Y86 Processor
Project Report

By:

Dikshant (2022102038)

Manas Deshmukh (2022102040)

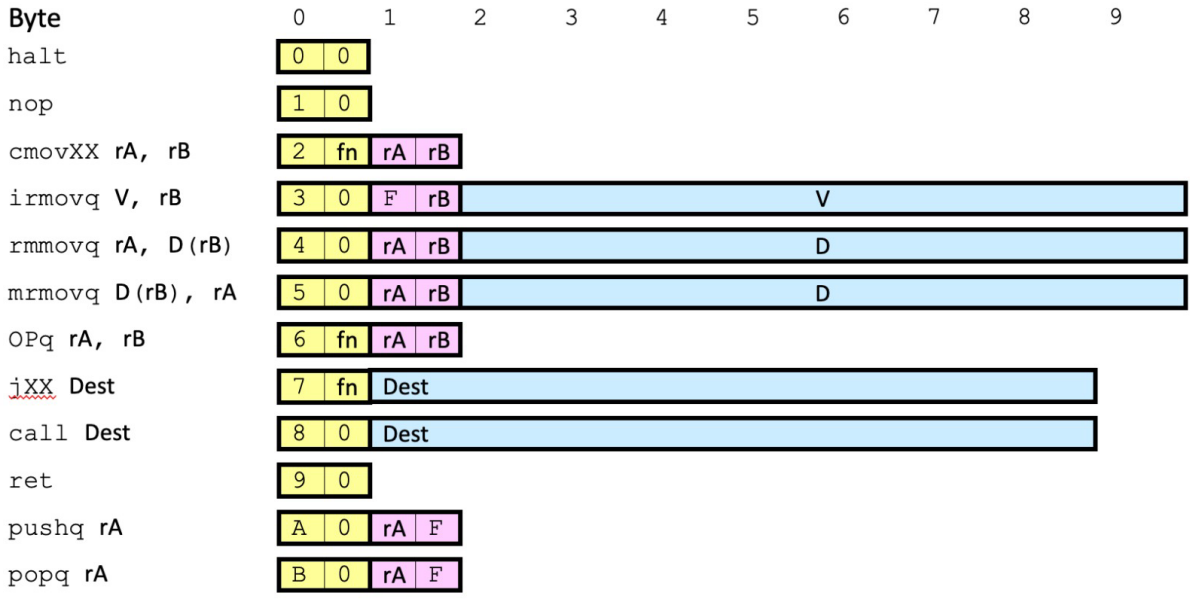
AIM:

Implement y86-64 Processor that's able to execute all instructions in Y86 ISA.

Brief About y86-64 ISA:

Y86-64 is the simplified version of x86-64 arch used by Intel. The y86-64 instruction set contains instructions for arithmetic and logical operations, comparison instructions, data movement instructions, stack instructions, jump instructions, and special instructions. The arithmetic and logic instructions include addition, subtraction, and bitwise operations like 'AND' and 'XOR'.

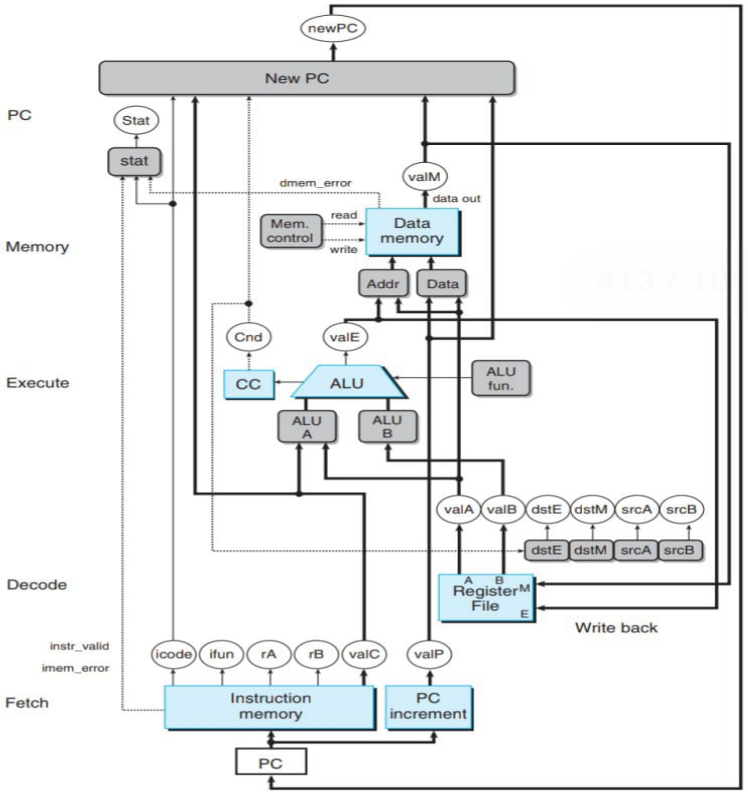
y86-64 Inst. Encodings:



Fun Codes for y86-64 Inst. Set:

Operations			Branches				Moves					
addq	6	0	jmp	7	0	jne	7	4	rrmovq	2	0	cmovne	2	4
subq	6	1	jle	7	1	jge	7	5	cmovle	2	1	cmovge	2	5
andq	6	2	jl	7	2	jg	7	6	cmovl	2	2	cmovg	2	6
xorq	6	3	je	7	3				cmove	2	3			

Sequential y-86 Processor:



Sequential Arch consists of 6 stages namely:

- Fetch
- Decode
- Execute
- Memory Stage
- Write Back in Registers
- PC Update

In a sequential processor, instructions are stored in memory, and the processor fetches instructions one by one from memory, decodes them, and executes them in order. The processor can perform

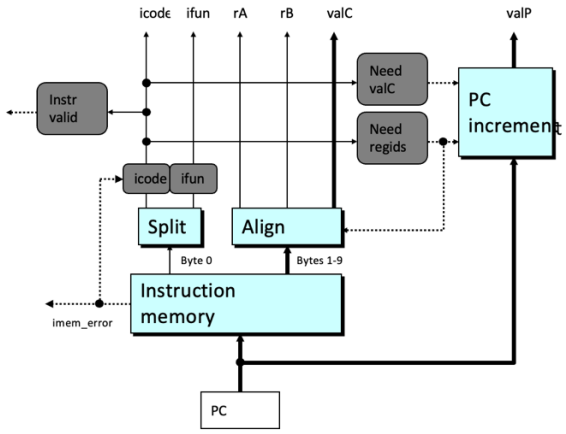
only one instruction at a time, and instructions cannot overlap in their execution.

The SEQ implementation of Y86-64 processor works according to the following table:

Stage	HALT	NOP	CMOV	IRMOVQ
Fch	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC + 10
Dec			valA ← R[rA]	
Exe	cpu.stat = HLT		valE ← valA Cnd ← Cond(CC,ifun)	valE ← valC
Mem				
WB			Cnd ? R[rB] ← valE	R[rB] ← valE
PC	PC ← 0	PC ← valP	PC ← valP	PC ← valP
Stage	**RMMOVQ**	**MRMOVQ**	**OPq**	**jXX**
Fch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC + 9
Dec	valA ← R[rA] valB ← R[rB]	 valB ← R[rB]	valA ← R[rA] valB ← R[rB]	
Exe	valE ← valB + valC	valE ← valB + valC	valE ← valB OP valA Set CC	Cnd ← Cond(CC,ifun)
Mem	M_8[valE] ← valA	valM ← M_8[valE]		
WB		R[rA] ← valM	R[rB] ← valE	
PC	PC ← valP	PC ← valP	PC ← valP	PC ← Cnd ? valC:valP
Stage	**CALL**	**RET**	**PUSHQ**	**POPQ**
Fch	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC + 9	icode:ifun ← M_1[PC] valP ← PC + 1	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC + 2
Dec	 valB ← R[RSP]	valA ← R[RSP] valB ← R[RSP]	valA ← R[rA] valB ← R[RSP]	valA ← R[RSP] valB ← R[RSP]
Exe	valE ← valB - 8	valE ← valB + 8	valE ← valB - 8	valE ← valB + 8
Mem	M_8[valE] ← valP	valM ← M_8[valA]	M_8[valE] ← valA	valM ← M_8[valA]
WB	R[RSP] ← valE	R[RSP] ← valE	R[RSP] ← valE	R[RSP] ← valE R[rA] ← valM
PC	PC ← valC	PC ← valM	PC ← valP	PC ← valP

SEQUENTIAL PROCESSOR

FETCH:



Fetch is the first stage of the execution cycle. It's responsible for fetching the next instruction from memory and storing it in inst. registers.

```
module fetch_seq(
    input [63:0] p_ctr,
    input clock,
    output reg [3:0] in_code,
    output reg [3:0] in_fun,
    output reg [3:0] rb,
    output reg [3:0] ra,
    output reg flag_halt,
    output reg [63:0] val_c,
    output reg [63:0] val_p,
    output reg bad_mem,
    output reg in_error
     );
```

Working:

- On positive edge of clock, instruction is fetched from Inst. memory based on current PC value. In the fetch module we've declared Inst. memory as 1024 bytes.
- If PC is greater than 1023 then it gives 'bad_mem' error.
- For a valid PC, 10 consecutive bytes (PC byte included) are considered as 'instruction'.

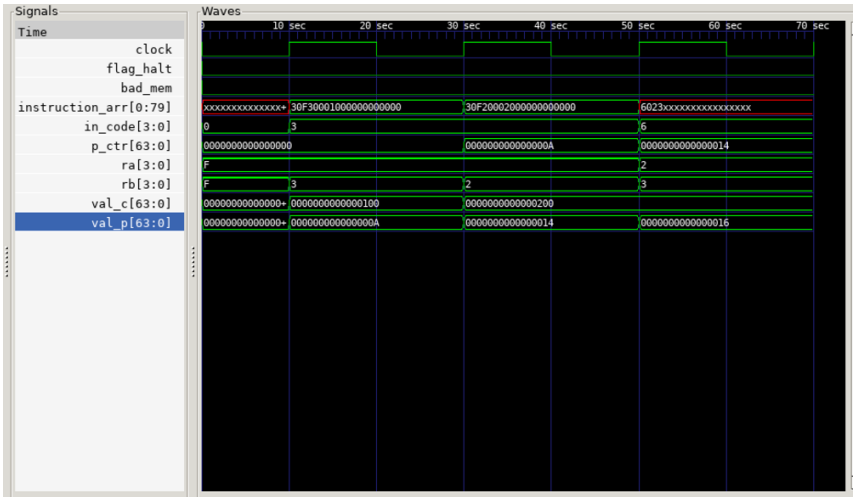
- in_code and in_fun corresponds to the first 4 and last 4 bits of the instruction respectively.
- If in_code is invalid it sets ‘in_error’ as 1.
- If in_code is 0000 then ‘halt_flag’ is set to 1.
- Depending on the value of in_code, other output values are obtained from instruction reg. as follows:
    - ra ≡ instruction_arr[8:11]
    - rb ≡ instruction_arr[12:15]
    - val_c ≡ offset/ destination/ value (read in reverse order).
    - val_p = PC + 1 for halt, nop, ret.
    - val_p = PC + 2 for halt, pop, push, rrmovq, opq.
    - val_p = PC + 9 for jxx, call.
    - val_p = PC + 10 for remaining operations.

