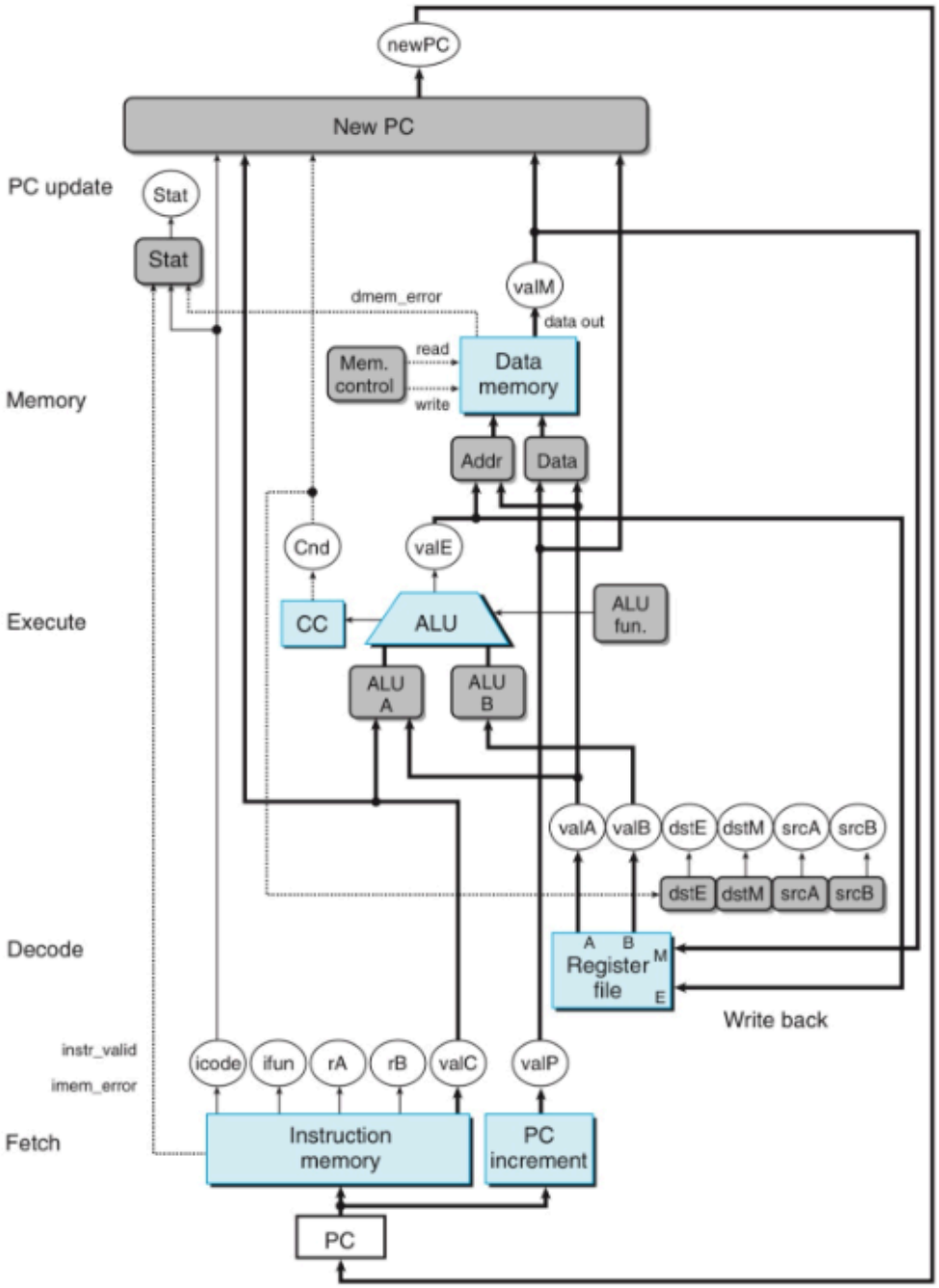
IPA Project Report

TASK 1 : 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**
   a. 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

b. 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

c. 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

d. The memory stage may write data to memory, or it may read data from memory (valM).

5 WRITE BACK

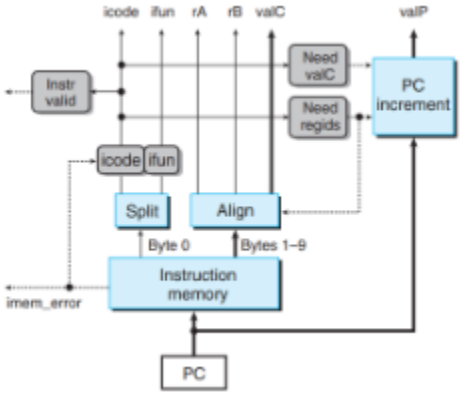
e. Writes up to two results to the register file to dstE and/or dstM.

6 PC UPDATE

f. Set the PC to the address of the next instruction.

1 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?

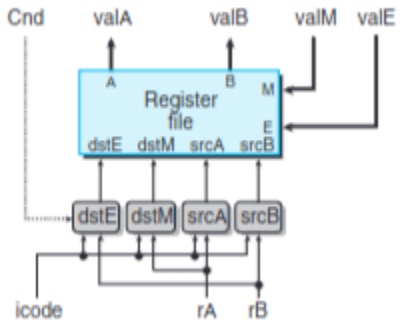
need_valC. Does this instruction include a constant word?

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.

According to these signals ,‘Align’ block splits bytes 1-9 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.

2 DECODE AND 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.

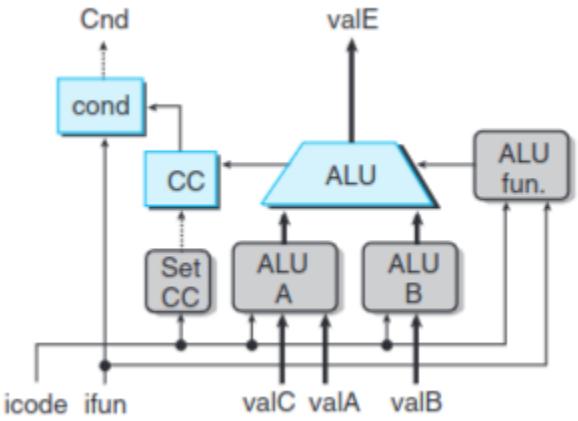


From the figure above, we can see that the register file has 4 ports - 2 read (A and B) and 2 write (E and M). Each port has an address - register ID and a data - 64 bit word connection. For the read ports the addresses are indicated by srcA and srcB and data as valA and valB respectively. Similarly, 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 cmovxx 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.

3 EXECUTE

This stage includes the ALU arithmetic logic unit which performs ADD, SUBTRACT, AND and XOR on aluA and aluB.



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

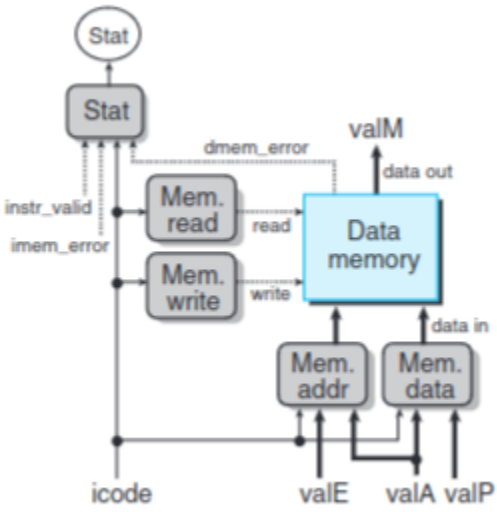
This stage also includes the condition code registers.ALU generates 3 signals based on which condition codes are based - zero flag,sign flag,overflow flag.
These flags are set only when set_cc = 1 which occurs during OPq instruction.

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

The condition code is set to 0 for all the other operations in OPq.

4 MEMORY

Reads or writes program data to memory.



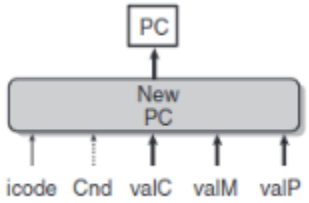
The data memory is defined in our code as a block of 1024 64-bit registers, initialized to zero.

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.

5 PC UPDATE

Generates a new value for Program Counter.



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.

