Y86-64 Processor Architecture

Harshvardhan Pandey
2022112006

Vaishnavi Shivkumar
2022102070

March 2024

The following is a report explaining the sequential and pipelined version of the Y86-64 processor.

1 Sequential Processor

1.1 Introduction

On each clock cycle, the sequential processor performs all the steps required to process a complete instruction. The sequential processor serves as a preliminary processor before the final pipelined version.

1.2 Fetch Stage

This is the stage where our processor reads the binary instruction file and fetches required value. Here Icode and Ifun tells us which operation and function to perform. rA and rB tell us which registers to access from our register file. PC update tells us the location of the next instruction in our instruction file. All fetching takes place in the positive edge of the clock.

Let us assume the following assembly code in its Y86-64 binary form is our input.

```
irmovq $0x100, %rbx
irmovq $0x200, %rdx
addq %rdx, %rbx
```

Figure 1: Code 1

Now the following picture illustrates what the fetch stage looks like:

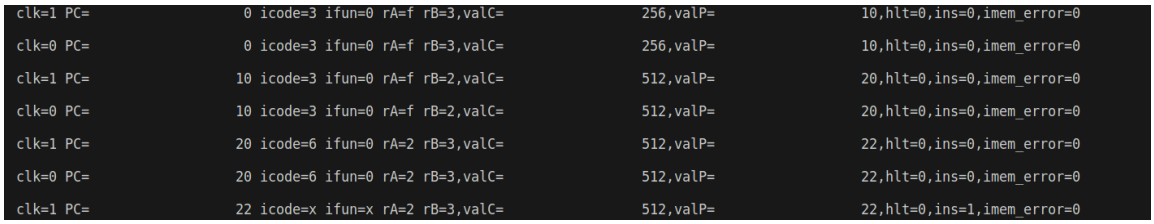


Figure 2: Fetch Output

1.3 Decode Stage

In this stage, we read the values of the fetched register indices from the register file and store it in variables valA and valB respectively.
Considering the same assembly code in Code 1, the following is our output for the decode stage.

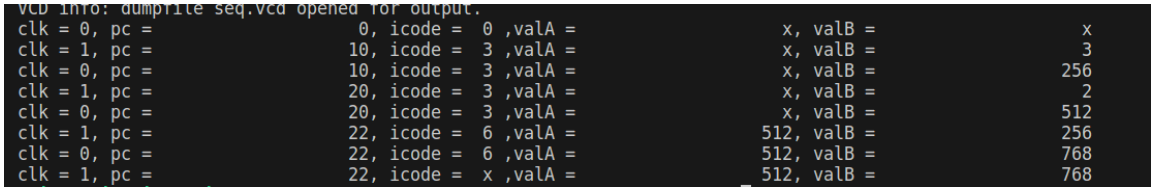


Figure 3: Decode Stage

1.4 Execute Stage

In this stage, the operations are executed. ValE stores the output of the execution. Some of the operations require conditional codes to be assigned. This too happens in the execute stage. We import the ALU to this stage, and all ALU operations occur in this stage.
Considering the assembly code in Code 1, the following will be the output of the execute stage:

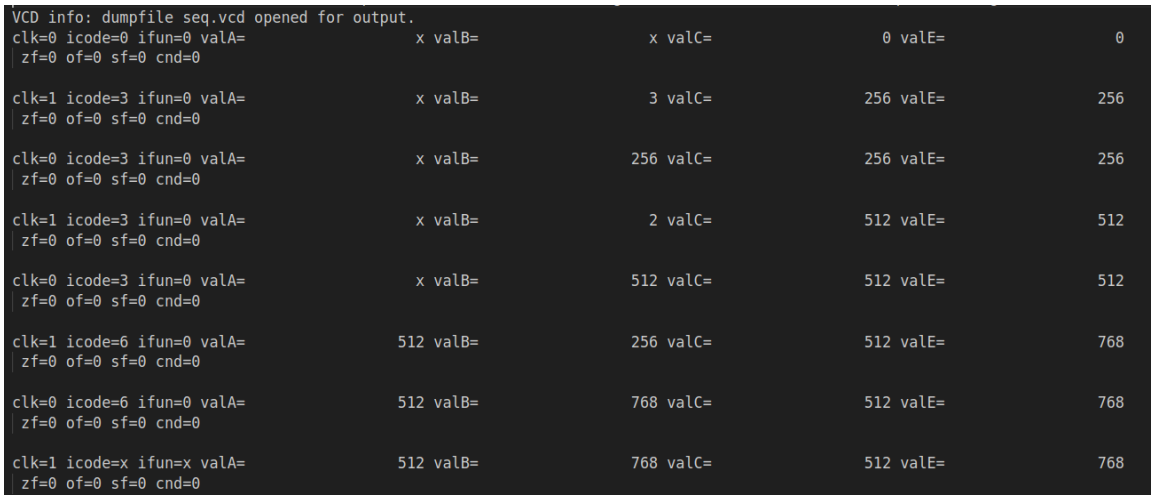


Figure 4: Execute Stage

1.5 Memory Stage

In this stage we interact with the memory of the processor. Based on the operation fetched, we either read the memory or write into the memory. All writing into the memory occurs in the clock's negative edge. If the operation asks us to read a value from memory, the value is stored in valM variable. If the operation asks us to write into memory, the value is written at the address stored in valE variable.
Considering Code 1, the following is the output of the memory stage.

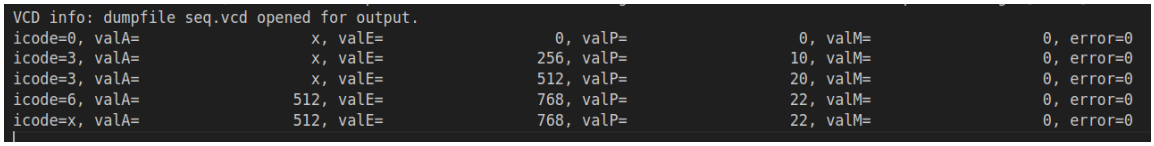


Figure 5: Memory Stage

1.6 Write Back Stage

In this stage, we update our register file with the new calculated values from the execute stage and values extracted from the memory. The write-back depends on the operation specified in the Icode. All write-back happens at the negative edge.
Considering Code 1, the following is the output of the write-back stage.

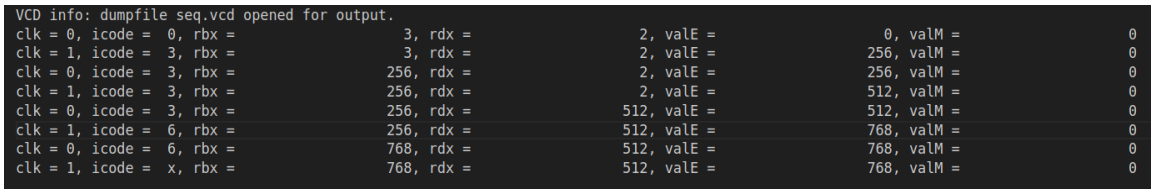


Figure 6: Write Back Stage

1.7 PC update

Here we update our PC value (our PC value acts as a cursor, iterating through all the instructions in our binary instruction file.) The new PC value depends on the operation but is usually the valP obtained from the fetch operation. Considering Code 1, the following is the output of the PC update stage.