Consider Inst. Code:

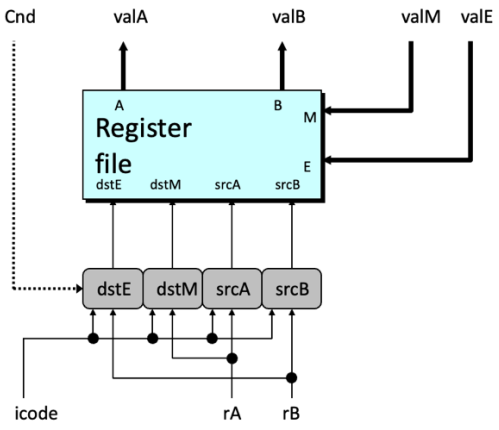
```
00110000
11110011
00000000
00000001
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00110000
11110010
00000000
00000010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
01100000
00100011
```

OUTPUT:

```
pingu@ubuntu ~/p/n/f/p/S/fetch (main)> iverilog -o out.vvp testbench.v design.v
pingu@ubuntu ~/p/n/f/p/S/fetch (main)> vvp out.vvp
VCD info: dumpfile fetch_tb.vcd opened for output.
                0    clk = 0 p_ctr = 0
              in_code = 0000 in_fun = 0000 ra = 1111 rb = 1111 val_c = 0 val_p = 0 bad_mem = 0, in_error = 0, halt = 0

               10    clk = 1 p_ctr = 0
              in_code = 0011 in_fun = 0000 ra = 1111 rb = 0011 val_c = 256 val_p = 10 bad_mem = 0, in_error = 0, halt = 0

               20    clk = 0 p_ctr = 0
              in_code = 0011 in_fun = 0000 ra = 1111 rb = 0011 val_c = 256 val_p = 10 bad_mem = 0, in_error = 0, halt = 0

               30    clk = 1 p_ctr = 10
              in_code = 0011 in_fun = 0000 ra = 1111 rb = 0010 val_c = 512 val_p = 20 bad_mem = 0, in_error = 0, halt = 0

               40    clk = 0 p_ctr = 10
              in_code = 0011 in_fun = 0000 ra = 1111 rb = 0010 val_c = 512 val_p = 20 bad_mem = 0, in_error = 0, halt = 0

               50    clk = 1 p_ctr = 20
              in_code = 0110 in_fun = 0000 ra = 0010 rb = 0011 val_c = 512 val_p = 22 bad_mem = 0, in_error = 0, halt = 0

               60    clk = 0 p_ctr = 20
              in_code = 0110 in_fun = 0000 ra = 0010 rb = 0011 val_c = 512 val_p = 22 bad_mem = 0, in_error = 0, halt = 0

               70    clk = 1 p_ctr = 20
              in_code = 0110 in_fun = 0000 ra = 0010 rb = 0011 val_c = 512 val_p = 22 bad_mem = 0, in_error = 0, halt = 0
```



DECODE:



Decode stage is responsible for decoding the instruction fetched in the prev. stage.

In decode stage val_a and val_b are read from the registers with address ra and rb respectively. Sometimes stack-pointer(%rsp) depending upon the in_code.

```
module decode_proc(
    input [3:0] in_fun,
    input [63:0] val_m,
    input [3:0] in_code,
    input [63:0] val_e,
    input clock,
    input [3:0] rb,
    input [3:0] ra,
    output reg [63:0] val_b,
    output reg [63:0] val_a
);
```

Working:

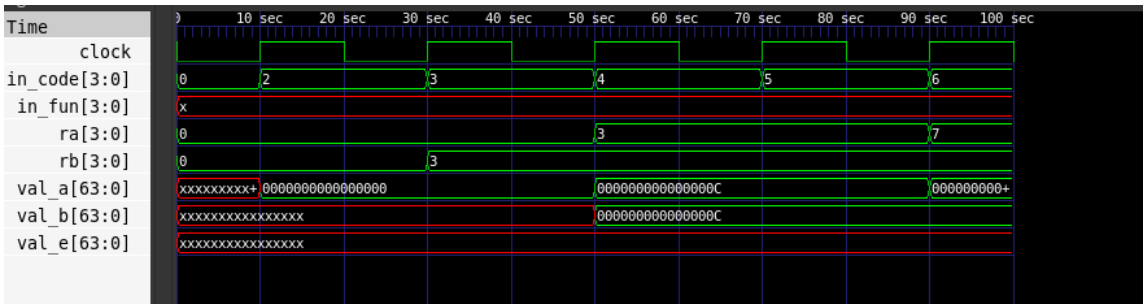
- We have implemented the register memory (hexadecimal format) in ‘common.txt’ it contains values stored in the registers in both decode and write_back stage.
- Val_a and val_b are present in the registers given by the address ra and rb. We use ra, rb and reg_chunk to obtain val_a and val_b.
- val_b = reg_chunk[rb] for rrmovq, irmovq, imrovq, opq.
- val_b = reg_chunk[4] (where 4 denotes %rsp)for call, pushq, popq, ret
- val_a = reg_chunk[ra] for rrmovq, rmmovq, opq, pushq.
- Val_a = reg_chunk[4] for popq, ret.

Reg_chunk:

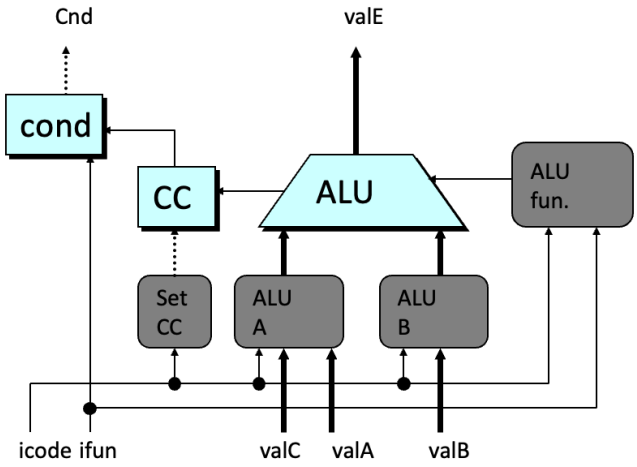
```
1   0000000000000000
2   000000000000000c
3   0000000000000010
4   000000000000000c
5   0000000000000005
6   0000000000000005
7   0000000000000008
8   0000000000000007
9   0000000000000009
10  0000000000000009
11  000000000000000d
12  000000000000000c
13  000000000000000e
14  000000000000000a
15  000000000000000a
```

OUTPUT:

```
pingu@ubuntu ~/p/n/f/p/S/decode (main)> iverilog -o out.vvp testbench.v design.v
pingu@ubuntu ~/p/n/f/p/S/decode (main)> vvp out.vvp
VCD info: dumpfile dump.vcd opened for output.
time= 0, Clock=0, in_code=0000, val_a=0, val_b=0,ra=0, rb=0
time= 10, Clock=1, in_code=0010, val_a=0, val_b=0,ra=0, rb=0
time= 20, Clock=0, in_code=0010, val_a=0, val_b=0,ra=0, rb=0
time= 30, Clock=1, in_code=0011, val_a=0, val_b=0,ra=0, rb=3
time= 40, Clock=0, in_code=0011, val_a=0, val_b=0,ra=0, rb=3
time= 50, Clock=1, in_code=0100, val_a=12, val_b=12,ra=3, rb=3
time= 60, Clock=0, in_code=0100, val_a=12, val_b=12,ra=3, rb=3
time= 70, Clock=1, in_code=0101, val_a=12, val_b=12,ra=3, rb=3
time= 80, Clock=0, in_code=0101, val_a=12, val_b=12,ra=3, rb=3
time= 90, Clock=1, in_code=0110, val_a=7, val_b=12,ra=7, rb=3
time= 100, Clock=0, in_code=0110, val_a=7, val_b=12,ra=7, rb=3
```



EXECUTE



In this stage Arithmetic Operations like Add, Subrtract, And, Xor are performed on A and B depending on in_fun and in_code. Condition codes are also set in this stage.

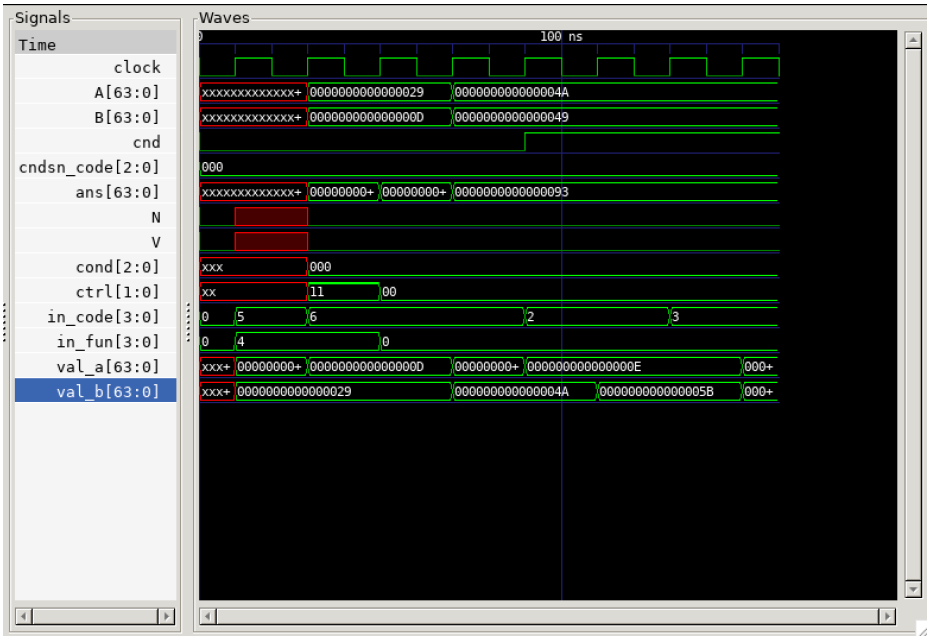
```
`include "alu.v"

module exe(
    input [3:0] in_code, in_fun,
    input [63:0] val_a, val_b, val_c,
    input clock,

    output reg [63:0] val_e,
    output reg cnd
);
```

Working:

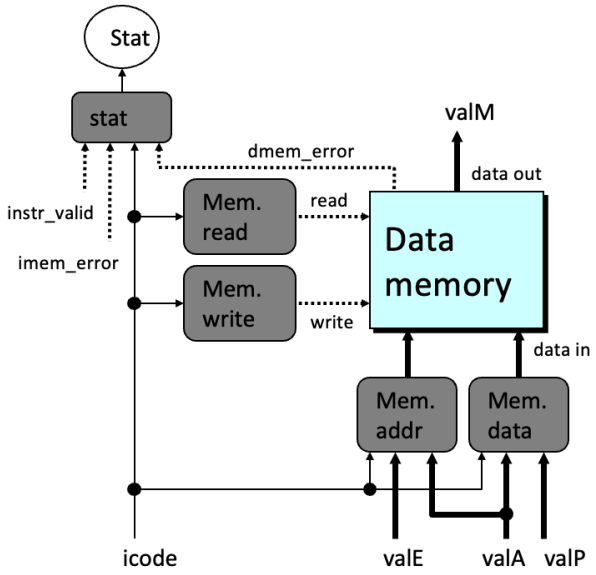
- Instructions are performed when the clock is high.
- ‘irmovq’: val_e = val_c
- ‘mrmovq/ rmmovq’: val_e = val_b + val_c
- ‘opq’: operation corresponding to in_fun is performed and ans is stored in val_e.
- ‘jxx/ cmov’: cnd is set according to condition codes.
- ‘call/ pushq’: val_e = val_b – 1
- ‘ret/ popq’: val_e = val_b + 1

OUTPUT:

```
pingu@ubuntu ~/p/n/f/p/S/execute (main)> iverilog -o out.vvp testbench.v design.v
pingu@ubuntu ~/p/n/f/p/S/execute (main)> vvp out.vvp
VCD info: dumpfile dump.vcd opened for output.
clock=0  in_fun=0000     in_code=0000    val_b=0         val_c=0         val_a=0         cnd=0   val_e=0

clock=1  in_fun=0100     in_code=0101    val_b=41        val_c=23        val_a=10        cnd=0   val_e=64

clock=0  in_fun=0100     in_code=0101    val_b=41        val_c=23        val_a=10        cnd=0   val_e=64

clock=1  in_fun=0100     in_code=0110    val_b=41        val_c=23        val_a=13        cnd=0   val_e=36

clock=0  in_fun=0100     in_code=0110    val_b=41        val_c=23        val_a=13        cnd=0   val_e=36

clock=1  in_fun=0000     in_code=0110    val_b=41        val_c=117       val_a=13        cnd=0   val_e=54

clock=0  in_fun=0000     in_code=0110    val_b=41        val_c=117       val_a=13        cnd=0   val_e=54

clock=1  in_fun=0000     in_code=0110    val_b=74        val_c=117       val_a=73        cnd=0   val_e=147

clock=0  in_fun=0000     in_code=0110    val_b=74        val_c=117       val_a=73        cnd=0   val_e=147

clock=1  in_fun=0000     in_code=0010    val_b=74        val_c=19        val_a=14        cnd=1   val_e=14

clock=0  in_fun=0000     in_code=0010    val_b=74        val_c=19        val_a=14        cnd=1   val_e=14

clock=1  in_fun=0000     in_code=0010    val_b=91        val_c=19        val_a=14        cnd=1   val_e=14

clock=0  in_fun=0000     in_code=0010    val_b=91        val_c=19        val_a=14        cnd=1   val_e=14

clock=1  in_fun=0000     in_code=0011    val_b=91        val_c=19        val_a=14        cnd=1   val_e=19

clock=0  in_fun=0000     in_code=0011    val_b=91        val_c=19        val_a=14        cnd=1   val_e=19

clock=1  in_fun=0000     in_code=0011    val_b=111       val_c=121       val_a=101       cnd=1   val_e=121

clock=0  in_fun=0000     in_code=0011    val_b=111       val_c=121       val_a=101       cnd=1   val_e=121

pingu@ubuntu ~/p/n/f/p/S/execute (main)>
```



MEMORY



The purpose of memory block is to read and write the values from the memory. Based on icode value, memory stage will be in read or write state.
Read state for instructions mrmovq, return, popq and write state for instructions rmmovq, call, pushq.
The memory stage in the sequential architecture of the Y86-64 processor is responsible for reading from and writing to memory. It retrieves the memory address from the previous stage and sends a request to the memory system to read or write data from or to that address. If the request is a read, the memory stage retrieves the data from memory and passes it on to the next stage. If the request is a write, the memory stage stores the data to memory at the specified address. Additionally, the memory stage also checks for memory-related errors, such as invalid memory accesses or page faults, and sets appropriate flags in the status register. The status register is then passed on to the next stage, where it can be used to make decisions or trigger interruptions.

DETAILS OF MEMORY BLOCK FOR EVERY INSTRUCTION:

WRITING IN MEMORY:

- We do memory_chunk[val_e] = val_a

(THIS IS FOR RMMOVQ AND PUSHQ INSTRUCTIONS)

- We do memory_chunk[val_e] = val_p

(FOR CALL INSTRUCTION)

READING FROM MEMORY:

- We do val_m = memory_chunk[val_e]

(FOR MRMOVQ AND POPQ)

- We do val_m = memory_chunk[val_e]

(FOR RETURN INSTRUCTION)

IMPLEMENTATION IN CODE:

We are writing or reading into the memory depending on the fetched instruction as discussed above for various instructions . In the previous memory implementation we had read_enable and write_enable to activate read and write but here we are directly doing operations based on the instruction fetched.

Memory chunk is already assigned and is dynamic to change before execution, at clock=1(high) we read/write from the memory

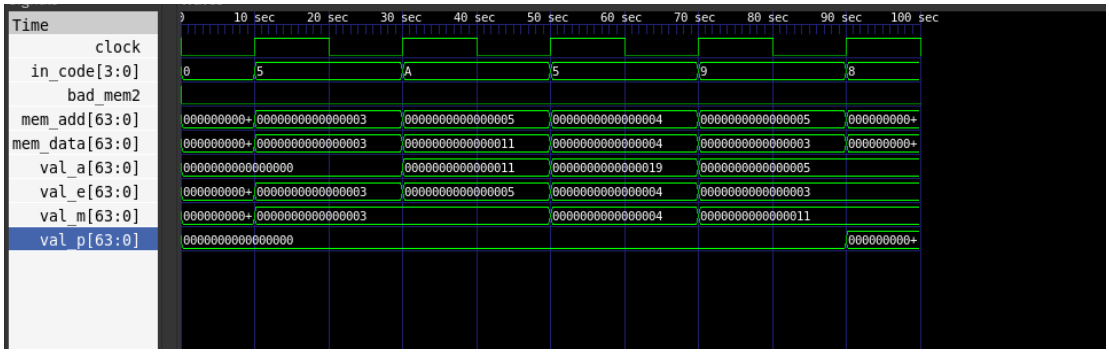
reg [63:0] mem_chunk[0:1023];

Memory is defined as 64 bit with 1023 values it can take. If memory address becomes out of bounds above 1023 we flag bad_mem2 error which becomes 1 that error has been flagged and memory has gone out of bounds.

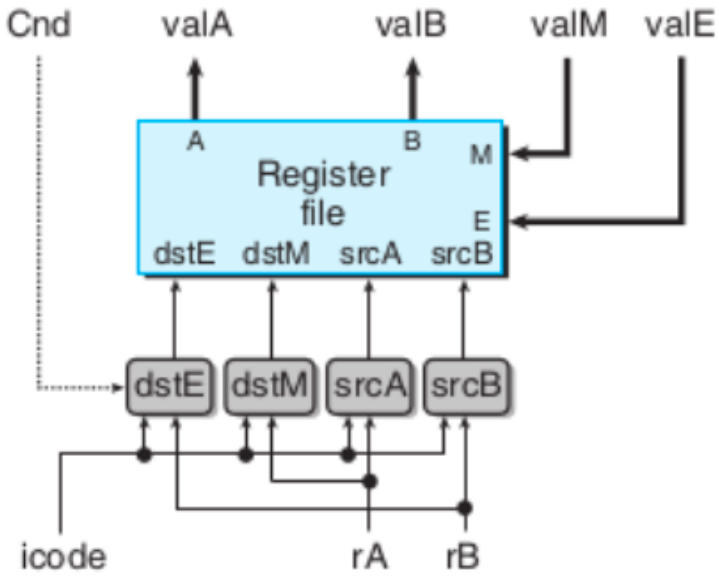
```
1   module mem(
2   input [3:0] in_code,
3   input [63:0] val_e,
4   input [63:0] val_p,
5   input [63:0] val_a,
6   input clock,
7   output reg bad_mem2,
8   output reg [63:0] mem_add,
9   output reg [63:0] val_m,
10  output reg [63:0] mem_data
11  );
```

OUTPUT:

```
pingu@ubuntu ~/p/n/f/p/S/memory (main)> vvp out.vvp
clock=0,Time=0, in_code= 0, val_e=0, val_p=0, val_a=0, bad_mem2=0, mem_add=0, val_m=0, mem_data=0
clock=1,Time=10, in_code= 5, val_e=3, val_p=0, val_a=0, bad_mem2=0, mem_add=3, val_m=3, mem_data=3
clock=0,Time=20, in_code= 5, val_e=3, val_p=0, val_a=0, bad_mem2=0, mem_add=3, val_m=3, mem_data=3
clock=1,Time=30, in_code=10, val_e=5, val_p=0, val_a=17, bad_mem2=0, mem_add=5, val_m=3, mem_data=17
clock=0,Time=40, in_code=10, val_e=5, val_p=0, val_a=17, bad_mem2=0, mem_add=5, val_m=3, mem_data=17
clock=1,Time=50, in_code= 5, val_e=4, val_p=0, val_a=25, bad_mem2=0, mem_add=4, val_m=4, mem_data=4
clock=0,Time=60, in_code= 5, val_e=4, val_p=0, val_a=25, bad_mem2=0, mem_add=4, val_m=4, mem_data=4
clock=1,Time=70, in_code= 9, val_e=3, val_p=0, val_a=5, bad_mem2=0, mem_add=5, val_m=17, mem_data=3
clock=0,Time=80, in_code= 9, val_e=3, val_p=0, val_a=5, bad_mem2=0, mem_add=5, val_m=17, mem_data=3
clock=1,Time=90, in_code= 8, val_e=3, val_p=999, val_a=5, bad_mem2=0, mem_add=3, val_m=17, mem_data=999
clock=0,Time=100, in_code= 8, val_e=3, val_p=999, val_a=5, bad_mem2=0, mem_add=3, val_m=17, mem_data=999
```



WRITE BACK



This stage updates register file corresponding to the in_code and in_fun of the current instruction, the values of val_e, val_m are passed to the register file, the val_e and val_m are the final values of the updated register. Between Val_e, Val_m the register is updated according to the in_code.

In the write back stage, values are written back into either of the main 15 registers, that is the values val_e, or val_m are written into the registers with addresses ra, rb or stack ptr register depending on the in_code.