SEQUENTIAL TIMING

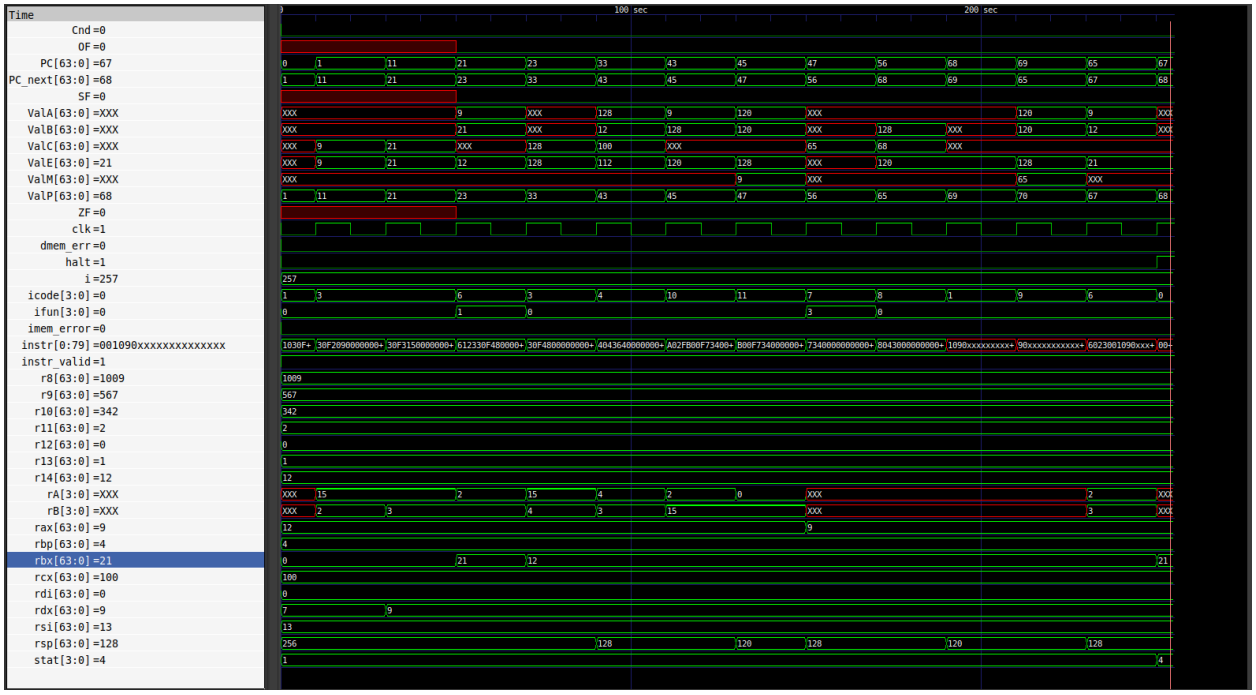
- An entire instruction is performed every clock cycle.
- Combinational logic does not require any sequencing or control from the clock and the values propagate through whenever the input changes.
- In Verilog it is denoted by always@(*). Also, the instruction memory can be treated as a combinational 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 at the posedge of the clock .
- 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 covers all the basics operations supported by Y-86 :

```
30 //irmovq
f2
09
00
00
00
00
00
00
00
30 //irmovq
f3
15
00
00
00
00
00
00
00
61 //subq
23
30 //irmovq
f4
80
00
00
00
00
00
00
00
40 //rmmovq
43
64
00
00
00
00
00
00
00
```

```
43  a0 //pushq
44  2f
45  b0 //popq
46  0f
47  73 //je
48  40
49  00
50  00
51  00
52  00
53  00
54  00
55  00
56  80 //call
57  43
58  00
59  00
60  00
61  00
62  00
63  00
64  00
65  60 //addq
66  23
67  00 //halt
68  10 //nop
69  90 //ret
```



From the above , jump doesnt take place as Cnd is set to 0 and we also see that the value of rbx is getting updated to 21 .

CHALLENGES FACED

a) Deciding the operation of stages based on the clock (posedge or negedge or *) .

b) The writeback was initially on the negedge so the result was computed first and the same was fetched again causing an error . This was later fixed by putting writeback on the posedge clk .

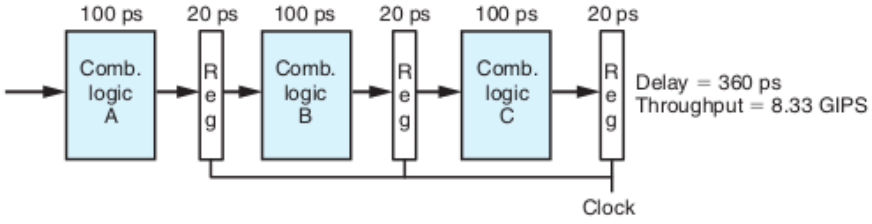
c) Dmem_error wasn't checking for ValE negative and hence wasn't flagging correctly .

d) While fetching , ValC wasn't flipped initially so all the Immediate values were fetched incorrectly .

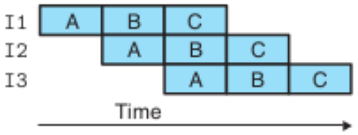
TASK 2 : PIPELINE IMPLEMENTATION

GENERAL PRINCIPLE OF PIPELINING

A key feature of pipelining is that it **increases the throughput** of the system (i.e., the number of instructions per unit time), but it may also slightly **increase the latency** (i.e., the time required to complete an entire instruction) .



(a) Hardware: Three-stage pipeline



(b) Pipeline diagram

The above one depicts a timing diagram in case of a pipelined system .
Here , the throughput is given by :

$$Throughput = \frac{1 \text{ instruction}}{(20 + 300) \text{ picoseconds}} \cdot \frac{1{,}000 \text{ picoseconds}}{1 \text{ nanosecond}} \approx 3.12 \text{ GIPS}$$

Since processing a single instruction requires 3 clock cycles, the latency of this pipeline is 3 × 120 = **360 ps.**

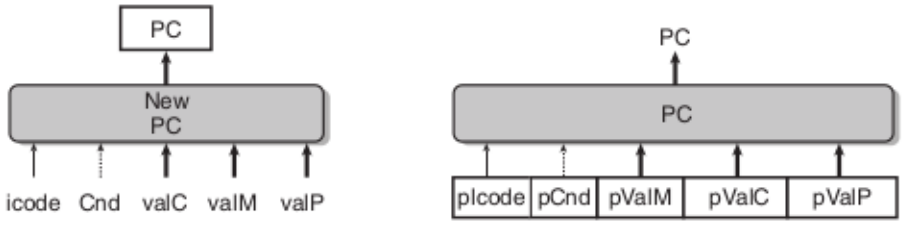
COMPARISON

We have increased the throughput of the system by a factor of 8.33/3.12 = 2.67 at the expense of some added hardware and a slight increase in the latency (360/320 = 1.12). The increased latency is due to the time overhead of the added pipeline registers.

PIPELINED Y86-64 IMPLEMENTATIONS

As a transitional step toward a pipelined design, we must slightly rearrange the order of the five stages in SEQ so that the PC update stage comes at the beginning of the clock cycle, rather than at the end.This transformation requires only minimal change to the overall hardware structure, and it will work better with the sequencing of activities within the pipeline stages. We refer to this modified design as SEQ+.

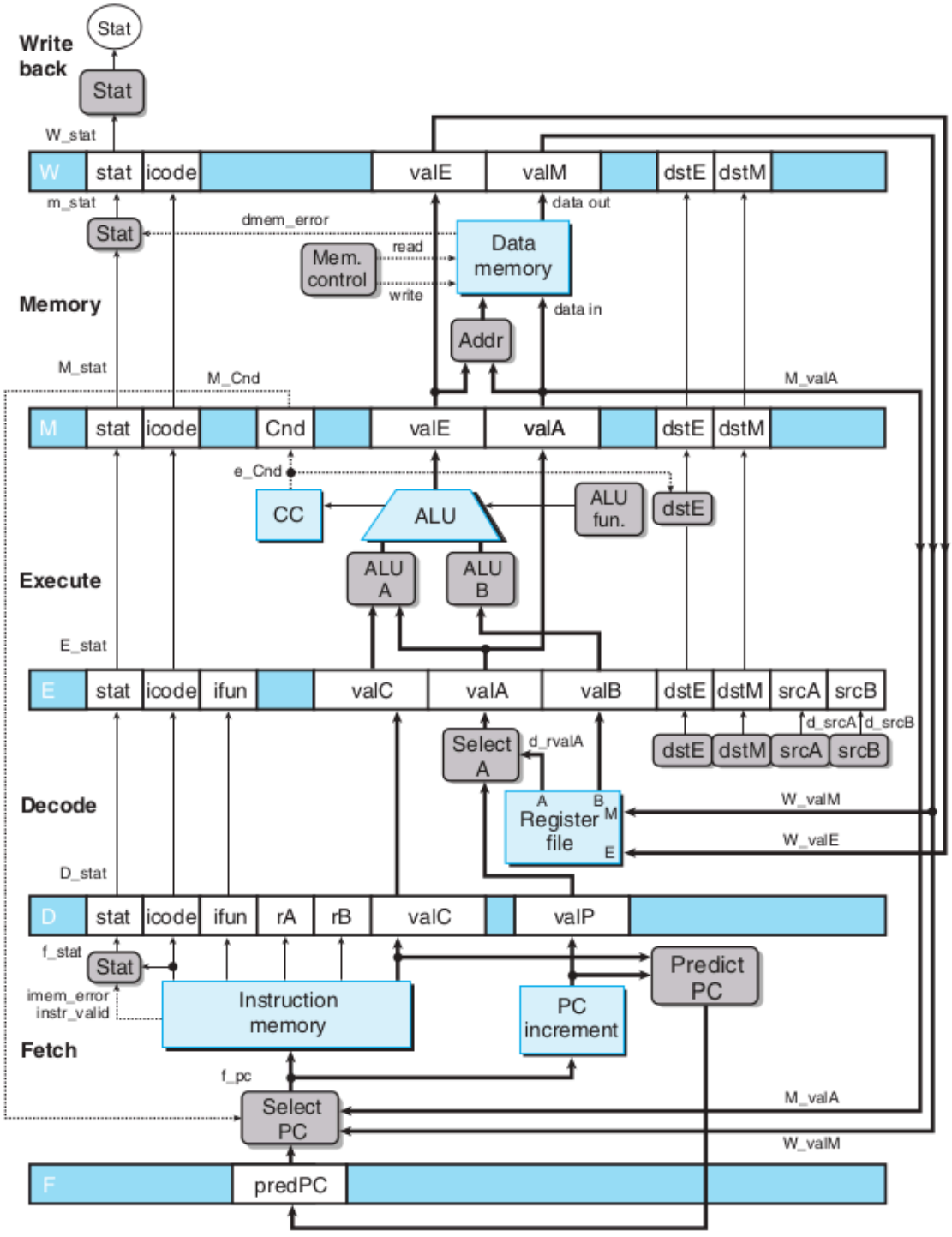
We can move the PC update stage so that its logic is active at the beginning of the clock cycle by making it compute the PC value for the current instruction. The below figure shows how SEQ and SEQ+ differ in their PC computation.