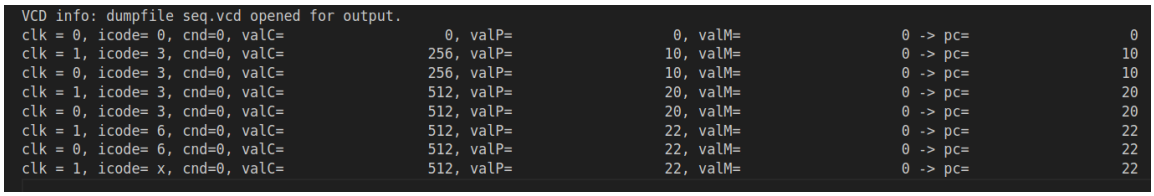


Figure 7: PC update stage

1.8 Test Cases

The following codes cover all operations.

1.8.1 rmmovq and mrmovq testing

The following assembly code is used to test rmmovq and mrmovq.

```
.pos 0
irmovq $5, %rax
irmovq $1, %rdx
rmmovq %rax, (%rdx)
mrmovq (%rdx), %rax
halt
```

Figure 8: Code 2

```
VCD info: dumpfile seq.vcd opened for output.
clk = 0, icode =  0, valA =                    x, valB =                    x,
 valC=                    0, valE =                    0, valM =                    0

clk = 1, icode =  3, valA =                    x, valB =                    0,
 valC=                    5, valE =                    5, valM =                    0

clk = 0, icode =  3, valA =                    x, valB =                    5,
 valC=                    5, valE =                    5, valM =                    0

clk = 1, icode =  3, valA =                    x, valB =                    2,
 valC=                    1, valE =                    1, valM =                    0

clk = 0, icode =  3, valA =                    x, valB =                    1,
 valC=                    1, valE =                    1, valM =                    0

clk = 1, icode =  4, valA =                    5, valB =                    1,
 valC=                    0, valE =                    1, valM =                    0

clk = 0, icode =  4, valA =                    5, valB =                    1,
 valC=                    0, valE =                    1, valM =                    0

clk = 1, icode =  5, valA =                    5, valB =                    1,
 valC=                    0, valE =                    1, valM =                    5

clk = 0, icode =  5, valA =                    5, valB =                    1,
 valC=                    0, valE =                    1, valM =                    5

clk = 1, icode =  0, valA =                    5, valB =                    1,
 valC=                    0, valE =                    1, valM =                    5
```

Figure 9: Output for Code 2

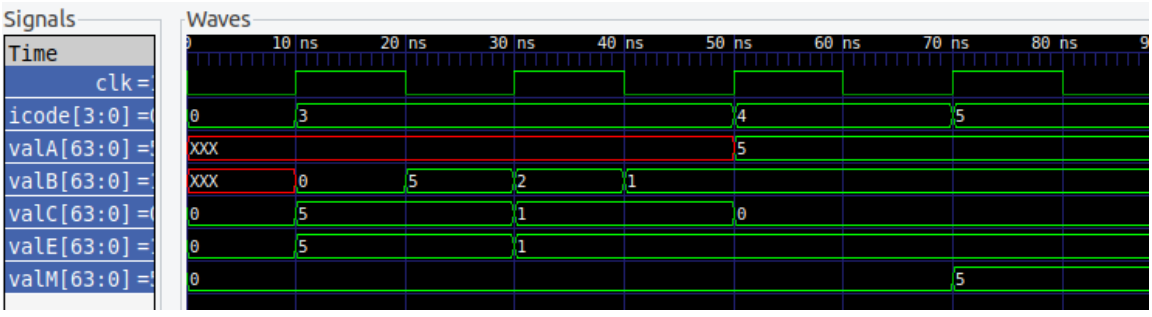


Figure 10: gtkwave for code 2

1.8.2 Push and Pop testing

The following assembly code is used to test pushq and popq operations.

```
irmovq $20, %rsp
irmovq $10, %rax
pushq %rax
popq %rbx
halt
```

Figure 11: Code 3

```
VCD info: dumpfile seq.vcd opened for output.
clk = 0, icode =  0, valA =                    x, valB =                    x,
 valE =                    0, valM =                    0

clk = 1, icode =  3, valA =                    x, valB =                    0,
 valE =                    5, valM =                    0

clk = 0, icode =  3, valA =                    x, valB =                    5,
 valE =                    5, valM =                    0

clk = 1, icode =  3, valA =                    x, valB =                    2,
 valE =                    1, valM =                    0

clk = 0, icode =  3, valA =                    x, valB =                    1,
 valE =                    1, valM =                    0

clk = 1, icode =  4, valA =                    5, valB =                    1,
 valE =                    1, valM =                    0

clk = 0, icode =  4, valA =                    5, valB =                    1,
 valE =                    1, valM =                    0

clk = 1, icode =  5, valA =                    5, valB =                    1,
 valE =                    1, valM =                    5

clk = 0, icode =  5, valA =                    5, valB =                    1,
 valE =                    1, valM =                    5

clk = 1, icode =  0, valA =                    5, valB =                    1,
 valE =                    1, valM =                    5
```

Figure 12: Output for Code 3

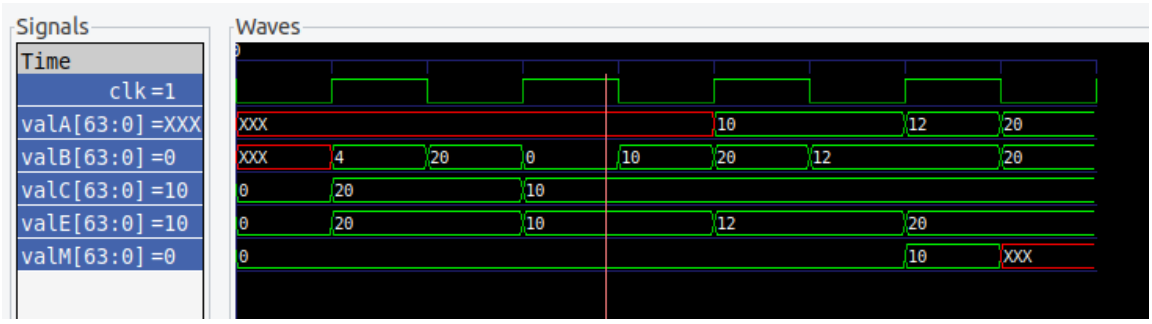


Figure 13: gtkwave for code 3

1.8.3 Call and Return testing

```
irmovq $10, %rsp
irmovq $5 , %rax
irmovq $4 , %rbx
call NF
xorq %rax, %rbx
halt

NF:
    addq %rax , %rbx
    ret
addq %rax, %rbx
```

