IPA Project Report

Shravan Gadbail
2022102025
Masumi Desai
2022102057

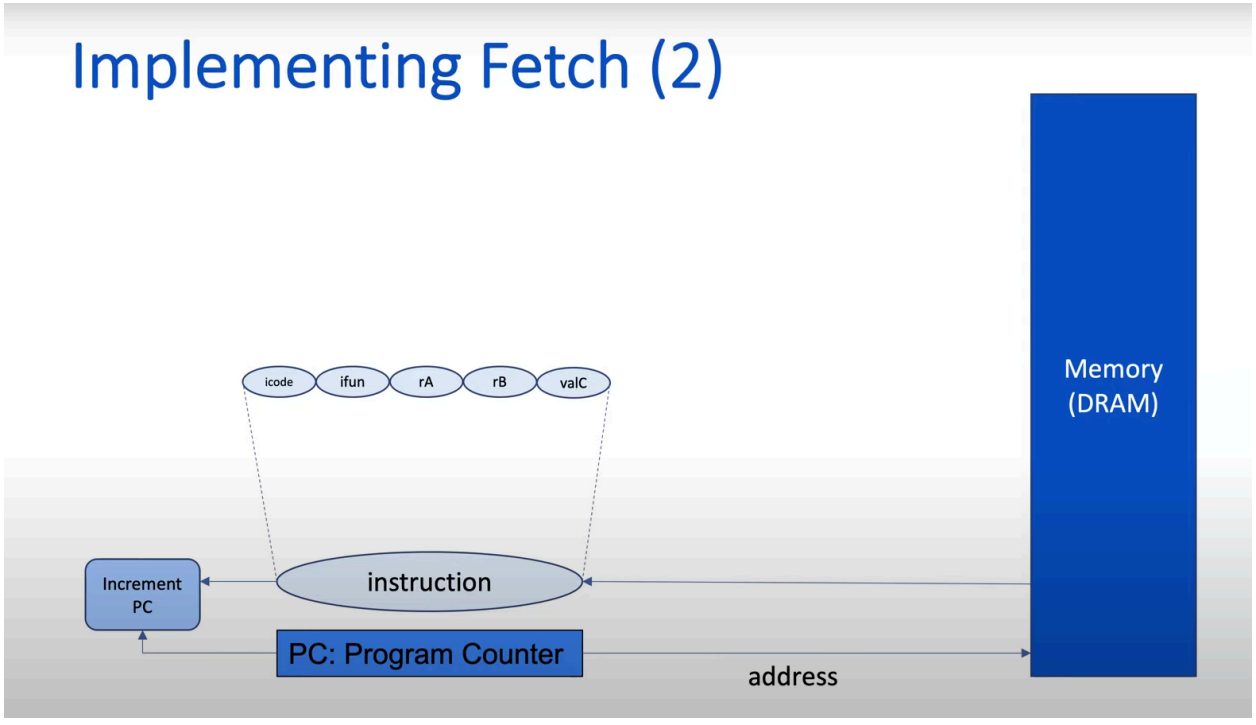
1. Sequential Implementation:

1.1 Introduction to Sequential Implementation:

In this implementation, we go instruction by instruction to execute a program. It consists of 6 stages:

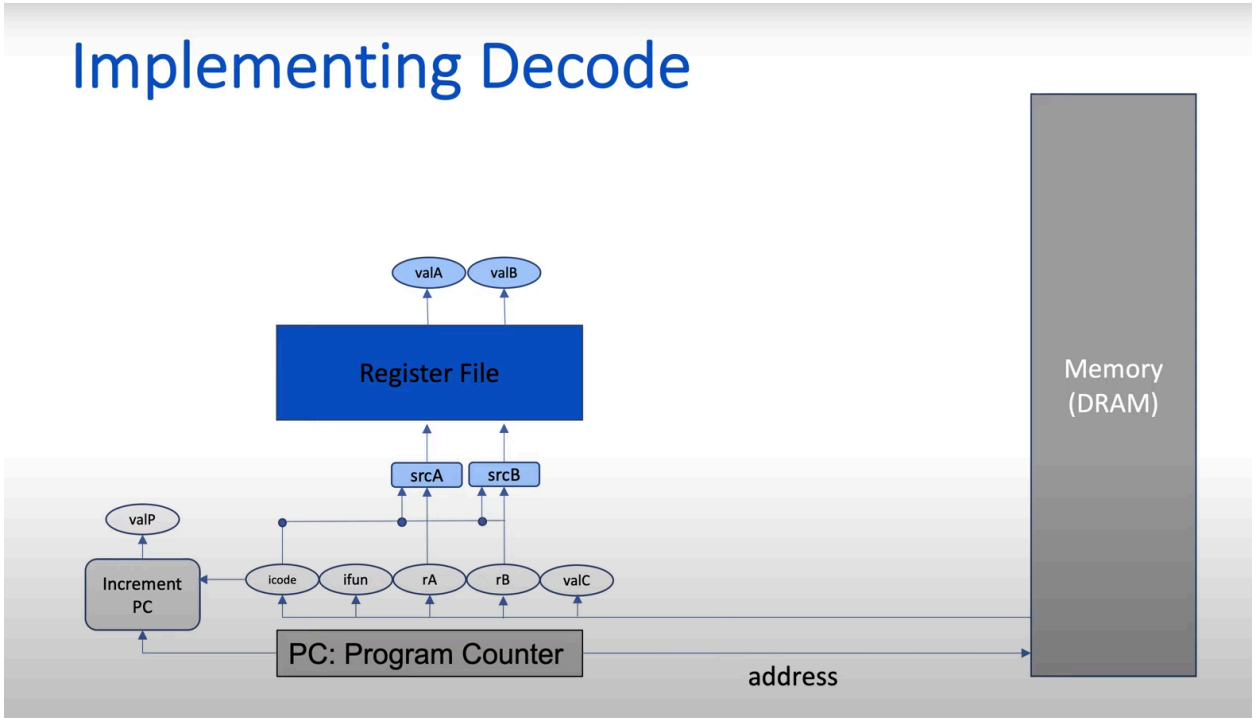
1. Fetch
2. Decode
3. Execute
4. Memory
5. Writeback
6. PC Update

1. Fetch:



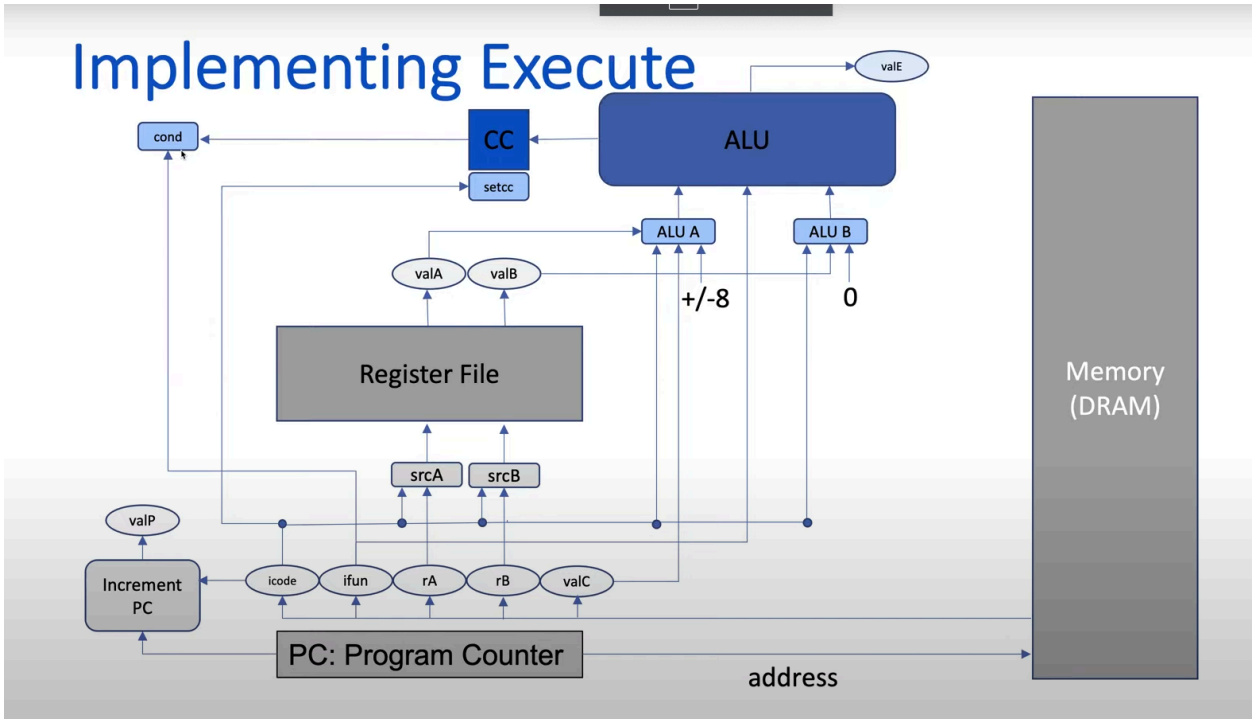
Here we fetch each instruction from the Data memory and identify the correct operands associated with the instruction.