The shift of state elements from SEQ to SEQ+ is an example of a general transformation known as **Circuit Retiming .**

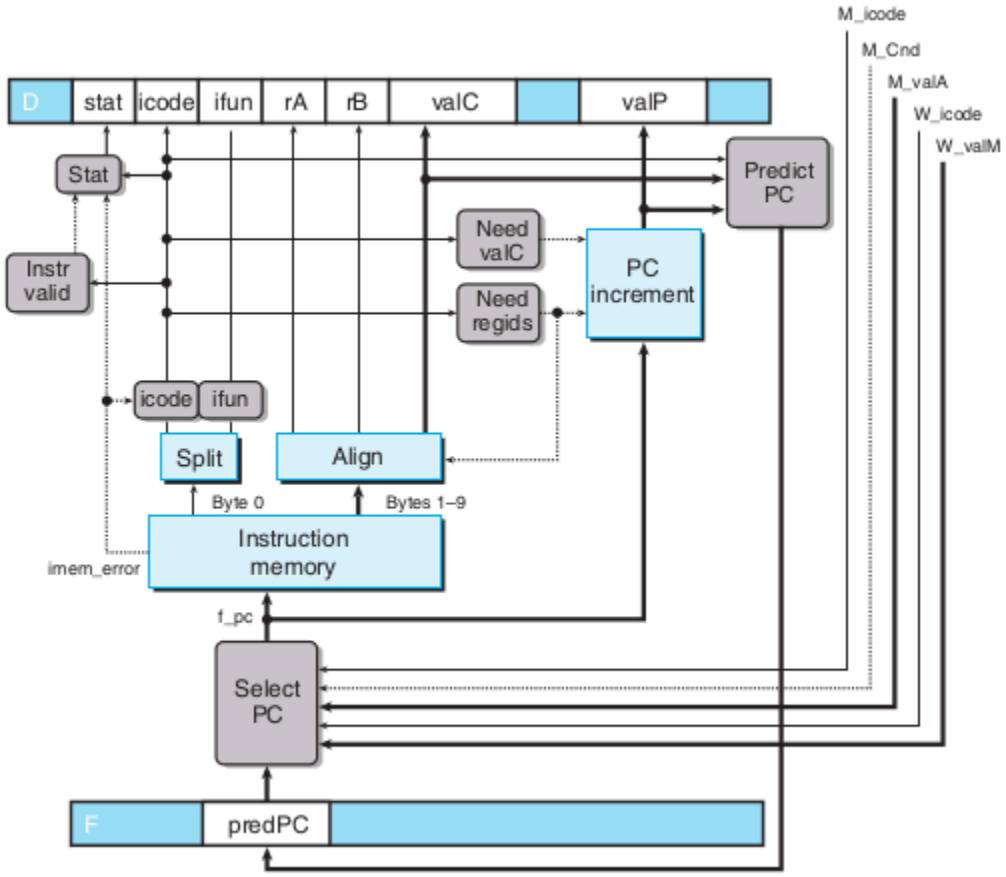
We insert 5 pipeline registers - F , D , E , M , W and construct a 5 - stage pipeline . These registers are given in capital letter and the register which are used in-stage are given by smaller letters .

HARDWARE IMPLEMENTATION



PIPE STAGE IMPLEMENTATION

1 PC SELECTION AND FETCH STAGE



This stage must also select a current value for the program counter and predict the next PC value .

The PC selection logic chooses between three program counter sources. As a mispredicted branch enters the memory stage, the value of valP for this instruction (indicating the address of the following instruction) is read from pipeline register M **(signal M_valA)** .When a ret instruction enters the write-back stage, the return address is read from pipeline register W (signal

W_valM). All other cases use the predicted value of the PC, stored in pipeline register F (**signal F_predPC**).

```
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;
];
```

The PC prediction logic chooses **valC** for the fetched instruction when it is either a **call or a jump, and valP** otherwise

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

The logic blocks labeled “Need regids,” and “Need valC” are the same as for SEQ and is used to compute the f_valP .

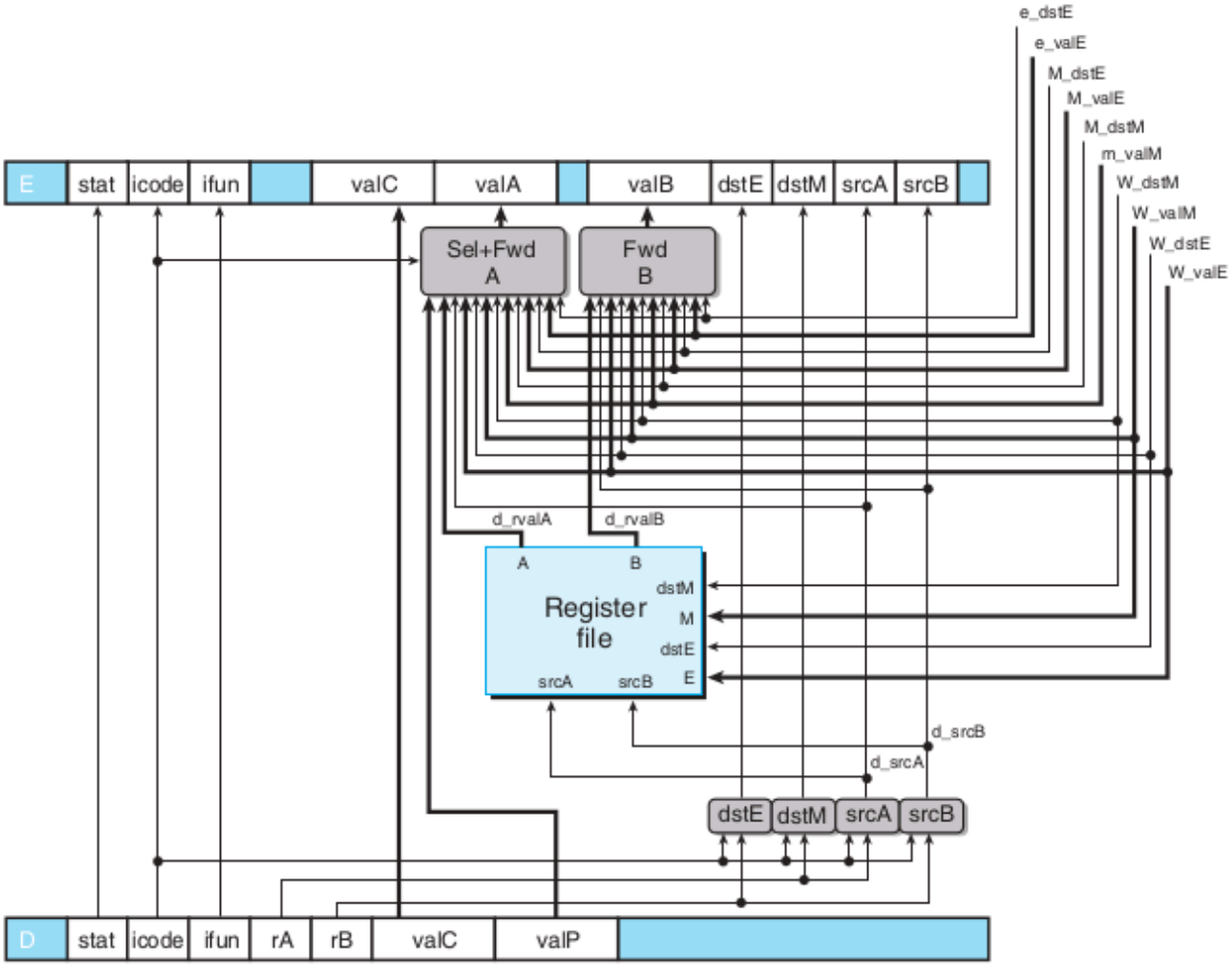
We split the computation of the instruction status intotwo parts. In the fetch stage, we can test for a memory error due to an out-of-range instruction address, and we can detect an illegal instruction or a halt instruction. Detecting an invalid data address must be deferred to the memory stage.

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

2 DECODE AND WRITEBACK

The below figure gives a detailed view of the decode and write-back logic for PIPE. The blocks labeled dstE, dstM, srcA, and srcB are very similar to their counterparts in the implementation of SEQ.

Note that the register IDs supplied to the write ports come from the write-back stage (signals W_dstE and W_dstM), rather than from the decode stage. This is because we want the writes to occur to the destination registers specified by the instruction in the write-back stage



The block labeled **“Sel+Fwd A”** serves two roles.

a) It **merges the valP signal into the valA** signal for later stages in order to reduce the amount of state in the pipeline register. The merging of signals valA and valP exploits the fact that only the call and jump instructions need the value of valP in later stages, and these instructions do not need the value read from the A port of the register file.

b) It also implements the **forwarding logic** for source operand valA.There are five different forwarding sources,each with a data word and a destination register ID . If none of the forwarding conditions hold, the block should select d_rvalA, the value read from register port A, as its output

Data word	Register ID	Source description
e_valE	e_dstE	ALU output
m_valM	M_dstM	Memory output
M_valE	M_dstE	Pending write to port E in memory stage
W_valM	W_dstM	Pending write to port M in write-back stage
W_valE	W_dstE	Pending write to port E in write-back stage

For source A :

```
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
];
```

For source B :

The order of priority between these multiple forwarding sources is of prime importance to handle .This priority is determined in the HCL code by the order in which the five destination register IDs are tested.

If any order other than the one shown were chosen, the pipeline would behave incorrectly for some programs.To imitate this behavior, our pipelined implementation should always **give priority to the forwarding source in the earliest pipeline stage**, since it holds the latest instruction in the program sequence setting the register. Thus, the logic in the HCL code above first tests the forwarding source in the **execute stage**, then those in the **memory stage,** and finally the sources in the **write-back stage.**

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.