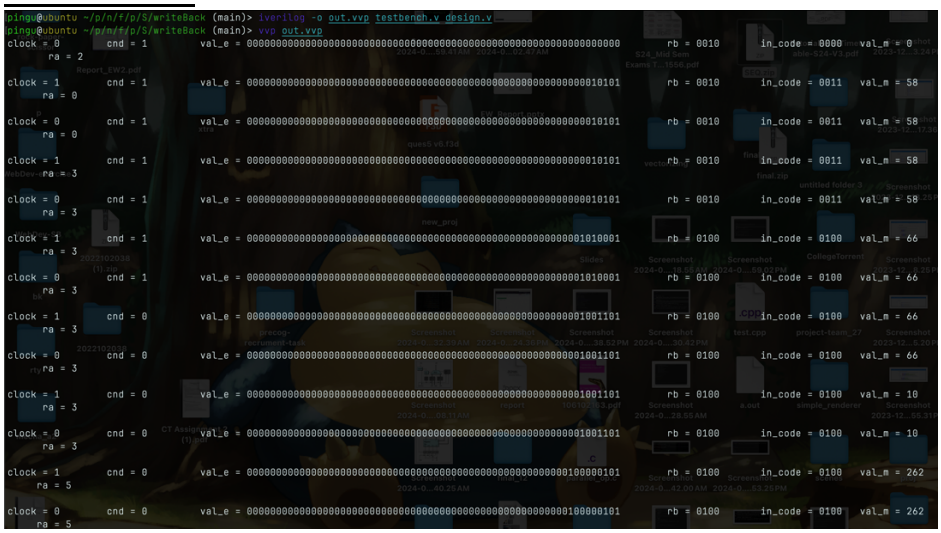
IMPLEMENTATION IN CODE:
We are reading and writing into a common.txt file maintaining value in all 15 registers of Y86 architecture.
Note:- We are reading at clock=1 and we are writing at clock=0.

WHY THIS APPROACH?
This helps us write two different blocks decode and write-back and changes made in both get reflected in the same common.txt file in the correct sequence and we can get the correct register values. Apart from this we can use 15 different variables as wires to store values or do decode and writeback in the same module at posedge and negedge respectively.
The values val_e or val_m are written into register with register code ra, rb or rsp
depending on the in_code as follows:
NOTE:REGISTER ARRAY IS reg_chunk
reg [63:0] reg_chunk [0:14];
o reg_chunk[ra] = val_m (value read from memory)
ABOVE IS FOR MRMOVQ AND POP INSTRUCTIONS
o reg_chunk[rb] = val_e
THIS IS FOR IRMOVQ FOR RRMOVQ WITH CND=1 AND FOR THE OPQ OPERATION FOR ADDITION,SUBTRACTION,AND AND XOR OPERATIONS
o reg_chunk[rsp] = val_e
ABOVE HAPPENS FOR INSTRUCTIONS CALL,RETURN,PUSH AND POP.

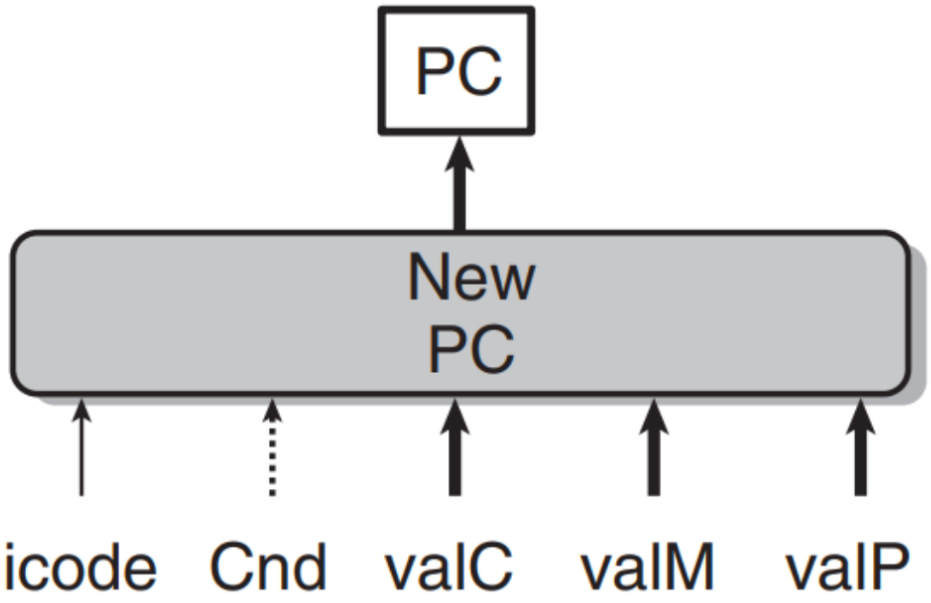
REGISTER FILE AFTER RUNNING

```
// 0x00000000
00000000000000000
00000000000000000
00000000000000000
00000000000000010
00000000000000000
00000000000000000
00000000000000000
00000000000000000
00000000000000000
00000000000000000
00000000000000000
00000000000000000
00000000000000000
00000000000000000
00000000000001000
```

OUTPUT:



PC-UPDATE



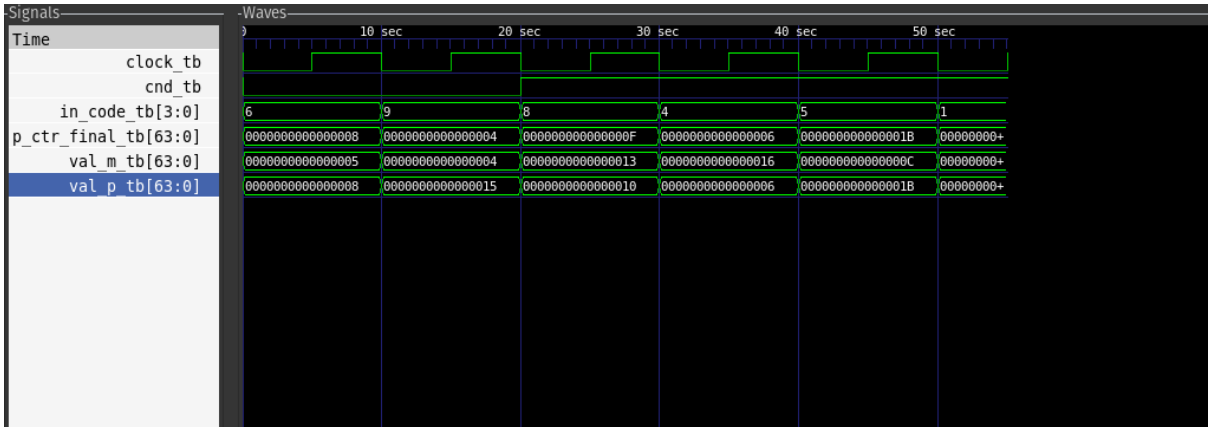
The main functionality of PC update is to point to the address of the next instruction accurately handling all cases. The PC Update block has in_code, cnd, val_c, val_m and val_p as inputs and the value of updated PC as final output , this helps us to fetch the next instruction.

PC is usually equal to val_p which is current program counter+length of the fetched instruction in that clock cycle.
But this doesn't hold for the following cases:-

- FOR CALL INSTRUCTION
  PC=p_ctr=val_c
- FOR RETURN INSTRUCTION
  PC=p_ctr=val_m
- FOR JXX(CONDITIONAL AND UNCONDITIONAL JUMPS)
  PC = p_ctr = val_c if cnd=1 jump to be taken Otherwise PC remains val_p.

```
module writeBack(
    input clock, cnd,
    input [3:0] in_code, ra, rb,

    input [63:0] val_e, val_m
);
```

OUTPUT




```
pingu@ubuntu ~/p/n/f/p/S/pc-update (main)> iverilog -o out.vvp testbench.v design.v
pingu@ubuntu ~/p/n/f/p/S/pc-update (main)> vvp out.vvp
clock=0, Time=0, in_code=6, val_m=5, val_c=3, val_p=8, cnd=0, p_ctr_final=8
clock=1, Time=5, in_code=6, val_m=5, val_c=3, val_p=8, cnd=0, p_ctr_final=8
clock=0, Time=10, in_code=9, val_m=4, val_c=91, val_p=21, cnd=0, p_ctr_final=4
clock=1, Time=15, in_code=9, val_m=4, val_c=91, val_p=21, cnd=0, p_ctr_final=4
clock=0, Time=20, in_code=8, val_m=19, val_c=15, val_p=16, cnd=1, p_ctr_final=15
clock=1, Time=25, in_code=8, val_m=19, val_c=15, val_p=16, cnd=1, p_ctr_final=15
clock=0, Time=30, in_code=4, val_m=22, val_c=6, val_p=6, cnd=1, p_ctr_final=6
clock=1, Time=35, in_code=4, val_m=22, val_c=6, val_p=6, cnd=1, p_ctr_final=6
clock=0, Time=40, in_code=5, val_m=12, val_c=23, val_p=27, cnd=1, p_ctr_final=27
clock=1, Time=45, in_code=5, val_m=12, val_c=23, val_p=27, cnd=1, p_ctr_final=27
clock=0, Time=50, in_code=1, val_m=27, val_c=29, val_p=31, cnd=1, p_ctr_final=31
clock=1, Time=55, in_code=1, val_m=27, val_c=29, val_p=31, cnd=1, p_ctr_final=31
** VVP Stop(0) **
```

COMBINED PROCESSOR

CODE:

processor.v M × | i.txt | common.txt

final_upload > project-team_27 > SEQ > combined > processor.v

```
1  `timescale 1ns/1ps
2
3
4  `include "decode.v"
5  `include "execute.v"
6  `include "fetch.v"
7  `include "memory.v"
8  `include "pc_update.v"
9  `include "writeback.v"
10
11
12 module combined();
13
14     reg [63:0] p_ctr;
15     reg clock;
16     reg [3:0] en_coder;
17     wire [3:0] in_code;
18     wire signed [63:0] val_b;
19     wire [63:0] val_m;
20     wire [3:0] in_fun;
21     wire signed [63:0] val_c;
22     wire flag_halt;
23     wire in_error;
24     wire [63:0] val_p;
25     wire [3:0] ra;
26     wire [3:0] rb;
27     wire signed [63:0] val_a;
28     wire zroFlag, N, V, cnd;
29     wire signed [63:0] val_e, mem_data, mem_add;
30     wire [63:0] p_ctr_final;
31     wire bad_mem,bad_mem2;
```

```
33      fetch_seq dut_1 (
34          .p_ctr(p_ctr),
35          .clock(clock),
36          .in_code(in_code),
37          .in_fun(in_fun),
38          .rb(rb),
39          .ra(ra),
40          .flag_halt(flag_halt),
41          .val_c(val_c),
42          .val_p(val_p),
43          .bad_mem(bad_mem),
44          .in_error(in_error)
45      );
46      decode_proc dut_2 (
47          .in_fun(in_fun),
48          .val_m(val_m),
49          .in_code(in_code),
50          .val_e(val_e),
51          .clock(clock),
52          .rb(rb),
53          .ra(ra),
54          .val_b(val_b),
55          .val_a(val_a)
56      );
57      exe dut_3(
58          .in_code(in_code),
59          .in_fun(in_fun),
60          .val_a(val_a),
61          .val_b(val_b),
62          .val_c(val_c),
63          .clock(clock),
```

```
64
65          .val_e(val_e),
66          .cnd(cnd)
67      );
68      mem dut_4 (
69          .in_code(in_code),
70          .val_e(val_e),
71          .val_p(val_p),
72          .val_a(val_a),
73          .clock(clock),
74          .bad_mem2(bad_mem2),
75          .mem_add(mem_add),
76          .val_m(val_m),
77          .mem_data(mem_data)
78      );
79      writeBack dut_5(
80          .clock(clock),
81          .cnd(cnd),
82          .in_code(in_code),
83          .ra(ra),
84          .rb(rb),
85          .val_e(val_e),
86          .val_m(val_m)
87      );
88      pc_update dut_6 (
89          .in_code(in_code),
90          .val_p(val_p),
91          .clock(clock),
92          .val_c(val_c),
93          .val_m(val_m),
```

```
94          .cnd(cnd),
95          .p_ctr_final(p_ctr_final)
96      );
97      initial begin
98          $dumpfile("dump.vcd");
99          $dumpvars(0);
100     end
101     initial
102     begin
103         clock = 1;
104         p_ctr = 64'd0;
105        en_coder=4'd8;
106     end
107
108
109     always #5 clock =~ clock;
110
111     always@(*)
112     begin
113         if(in_error)
114         begin
115         en_coder=4'd4;
116         $finish;
117         end
118         if(bad_mem2== 1)
119         begin
120         en_coder=4'd1;
121         $finish;
122         end
123         if(flag_halt)
```

