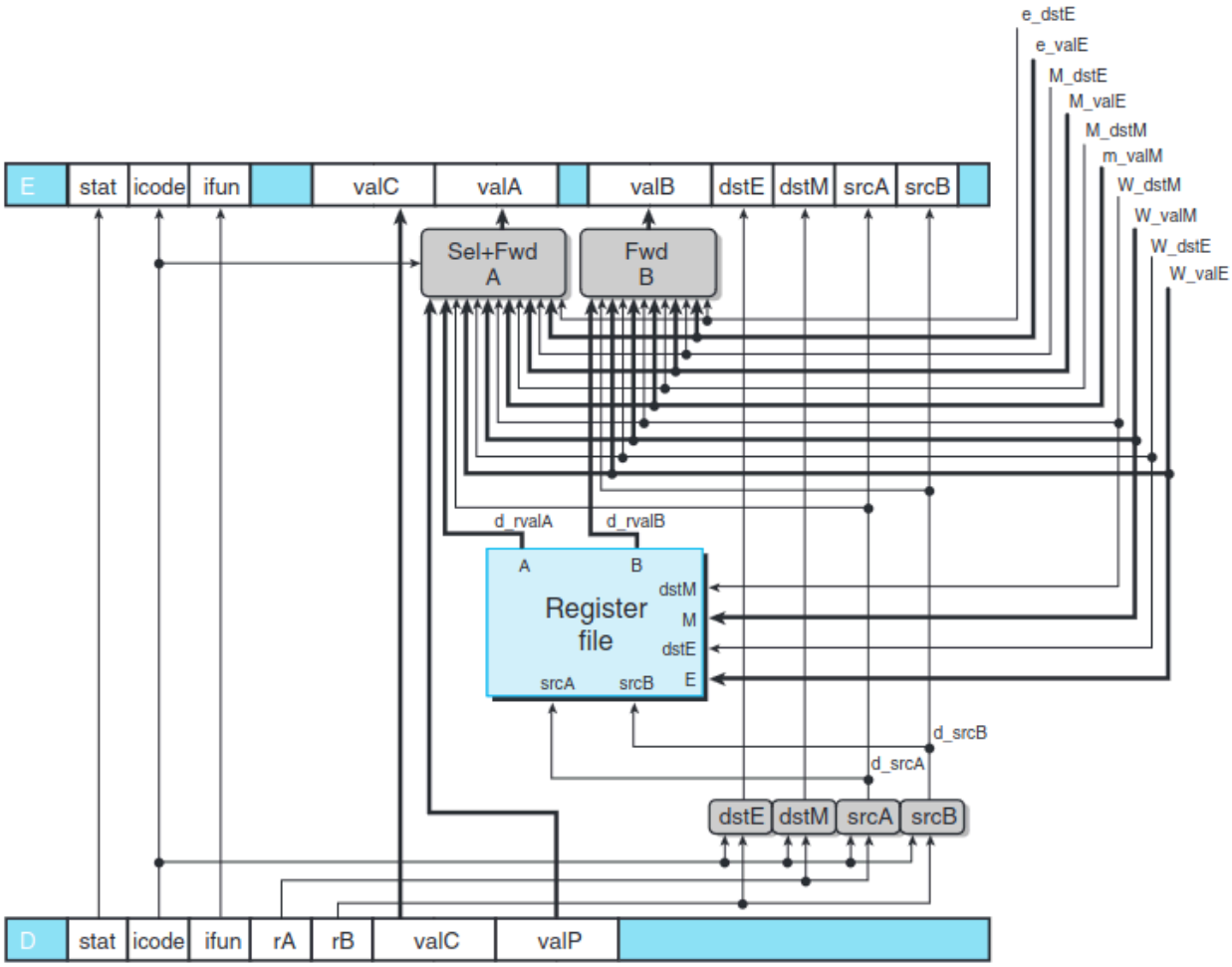
```
word f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Cnd : M_valA;
        # Completion of RET instruction
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];
```

```
word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];
```

f_stat is also generated providing the provisional status for the fetched instruction.

```
word f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];
```

DECODE & WRITE BACK : The blocks labeled dstE, dstM, srcA, and srcB closely resemble their equivalents in the SEQ implementation. However, instead of the decode stage, register IDs for write ports come from the write-back stage (signals W_dstE and W_dstM) to ensure writes occur to the specified destination registers during the write-back stage.



To optimize pipeline efficiency, valP (required only by call and jump) and the value read from register port A are merged into a single signal called valA for subsequent stages. The "Sel+Fwd A" block consolidates valP and valA and handles forwarding logic for source operand valA. Priority among the five forwarding sources is crucial and is determined by the order in which destination register IDs are tested.

```
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];
```

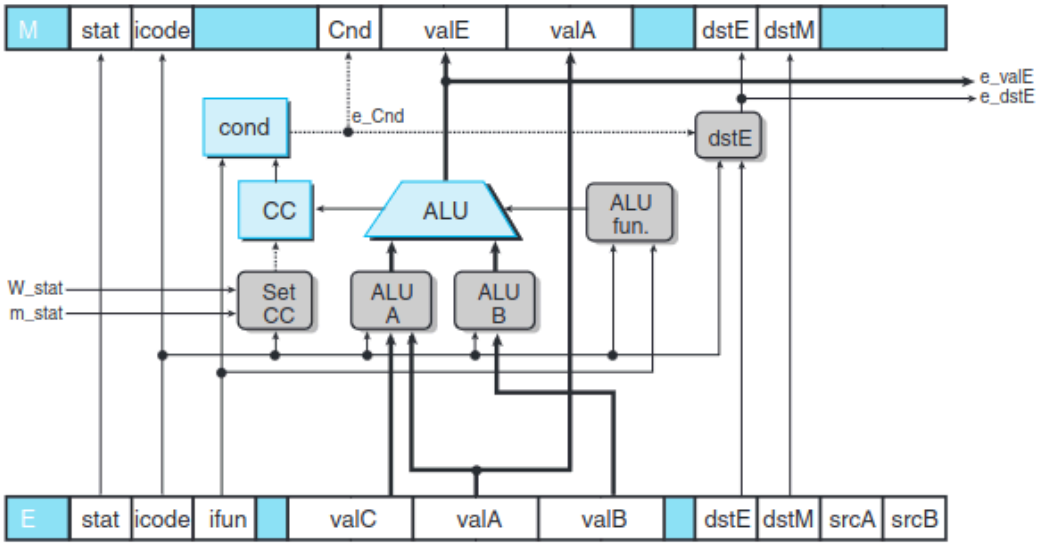
While the "Fwd B" block manages forwarding logic for source operand valB.

```
word d_valB = [
        d_srcB == e_dstE : e_valE;    # Forward valE from execute
        d_srcB == M_dstM : m_valM;    # Forward valM from memory
        d_srcB == M_dstE : M_valE;    # Forward valE from memory
        d_srcB == W_dstM : W_valM;    # Forward valM from write back
        d_srcB == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalB;  # Use value read from register file
];
```

Processor status (Stat) is computed based on the status value in pipeline register W, reflecting the overall processor status. This is logical as pipeline register W holds the state of the most recently completed instruction. However, special consideration is required when there's a bubble in the write-back stage.

```
word Stat = [
        W_stat == SBUB : SAOK;
        1 : W_stat;
];
```