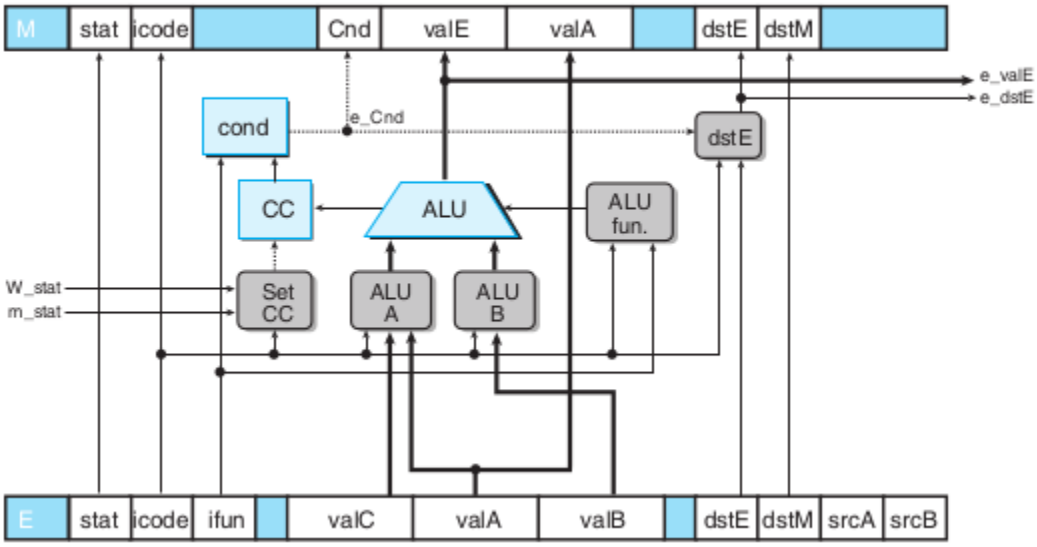
```
word Stat = [
        W_stat == SBUB : SAOK;
        1 : W_stat;
];
```

3 EXECUTE

The below figure shows the execute stage logic for PIPE. The hardware units and the logic blocks are identical to those in SEQ, with an appropriate renaming of signals.

We can see the signals **e_valE and e_dstE** directed toward the decode stage as one of the **forwarding sources**. One difference is that the logic

labeled “Set CC,” which determines whether or not to update the condition codes, has signals m_stat and W_stat as inputs. These signals are used to detect cases where an instruction causing an exception is passing through later pipeline stages, and therefore any updating of the condition codes should be suppressed.

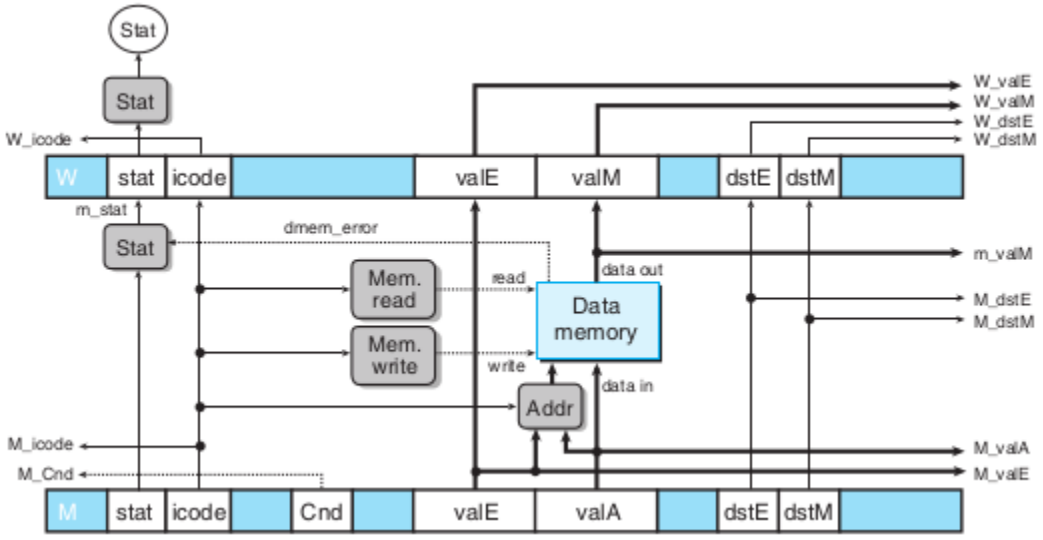


The conditional Codes will get updated only if the operation taking place is an OPq and both m_stat and W_stat doesn’t show up any errors

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

4 MEMORY

The below figure shows the memory stage logic for PIPE. Comparing this to the memory stage for SEQ , we see that, as noted before, the block labeled **“Mem data”** in SEQ is not present in PIPE. This block served to select between data sources valP (for call instructions) and valA, but this selection is now performed by the block labeled **“Sel+Fwd A”** in the decode stage.

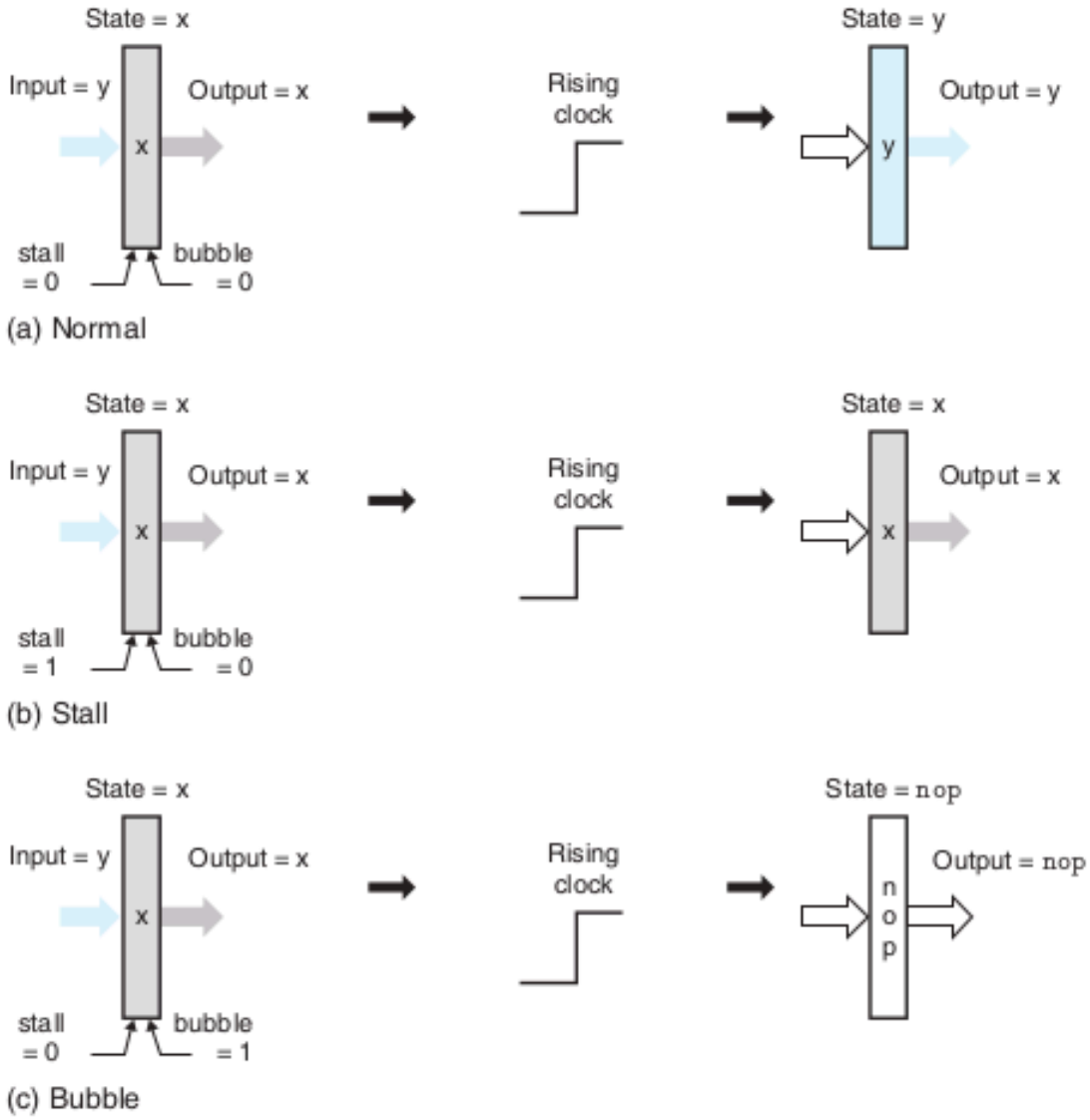


Most other blocks in this stage are identical to their counterparts in SEQ, with an appropriate renaming of the signals. In this figure, you can also see that many of the values in pipeline registers and M and W are supplied to other parts of the circuit as part of the **forwarding and pipeline control logic.**

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

PIPELINE CONTROL MECHANISM

The below figure shows low-level mechanisms that allow the pipeline control logic to hold back an instruction in a pipeline register or to inject a bubble into the pipeline.