2. Decode:



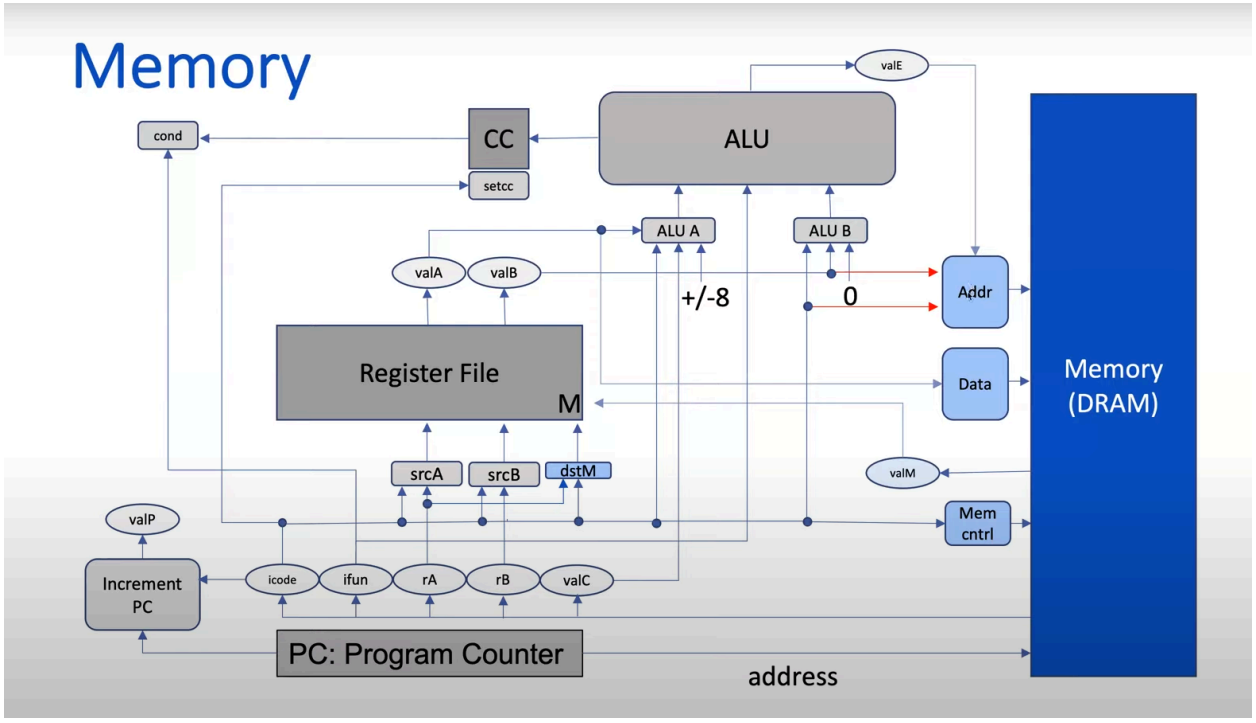
Here we decode the values of the registers stored in registers denoted by rA and rB.

3. Execute:



Here we call the ALU for certain functions and let it perform either address computation or arithmetic computations.

4. Memory:



Here we either write in the memory or read from the memory(DRAM).

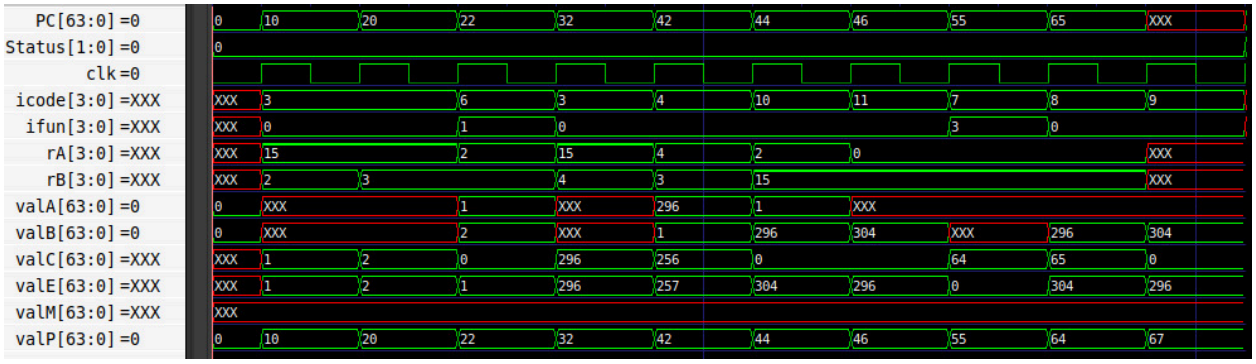
5. Writeback:

   In this stage we write the values that are to be written in the registers. dstM shown in the above photo is the location where the data read from the memory has to be written and dstE is the location where the data read either from the register or the stack ofr immediate value has to be stored in the register.

1.2 Results and Analysis:

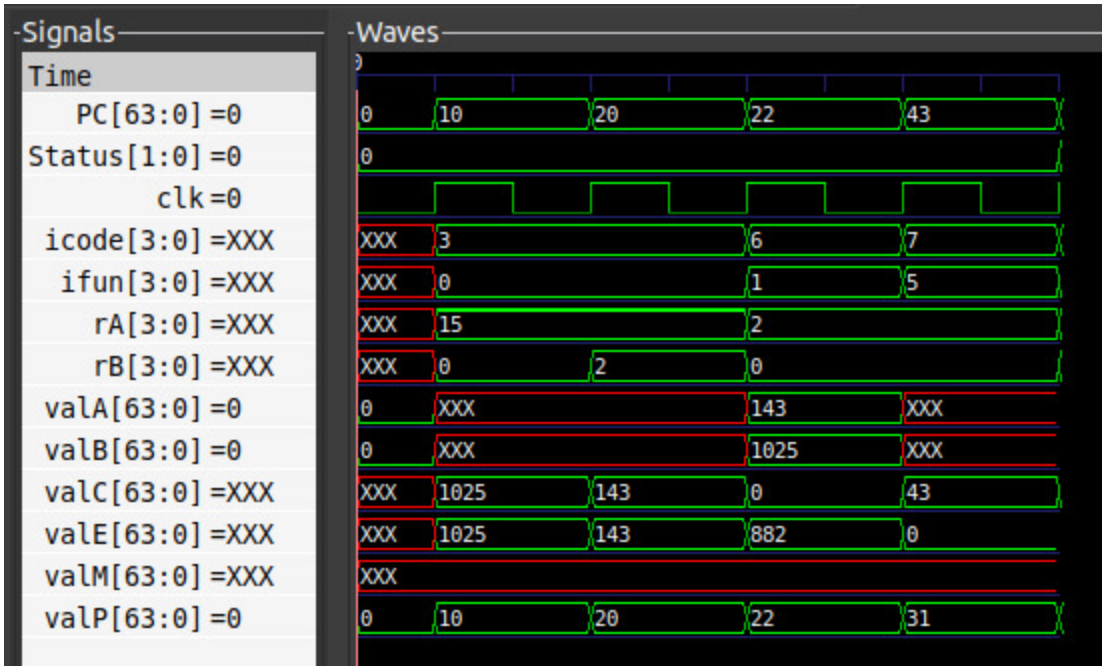
Test case 1:

```
1   irmovl $1,  %rdx
2   irmovl $2, %rbx
3   subl %rdx, %rbx
4   irmovl $128,%rsp
5   rmmovl %rsp, 100(%rbx)
6   pushl %rdx
7   popl  %rax
8   je done
9   call proc
10  addq %rdx, %rcx    ///////
11  done:
12      halt
13  proc:
14      mrmovq 100(%rbx), %rcx   //////
15      ret
```