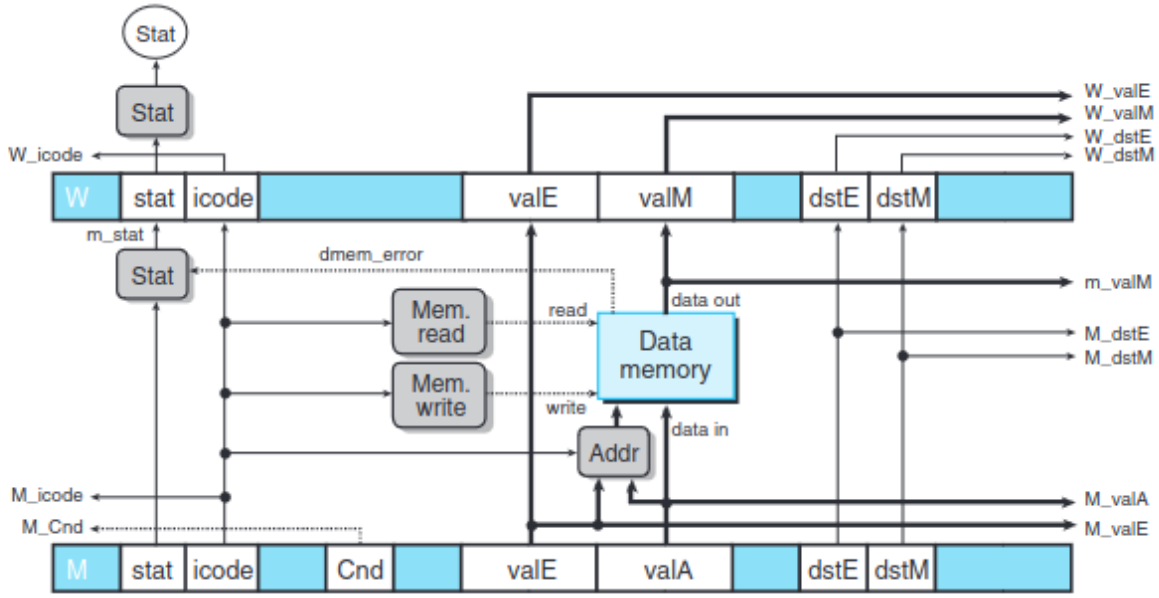
EXECUTE : The hardware units and logic blocks remain the same as in SEQ, with signals renamed accordingly. Notably, signals like e_valE and e_dstE are directed towards the decode stage as forwarding sources.



One distinction is found in the "Set CC" logic, which decides whether to update condition codes. It takes inputs from m_stat and W_stat signals to identify situations where an instruction causing an exception is progressing through later pipeline stages, hence preventing any updates to the condition codes.

```
bool set_cc = E_icode == IOPQ &&
       # State changes only during normal operation
       !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };
```

MEMORY : Comparing the memory stage of PIPE to that of SEQ (Figure 4.30), we observe that the "Mem. data" block present in SEQ, responsible for selecting between data sources valP and valA, is absent in PIPE. Instead, this selection is now handled by the "Sel+Fwd A" block in the decode stage. Most other blocks in this stage remain unchanged from SEQ, albeit with signal renaming.



Furthermore, many values in pipeline registers and M and W are utilized in other parts of the circuit for forwarding and pipeline control logic. In this stage, the computation of the status code Stat is finalized by detecting cases of invalid memory addresses for the data memory.

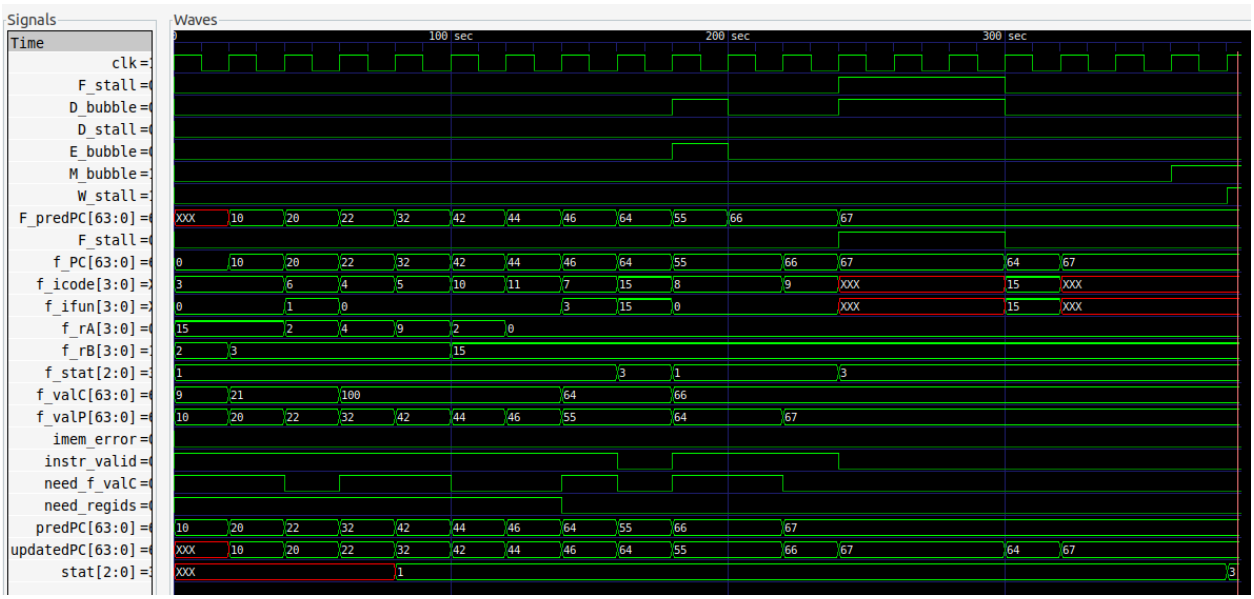
```
word m_stat = [
        dmem_error : SADR;
        1 : M_stat;
];
```

<u>Test case</u> - each clock cycle is 20 sec long

Input 1: Running the same test case from sequential. This case has no load/use hazard.