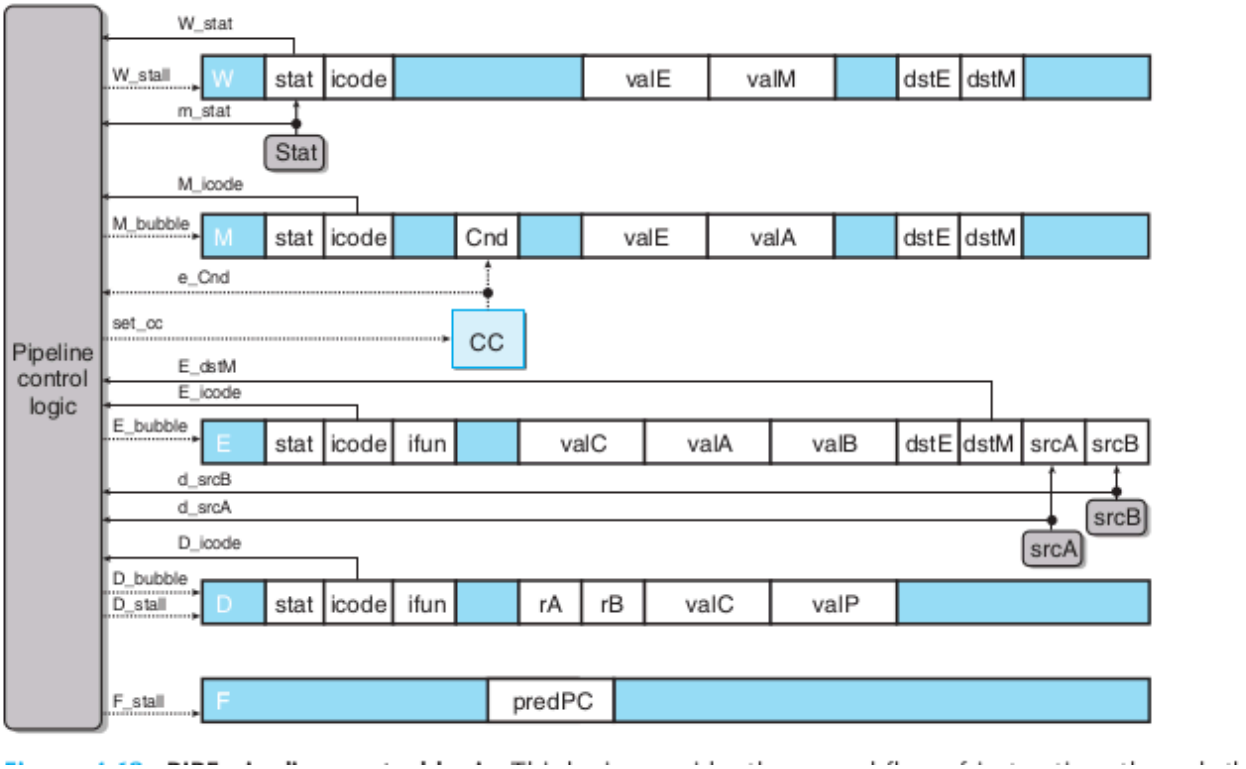


3 CASES

a) Normal : Under normal operation , both of these inputs are set to 0, causing the register to load its input as its new state.

b) Stall: When the stall signal is set to 1 , the updating of the state is disabled. Instead, the register will remain in its previous state. This makes it possible to hold back an instruction in some pipeline stage .

c) Bubble : When the bubble signal is set to 1 , the state of the register will be set to some fixed reset configuration, giving a state equivalent to that of a nop instruction.



Based on signals from the pipeline registers and pipeline stages, the control logic generates stall and bubble control signals for the pipeline registers and also determines whether the condition code registers should be updated.

1) **F_stall : Load/Use Hazard (or) Ret**

```
bool F_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB } ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
```

2) D_bubble : Mispredicted Jump (or) Ret but not load/use

```
bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
          IRET in { D_icode, E_icode, M_icode };
```

3) E_bubble : Mispredicted Jump (or) load/use

```
bool E_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB};
```

4) D_stall : Load / Use hazard

```
bool D_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB };
```

5) M_bubble : Exception detection

```
# Start injecting bubbles as soon as exception passes through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
```

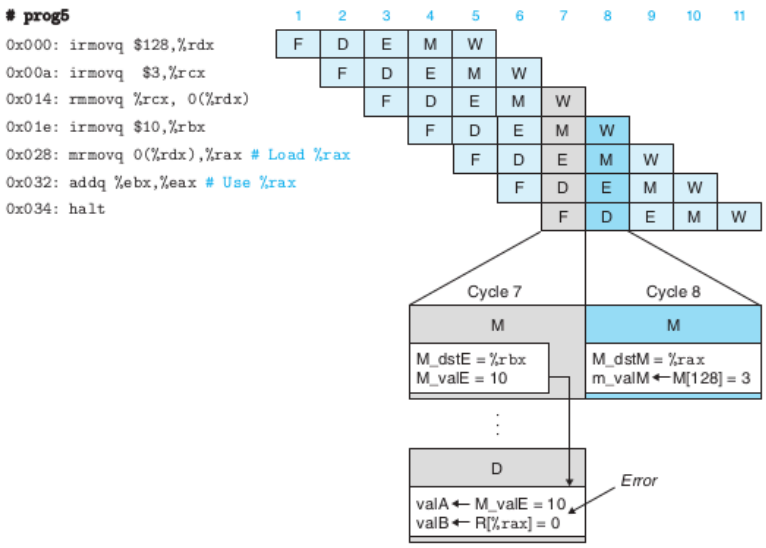
6) W_stall : Checking status code of WB stage

```
bool W_stall = W_stat in { SADR, SINS, SHLT };
```

PIPELINE CONTROL LOGIC

Certain logics can't be just handled by normal methods .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 HAZARD



The value of rax has to be fetched from memory and therefore it must stall for one cycle .
This use of a stall to handle a load/use hazard is called a **load interlock.**

The pipeline must stall for one cycle between an instruction
that reads a value from memory and an instruction that uses this value.

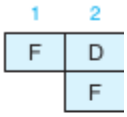
Load/use hazard	E_icode ∈ {IMRMOVQ, IPOPQ} && E_dstM ∈ {d_srcA, d_srcB}				
Load/use hazard	stall	stall	bubble	normal	normal

TEST CASE

INSTRUCTION MEMORY

In assembly :

```
# prog5
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
0x032: addq %ebx,%eax # Use %rax
0x034: halt
```



```
30  //irmovq
f2
80
00
00
00
00
00
00
00
30  //irmovq
f1
03
00
00
00
00
00
00
00
40 //rmmovq
12
00
00
00
00
00
00
00
30 //irmovq
f3
0A
00
00
00
00
00
00
00
50 //mrmovq
02
00
00
00
00
00
00
00
00
60 //addq
30
00 //halt

```

In this test case ,

1)rdx first gets the value of 128

2) rcx gets the value of 3

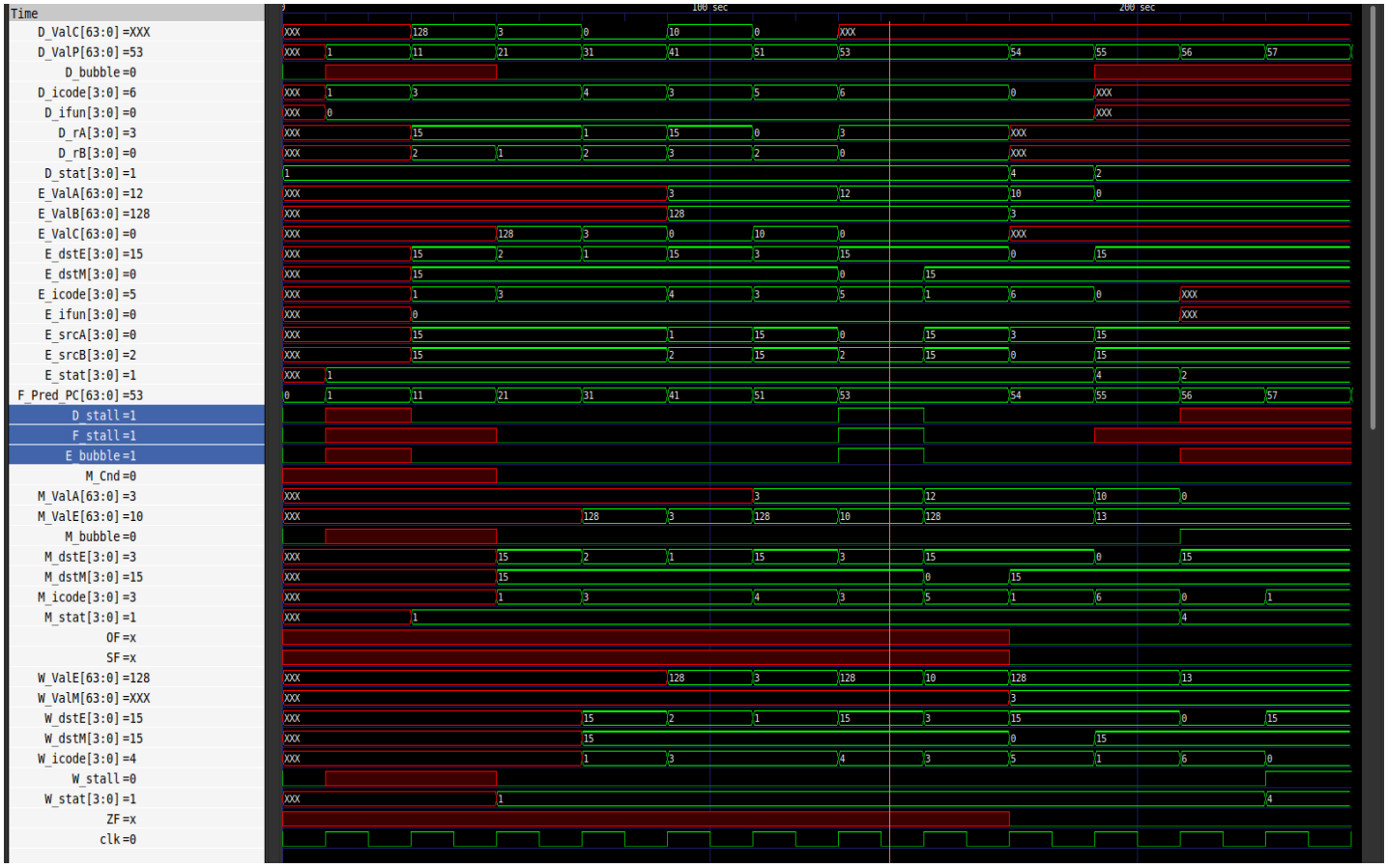
3) Value 3 is moved to memory index 128 i.e, M[128] = 3