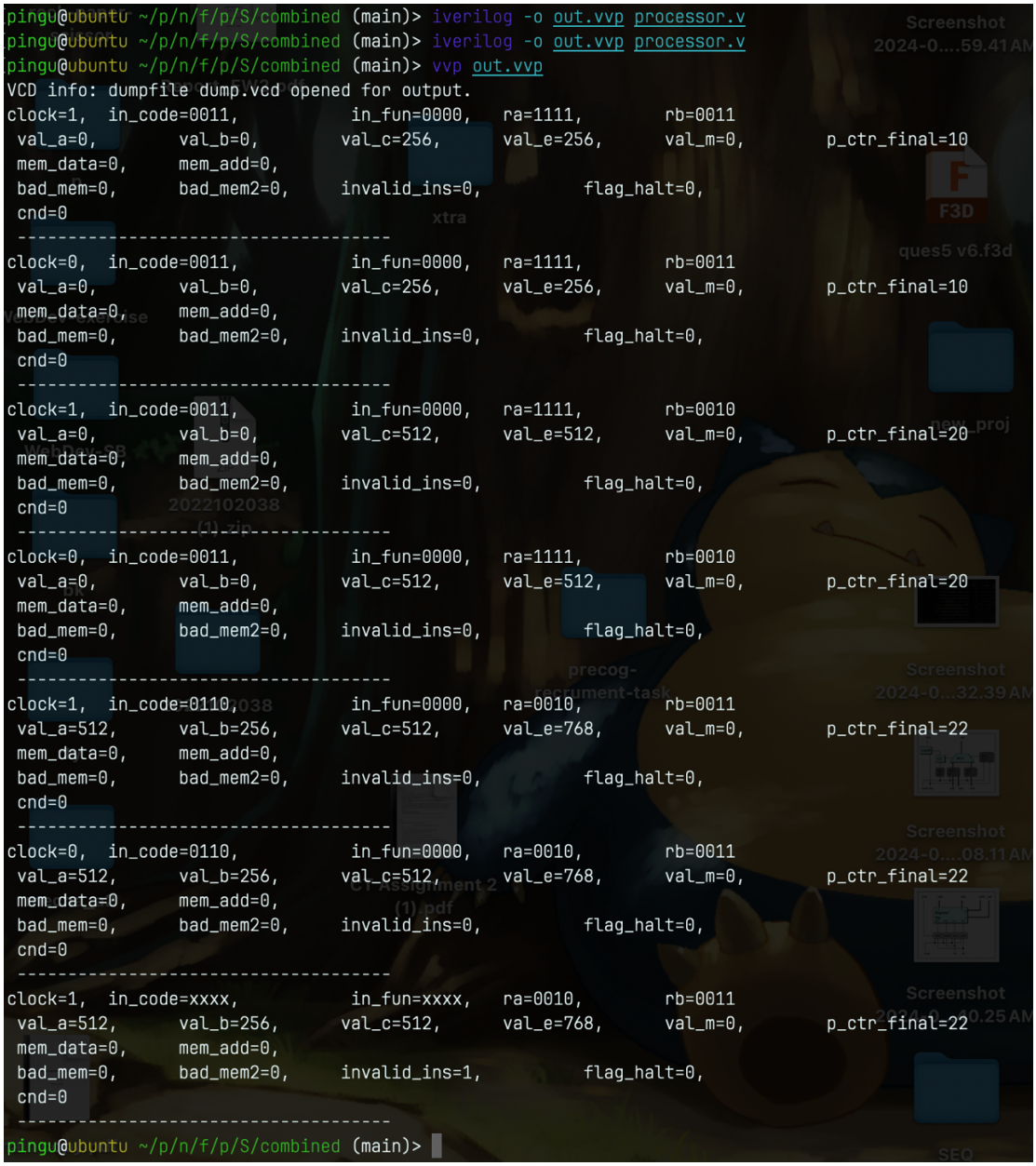
```
123         if(flag_halt)
124         begin
125         en_coder=4'b0010;
126         $finish;
127         end
128         if(bad_mem== 1)
129         begin
130         en_coder=4'b0001;
131         $finish;
132         end
133         else
134         begin
135        en_coder=4'b1000;
136         end
137     end
138 always@(*)
139 begin
140 p_ctr=p_ctr_final;
141 end
142
143    always@(*)
144     begin
145         if(en_coder==4'b0010)
146         begin
147             $finish;
148         end
149         if(en_coder==4'b0011) begin
150             $finish;
151         end
152
153         if(en_coder==4'b0100 || en_coder==4'b0101 || en_coder==4'b1101 || en_coder==4'b0110 || en_cod
154         begin
155             $finish;
156         end
157
158
159         if(en_coder==4'b1010) begin
160             $finish;
161         end
162
163     end
164     initial begin
165         $monitor("clock=%d,  in_code=%b, \t  in_fun=%b, \t ra=%b, \t rb=%b \n val_a=%g, \t val_b=%g,
166     end
167 endmodule
```

Register file before execution:

```
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

Register file after execution:

```
// 0x00000000
0000000000000000
0000000000000000
0000000000000200
0000000000000300
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```



Explanation:

From line 1 in_code = 3, in_fun=0 i.e is irmovq Since in rrmovq ra is always F and over here rb = 3 and V = 256. So reg_chunk[rb]<-256.

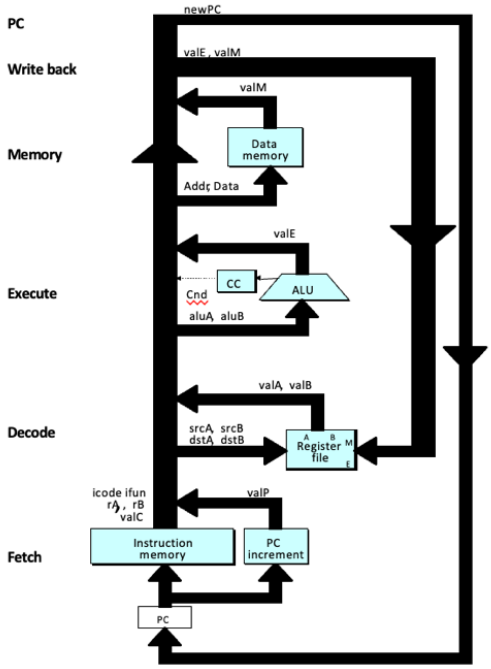
Now PC will jump to line 11, in_code=3, rb = 2 and V = 512. So reg_chunk[rb]<-512.

Now PC will jump to line 21 and since in_code=6, in_fun=0 it'll perform addition on values reg_chunk[rb] and reg_chunk[ra]. And store their result in reg_chunk[rb].

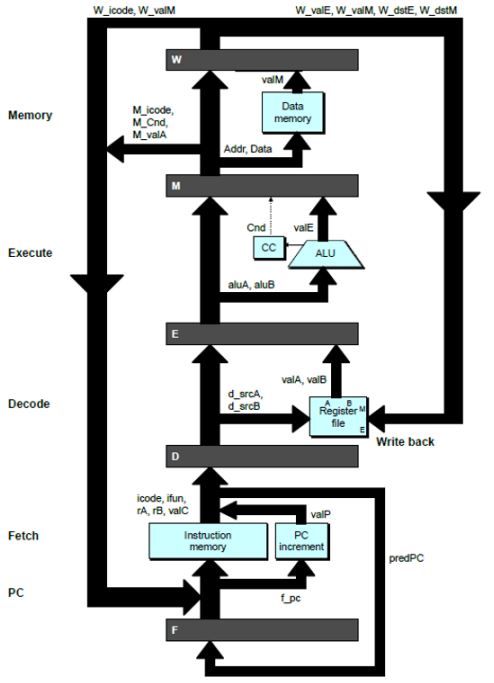
```
00110000
11110011
00000000
00000001
00000000
00000000
00000000
00000000
00000000
00000000
00110000
11110010
00000000
00000010
00000000
00000000
00000000
00000000
00000000
00000000
01100000
00100011
```

PIPELINE ARCHITECTURE

Implementation of the pipeline involved the addition of the registers between each e that can pass designated outputs from one stage register as inputs to the next stage register. Another change is in the PC update. We do not have a separate PC update block, instead, we integrated PC update with the Fetch so that before every instruction executes, the Fetch stage will determine the current PC value and also the next predicted PC. In addition to these, there is a control logic too involved for Bubble, Stall and Forwarding implementation to eliminate pipeline hazards.



Seq Arch



Pipelined Arch

DIFFERENCE B/w SEQ and PIPELINED ARCH

- SEQUENTIAL:
  - In a sequential Y86 design, instructions are executed one after the other without any overlap.
  - Each stage of instruction execution, such as instruction fetch, decode, execute, memory access, and write back, is completed for one instruction before moving on to the next instruction.
  - This approach is straightforward and easy to understand, making it suitable for educational purposes and introductory courses on computer architecture.
- PIPELINED ARCH:

- In a pipelined Y86 design, the instruction execution process is divided into stages, and multiple stages can be active simultaneously for different instructions.
- As one instruction progresses through the pipeline stages, the next instruction can enter the pipeline, leading to potential overlap in the execution of multiple instructions.
- Pipelining aims to improve overall instruction throughput by allowing different stages to work concurrently on different instructions.
- However, it introduces challenges such as handling hazards (e.g., data hazards, control hazards) that may arise due to the overlap of instructions.