Output 1

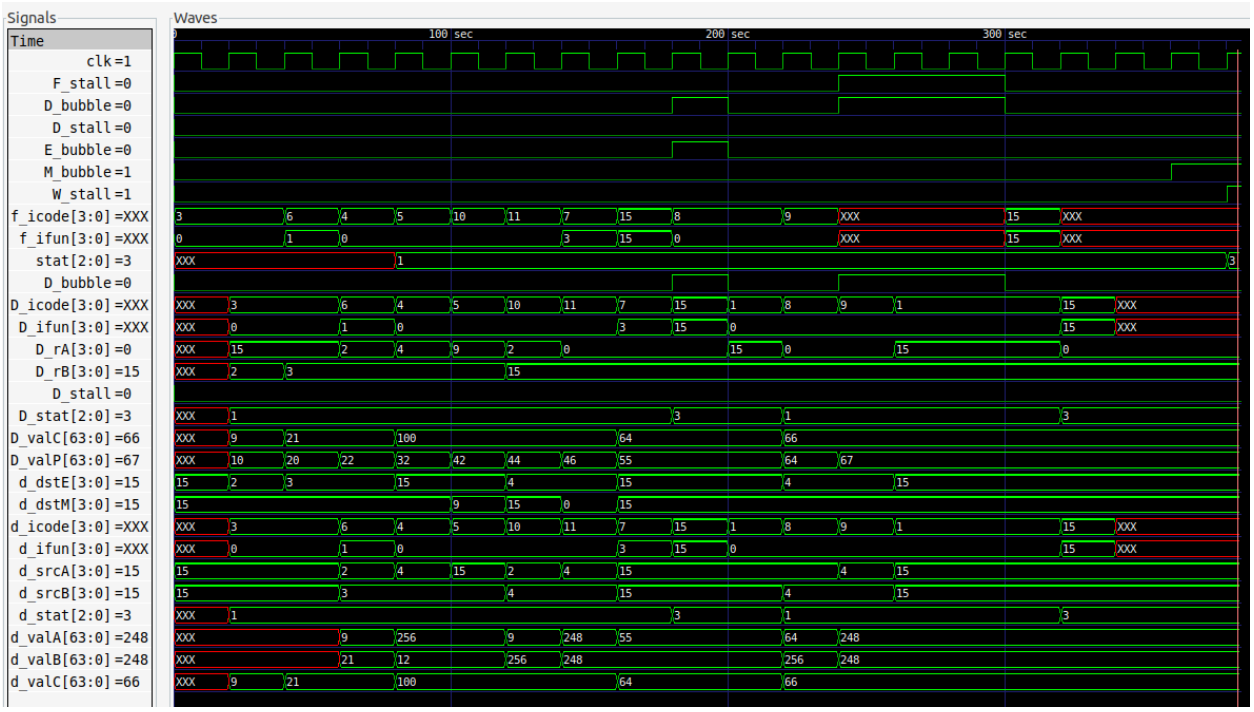
<u>Fetch:</u>



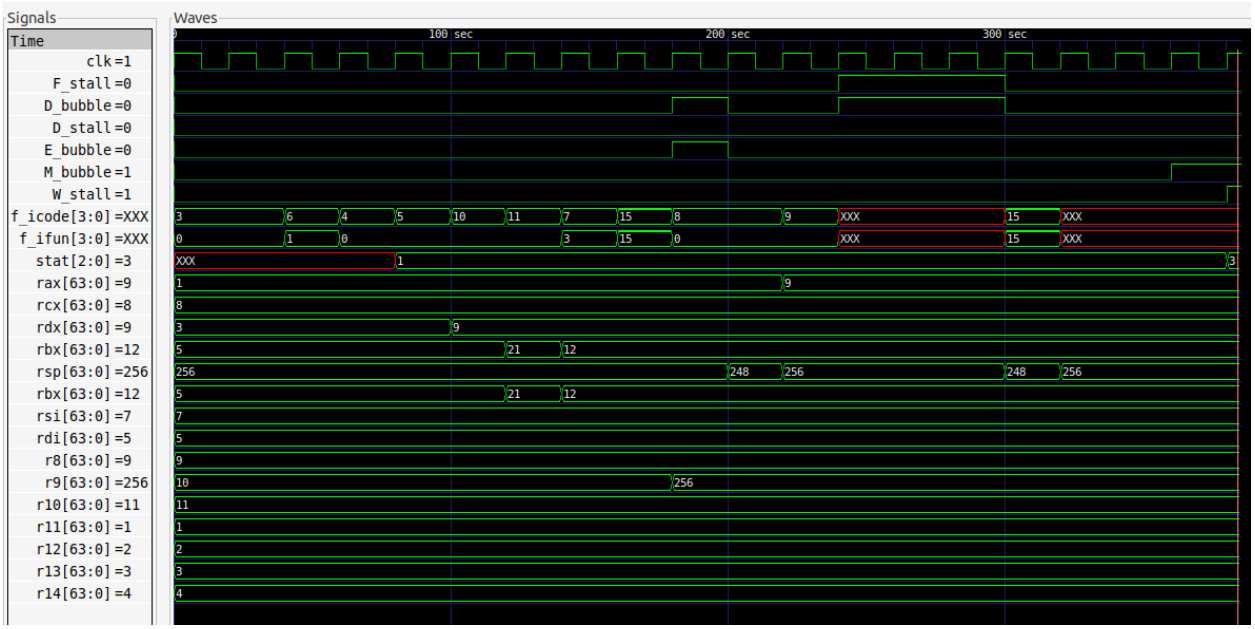
Until f_icode becomes 7, no hazards are introduced and the program runs smoothly.By PC prediction for jXX instruction, the updated PC would be f_valC which would lead to a mispredicted branch, which happens at 140 sec in the plot above. However, due to the pipeline control logic this is fixed by inserting a bubble in the decode and the execute stage which happens at 180 sec as a misprediction can only be realized in the execute cycle. This prevents the wrongly chosen instruction FF after jump to not be executed. If it was executed, it would have caused the program to terminate as Stat = Invalid instruction.

At 240 sec, a return instruction is executed and we observe stall in F and bubble in D stage to ensure ret reaches Write Back stage and we get the address for our next instruction(here = 64), which has an invalid instruction and thus the program terminates once it reaches write back stage(Exception handling). Two cycles after the excepting instruction was encountered, a bubble in M stage and in the next cycle stall in W occurs.

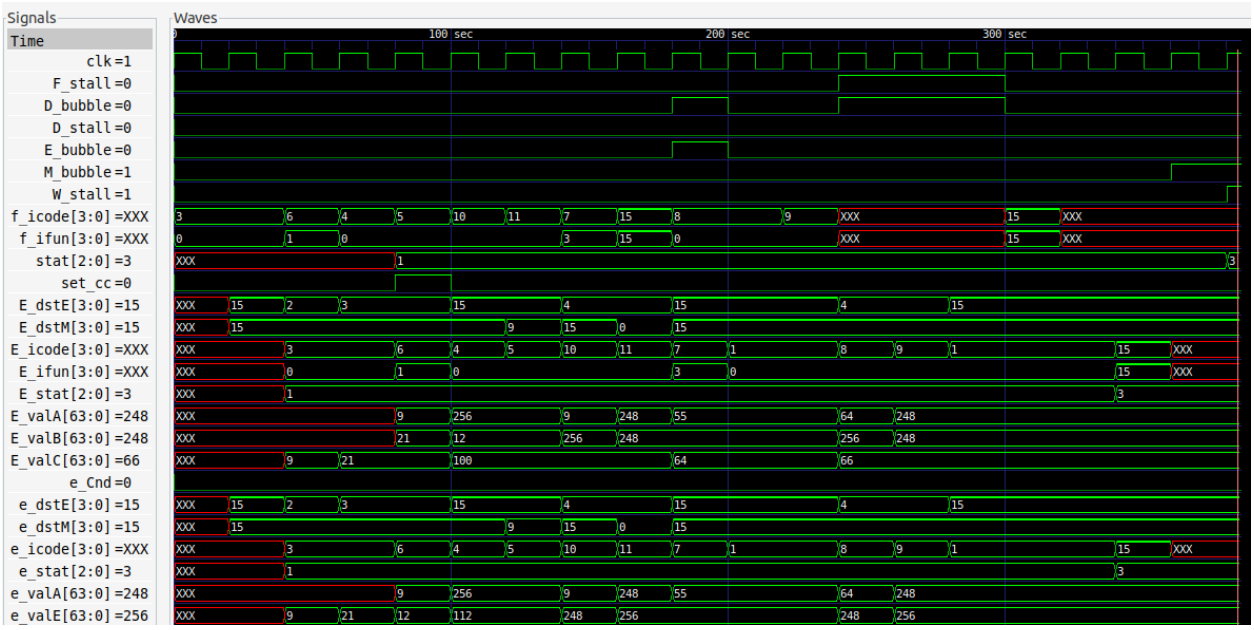
Decode:



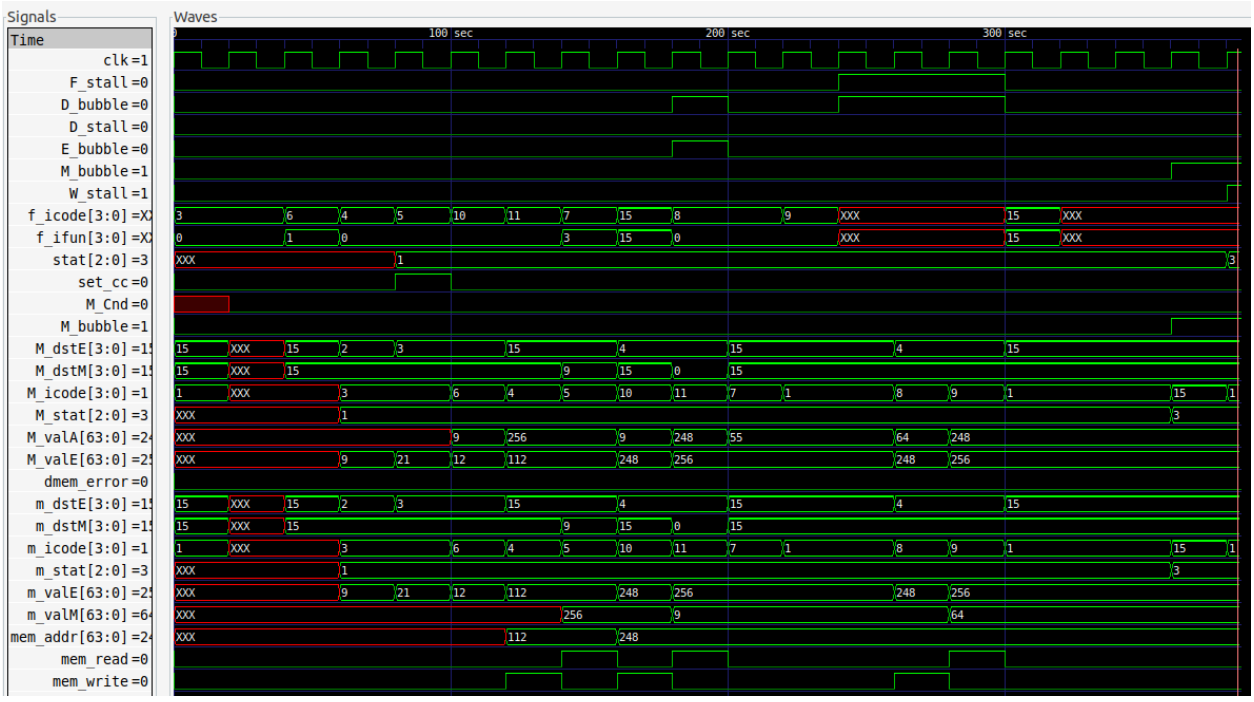
Register file



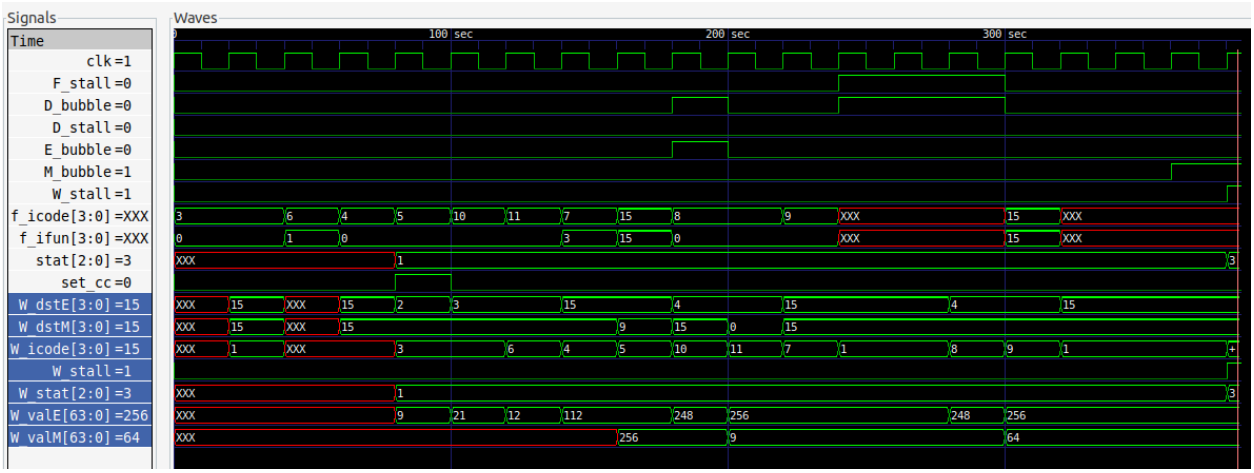
Execute:



Memory:



Write Back:


Signals
Waves
Time
clk =1
F_stall =0
D_bubble =0
D_stall =0
E_bubble =0
M_bubble =1
W_stall =1
f_icode[3:0] =XXX
f_ifun[3:0] =XXX
stat[2:0] =3
set_cc =0
W_dstE[3:0] =15
W_dstM[3:0] =15
W_icode[3:0] =15
W_stall =1
W_stat[2:0] =3
W_valE[63:0] =256
W_valM[63:0] =64
100 sec
200 sec
300 sec

Input 2:Load/use hazard

Instructions- irmovq $9,rdx

irmovq $15,rcx

irmovq $15,rbx

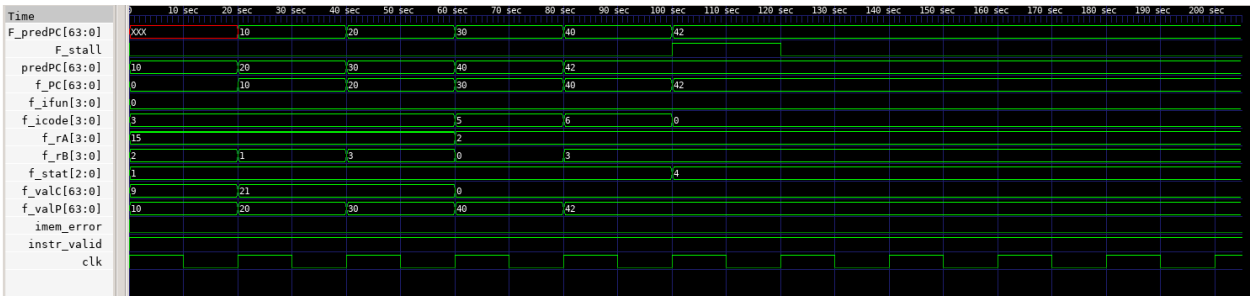
mrmovq 0(rdx),rbx

addq rdx,rbx

Halt

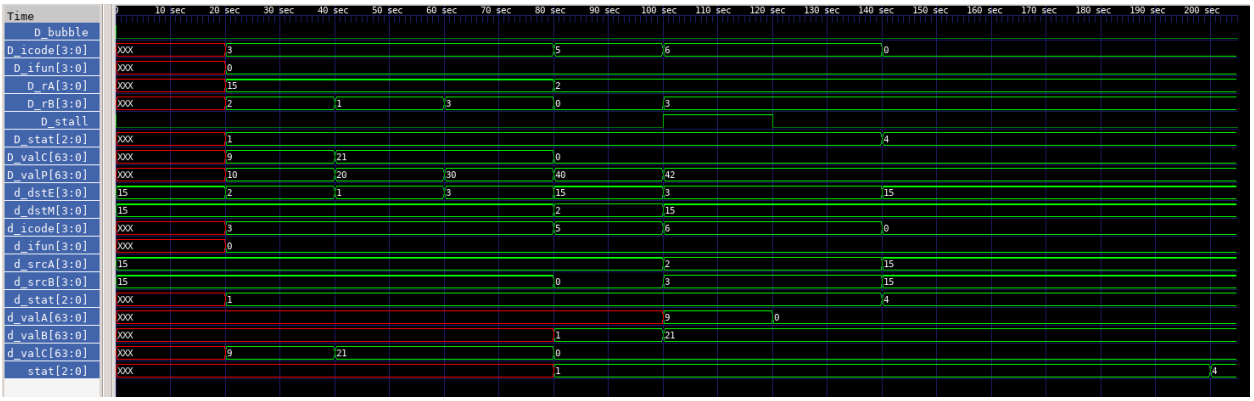
Output 2:

Fetch-



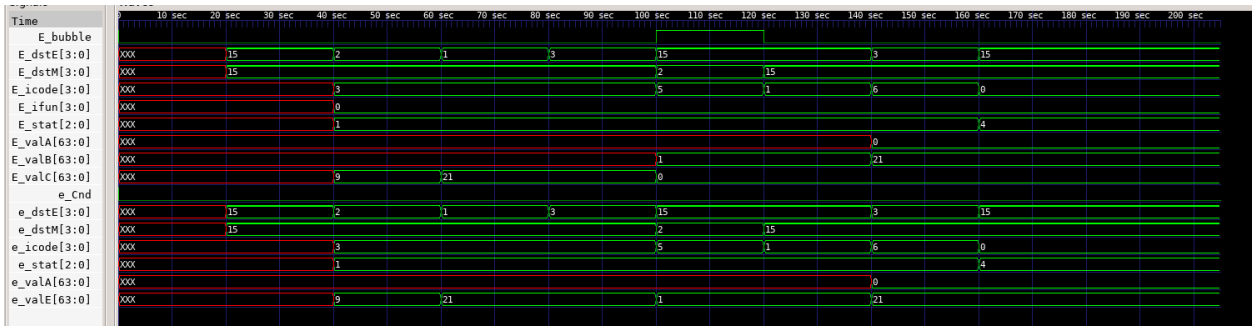
Since irmovq instruction is fetched f_icode remains 3 for 3 clock cycles. rA is F since no register for immediate value.rb is 2,1,3 as rdx,rcx,rbx are destination registers.Load use hazard causes no change in order of instructions in fetch therefore after every clock cycle as PC is updated we get the new fetched instruction.