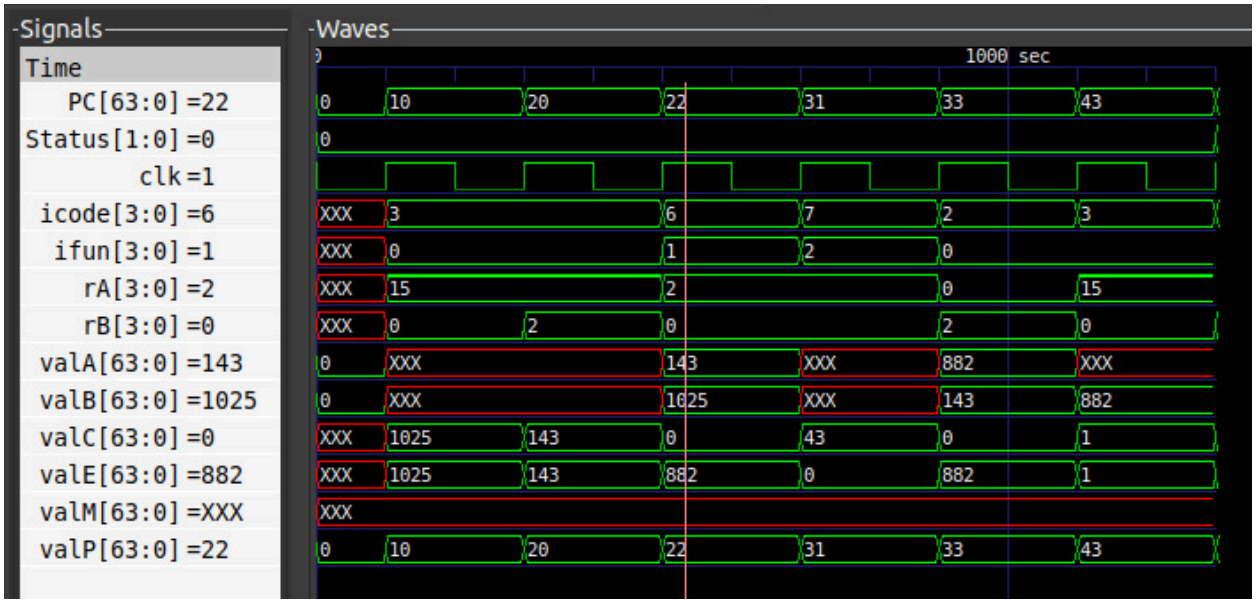
Test case 2: (Jump taken)

```
1   irmovq $0x401, %rax
2   irmovq $0x8F, %rdx
3   subq %rdx, %rax
4   jge  $0x2B
5   rrmovq %rax, %rdx
6   irmovq $0x1, %rax
7   halt
```



Test case 3:

```
1   irmovq $0x401, %rax
2   irmovq $0x8F, %rdx
3   subq %rdx, %rax
4   jl $0x2B
5   rrmovq %rax, %rdx
6   irmovq $0x1, %rax
7   halt
```



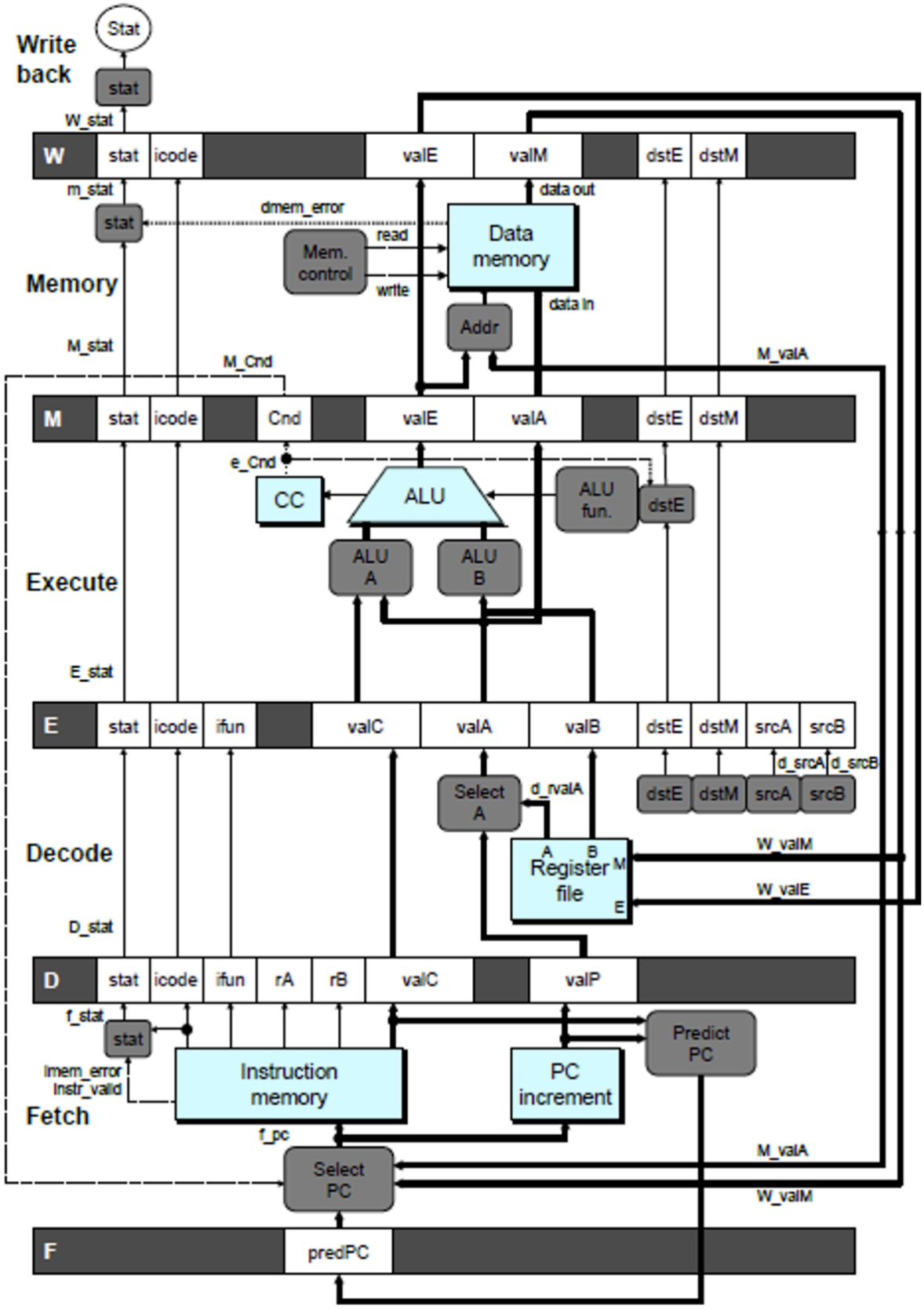
2. Pipelined Implementation:

2.1 Introduction to Pipelined Implementation:

In this type of implementation, our goal is to not leave any stages idle in any clock cycle (for the most part).

This is achieved by running multiple instructions parallelly. Basically, in steady state, in every clock edge, there is one new instruction entering the pipeline and one instruction leaving the pipeline.

In this implementation we first bring the PC update stage to the start of the program, where the next PC will be calculated as soon as the current instruction is fetched.


Write back
Stat
stat
W_stat
W
stat
icode
valE
valM
dstE
dstM
m_stat
stat
dmem_error
data out
Memory
Mem. control
read
write
Data memory
data in
Addr
M_stat
M_Cnd
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
dstE
Execute
ALU A
ALU B
E_stat
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
Select A
d_rvalA
dstE
dstM
srcA
srcB
Decode
A
B
Register file
M
E
W_valM
W_valE
D_stat
D
stat
icode
ifun
rA
rB
valC
valP
f_stat
stat
Predict PC
Imem_error
Instr_valid
Instruction memory
PC increment
Fetch
f_pc
M_valA
Select PC
W_valM
F
predPC

We introduce pipeline registers which store the information of the previous instruction which was carried out in the previous stage.

2.3 Pipeline Control Logic:

Here, the key feature of the pipelined architecture is that at every clock cycle, we update the registers with the values computed in the previous stage. This leads to the updation of the PC and a new instruction moving into the pipeline alongside the previous instruction moving from Fetch to Decode, the instruction before that moving from Decode to Execute and so on.

In such an implementation, the clock cycle may be larger than the Sequential implementation, but the total time spent in completing a number of programs is way lesser than that in Sequential implementation.