- Main difference between SEQ and PIPELINED Arch is that 'Pipeline has much higher throughput' and insertion of pipeline registers F,D,E,M,W and basic concepts of 'Data Forwarding'.

F pipeline register Holds a predicted value of the program counter, as will be discussed shortly.

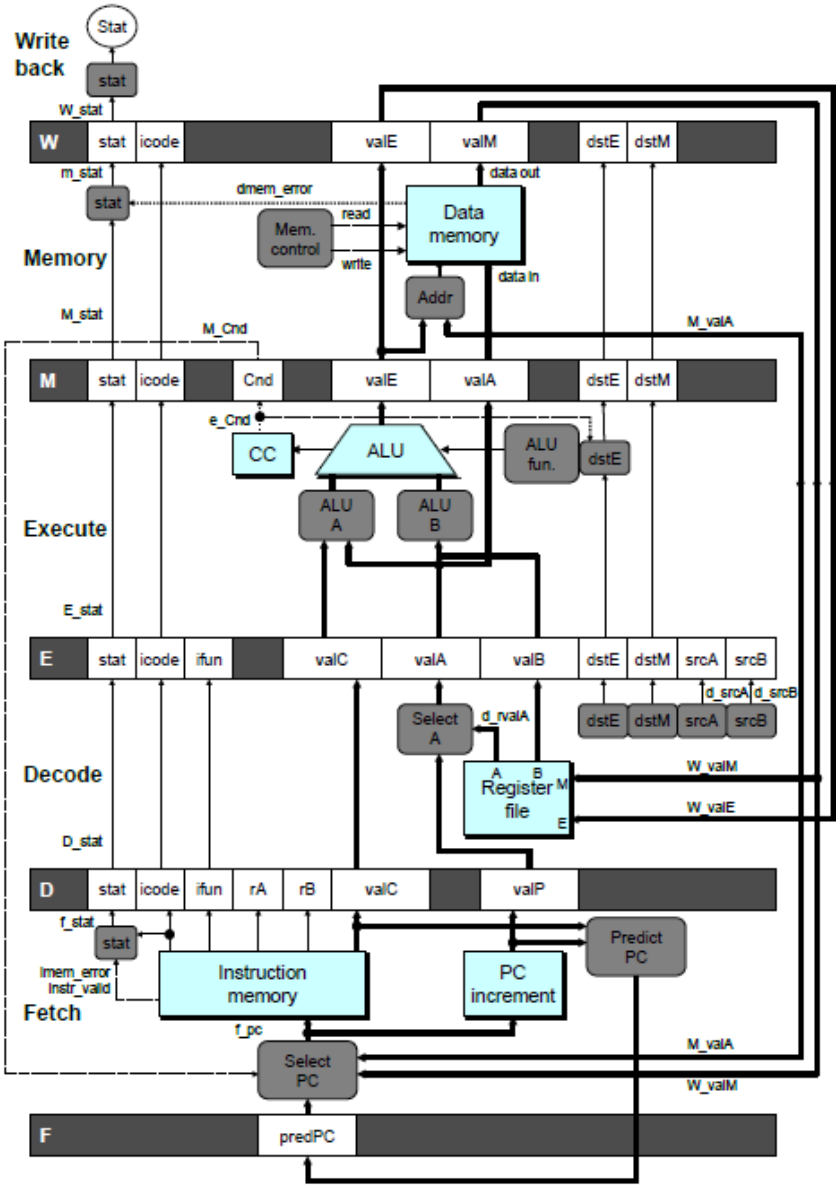
D pipeline register Sits between the fetch and decode stages. It holds information about the most recently fetched instruction for processing by the decode stage.

E pipeline register Sits between the decode and execute stages. It holds information about the most recently decoded instruction and the values read from the register file for processing by the execute stage.

M pipeline register Sits between the execute and memory stages. It holds the results of the most recently executed instruction for processing by the memory stage. It also holds information about

branch conditions and branch targets for processing conditional jumps.

W pipeline register Sits between the memory stage and the feedback paths that supply the computed results to the register file for writing and the return address to the PC selection logic when completing a ret instruction.



HOW TO AVOID DATA HAZARDS

Avoiding Data Hazards by Stalling One very general technique for avoiding hazards involves stalling, where the processor holds back one or more instructions in the pipeline until the hazard condition no longer holds. Stalling involves holding back one group of instructions in their stages while allowing other instructions to continue flowing through the pipeline. The stages that should normally be processing would be injected with a bubble. A bubble is like a dynamically generated nop instruction—it does not cause any changes to the registers, the memory, the condition codes, or the program status.

Avoiding Data Hazards by Forwarding The technique of passing a result value directly from one pipeline stage to an earlier one is commonly known as data forwarding. Data forwarding requires adding additional data connections and control logic to the basic hardware structure. o It can also use the ALU output (signal e_vale) for operand val_a or val_b. o It can use the value that has just been read from the data memory (signal m_val_m) for operand val_a or val_b. o It can use the value in the memory stage (signal M_val_e) for operand val_a or val_b. o It can use the value in the write-back stage (signal W_val_e or signal W_val_m) for operand val_a or val_b.

Avoiding Control Hazards: Control hazards arise when the processor cannot reliably determine the address of the next instruction based on the current instruction in the fetch stage. control hazards can only occur in our pipelined processor for ret and jump instructions. In case of ret, the processor is stalled for 3 clock cycles after the ret instruction. While in case of jump misprediction, the pipeline can simply cancel the two misfetched instructions by injecting bubbles into the decode and execute stages on the following cycle while also fetching the instruction following the jump instruction.

Avoiding Load/Use Data Hazards: One class of data hazards cannot be handled purely by forwarding, because memory reads occur late in the pipeline. These are called load/use hazards and they occur when one instruction reads a value from memory for register while the next instruction needs this value as a source operand. We can avoid a

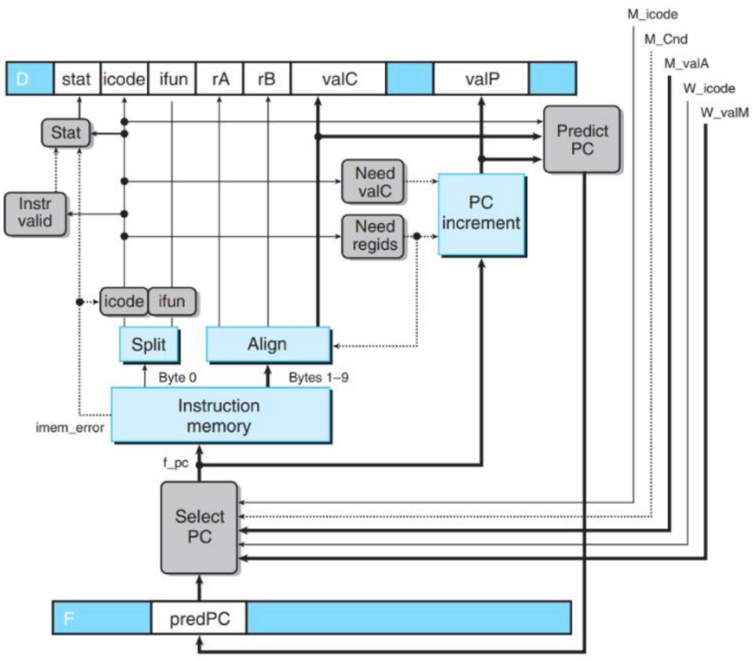
load/use data hazard with a combination of stalling and forwarding. This requires modifications of the control logic. Avoiding Control Hazards: Control hazards arise when the processor cannot reliably determine the address of the next instruction based on the current instruction in the fetch stage. control hazards can only occur in our pipelined processor for ret and jump instructions. In case of ret, the processor is stalled for 3 clock cycles after the ret instruction. While in case of jump misprediction, the pipeline can simply cancel the two miss-fetched instructions by injecting bubbles into the decode and execute stages on the following cycle while also fetching the instruction following the jump instruction.

THE PIPELINE ARCHITECTURE:-

1) FETCH (WITH PC SELECTION)
2) DECODE
3) EXECUTE
4) MEMORY
5) WRITE-BACK

FETCH BLOCK ALONG WITH PC SELECTION

HARDWARE IMPLEMENTATION



This happens at the positive edge of the clock.

This stage selects a current value for the program counter and predicts the next PC value from the previous clock cycle. The hardware units for reading the instruction from memory etc are same as SEQ architecture. If bubble (D_bub=1) then nop instruction is executed.
THE BASIC LOGIC OF PROGRAM REMAINS SAME AS THE SEQUENTIAL ARCHITECTURE The PC selection logic chooses between three program counter sources at an instant.

- f_prediction_pc = W_val_m (again read from memory) for the return instruction
- f_prediction_pc = M_val_a(value read from memory) if in code in memory is 7 and cnd is not set
- f_prediction_pc = F_prediction_pc for rest cases.
  The PC prediction logic is as follows:
- F_prediction_pc = f_val_c
  ABOVE IS FOR JXX IN_CODE=7 AND FOR CALL(8)
  F_prediction_pc = f_val_p for rest of the cases.

  Logic for updating pc in always(*) block:

```
begin
    holder=F_prediction_pc;
    p_ctr=holder;
end
```

PC prediction Logic

Instructions that Don't Transfer Control
- Predict next PC to be valP
- Always reliable

Call and Unconditional Jumps
- Predict next PC to be valC (destination)
- Always reliable

Conditional Jumps
- Predict next PC to be valC (destination)
- Only correct if branch is taken
  - Typically right 60% of time

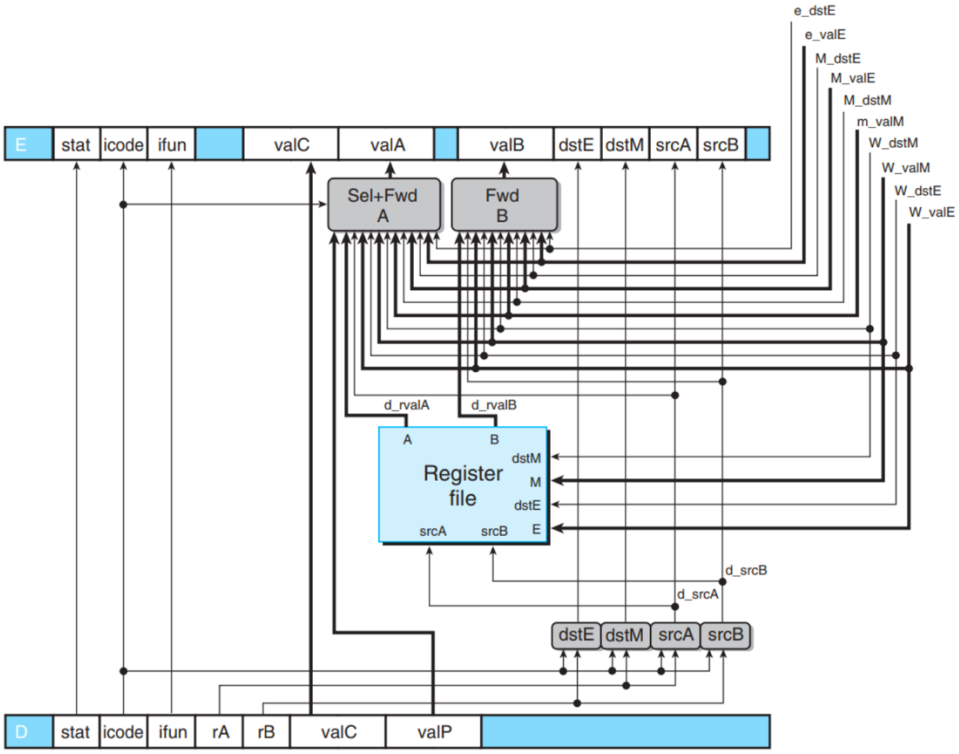
Return Instruction
- Don't try to predict