Decode-



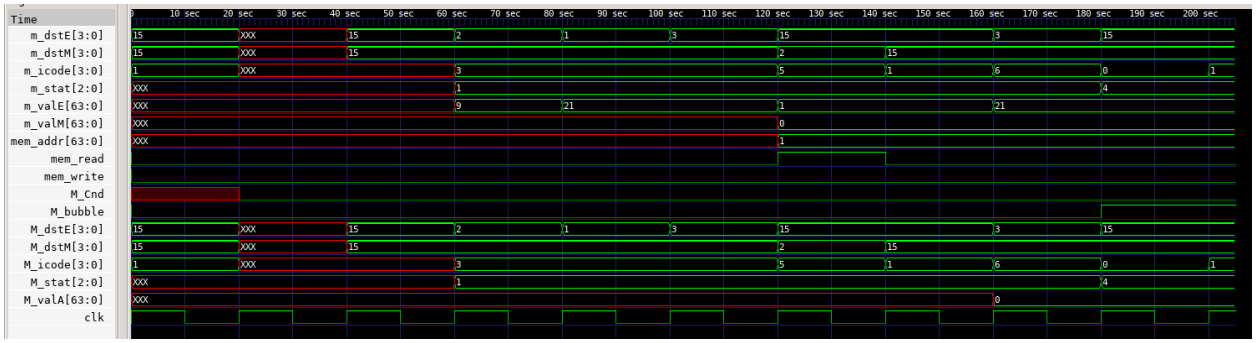
For the first clock cycle since we dont have any execution in decode the values remain x. On posedge clk all D_ values from decode register are updated.We know for load hazard since addq instruction wont read as the value hasnt been read from memory ,hence we should stall the decode stage for 1 cycle so that read from memory takes place.Here we see D_icode stays 6 for another clock cycle as the decode has been stalled.Normal halt instruction is read after this cycle.

Execute-

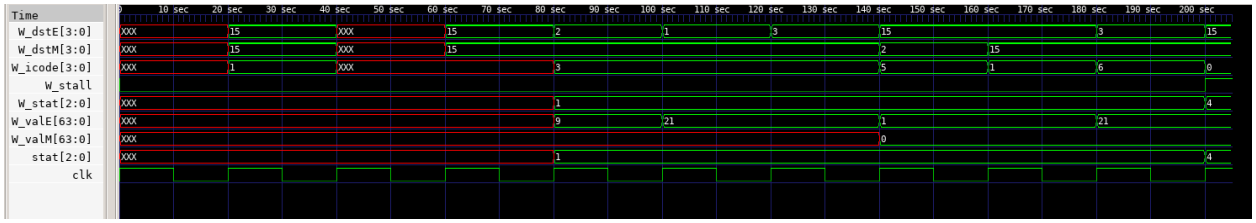


When a bubble enters the execute stage , we execute a nop instruction in the next clock cycle .

Memory-



Write-back-



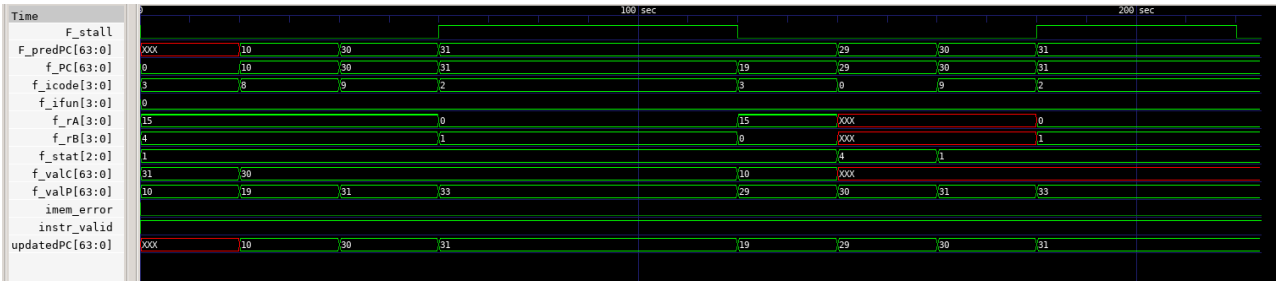
Input 3: Processing Return

Testcase

```
0x000:    irmovq stack,%rsp  #   Initialize stack pointer
0x00a:    call proc          #   Procedure call
0x013:    irmovq $10,%rdx    #   Return point
0x01d:    halt
0x020: .pos 0x20
0x020: proc:                 # proc:
0x020:    ret                #   Return immediately
0x021:    rrmovq %rdx,%rbx   #   Not executed
0x030: .pos 0x30
0x030: stack:                # stack: Stack pointer
```

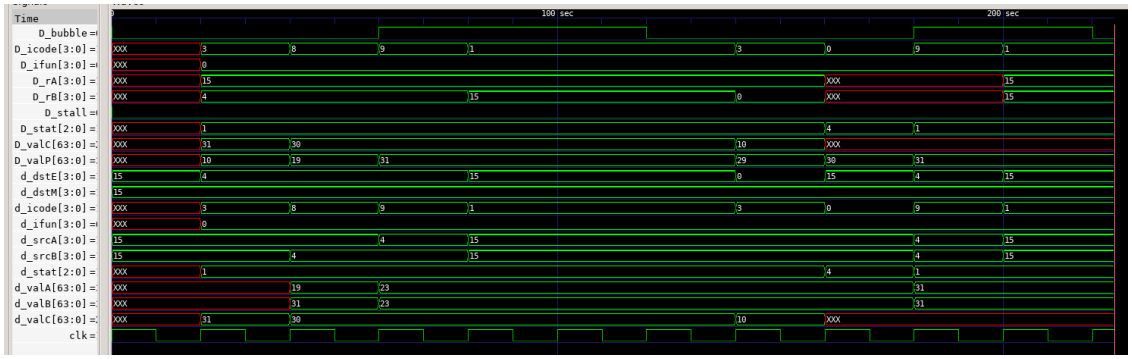
Output 3:

Fetch-



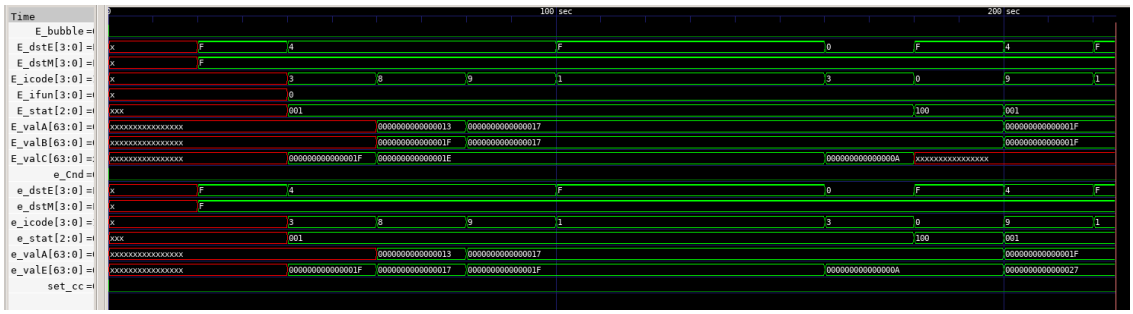
When a return is encountered we stall fetch and add bubble in decode execute memory. Hence when f_icode =9 (return) F_stall goes to 1 in the next posedge clock cycle and the next instruction written after fetch rrmovq is stalled for next 3 clock cycles until write back takes place.All the values in fetch remain constant for the next 3 clock cycles. Another thing to note was when the f_icode goes to 0 or halt. We still fetch the instructions after it . The f_PC increment takes place hence again the return is encountered and Fstall becomes 1. However when the halt f_icode is written back W_stat is updated and the final stat goes to halt the processor execution stops.