Figure 14: Code 4

```
WARNING: ./fetch.v:22: $readmemb(../SampleTestcase/call_ret.txt): Not enough words in the file for the requested range [0:511].
VCD info: dumpfile seq.vcd opened for output.
icode =  0, valA =                    x, valB =                    x, valC =                    0, valE =                    0,
 valM =                    0
icode =  3, valA =                    x, valB =                    4, valC =                   39, valE =                   39,
 valM =                    0
icode =  3, valA =                    x, valB =                   39, valC =                   39, valE =                   39,
 valM =                    0
icode =  3, valA =                    x, valB =                    0, valC =                    5, valE =                    5,
 valM =                    0
icode =  3, valA =                    x, valB =                    5, valC =                    5, valE =                    5,
 valM =                    0
icode =  3, valA =                    x, valB =                    3, valC =                    4, valE =                    4,
 valM =                    0
icode =  3, valA =                    x, valB =                    4, valC =                    4, valE =                    4,
 valM =                    0
icode =  8, valA =                    x, valB =                   39, valC =                   42, valE =                   31,
 valM =                    0
icode =  8, valA =                    x, valB =                   31, valC =                   42, valE =                   31,
 valM =                    0
icode =  6, valA =                    5, valB =                    4, valC =                   42, valE =                    9,
 valM =                    0
icode =  6, valA =                    5, valB =                    9, valC =                   42, valE =                    9,
 valM =                    0
icode =  9, valA =                   31, valB =                   31, valC =                   42, valE =                   39,
 valM =                   39
icode =  9, valA =                   39, valB =                   39, valC =                   42, valE =                   39,
 valM =                    x
icode =  6, valA =                    5, valB =                    9, valC =                   42, valE =                   12,
 valM =                    x
icode =  6, valA =                    5, valB =                   12, valC =                   42, valE =                   12,
 valM =                    x
icode =  0, valA =                    5, valB =                   12, valC =                   42, valE =                   12,
 valM =                    x
```

Figure 15: Output of Code 4

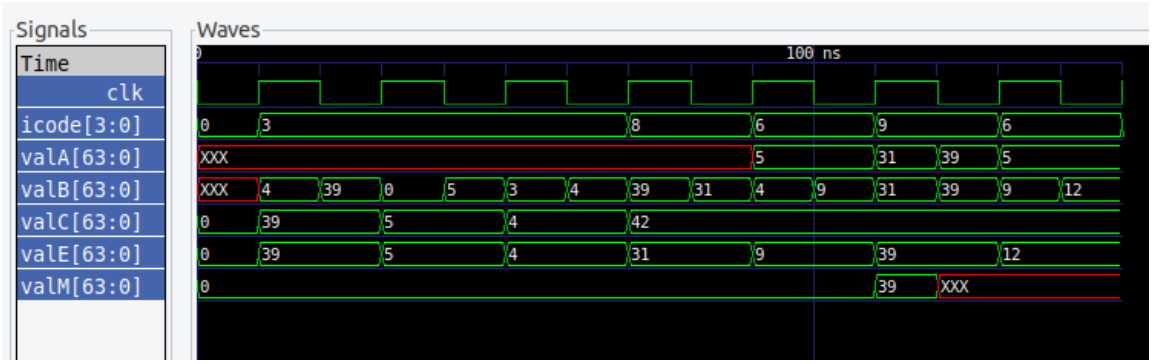


Figure 16: gtkwave for code 4

1.9 Problems Faced

1.9.1 Timing

We faced a lot of issues with when which stage should be active and modifying the state of the processor. To handle this we divided the clock cycle into sections and stages worked in their assigned sections. The clock cycle was divided as

follows:

1. Positive Edge: Fetch was done in the positive edge
2. Clock is at 1: Execute stage happens when the clock is high
3. Negative Edge: Write back and Memory Write happened at the positive edge

Decode and Memory Read could be treated as combinational blocks and hence did not need any specific clock section assignment. The Fetch-Execute-Writeback order is ensured by the clock allotment given.

1.9.2 valP Issues for RET instruction

During RET valP went to 'x' stage during the low part of the clock. I still don't know why. This issue was mitigated by doing PC Update only when the clock is HIGH.

2 Pipelined Processor

2.1 Fetch and PC Prediction

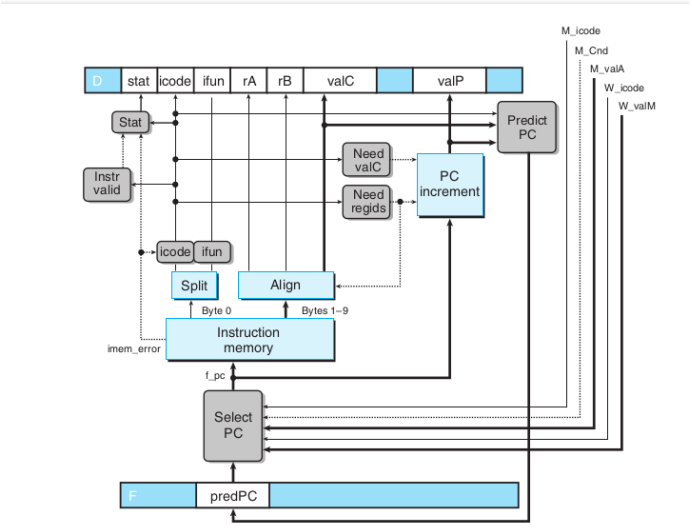


Figure 17: PIPE PC selection and Fetch Logic

This stage consists of the fetch register and the fetch stage. This stage selects a current value for the program counter and predicts the next PC value.
The fetch register checks for stalls coming from the control system of the pipeline

processor. If there is no stalling, the PC takes the value of the predicted PC value from the fetch stage.**(1)**
The PC selection logic chooses between three program counter sources.

1. When the mispredicted branch enters the memory stage, the value of valP for this instruction (next address) is read from the memory register **M_valA**.

2. When a return instruction enters the write-back stage, the return address is read from the write-back register **W_valM**.

3. In all other cases, we use the predicted value of the PC stored in the fetch register (refer to (1)).

Our fetch stage happens at the negative edge of the clock, while the register loads at the positive edge of the clock.

2.2 Decode and Writeback Stage

The decode stage primarily calculates the values of d_valA, and d_valB, which are then propagated through the pipeline. We use the methods of data-forwarding and stalling which are described in sections 2.6.1 and 2.6.2.