2.4 Data Forwarding and Hazards:

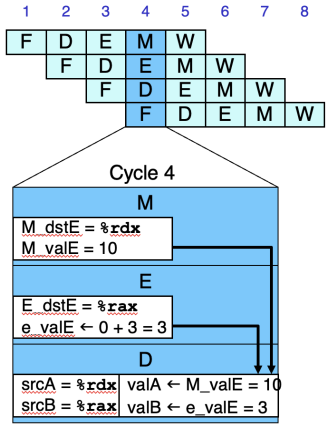
2.4.1 Data Forwarding

In some scenarios, there is data dependency between some instructions. An example would be

>>> irmovq $105,%rbx
>>> irmovq $200,%rcx
>>> addq %rbx,%rcx

In such a situation the instructions
>>> irmovq $105,%rbx
and
>>> irmovq $105,%rcx

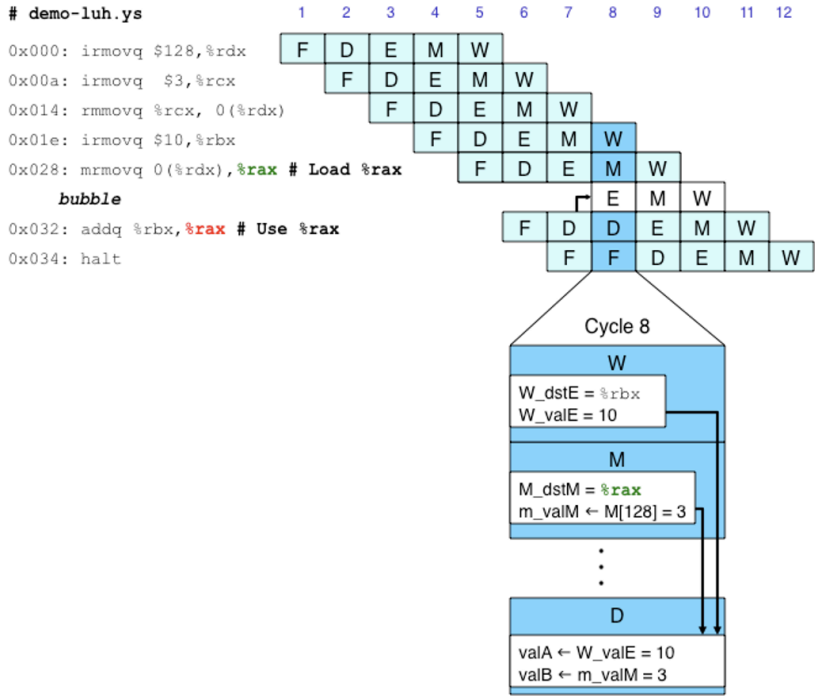
will not be updated until the instructions reach the Writeback stages which isn't until the addq instruction reaches the Memory stage.
So, we can either stall the processor for 3 clock cycles or forward the data from the Execute and memory stages as shown in the diagram below:



2.4.2 Data Hazard

Load/Use Hazard

A Load/Use Hazard occurs when there is a data dependency between a load instruction (which reads data from memory) and a subsequent instruction that uses the data loaded by the load instruction.



As can be seen above, in this case we need to stall for one clock-cycle first before forwarding as the load instruction needs to reach the memory stage in order to know the valM (i.e. value read from the memory).

Therefore, in cases like the above, we stall for one cycle then forward m_valM from the memory to valB in the decode stage.

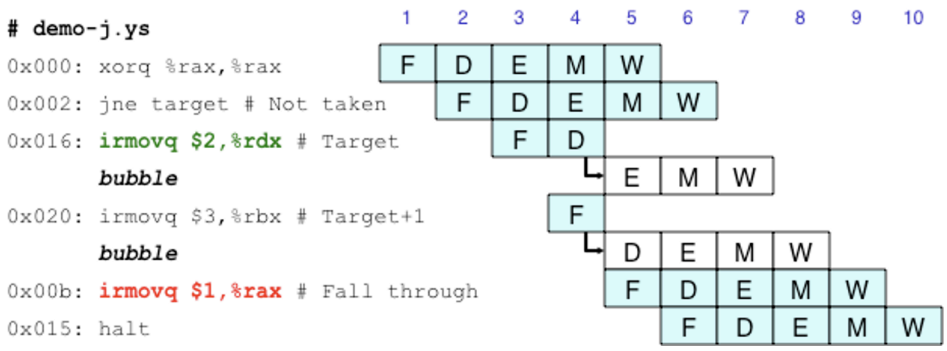
Control for Load/Use Hazard

- Stall instructions in fetch and decode stage
- Insert a bubble in the execute stage

Condition	F	D	E	M	W
Load/Use Hazard	stall	stall	bubble	normal	normal

2.4.3 Control Hazard

Branch Misprediction



As can be seen above, in case of a wrong branch taken, two wrong instructions (two clock-cycle passes by) are fetched and processed till we would know from the execute stage that the predicted branch was incorrect.
This calls for flushing out the two wrongly fetched instructions which are currently in decode and execute and so we insert two bubbles, one in decode and one in execute.

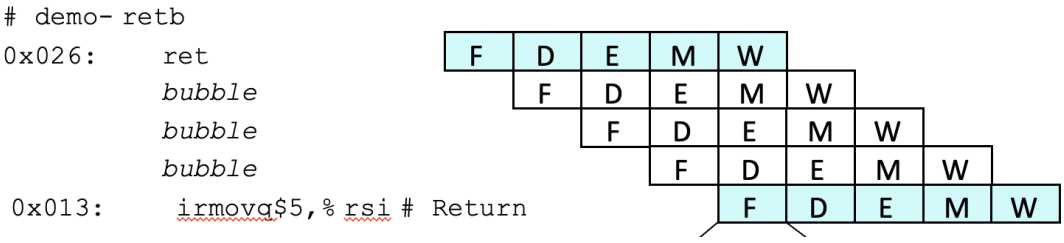
Control for Misprediction

- Insert a bubble in the decode stage
- Insert another bubble in the execute stage

to flush out the wrong instructions out of the processor and start afresh by fetching the correct branch this time.

Return

While ret passes through the pipeline stages - Decode, Execute and Memory, stall the fetch stage so that the correct return-address is given for the next instruction to be fetched.