Decode-

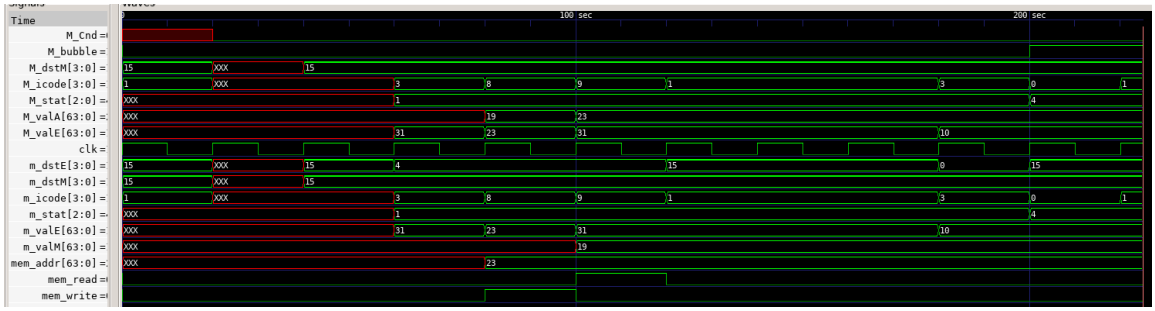


We see that D_bubble is 1 when D_icode becomes 9 or when return is processed in decode stage.A nop instruction is executed in the next clock cycle in decode stage for the next 3 clock cycles.

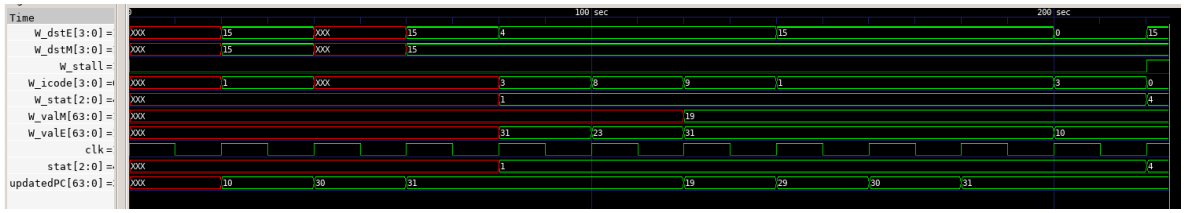
Execute-



Memory-



Write-back-

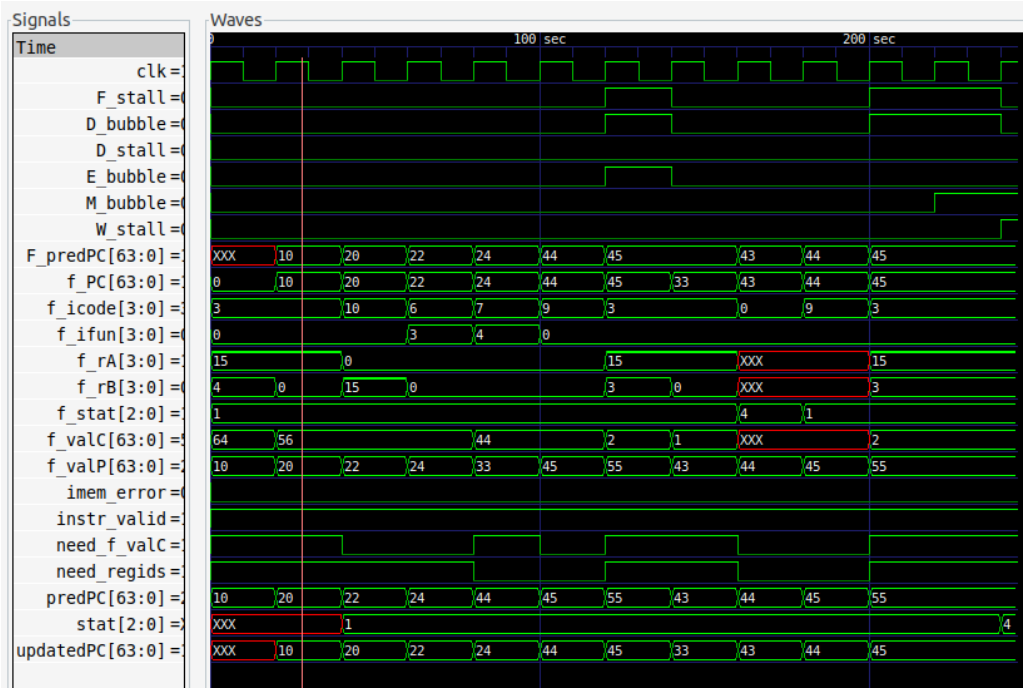


Input 4 : Control Combination A

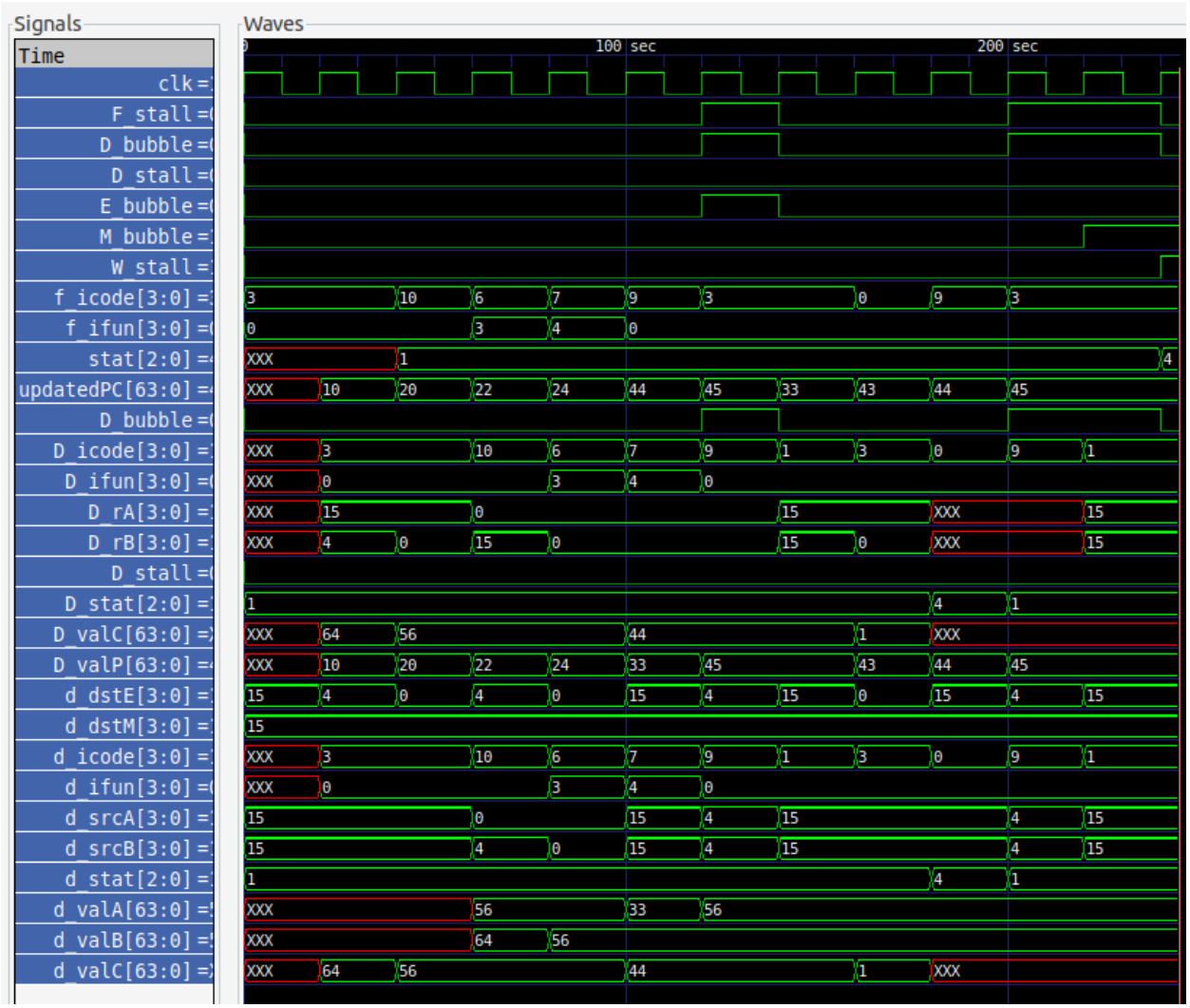
0x0000: 30f44000000000000000 | irmovq Stack, %rsp
0x000a: 30f03800000000000000 | irmovq rtnp,%rax
0x0014: a00f | pushq %rax # Set up return pointer
0x0016: 6300 | xorq %rax,%rax # Set Z condition code
0x0018: 742c00000000000000 | jne target # Not taken (First part of combination)
0x0021: 30f00100000000000000 | irmovq $1,%rax # Should execute this
0x002b: 00 | halt
0x002c: 90 | target: ret # Second part of combination
0x002d: 30f30200000000000000 | irmovq $2,%rbx # Should not execute this
0x0037: 00 | halt
0x0038: 30f20300000000000000 | rtnp: irmovq $3,%rdx # Should not execute this
0x0042: 00 | halt
0x0043: | .pos 0x40
0x0040: | Stack:

This program is designed so that if something goes wrong (for example, if the ret instruction is actually executed), then the program will execute one of the extra irmovq instructions and then halt. Thus, an error in the pipeline would cause some register to be updated incorrectly. Clearly neither rax nor rdx are updated.

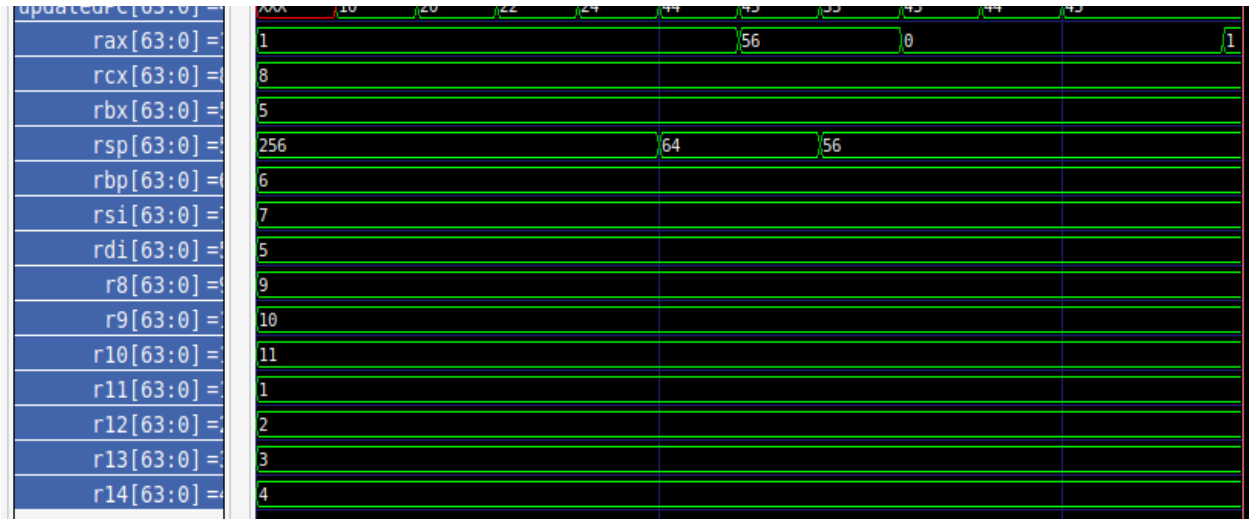
Output 4:
Fetch



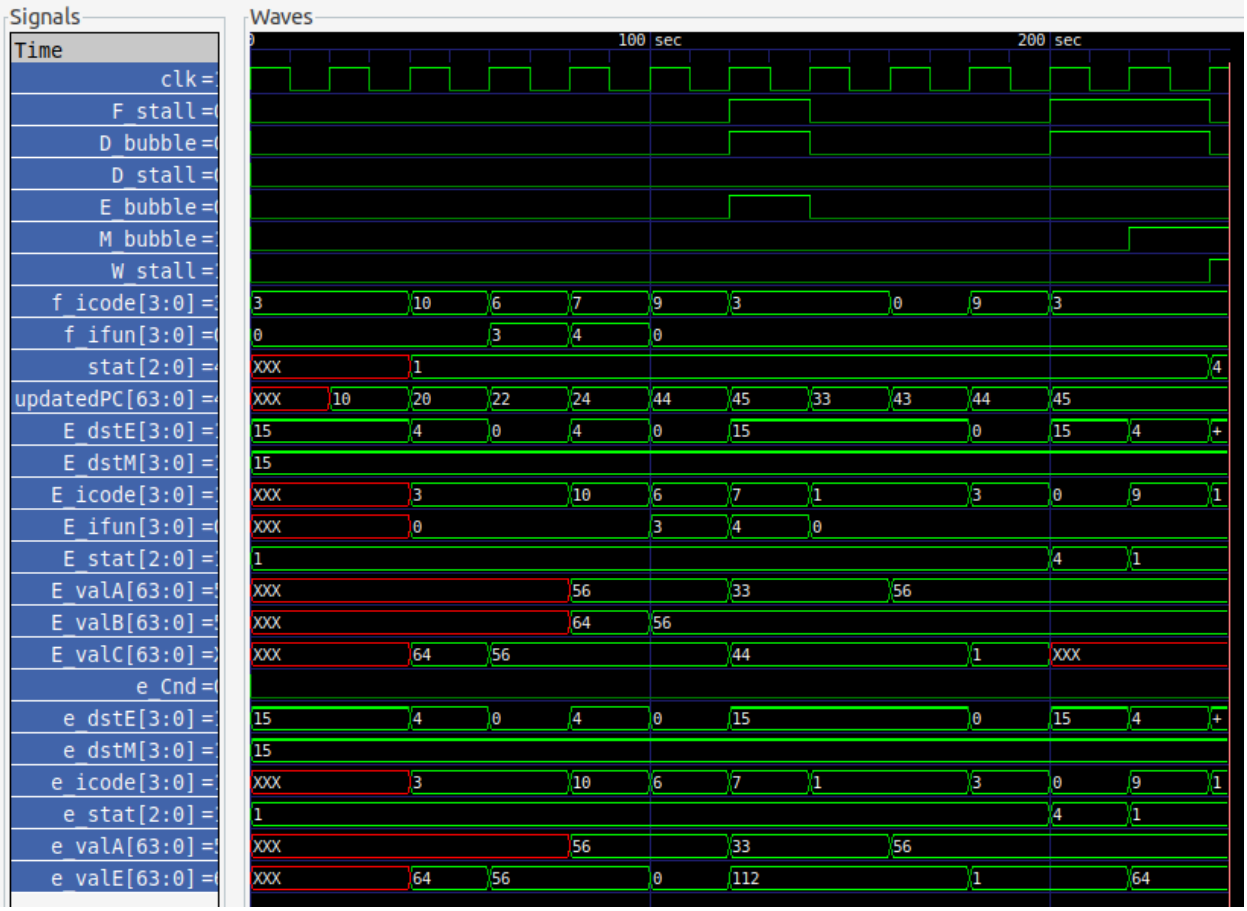
Decode-



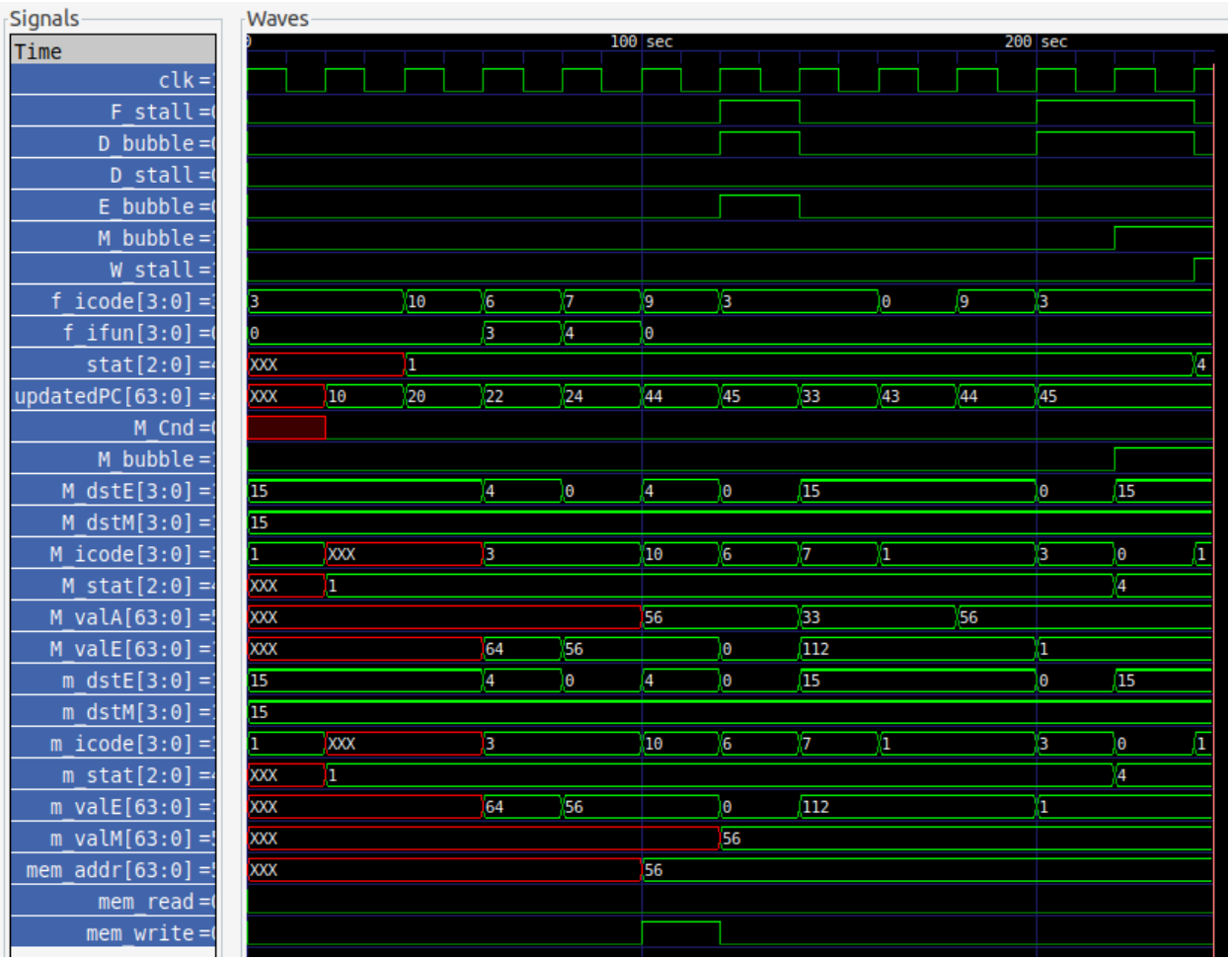
Register file-



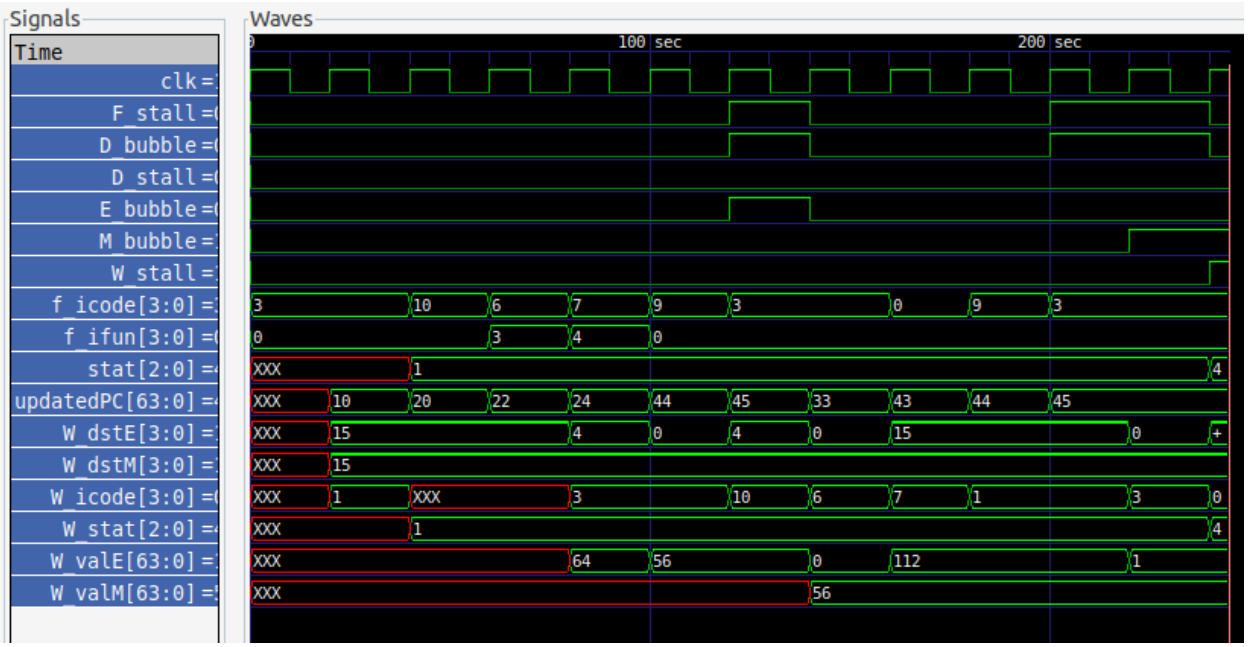
Execute-


Signals
Waves
Time
clk
F_stall
D_bubble
D_stall
E_bubble
M_bubble
W_stall
f_icode[3:0]
f_ifun[3:0]
stat[2:0]
updatedPC[63:0]
E_dstE[3:0]
E_dstM[3:0]
E_icode[3:0]
E_ifun[3:0]
E_stat[2:0]
E_valA[63:0]
E_valB[63:0]
E_valC[63:0]
e_Cnd
e_dstE[3:0]
e_dstM[3:0]
e_icode[3:0]
e_stat[2:0]
e_valA[63:0]
e_valE[63:0]
100 sec
200 sec

Memory



Write back-



Challenges

1. Understanding the timing of pipeline implementation- all pipeline registers and write back stage update only on posedge while everything else updates when any input changes.
2. Passing D_stat directly to M_stat as no stage in between can cause a change in program status. This made sure all stat and set_cc conditions were timely updated for any opq instruction.
3. Also if a mispredicted branch was taking us to a invalid instruction/halt/invalid address exception we need to ensure this is not propagated to the memory or write back stage as that would insert a false bubble in memory and put a stall in W leading to incorrect results. This was fixed by putting an explicit condition in E_register that reset E_stat to 1 for a mispredicted jump that led to D_stat becoming not AOK.
4. All the hazards and their combinations were dealt with according to pipeline control logic.