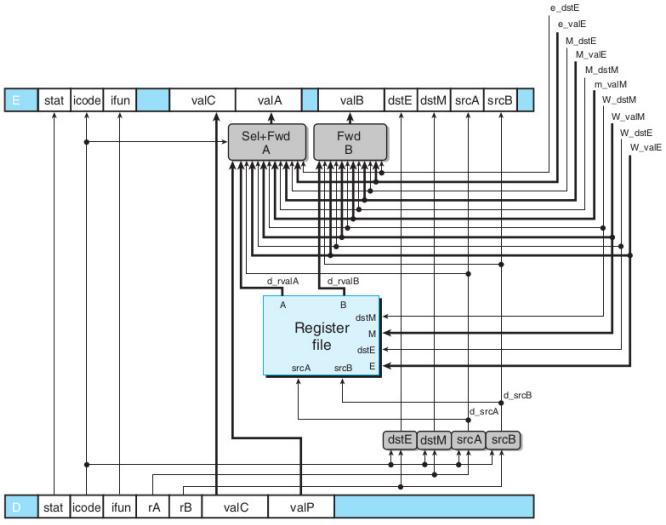


Figure 18: Decode and Write-back Block Diagram

Writeback is relatively straightforward. We set register W_dstE to the value W_valE, and W_dstM to the value W_valM.

2.3 Execute

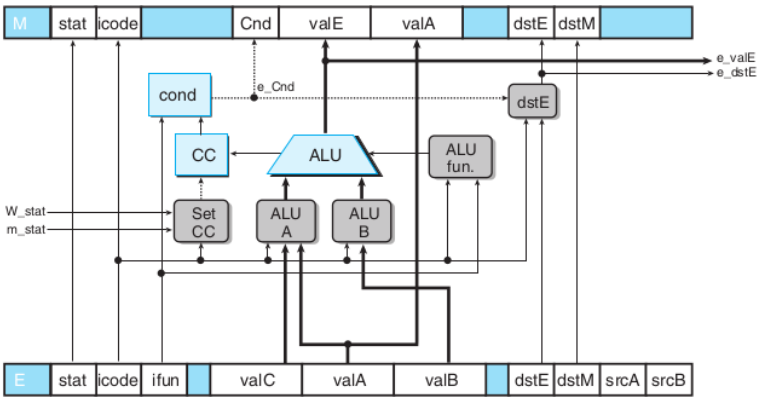


Figure 4.60 PIPE execute stage logic. This part of the design is very similar to the logic in the SEQ implementation.

Figure 19: Execute logic

The execute register checks for a bubble. If there is a bubble it changes operation to a nop. Else, we proceed as normal. The instruction is loaded at the positive edge of the clock.
The structuring of the execute block in the pipeline is the same as sequential. One difference is that the conditional codes (set in the control system) consider status inputs from the memory and writeback stage. This is in order to not update the programmer's visible state in case of an exception.

2.4 Memory

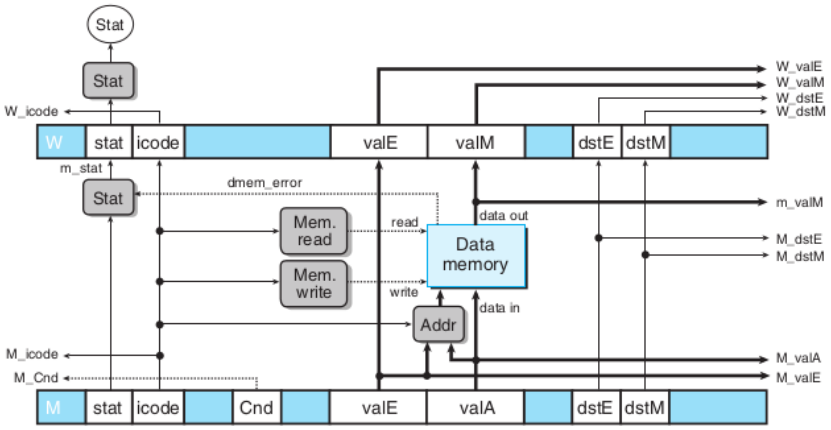


Figure 20: Pipe Memory logic

The memory register checks for a bubble. If there is a bubble it changes operation to a nop. Else, we proceed as normal. The instruction is loaded at the positive edge of the clock.
The structuring of memory in the pipeline is similar to that of sequential. The status gets updated in case of any data address errors.

2.5 Control System

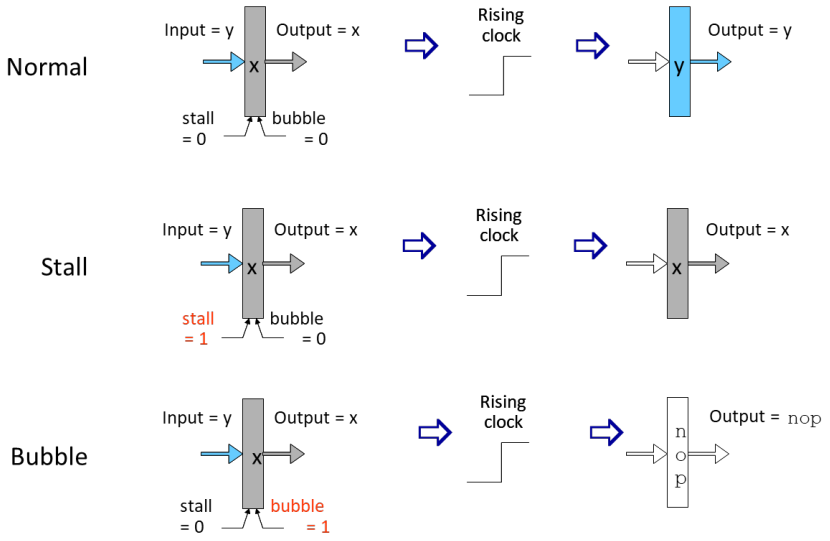
Pipeline registers work in three modes:

1. Normal
2. Stall
3. Bubble

To choose between modes each pipeline register has required input signals. The response to the input signal is as follows: For each hazard, we need to first analyze the detection condition.

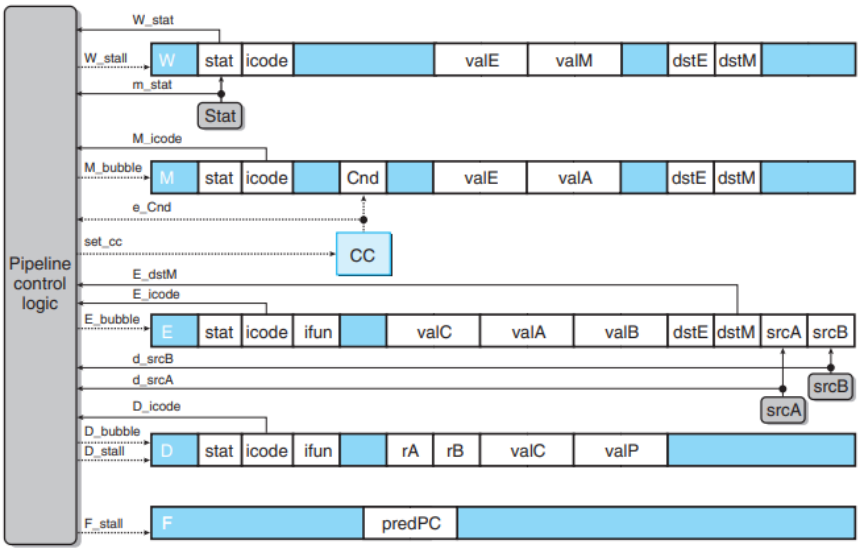
Hazard	Condition
load-use	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
return	IRET in { D_icode, E_icode, M_icode }
branch misprediction	E_icode = IJXX && !e_Cnd
non-AOK status	m_stat in {HLT, ADS, INS} && W_stat in {HLT, ADS, INS}