4) rbx gets the value of 10

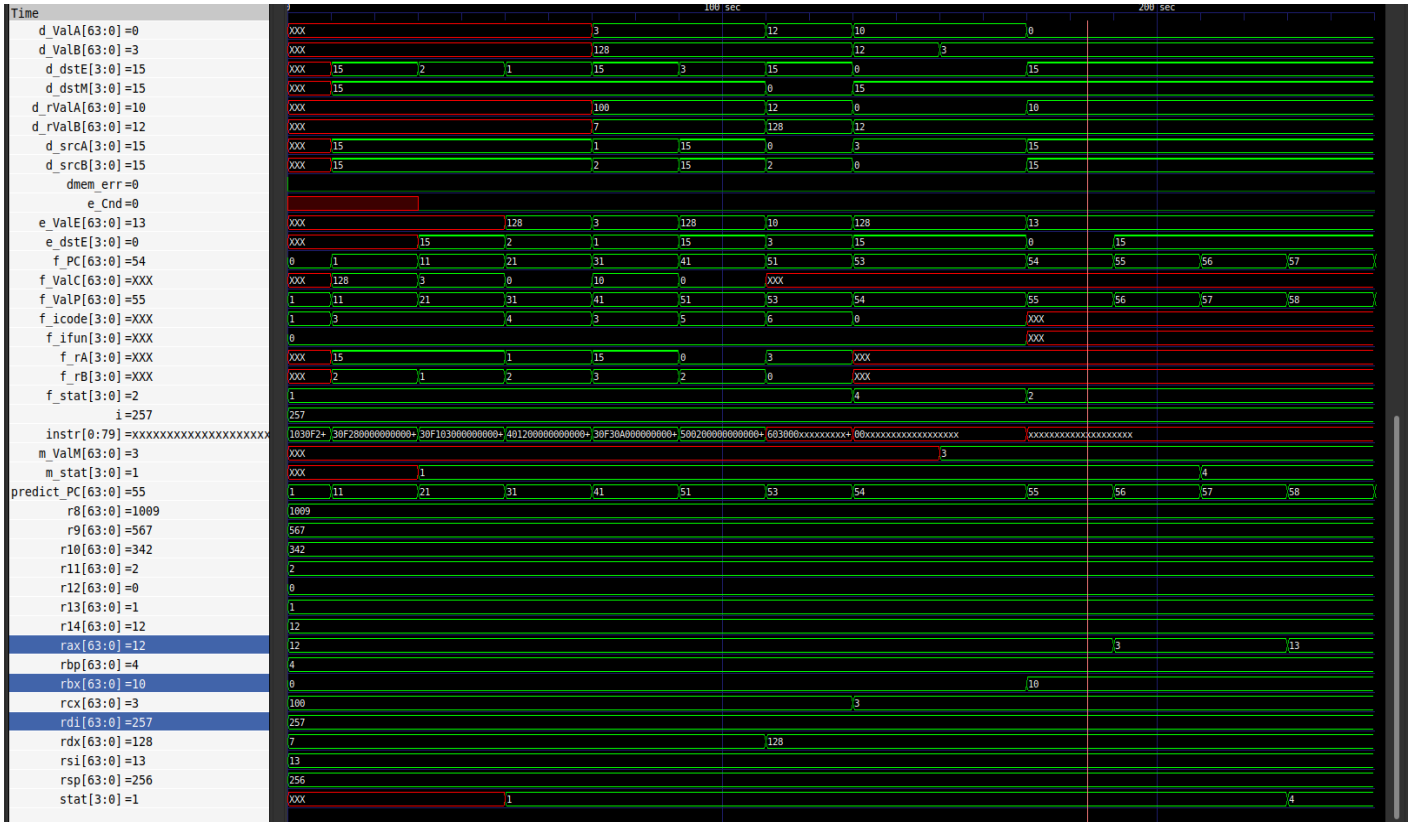
5) rax will get the value in M[128] = 3

6) add the ebx and eax value = 0+3 = 3 and store it in rax

RESULT - OUTPUT PLOTS

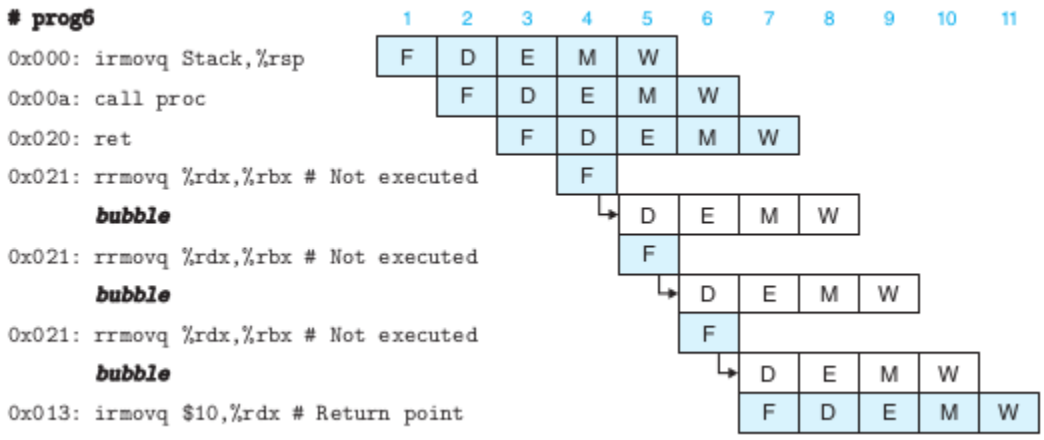


Notice that the value of **D_stall , F_stall and E_bubble is set to 1** which is required to handle load/use hazard .



From the above we can see that rdx gets 128 , rcx gets 3 , rbx gets 10 , rax gets 3 and the addition result = 13 is stored in rax and hence rax gets updated to 13 .

2 PROCESSING RET



Ret must take value from the memory for the next instruction , so until it reaches writeback , no other instruction must be decoded.

The instant it reaches W , next correct instruction can be fetched and further processing can be done .

The pipeline must stall until the ret instruction reaches the write-back stage.

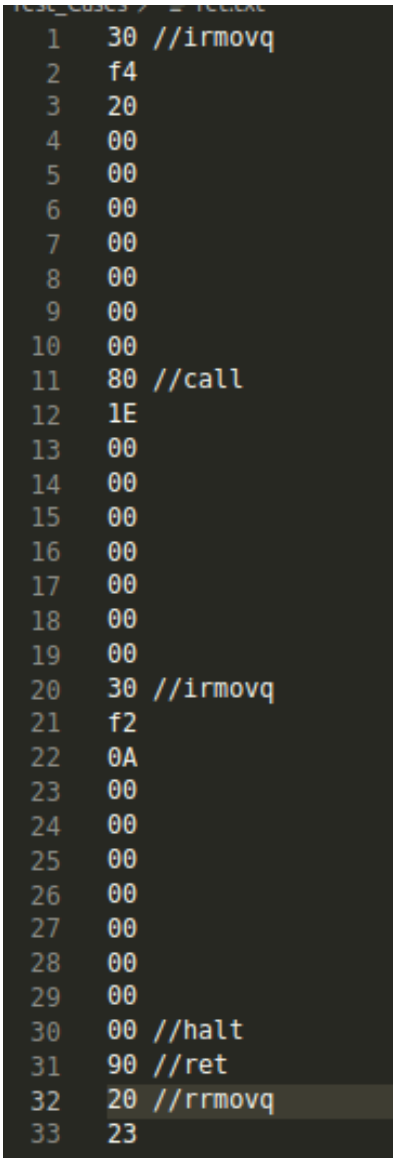
Processing ret	IRET ∈ {D_icode, E_icode, M_icode}				
Processing ret	stall	bubble	normal	normal	normal

So , we need to **STALL fetch** and **BUBBLE decode** .

TEST CASE

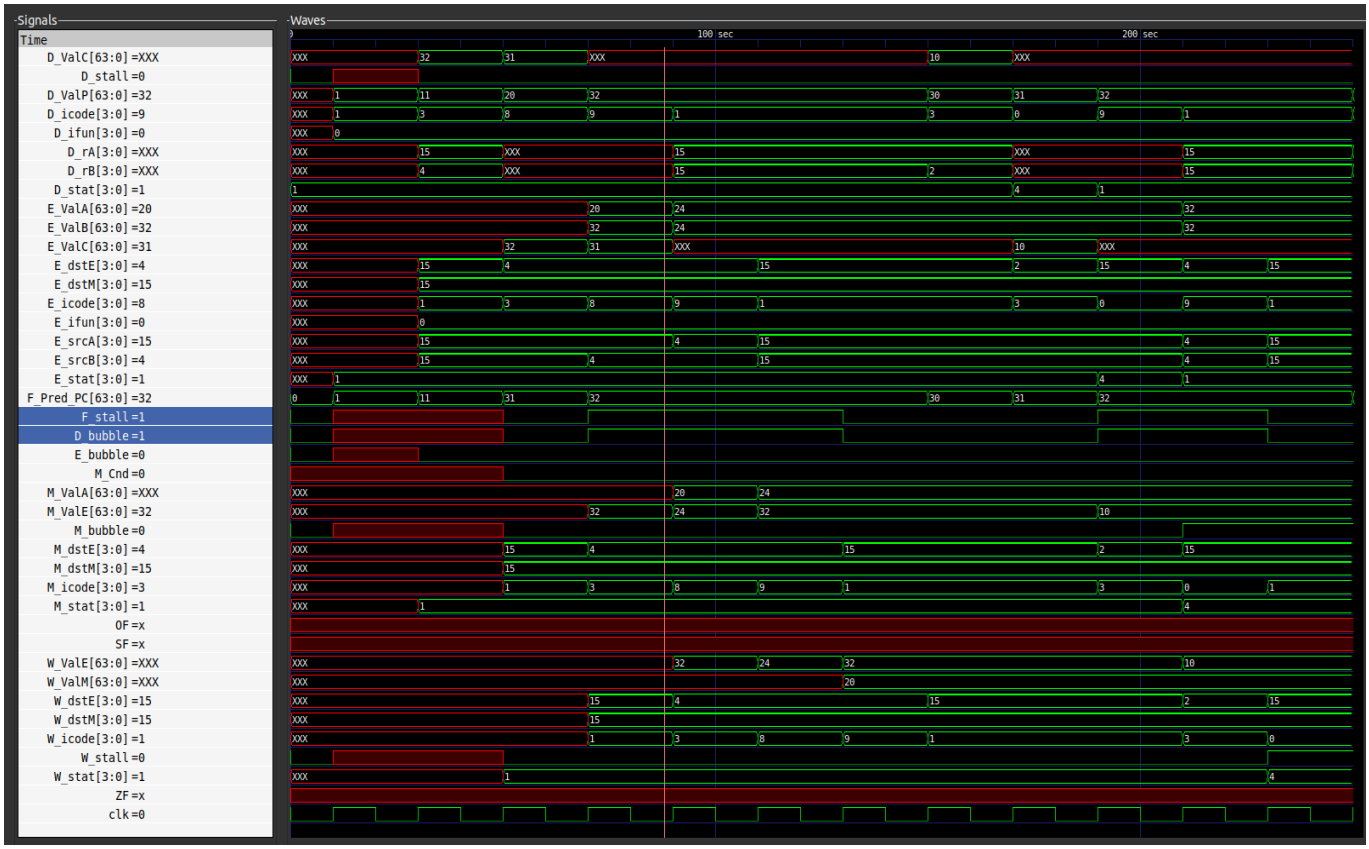
ASSEMBLY CODE

```
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
```