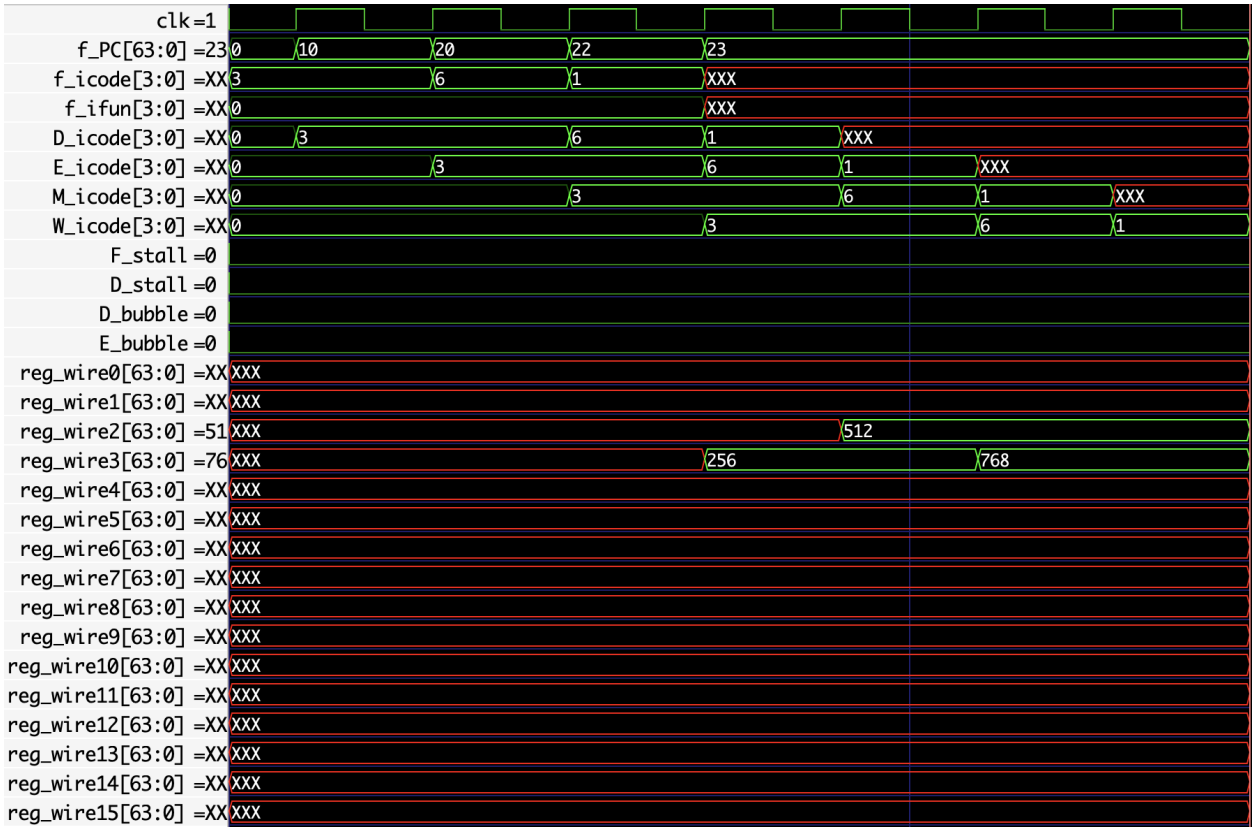
Control for return instruction

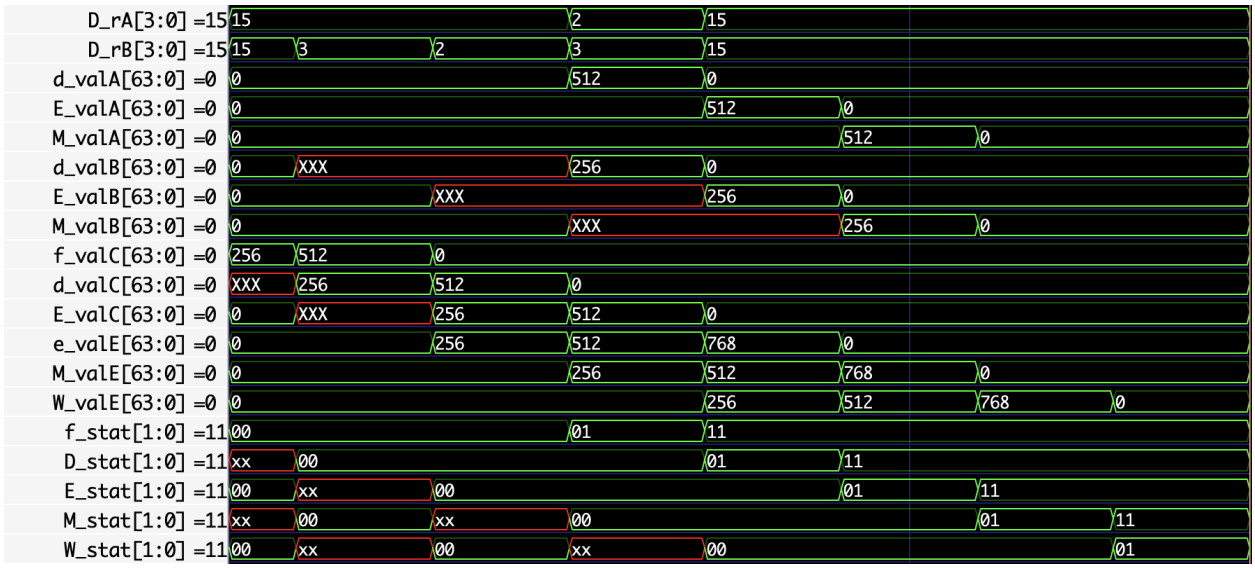
- Insert three bubbles back to back in decode stage
- Keep the fetch stage stalled till ret reaches writeback stage (3 clock-cycles)

2.5 Final Test Case vs Output Photos:

Test case 1:

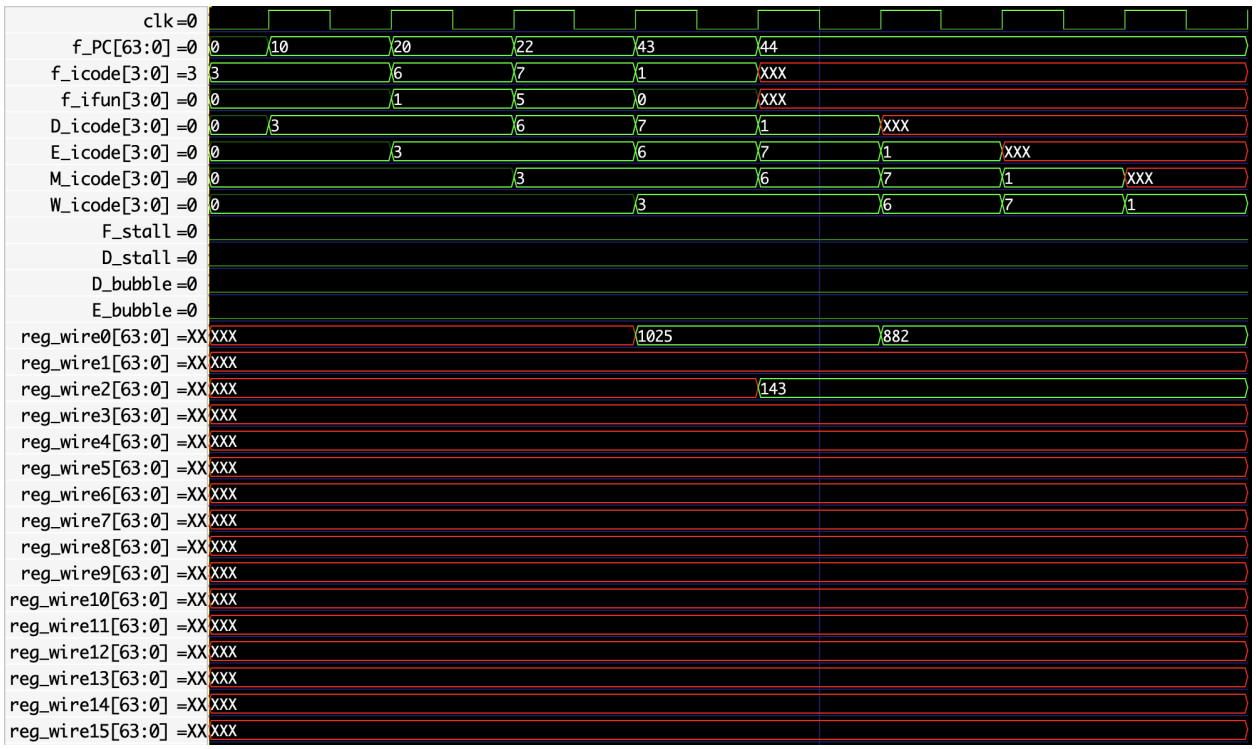
```
1    irmovq $0x100, %rbx
2    irmovq $0x200, %rdx
3    addq %rdx, %rbx
4    halt
```

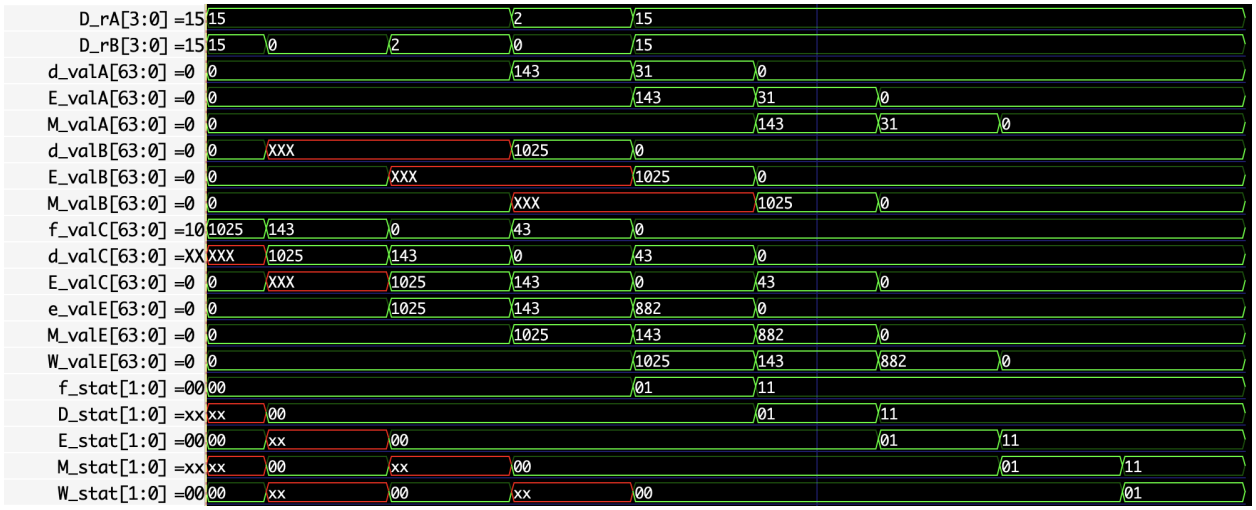



D_rA[3:0] =15
D_rB[3:0] =15
d_valA[63:0] =0
E_valA[63:0] =0
M_valA[63:0] =0
d_valB[63:0] =0
E_valB[63:0] =0
M_valB[63:0] =0
f_valC[63:0] =0
d_valC[63:0] =0
E_valC[63:0] =0
e_valE[63:0] =0
M_valE[63:0] =0
W_valE[63:0] =0
f_stat[1:0] =11
D_stat[1:0] =11
E_stat[1:0] =11
M_stat[1:0] =11
W_stat[1:0] =11

Test case 2:

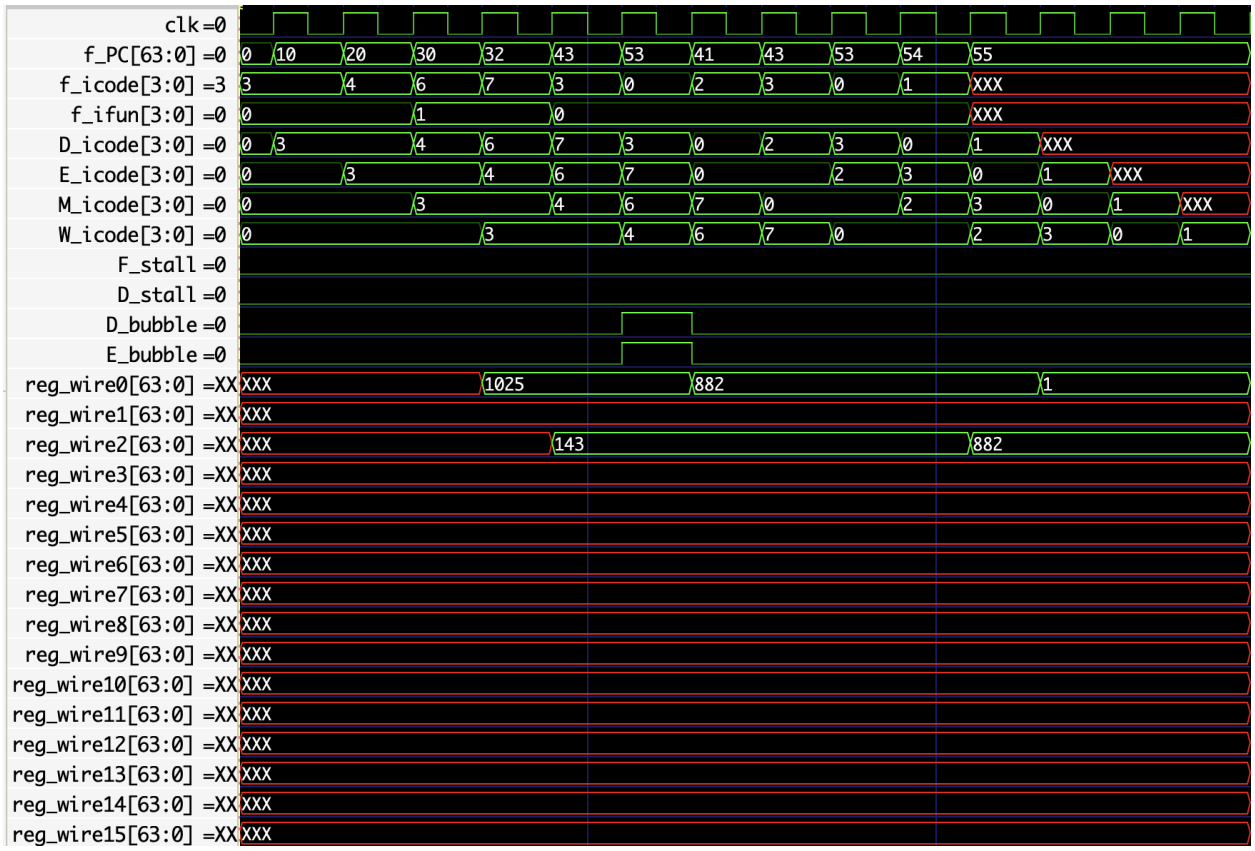
```
1    irmovq $0x401, %rax
2    irmovq $0x8F, %rdx
3    subq %rdx, %rax
4    jge $0x2B
5    rrmovq %rax, %rdx
6    irmovq $0x1, %rax
7    halt
```

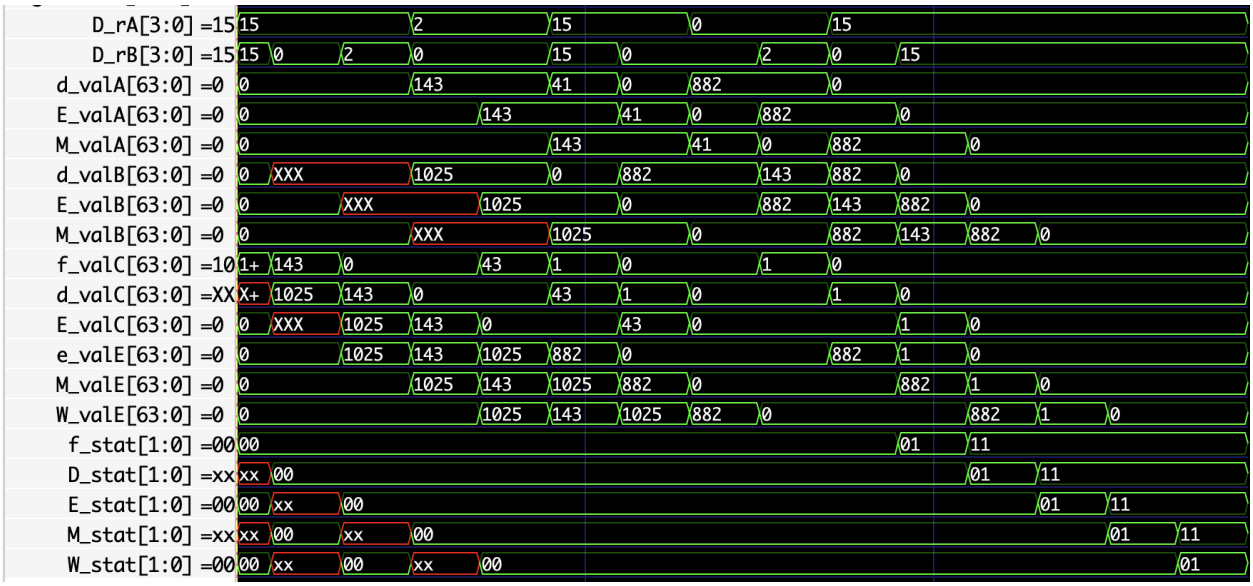



D_rA[3:0] =15
D_rB[3:0] =15
d_valA[63:0] =0
E_valA[63:0] =0
M_valA[63:0] =0
d_valB[63:0] =0
E_valB[63:0] =0
M_valB[63:0] =0
f_valC[63:0] =10
d_valC[63:0] =XX
E_valC[63:0] =0
e_valE[63:0] =0
M_valE[63:0] =0
W_valE[63:0] =0
f_stat[1:0] =00
D_stat[1:0] =xx
E_stat[1:0] =00
M_stat[1:0] =xx
W_stat[1:0] =00

Test case 3:

```
1    irmovq $0x401, %rax
2    irmovq $0x8F, %rdx
3    rmmovq %rdx, (%rax)
4    subq %rdx, %rax
5    jne $0x2B
6    rrmovq %rax, %rdx
7    irmovq $0x1, %rax
8    nop
9    halt
```