We analyze different control hazards to see the mode of operation of the pipeline registers



Hazard	Fetch	Decode	Execute	Memory	Write Back
load-use	Stall	Stall	Bubble	Normal	Normal
return	Stall	Bubble	Normal	Normal	Normal
branch misprediction	normal	bubble	bubble	normal	normal
non-AOK status	normal	normal	normal	bubble	stall

Now, when we encounter a combination of load-use and return hazard, both bubble and stall conditions of D registers are true. In this case, we just stall to "wait out" any hazards.
There is another very important parameter for control that is set_cc, which ensures that condition codes are not altered after an instruction that triggers non-zero status is encountered.
This gives us the following block diagram of the control block:



2.6 Pipeline Hazards

2.6.1 Data Dependencies

Consider the following code snippet:

```
1    irmovq $50, %rax
2    addq %rax, %rbx
3    mrmovq 100(%rbx), %rdx
```

While decoding instruction 2, the value of register rax is required. However, when instruction 2 is in the decode stage, instruction 1 is in the execute stage. Hence, the register has not been written yet. A similar issue is dealt with in instruction 3, which depends on instruction 2 output. For our project, we have dealt with this issue with data-forwarding which is clock cycle efficient.
Data forwarding works by establishing a feedback path from the later stages to the decode stage to use computed values. So for decoding a register, the processor goes in the following order:

1. Check if e_valE is the required value. If not go to the next stage.
2. Check if M_valE or m_valM is the required value. If not go to the next stage
3. Check if W_valE or W_valM is required value. If not go to next step.

4. Read from register

This strategy doesn't work for load-use hazards, which will be covered in the next section.

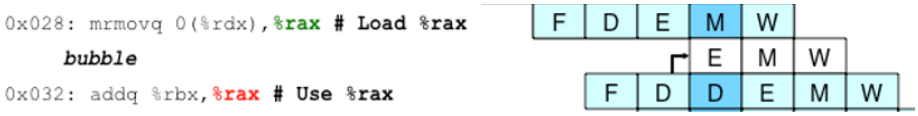
2.6.2 Load-Use Hazard

Load-use hazard occurs when a value is read from memory and used in the next instruction.

```
mrmovq %rdx, %rax //loads rax
addq %rbx, %rax  //uses rax
```

Figure 21: load use example

Now, why can't we just forward data here? It's because the data is not in the processor when the use instruction is in the decode phase, since the load instruction is in the execute stage. The solution is to stall decoding for a cycle and insert a bubble in the execute register. In the next cycle, the load instruction would be in the memory stage, and m_valM can be forwarded to the decode stage.

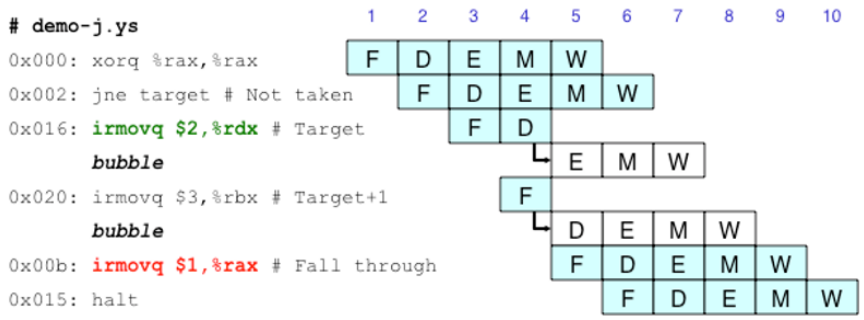


2.6.3 Branch Misprediction

In conditional jumps, we can not predict the next instruction with full certainty. To deal with this, we use the following strategy:

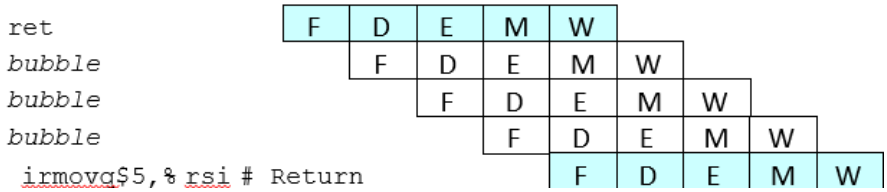
1. "Guess" next PC.
2. If we find later that the guess is wrong, fix the "programmer visible state".

The benefit of using this approach is that if our guess is right, we are clock-cycle efficient, if it is wrong, we are only as bad as stopping the pipeline till the jump condition is evaluated.
Our guessing algorithm would be "always take the jump"
If the guess is wrong, that condition is called "branch-misprediction". We need to ensure the visible state of the processor is not altered by the wrong instructions that have entered the pipeline. Notice, that we find out that our guess was wrong in the execute stage. Hence, the wrong instructions are in the fetch and decode stage, and have not altered the memory, condition flags or the registers. So, we insert a bubble in D register and E register to flush the instructions' effects out of the processor.



2.6.4 Return

We cannot predict even a guess for the next PC when a return instruction is encountered. So, we need to wait till the return instruction is in the memory stage to see where to go next. Thus, we stall the entire pipeline for 3 clock cycles when a return instruction is encountered.



2.7 Test Cases

2.7.1 Operations and irmovq Testing

We consider the following assembly code to test the operations and irmovq commands. This also checks if data dependencies are working fine.

```
irmovq $0x100, %rbx
irmovq $0x200, %rdx
addq %rdx, %rbx
```

Figure 22: Code 1

The following are the variables as the pipeline processor tries to run code 1.