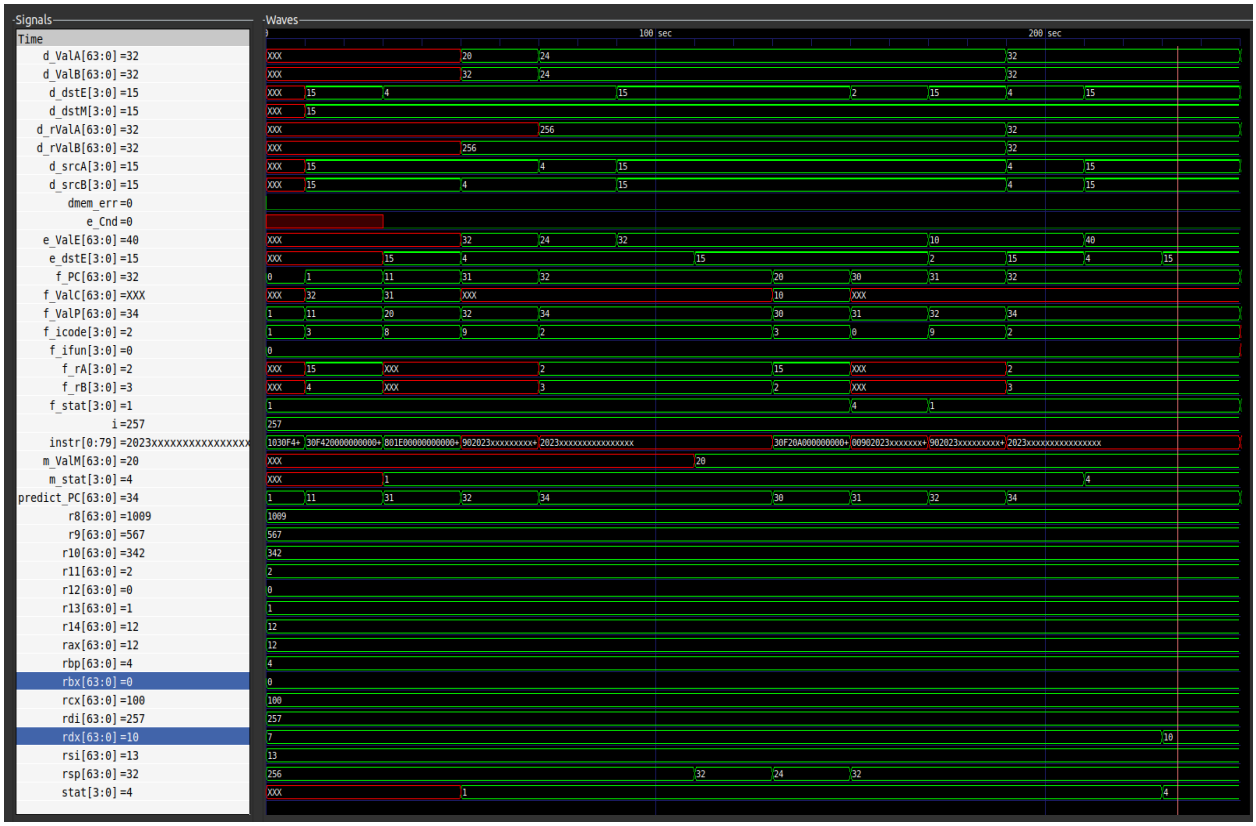


1) rsp gets the value of 32 due to irmovq
2) We call the .proc and then it immediately return i.e, the rrmovq instruction following the return should not be executed and hence the value of rbx should not get changed .
3) rdx gets the value of 10 after returning

RESULT - OUTPUT PLOTS



Observe that the value of F_stall and D_bubble is set to 1 to handle ret instruction , and it is held for 3 clock cycles so that the return instruction can reach writeback stage .



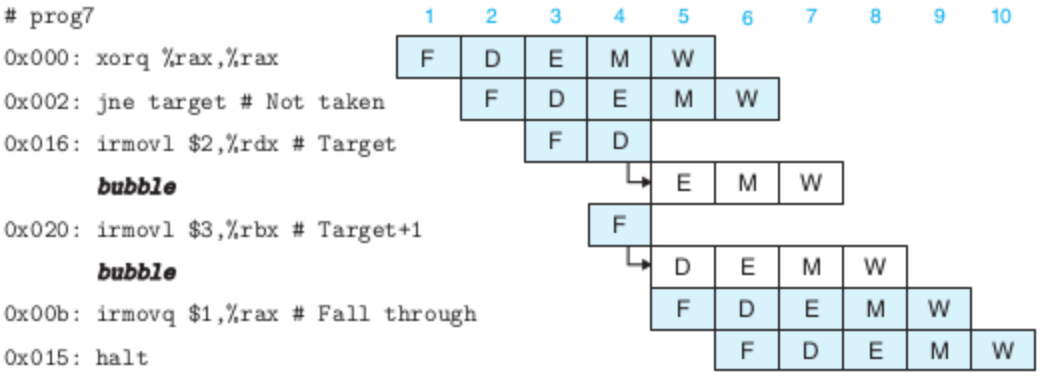
From the above its clear that the value of rbx is never updated indicating the **perfect working for return and call function .**

The value of rdx is set to 10 at the end after returning .

3 MISPREDICTED BRANCHES

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. These instructions must be canceled, and fetching should begin at the instruction following the jump instruction.

		Mispredicted branch	E_icode = IJXX && !e_Cnd		
Mispredicted branch	normal	bubble	bubble	normal	normal



From above , we can see that to handle the mispredicted jumps , we need to just **BUBBLE DECODE and BUBBLE EXECUTE** so that the instruction in the mistarget will be canceled .

TEST CASE

ASSEMBLY CODE

```
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
```

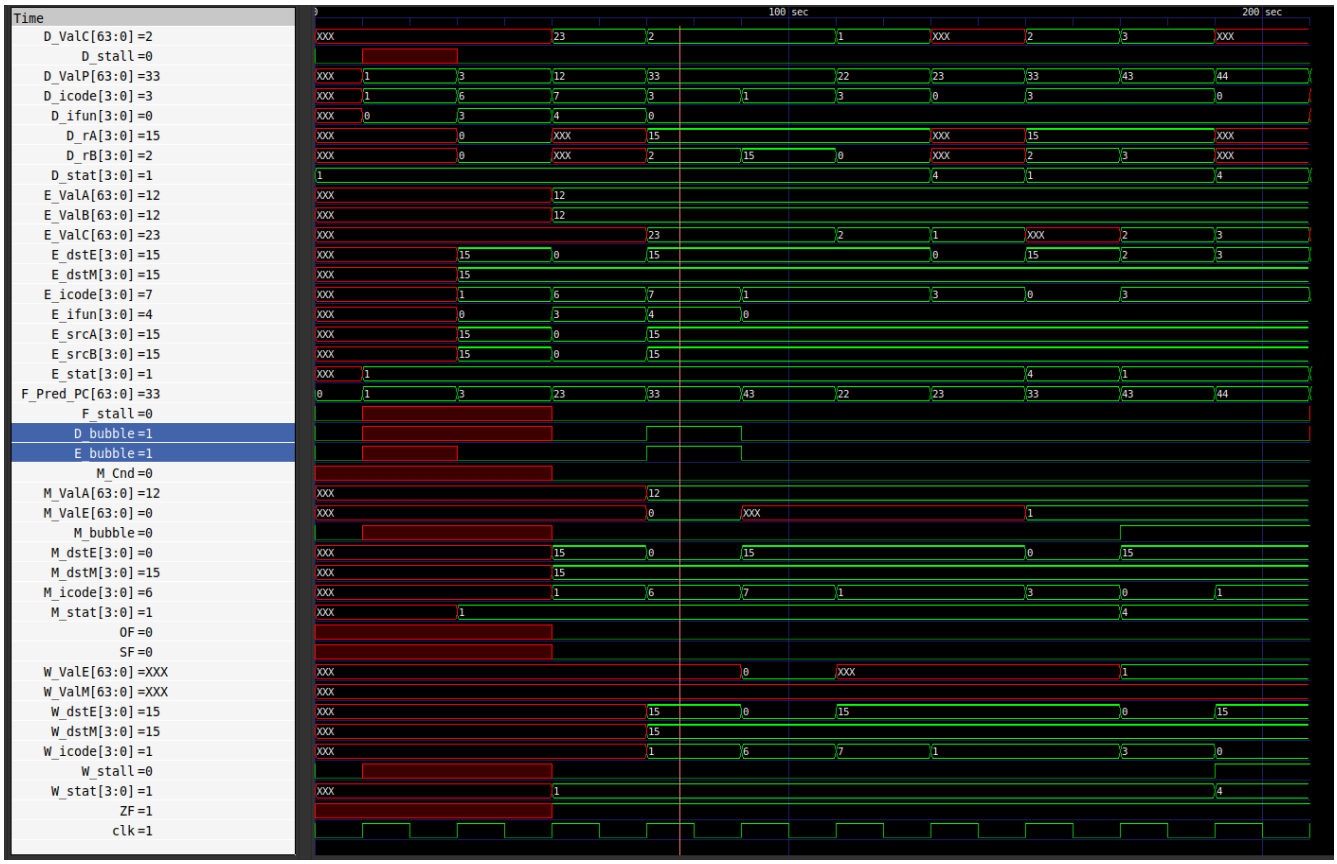
So , the instructions in the mistarget should not be executed i.e, rdx and rbx should not get updated to 2 and 3 respectively .