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;
];
```

DECODE BLOCK:

HARDWARE IMPLEMENTATION



The code and implementation of this stage is like that of the decode stage in the sequential processor. Most of the complexity of this stage is associated with the forwarding logic to avoid hazards discussed above. Here also decode and writeback is done on the same file to ensure accurate results since we cannot transmit the array from one block to another.

At higher(1)/positive edge of the clock all the values are read from the common.txt file and val_a and val_b are computed accordingly.

Note : AT end of the code E register is filled with all required values by values from d happening in decode.

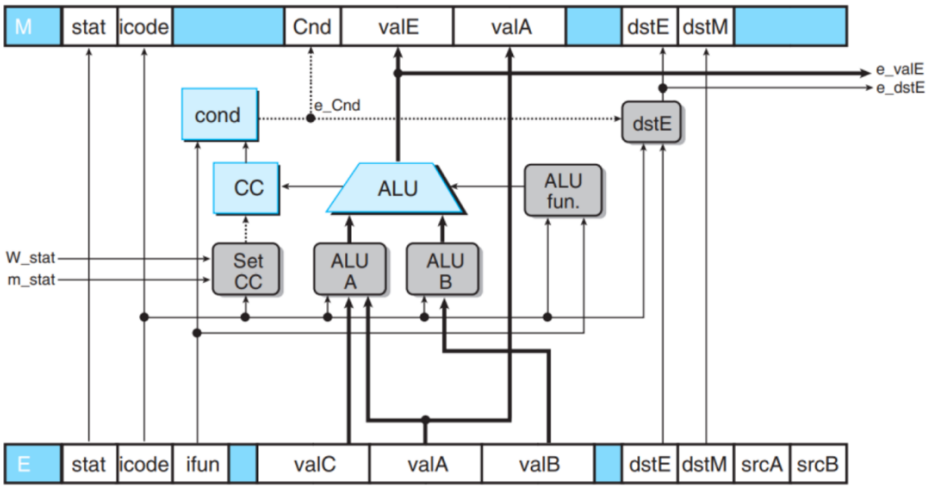
IF E_bub is set then nop instruction is set doing nothing.

DATA FORWARDING LOGIC

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

THIS IS DONE SO THAT WE DON'T HAVE TO WAIT FOR WRITEBACK STAGE/MEMORY/EXECUTE STAGES OF THE PREVIOUS INSTRUCTIONS TO ENSURE PROPER EXECUTION OF THE CURRENT AND NEXT INSTRUCTIONS WITHOUT ANY BUG/ERROR.

EXECUTE BLOCK:



The implementation of the execute block of the pipeline architecture is exactly the same implementation as in the sequential architecture that is val_e value is calculated for storing particular value/address to perform the particular operation and conditional flags are set for ALU operations

ensuring proper execution of future conditional jump instructions if any also flags are set which help in determining whether to jump or not.

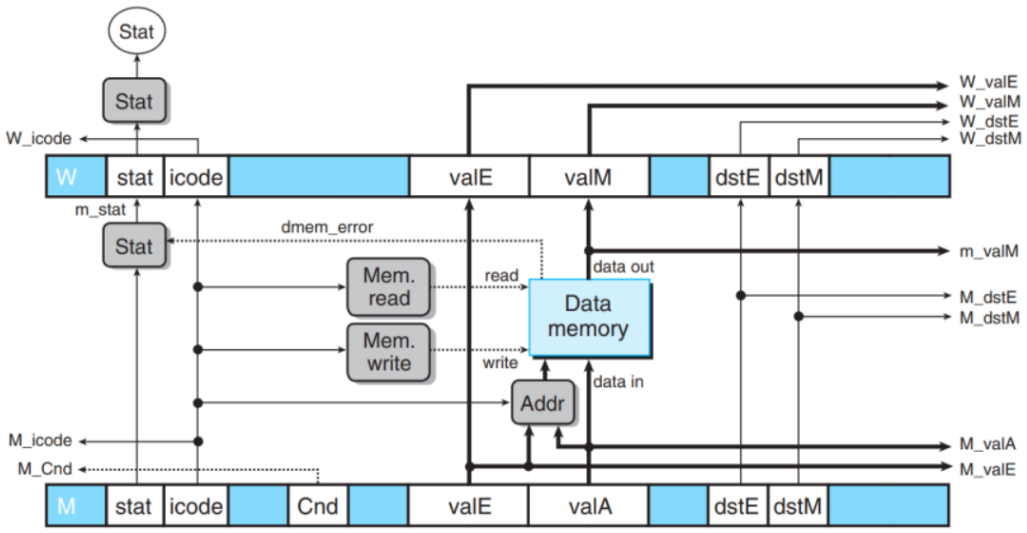
MAIN DIFFERENCE BETWEEN SEQUENTIAL AND PIPELINING ARCHITECTURE IS THAT:

The values of e_val_e and e_cnd are set during this execution block execution of the pipeline.

NOTE: All e_values (computed during execute stage) along with values in E pipeline register are fed into the M pipeline register at high(1)/positive edge of the clock together in a single trigger.

```
134     always@(posedge clock) begin
135         M_stat <= E_stat;
136         M_in_code <= E_in_code;
137         M_cnd <= e_cnd;
138         M_val_e <= e_val_e;
139         M_val_a <= E_val_a;
140         M_dst_e <= e_dst_e;
141         M_dst_m <= E_dst_m;
142     end
```

MEMORY BLOCK:



THE IMPLEMENTATION OF THIS BLOCK TOO IS VERY SIMILAR TO THE MEMORY BLOCK IMPLEMENTED IN THE SEQUENTIAL ARCHITECTURE OF THE Y86 PROCESSOR.

WE READ / WRITE FROM THE MEMORY DEPENDING ON THE INSTRUCTION FETCHED.
THE IMPLEMENTATION IS EXACTLY SAME AS DISCUSSED IN THE MEMORY BLOCK OF THE REPORT IN SEQUENTIAL PART .

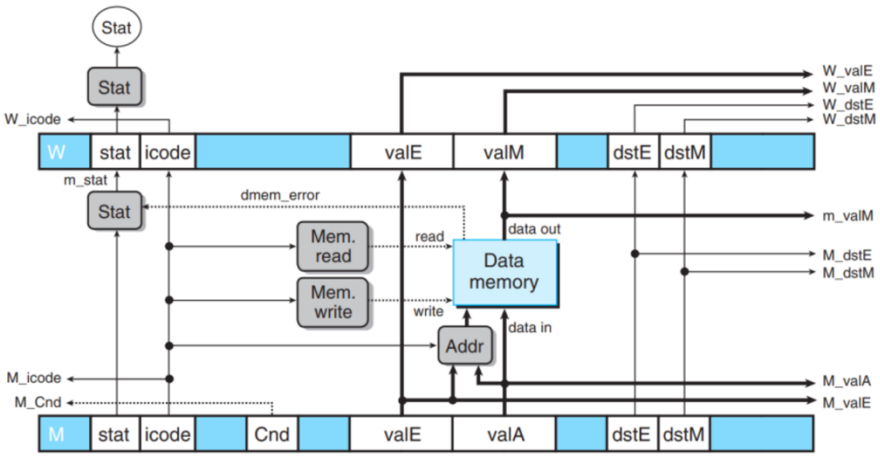
MAIN DIFFERENCE BETWEEN SEQUENTIAL AND PIPELINE:

All values m_values (computed during memory stage) and M_values(from the memory pipeline register computed in the previous clock cycle are fed into the W write-back pipeline register at positive edge of the clock together in single trigger.

CODE FOR ABOVE STORING IN W PIPELINE REGISTER:

```
108   //changes at positive trigger
109   always@(posedge clock) begin
110       W_in_code<=M_in_code;
111       W_dst_m<=M_dst_m;
112         W_val_e<=M_val_e;
113       W_dst_e<=M_dst_e;
114       W_stat<=m_stat;
115       W_val_m<=m_val_m;
116   end
117   endmodule
```

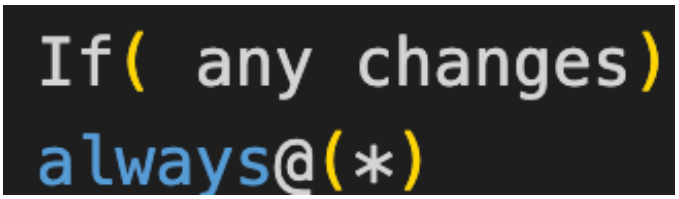
WRITE BACK BLOCK:



The implementation of this block too is exactly same as the SEQ architecture implementation. We make changes write back or make changes in the register values(/read) in the common.txt file here as well as in the decode stage.

DIFFERENCES B/w SEQ AND PIPELINE ARCH:

- Instead of performing writing or reading from the register memory depending on the clock we do so whenever we sense some changes in register values do to instructions.



- Basically we write into register memory that is make changes whenever changes are brought by fetching instructions and we identify instructions by W_in_code which is stored in the W pipeline register.

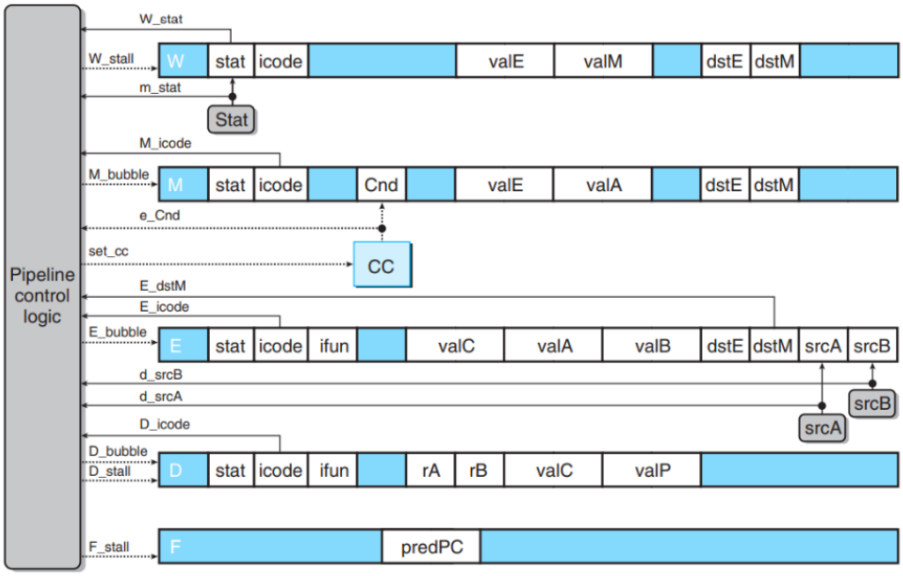
PIPE CONTROL:

For controlling all kinds of data/ control hazards (bubble, stall, data forwarding).

- The return instruction can be detected using the condition given below, { D_icode, E_icode, M_icode} any of these must be 9.

- Branch misprediction can be detected using the condition given below, o E_icode = 7(conditional jump) & !e_cnd (if jump was not to be taken but we take with 60 percent probability)

- Load/use hazard can be detected using the condition given below, o E_icode equal to 5 or eleven that is for mrmovq and popq instructions && E_dstM is in either d_src_b or d_src_a.



STALLING/BUBBLING ENSURES THAT THE EARLIER STAGES OF THE RECENT INSTRUCTIONS DON'T READ/WRITE WRONG VALUES SO THEY STALL/BUBBLE FOR THE CRUCIAL STAGES OF THE OLDER INSTRUCTIONS TO GET OVER.
STALL/BUBBLE/NORMAL FOR ALL 5 STAGES IMPLEMENTED IN PIPE CONTROL FOR ALL 3 HAZARDS TO COMBAT THEM.

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

CODE FOR STALLING/BUBBLING VARIOUS BLOCKS TO ENSURE PROPER EXECUTION OF ALL BLOCKS IN PIPELINE ARCHITECTURE.

(ALL 3 HAZARDS WITH CONDITIONS ALREADY MENTIONED IN THE STARTING)