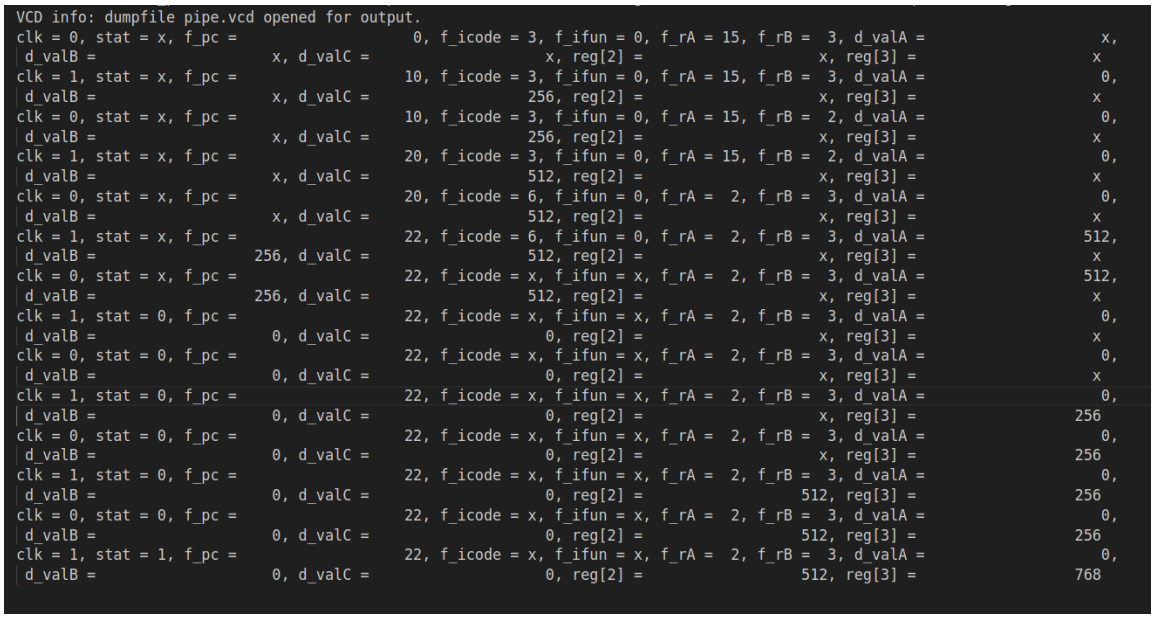


Figure 23: PIPE output of Code 1

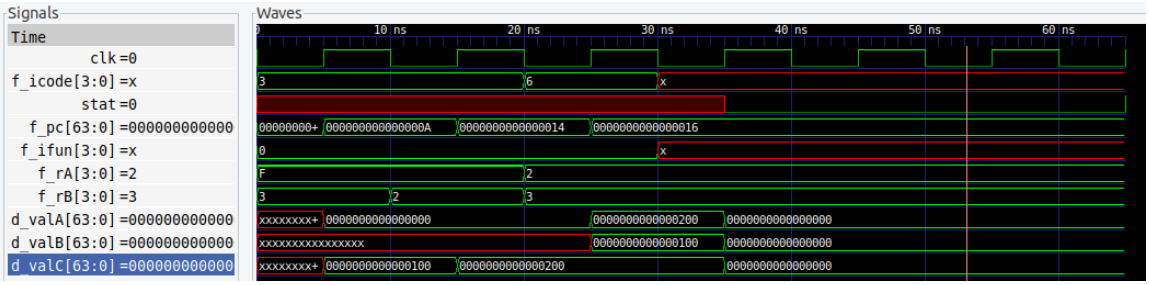


Figure 24: Code 1 PIPE gtkwave

2.7.2 Rmmovq and mrmovq testing

We consider the following assembly code to test rmmovq and mrmovq commands.

```
irmovq $5, %rax
irmovq $1, %rdx
rmmovq %rax, (%rdx)
mrmovq (%rdx), %rax
halt
```

Figure 25: Code 2

The following are the variables as the pipeline processor tries to run code 2.

```
clk = 0, f_icode =  3, d_valA =                    x, d_valB =                    x,
 f_valC=                    5, e_valE =                    0, m_valM =                    0
clk = 1, f_icode =  3, d_valA =                    0, d_valB =                    x,
 f_valC=                    5, e_valE =                    0, m_valM =                    0
clk = 0, f_icode =  3, d_valA =                    0, d_valB =                    x,
 f_valC=                    1, e_valE =                    0, m_valM =                    0
clk = 1, f_icode =  3, d_valA =                    0, d_valB =                    x,
 f_valC=                    1, e_valE =                    5, m_valM =                    0
clk = 0, f_icode =  4, d_valA =                    0, d_valB =                    x,
 f_valC=                    0, e_valE =                    5, m_valM =                    0
clk = 1, f_icode =  4, d_valA =                    5, d_valB =                    1,
 f_valC=                    0, e_valE =                    1, m_valM =                    0
clk = 0, f_icode =  5, d_valA =                    5, d_valB =                    1,
 f_valC=                    0, e_valE =                    1, m_valM =                    0
clk = 1, f_icode =  5, d_valA =                    0, d_valB =                    1,
 f_valC=                    0, e_valE =                    1, m_valM =                    0
clk = 0, f_icode =  0, d_valA =                    0, d_valB =                    1,
 f_valC=                    0, e_valE =                    1, m_valM =                    0
clk = 1, f_icode =  0, d_valA =                    0, d_valB =                    0,
 f_valC=                    0, e_valE =                    1, m_valM =                    0
clk = 0, f_icode =  0, d_valA =                    0, d_valB =                    0,
 f_valC=                    0, e_valE =                    1, m_valM =                    0
clk = 1, f_icode =  0, d_valA =                    0, d_valB =                    0,
 f_valC=                    0, e_valE =                    1, m_valM =                    5
clk = 0, f_icode =  0, d_valA =                    0, d_valB =                    0,
 f_valC=                    0, e_valE =                    1, m_valM =                    5
clk = 1, f_icode =  0, d_valA =                    0, d_valB =                    0,
 f_valC=                    0, e_valE =                    1, m_valM =                    5
clk = 0, f_icode =  0, d_valA =                    0, d_valB =                    0,
 f_valC=                    0, e_valE =                    1, m_valM =                    5
clk = 1, f_icode =  0, d_valA =                    0, d_valB =                    0,
 f_valC=                    0, e_valE =                    1, m_valM =                    5
```

Figure 26: PIPE output for code 2

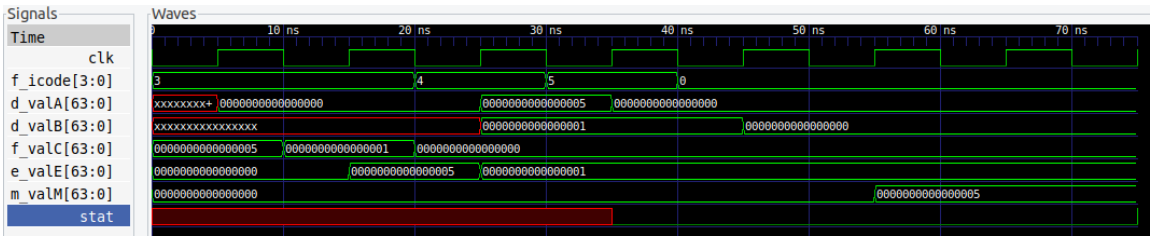


Figure 27: gtkwave for code 2

2.7.3 Push and Pop Testing

We consider the following assembly code to test pushq and popq commands.

```
irmovq $20, %rsp
irmovq $10, %rax
pushq %rax
popq %rbx
halt
```