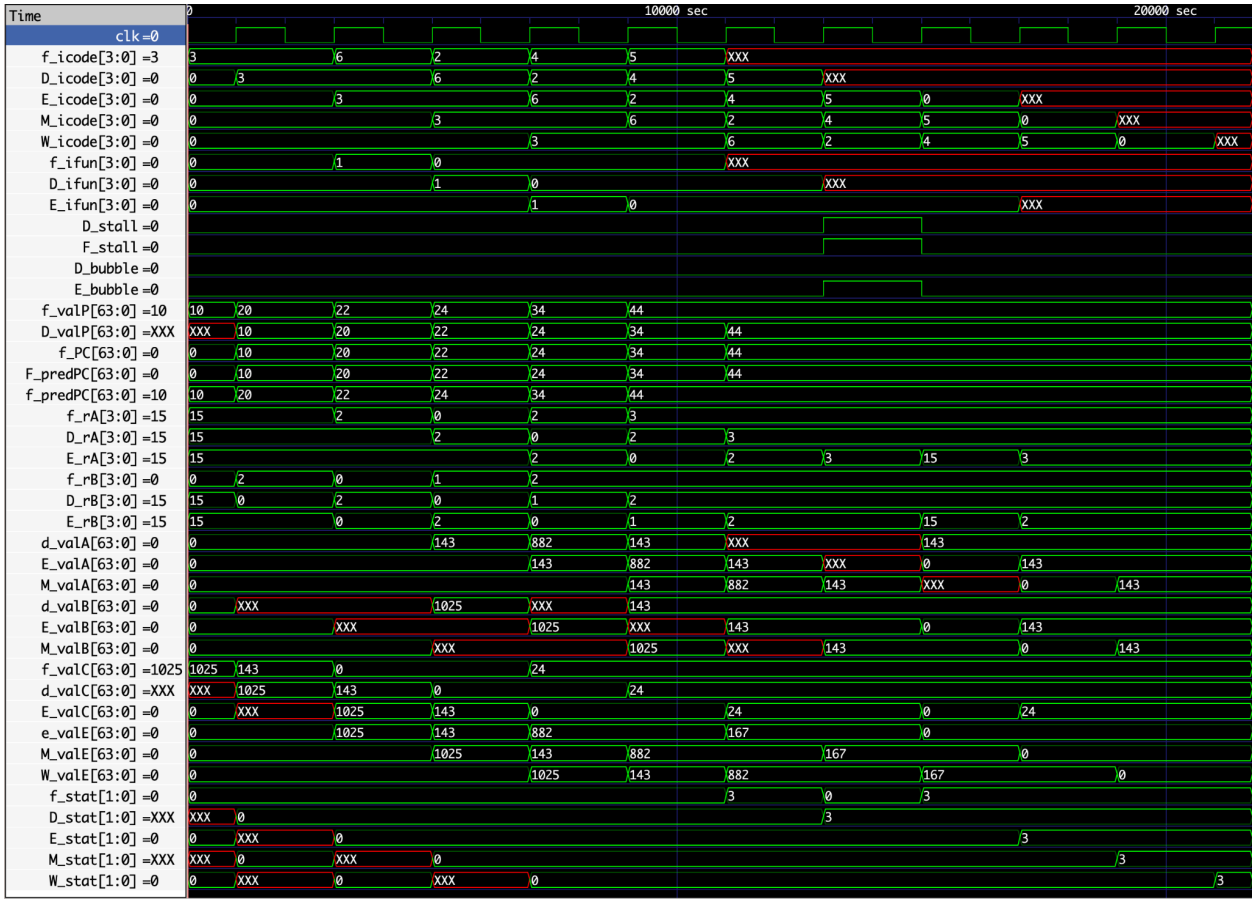


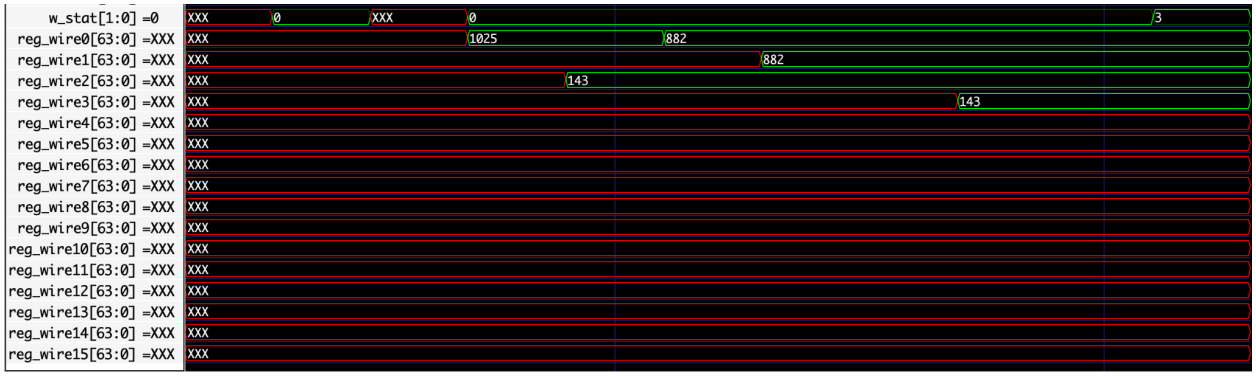

D_rA[3:0] =15
D_rB[3:0] =15
d_valA[63:0] =0
E_valA[63:0] =0
M_valA[63:0] =0
d_valB[63:0] =0
E_valB[63:0] =0
M_valB[63:0] =0
f_valC[63:0] =10
d_valC[63:0] =XX
E_valC[63:0] =0
e_valE[63:0] =0
M_valE[63:0] =0
W_valE[63:0] =0
f_stat[1:0] =00
D_stat[1:0] =xx
E_stat[1:0] =00
M_stat[1:0] =xx
W_stat[1:0] =00

Test case 4:

4_assembly.txt

```
1    irmovq $0x401, %rax
2    irmovq $0x8F, %rdx
3    subq %rdx, %rax
4    cmove %rax, %rcx
5    rmmovq %rdx, $0x18(%rdx)
6    mrmovq $0x18(%rdx), %rbx
```

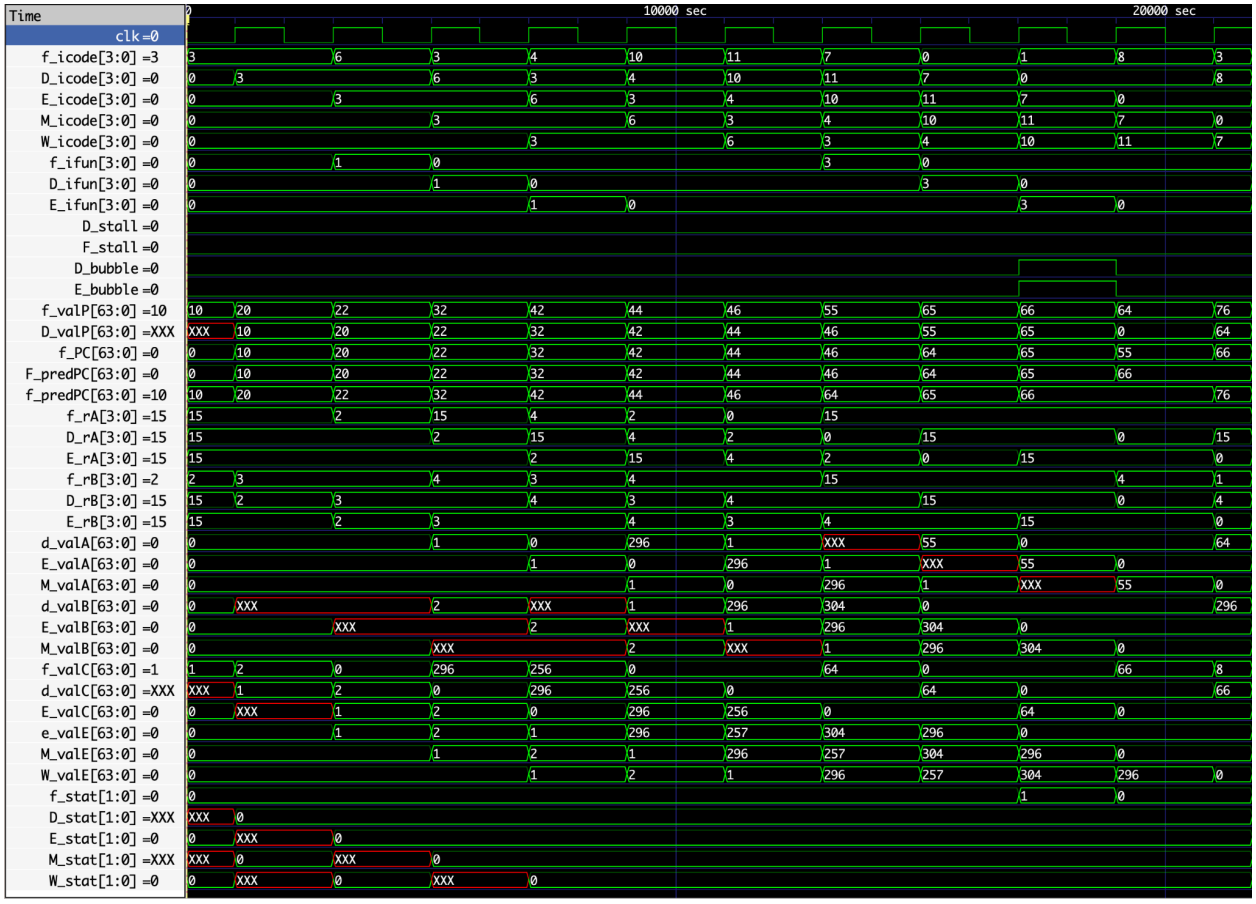


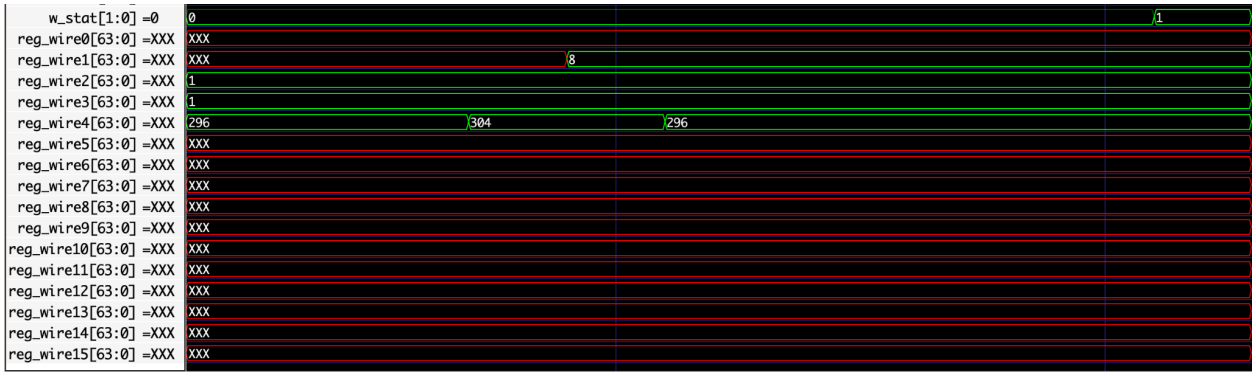

w_stat[1:0] =0
reg_wire0[63:0] =XXX
reg_wire1[63:0] =XXX
reg_wire2[63:0] =XXX
reg_wire3[63:0] =XXX
reg_wire4[63:0] =XXX
reg_wire5[63:0] =XXX
reg_wire6[63:0] =XXX
reg_wire7[63:0] =XXX
reg_wire8[63:0] =XXX
reg_wire9[63:0] =XXX
reg_wire10[63:0] =XXX
reg_wire11[63:0] =XXX
reg_wire12[63:0] =XXX
reg_wire13[63:0] =XXX
reg_wire14[63:0] =XXX
reg_wire15[63:0] =XXX
XXX
0
XXX
0
3
1025
882
882
143
143