After returning back , value 1 is moved to rax and then the program is halted .

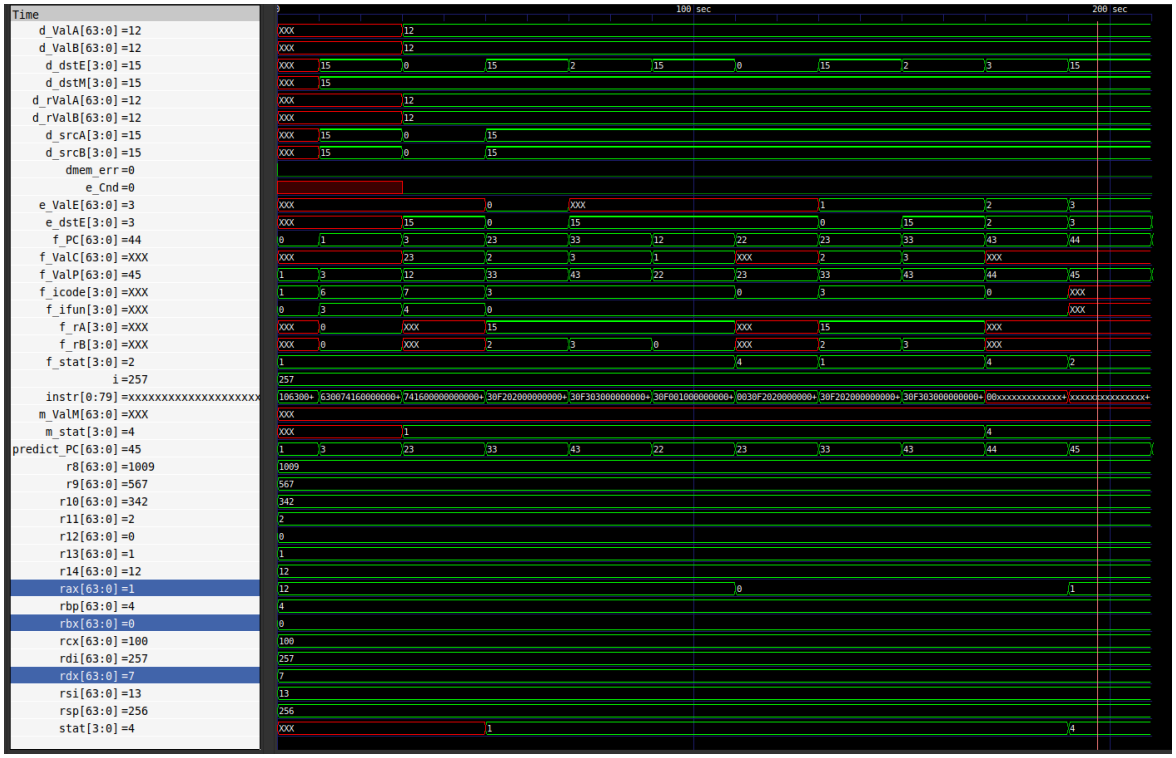
INSTRUCTION MEMORY

```
63 //xorq
00
74 //jne
16
00
00
00
00
00
00
00
30 //irmovq
f0
01
00
00
00
00
00
00
00
00 //halt
30 //irmovq
f2
02
00
00
00
00
00
00
00
00
30 //irmovq
f3
03
00
00
00
00
00
00
00
00
00 //halt
```

RESULT - OUTPUT PLOTS



Clearly from the above figure , we see that D_bubble and E_bubble are set to 1 to handle the mispredicted jump .i.e, to cancel to two wrong instructions in the mistarget .



From the above we see that the values of rdx and rbx doesn't change as they are in the mistarget and shouldn't be executed .

The value of rax is set to 1 as it returns back to the correct location after the mispredicted jump .

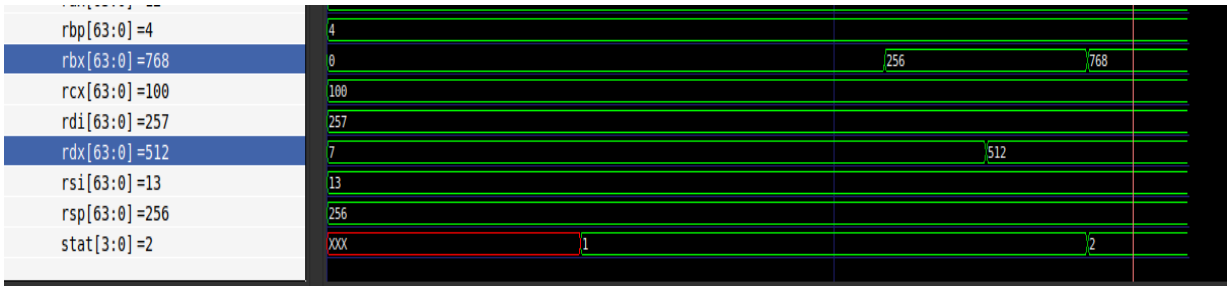
GENERAL TEST CASES

1) Two irmovq operation followed by operation on those two updated values

Test_Cases > forwarding_no_nop.txt

```
00110000 //irmovq
11110011
00000000
00000001
00000000
00000000
00000000
00000000
00000000
00000000
00110000  //irmovq
11110010
00000000
00000010
00000000
00000000
00000000
00000000
00000000
00000000
01100000  //addq
00100011
```

OUTPUT



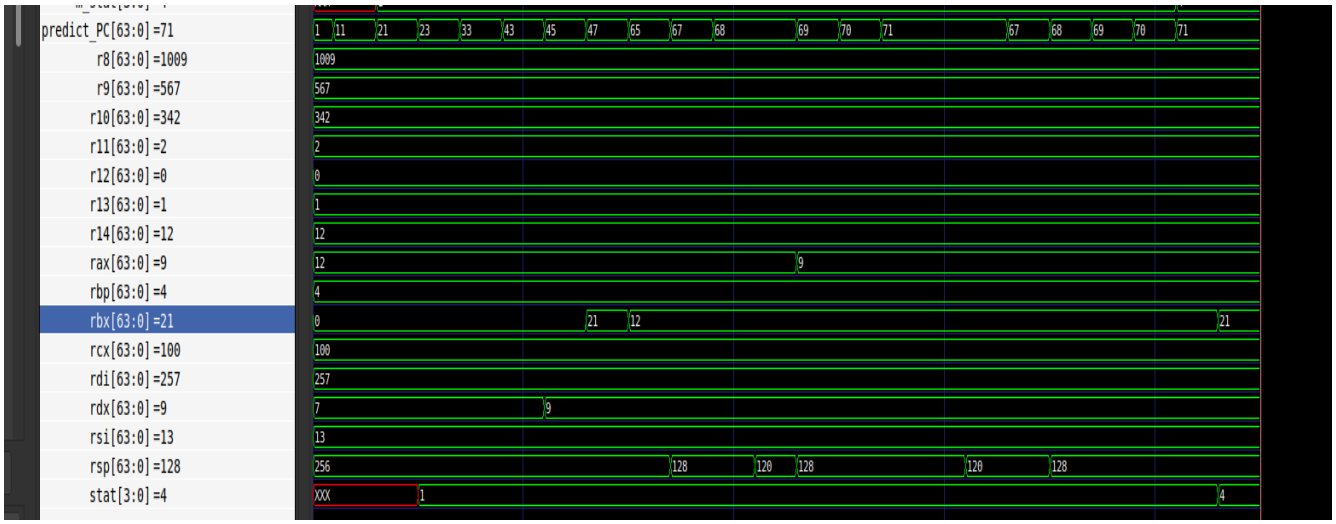
So value of rbx is set to 256 initially and rdx to 512 and then addq is performed on these two resulting in 768 getting stored in rbx .

2) Multiple instructions combined

```
30 //irmovq
f2
09
00
00
00
00
00
00
30 //irmovq
f3
15
00
00
00
00
00
00
00
61 //subq
23
30 //irmovq
f4
80
00
00
00
00
00
00
00
40 //rmmovq
43
64
00
00
00
00
00
00
00
00
```

```
00
a0 //pushq
2f
b0 //popq
0f
73 //je
40
00
00
00
00
00
00
00
80 //call
43
00
00
00
00
00
00
00
60 //addq
23
00 //halt
10 //nop
90 //ret
```

1) Value 9 should move to rdx
2) Value 21 is moved to rbx
3) Subq operates on these 2 values and the result = 12 is stored in rbx .
4) Push and pop occurs
5) Jump operation is mispredicted
6) Calls the instruction in 68 and nop is fetched
7) It returns and then executes addq and stores the result 21 in rbx



All the required movements are happening and the registers are getting updated accordingly.

Various other test cases :

1) Forwarding priority
2) Kmp1.txt (seq evals test file)
3) Mem_exceptions (2 test cases)
4) Mr and rm movq test cases
5) Target_error (when there is an error in the mistarget location)
6) **COMBINATIONAL LOGIC ERRORS** were also tested and code works .

All the above test cases have been tested and added in the git

CHALLENGES FACED

1) Deciding the operation of stages based on the clock (posedge or negedge or *)
2) Initially , we didn't set the set_cc condition , so it was updating the Conditional Codes when there was an error in memory and writeback
3) Handling the Target error i.e, the mistargeted location has an invalid instruction .
4) D_bubble should check for (!load/hazard && ret) as it shouldn't both set both bubble and stall to 1 which will result in error.
5) Initially , forwarding priority wasn't taken care of which was later fixed .