Figure 28: Code 3

The following are the variables as the pipeline processor runs code 3.

```
WARNING: ./fetch_pc.v:48: $readmemb(../SampleTestcase/push_pop.txt): Not enough words in the file for the requested range
VCD info: dumpfile pipe.vcd opened for output.
clk = 0, f_icode =  3, d_valA =                  x, d_valB =                  x, f_valC =                 20,
 e_valE =                  0, m_valM =                  0, status = x
clk = 1, f_icode =  3, d_valA =                  0, d_valB =                  x, f_valC =                 20,
 e_valE =                  0, m_valM =                  0, status = x
clk = 0, f_icode =  3, d_valA =                  0, d_valB =                  x, f_valC =                 10,
 e_valE =                  0, m_valM =                  0, status = x
clk = 1, f_icode =  3, d_valA =                  0, d_valB =                  x, f_valC =                 10,
 e_valE =                 20, m_valM =                  0, status = x
clk = 0, f_icode = 10, d_valA =                  0, d_valB =                  x, f_valC =                 10,
 e_valE =                 20, m_valM =                  0, status = x
clk = 1, f_icode = 10, d_valA =                 10, d_valB =                 20, f_valC =                 10,
 e_valE =                 10, m_valM =                  0, status = x
clk = 0, f_icode = 11, d_valA =                 10, d_valB =                 20, f_valC =                 10,
 e_valE =                 10, m_valM =                  0, status = x
clk = 1, f_icode = 11, d_valA =                 12, d_valB =                 12, f_valC =                 10,
 e_valE =                 12, m_valM =                  0, status = 0
clk = 0, f_icode =  0, d_valA =                 12, d_valB =                 12, f_valC =                 10,
 e_valE =                 12, m_valM =                  0, status = 0
clk = 1, f_icode =  0, d_valA =                  0, d_valB =                  0, f_valC =                 10,
 e_valE =                 20, m_valM =                  0, status = 0
clk = 0, f_icode =  0, d_valA =                  0, d_valB =                  0, f_valC =                 10,
 e_valE =                 20, m_valM =                  0, status = 0
clk = 1, f_icode =  0, d_valA =                  0, d_valB =                  0, f_valC =                 10,
 e_valE =                 20, m_valM =                 10, status = 0
clk = 0, f_icode =  0, d_valA =                  0, d_valB =                  0, f_valC =                 10,
 e_valE =                 20, m_valM =                 10, status = 0
clk = 1, f_icode =  0, d_valA =                  0, d_valB =                  0, f_valC =                 10,
 e_valE =                 20, m_valM =                 10, status = 0
clk = 0, f_icode =  0, d_valA =                  0, d_valB =                  0, f_valC =                 10,
 e_valE =                 20, m_valM =                 10, status = 0
clk = 1, f_icode =  0, d_valA =                  0, d_valB =                  0, f_valC =                 10,
 e_valE =                 20, m_valM =                 10, status = 1
```

Figure 29: Pipe output for code 3

image

Figure 30: gtkwave for code 3

2.7.4 Call and Return Testing

We consider the following assembly code to test call and return commands.

```
irmovq $10, %rsp
irmovq $5 , %rax
irmovq $4 , %rbx
call NF
xorq %rax, %rbx
halt

NF:
    addq %rax , %rbx
    ret
addq %rax, %rbx
```

Figure 31: Code 4

The following are the variables as the pipeline processor runs code 4.

```
WARNING: ./fetch_pc.v:48: $readmemb(../SampleTestcase/push_pop.txt): Not enough words in the file for the requested ra
VCD info: dumpfile pipe.vcd opened for output.
clk = 0, f_icode =  3, d_valA =                   x, d_valB =                   x, f_valC =                  20,
 e_valE =                   0, m_valM =                   0, status = x
clk = 1, f_icode =  3, d_valA =                   0, d_valB =                   x, f_valC =                  20,
 e_valE =                   0, m_valM =                   0, status = x
clk = 0, f_icode =  3, d_valA =                   0, d_valB =                   x, f_valC =                  10,
 e_valE =                   0, m_valM =                   0, status = x
clk = 1, f_icode =  3, d_valA =                   0, d_valB =                   x, f_valC =                  10,
 e_valE =                  20, m_valM =                   0, status = x
clk = 0, f_icode = 10, d_valA =                   0, d_valB =                   x, f_valC =                  10,
 e_valE =                  20, m_valM =                   0, status = x
clk = 1, f_icode = 10, d_valA =                  10, d_valB =                  20, f_valC =                  10,
 e_valE =                  10, m_valM =                   0, status = x
clk = 0, f_icode = 11, d_valA =                  10, d_valB =                  20, f_valC =                  10,
 e_valE =                  10, m_valM =                   0, status = x
clk = 1, f_icode = 11, d_valA =                  12, d_valB =                  12, f_valC =                  10,
 e_valE =                  12, m_valM =                   0, status = 0
clk = 0, f_icode =  0, d_valA =                  12, d_valB =                  12, f_valC =                  10,
 e_valE =                  12, m_valM =                   0, status = 0
clk = 1, f_icode =  0, d_valA =                   0, d_valB =                   0, f_valC =                  10,
 e_valE =                  20, m_valM =                   0, status = 0
clk = 0, f_icode =  0, d_valA =                   0, d_valB =                   0, f_valC =                  10,
 e_valE =                  20, m_valM =                   0, status = 0
clk = 1, f_icode =  0, d_valA =                   0, d_valB =                   0, f_valC =                  10,
 e_valE =                  20, m_valM =                  10, status = 0
clk = 0, f_icode =  0, d_valA =                   0, d_valB =                   0, f_valC =                  10,
 e_valE =                  20, m_valM =                  10, status = 0
clk = 1, f_icode =  0, d_valA =                   0, d_valB =                   0, f_valC =                  10,
 e_valE =                  20, m_valM =                  10, status = 0
clk = 0, f_icode =  0, d_valA =                   0, d_valB =                   0, f_valC =                  10,
 e_valE =                  20, m_valM =                  10, status = 0
clk = 1, f_icode =  0, d_valA =                   0, d_valB =                   0, f_valC =                  10,
 e_valE =                  20, m_valM =                  10, status = 1
```