Test case 5:

5_assembly.txt

```
1    irmovl $1, %rdx
2    irmovl $2, %rbx
3    subl %rdx, %rbx
4    irmovl $128,%rsp
5    rmmovl %rsp, 100(%rbx)
6    pushl %rdx
7    popl  %rax
8    je done
9    call proc
10   done:
11       nop
12       halt
13   proc:
14       irmovq $8, %rcx
15       ret
```

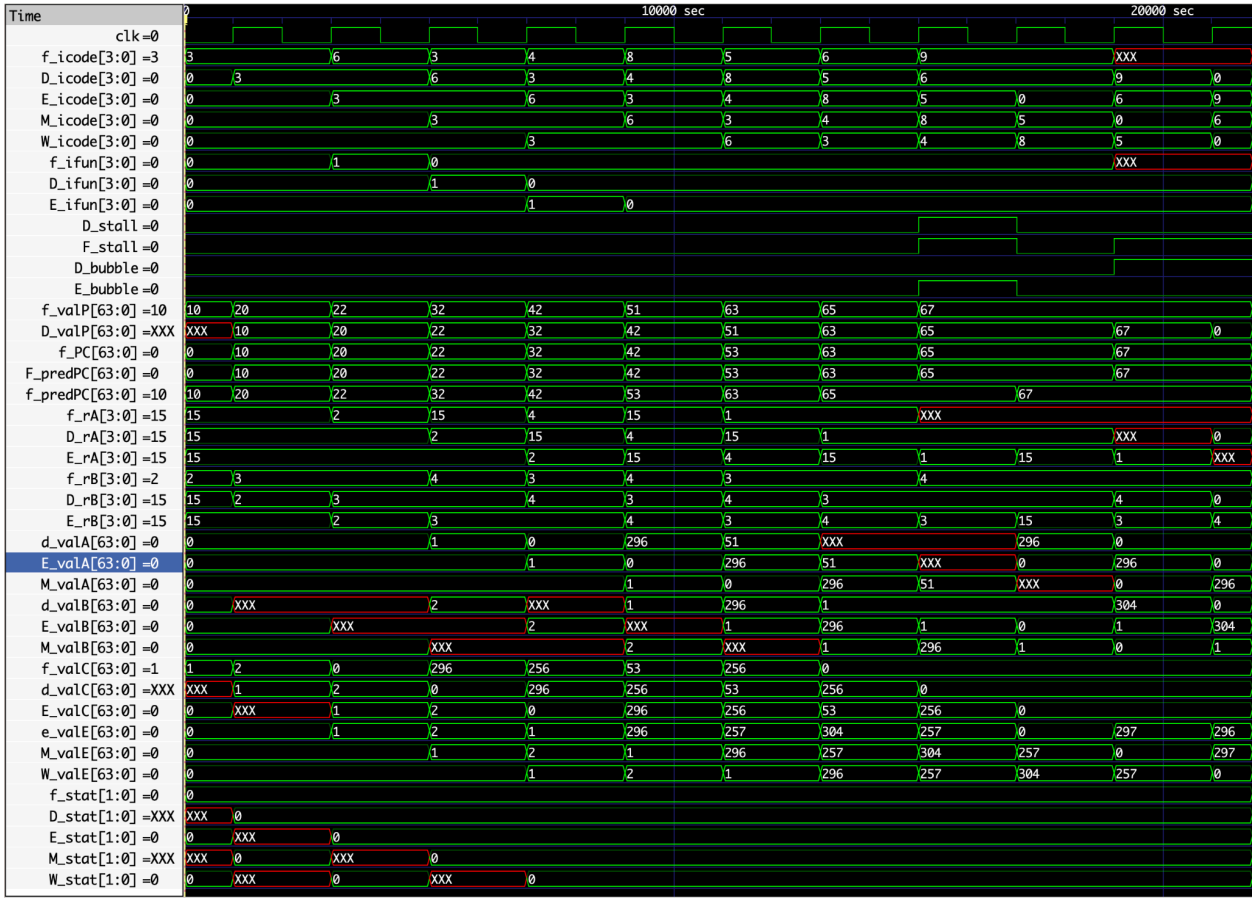


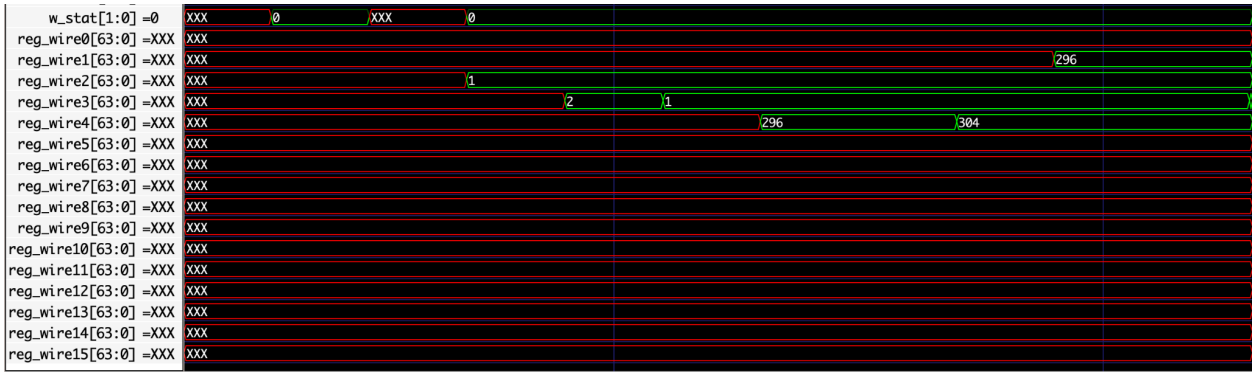

w_stat[1:0] =0
reg_wire0[63:0] =XXX
reg_wire1[63:0] =XXX
reg_wire2[63:0] =XXX
reg_wire3[63:0] =XXX
reg_wire4[63:0] =XXX
reg_wire5[63:0] =XXX
reg_wire6[63:0] =XXX
reg_wire7[63:0] =XXX
reg_wire8[63:0] =XXX
reg_wire9[63:0] =XXX
reg_wire10[63:0] =XXX
reg_wire11[63:0] =XXX
reg_wire12[63:0] =XXX
reg_wire13[63:0] =XXX
reg_wire14[63:0] =XXX
reg_wire15[63:0] =XXX
0
1
XXX
8
1
1
296
304
296

Test case 6:

6_assembly.txt

```
1    irmovl $1,  %rdx
2    irmovl $2, %rbx
3    subl %rdx, %rbx
4    irmovl $128,%rsp
5    rmmovl %rsp, 100(%rbx)
6    call proc
7    nop
8    halt
9    proc:
10       mrmovq $100(%rbx), %rcx
11       addq %rcx,%rbx
12       ret
```




w_stat[1:0] =0
reg_wire0[63:0] =XXX
reg_wire1[63:0] =XXX
reg_wire2[63:0] =XXX
reg_wire3[63:0] =XXX
reg_wire4[63:0] =XXX
reg_wire5[63:0] =XXX
reg_wire6[63:0] =XXX
reg_wire7[63:0] =XXX
reg_wire8[63:0] =XXX
reg_wire9[63:0] =XXX
reg_wire10[63:0] =XXX
reg_wire11[63:0] =XXX
reg_wire12[63:0] =XXX
reg_wire13[63:0] =XXX
reg_wire14[63:0] =XXX
reg_wire15[63:0] =XXX
XXX
0
XXX
0
296
1
2
1
296
304

3 Challenges Faced in Both Implementations

SEQ

A. We struggled a little with implementing the sequential nature of this type of processor in verilog as we were confused as to when to trigger a particular stage (for example at posedge clk or whenever its inputs change etc.)
B. Integrating all the different modules and stages into one processor and keeping in check the entire flow of the instructions was a bit of a task for us.
C. As it was implementing a whole processor from scratch, we made a lot of silly logical errors throughout the process, which took us a lot of time to rectify and a lot of “dry running” of various instructions to finally make flawless y86-64 SEQ.

Pipeline

A. Pipeline implementation was very complex to think about.
B. We struggled the most with debugging the code, where we had to jump within all modules in order to debug one small part of the code.
C. One more thing that we struggled with was that we had to change a lot of ports in each and every module of SEQ to convert that to Pipeline. We were clueless when we would miss some or the other module and keep trying to debug the code while the error was just changing the ports.