Figure 32: Pipe output for code 4

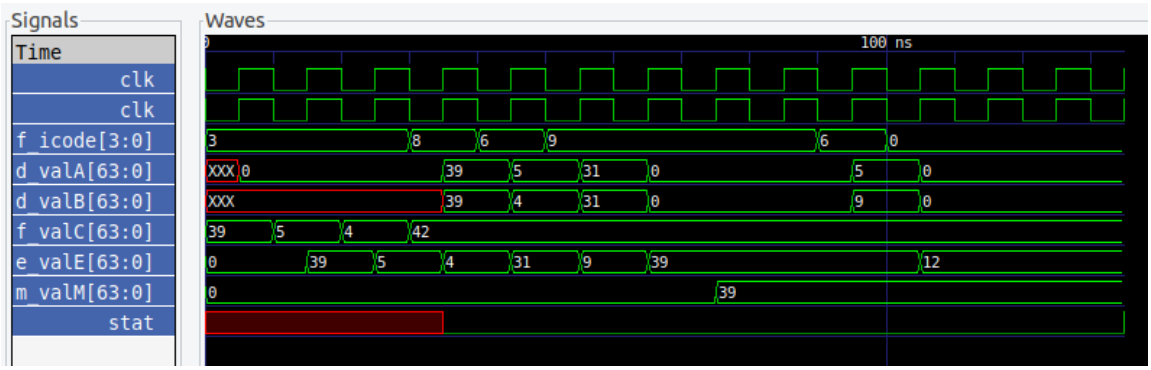


Figure 33: gtkwave for code 4

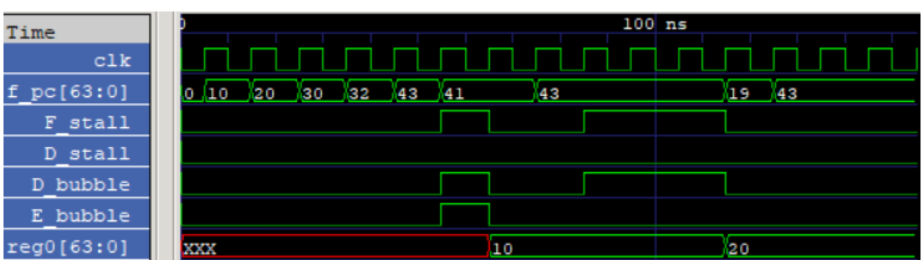
2.7.5 Return and Branch Misprediction Combination

To test this combination we need a branch misprediction where the wrong branch has a return statement.

```
irmovq $100, %rsp
call func
halt
func:
    irmovq $10, %rax
    andq %rax, %rax
    je L
    addq %rax, %rax
    L:
        ret
```

Figure 34: Code 5

```
clk = 0, f_pc =                    0, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 1, f_pc =                   10, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 0, f_pc =                   10, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 1, f_pc =                   20, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 0, f_pc =                   20, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 1, f_pc =                   30, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 0, f_pc =                   30, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 1, f_pc =                   32, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 0, f_pc =                   32, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 1, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 0, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                    x
clk = 1, f_pc =                   41, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 1, rax =                    x
clk = 0, f_pc =                   41, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 1, rax =                    x
clk = 1, f_pc =                   41, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   10
clk = 0, f_pc =                   41, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   10
clk = 1, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   10
clk = 0, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   10
clk = 1, f_pc =                   43, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax =                   10
clk = 0, f_pc =                   43, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax =                   10
clk = 1, f_pc =                   43, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax =                   10
clk = 0, f_pc =                   43, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax =                   10
clk = 1, f_pc =                   43, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax =                   10
clk = 0, f_pc =                   43, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax =                   10
clk = 1, f_pc =                   19, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   20
clk = 0, f_pc =                   19, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   20
clk = 1, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   20
clk = 0, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   20
clk = 1, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   20
clk = 0, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   20
clk = 1, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   20
clk = 0, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   20
processor.v:306: $finish called at 155000 (1ps)
clk = 1, f_pc =                   43, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax =                   20
```



As we can see the bubble and stall signals are created appropriately

2.7.6 Return and Load-Use Combination

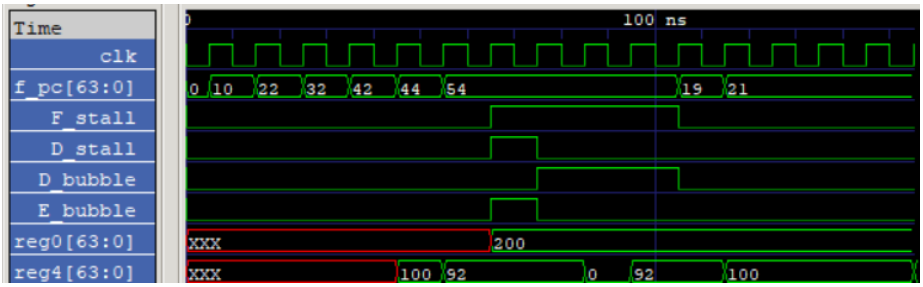
To test this combination we need to load rsp from memory and then return from function.

```
irmovq $100, %rsp
call func
addq %rsp, %rsp
halt

func:
    irmovq $200, %rax
    rmmovq %rsp, (%rax)
    subq %rsp, %rsp
    mrmovq (%rax), %rsp
    ret
```

Figure 35: Code 6

```
clk = 0, f_pc = 0, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = x
clk = 1, f_pc =10, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = x
clk = 0, f_pc =10, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = x
clk = 1, f_pc =22, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = x
clk = 0, f_pc =22, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = x
clk = 1, f_pc =32, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = x
clk = 0, f_pc =32, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = x
clk = 1, f_pc =42, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = x
clk = 0, f_pc =42, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = x
clk = 1, f_pc =44, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = 100
clk = 0, f_pc =44, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp = 100
clk = 1, f_pc =54, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp =92
clk = 0, f_pc =54, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = x, rsp =92
clk = 1, f_pc =54, F_stall = 1, D_stall = 1, D_bubble = 0, E_bubble = 1, rax = 200, rsp =92
clk = 0, f_pc =54, F_stall = 1, D_stall = 1, D_bubble = 0, E_bubble = 1, rax = 200, rsp =92
clk = 1, f_pc =54, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax = 200, rsp =92
clk = 0, f_pc =54, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax = 200, rsp =92
clk = 1, f_pc =54, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax = 200, rsp = 0
clk = 0, f_pc =54, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax = 200, rsp = 0
clk = 1, f_pc =54, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax = 200, rsp =92
clk = 0, f_pc =54, F_stall = 1, D_stall = 0, D_bubble = 1, E_bubble = 0, rax = 200, rsp =92
clk = 1, f_pc =19, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp =92
clk = 0, f_pc =19, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp =92
clk = 1, f_pc =21, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp = 100
clk = 0, f_pc =21, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp = 100
clk = 1, f_pc =21, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp = 100
clk = 0, f_pc =21, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp = 100
clk = 1, f_pc =21, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp = 100
clk = 0, f_pc =21, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp = 100
clk = 1, f_pc =21, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp = 100
clk = 0, f_pc =21, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp = 100
processor.v:306: $finish called at 155000 (1ps)
clk = 1, f_pc =21, F_stall = 0, D_stall = 0, D_bubble = 0, E_bubble = 0, rax = 200, rsp = 200
```



As we can see the stall and bubble values are created appropriately.

2.8 Problems Faced

The only major issue faced was responding to the status codes. In the SEQ processor, whenever a non-AOK status is reached, the processor is stopped. In a pipelined processor, this can't be done because a non-zero status code might be detected while an OK instruction is in a later stage. If we stop the processor, the later instruction won't be executed completely. Hence we need to propagate the status towards the end and only halt when we reach write-back. Although seemingly simple, it was difficult to maintain the "bubble" and "stall" requirements of the pipeline registers.