```
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 };

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

bool D_bubble =
      # Mispredicted branch
      (E_icode == IJXX && !e_Cnd) ||
      # Stalling at fetch while ret passes through pipeline
       IRET in { D_icode, E_icode, M_icode };

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

CODE:

```
module p_control(
output reg F_st,
input e_cnd,
input [3:0] d_src_b,
input [3:0] M_in_code,
input [3:0] d_src_a,
output reg set_cc,
input [3:0] D_in_code,
output reg D_st,
input [1:0] W_stat,
output reg D_bub,
input [3:0] E_in_code,
input [3:0] E_dst_m,
output reg E_bub,
input [1:0] m_stat
);
reg pip_control_start;
reg pip_control_end;
initial begin
    pip_control_end=1'b0;
end
initial begin
    pip_control_start=1'b1;
end
always@(*)
begin
    set_cc=1;
    F_st=0;
    D_st=0;
    E_bub=0;
    D_bub=0;
        if(E_in_code==4'd5 & ((E_dst_m==d_src_b)|(E_dst_m==d_src_a)))
  begin
    pip_control_end=1;
    E_bub=1;
    D_st=1;
    F_st=1;
  end
```

```
    else if(E_in_code==4'd7)
    begin
        if(e_cnd==1)
        begin
            // no need
        end
        if(e_cnd==0)
        begin
          E_bub=1;
          D_bub=1;
        end
        pip_control_end=1;
    end

  else if(E_in_code==4'd11 & ((E_dst_m==d_src_b)|(E_dst_m==d_src_a)))
  begin
    pip_control_end=1;
    E_bub=1;
    D_st=1;
    F_st=1;
  end

  else if(E_in_code==4'd9 | D_in_code==4'd9 | M_in_code==4'd9)
  begin
    D_bub=1;
    F_st=1;
    pip_control_end=1;
  end

else if (m_stat!=2'd0 | W_stat!=2'd0 |E_in_code==0)
begin
    pip_control_end=1;
    set_cc=1'b0;
end
end
endmodule
```

PIPELINE PROCESSOR:

CODE:

```v
`include "fetch.v"
`include "decode.v"
`include "exe1.v"
`include "memory.v"
`include "writeBack.v"
`include "pipeCtrl.v"

module pipe();
    reg clock;
    reg [63:0] F_prediction_pc;

    reg [1:0] stat;

    wire [63:0] f_prediction_pc;

    wire [3:0] d_src_a, d_src_b;

    wire [3:0] D_in_code;
    wire [3:0] D_in_fun;
    wire [3:0] D_ra;
    wire [3:0] D_rb;
    wire [63:0] D_val_c;
    wire [63:0] D_val_p;
    wire [1:0] D_stat;
    wire [63:0] d_val_a, d_val_b;

    wire [3:0] M_in_code;
    wire [63:0] M_val_a;
    wire [63:0] M_val_e;
    wire [3:0] M_dst_m;
    wire [3:0] M_dst_e;
    wire M_cnd;
    wire[1:0] M_stat;

    wire [3:0] E_in_code;
    wire [3:0] E_in_fun;
    wire [3:0] E_src_a;
    wire [3:0] E_src_b;
    wire [63:0] E_val_a;
    wire [63:0] E_val_b;
    wire [63:0] E_val_c;
    wire [3:0] E_dst_e;
    wire [3:0] E_dst_m;
    wire [1:0] E_stat;

    wire [3:0] e_dst_e;
    wire [63:0] e_val_e;
    wire e_cnd;

    wire [3:0] W_in_code;
    wire [3:0] W_dst_e;
    wire [3:0] W_dst_m;
    wire [63:0] W_val_e;
    wire [63:0] W_val_m;
    wire [1:0] W_stat;

    wire [63:0] m_val_m;
    wire [1:0] m_stat;

    wire F_st, D_st, D_bub, E_bub;
    wire in_error;
    wire set_cc;

    fetch_pipe uut (
        .F_prediction_pc(F_prediction_pc),
        .D_ra(D_ra),
        .D_rb(D_rb),
        .M_val_a(M_val_a),
        .D_val_c(D_val_c),
        .M_cnd(M_cnd),
        .D_st(D_st),
        .D_val_p(D_val_p),
        .W_in_code(W_in_code),
        .D_in_code(D_in_code),
        .D_in_fun(D_in_fun),
        .W_val_m(W_val_m),
        .D_stat(D_stat),
        .F_st(F_st),
        .M_in_code(M_in_code),
        .clock(clock),
        .D_bub(D_bub),
        .f_prediction_pc(f_prediction_pc),
        .in_error(in_error)
    );
```

```
    exe dut2(
        .E_in_code(E_in_code),
        .E_in_fun(E_in_fun),
        .M_in_code(M_in_code),
        .M_dst_e(M_dst_e),
        .M_dst_m(M_dst_m),
        .e_dst_e(e_dst_e),
        .E_val_a(E_val_a),
        .E_val_b(E_val_b),
        .E_val_c(E_val_c),
        .M_val_e(M_val_e),
        .M_val_a(M_val_a),
        .e_val_e(e_val_e),
        .E_dst_e(E_dst_e),
        .E_dst_m(E_dst_m),
        .M_stat(M_stat),
        .E_stat(E_stat),
        .clock(clock),
        .set_cc(set_cc),
        .e_cnd(e_cnd),
        .M_cnd(M_cnd)
    );

    wb_pipe uut2(
        .W_dst_e(W_dst_e),
        .W_in_code(W_in_code),
        .W_dst_m(W_dst_m),
        .W_val_e(W_val_e),
        .W_val_m(W_val_m),
        .clock(clock)
    );

```

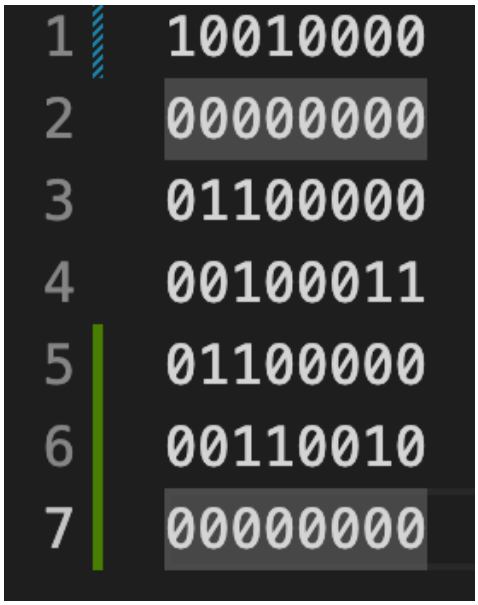
```
    memory_pipe uut4(
        .M_stat(M_stat),
        .W_dst_m(W_dst_m),
        .m_val_m(m_val_m),
        .M_dst_e(M_dst_e),
        .W_in_code(W_in_code),
        .W_stat(W_stat),
        .M_in_code(M_in_code),
        .W_val_m(W_val_m),
        .m_stat(m_stat),
        .M_dst_m(M_dst_m),
        .W_val_e(W_val_e),
        .M_cnd(M_cnd),
        .M_val_e(M_val_e),
        .M_val_a(M_val_a),
        .clock(clock),
        .W_dst_e(W_dst_e)
    );

    p_control uu3(
        .F_st(F_st),
        .e_cnd(e_cnd),
        .d_src_b(d_src_b),
        .M_in_code(M_in_code),
        .d_src_a(d_src_a),
        .set_cc(set_cc),
        .D_in_code(D_in_code),
        .D_st(D_st),
        .W_stat(W_stat),
        .D_bub(D_bub),
        .E_in_code(E_in_code),
        .E_dst_m(E_dst_m),
        .E_bub(E_bub),
        .m_stat(m_stat)
    );
```

```
    decode_proc dut(
        .D_in_fun(D_in_fun),
        .D_val_c(D_val_c),
        .D_in_code(D_in_code),
        .D_val_p(D_val_p),
        .W_dst_m(W_dst_m),
        .M_dst_e(M_dst_e),
        .M_dst_m(M_dst_m),
        .clock(clock),
        .E_bub(E_bub),
        .D_rb(D_rb),
        .D_ra(D_ra),
        .e_val_e(e_val_e),
        .M_val_e(M_val_e),
        .W_val_e(W_val_e),
        .e_dst_e(e_dst_e),
        .W_dst_e(W_dst_e),
        .D_stat(D_stat),
        .m_val_m(m_val_m),
        .W_val_m(W_val_m),
        .E_val_b(E_val_b),
        .E_val_a(E_val_a),
        .d_val_a(d_val_a),
        .E_val_c(E_val_c),
        .E_src_b(E_src_b),
        .E_in_code(E_in_code),
        .d_src_b(d_src_b),
        .E_dst_m(E_dst_m),
        .E_src_a(E_src_a),
        .E_in_fun(E_in_fun),
        .E_dst_e(E_dst_e),
        .E_stat(E_stat),
        .d_val_b(d_val_b),
        .d_src_a(d_src_a)
    );

    always@(W_stat)
    begin
        stat = W_stat;
    end

```

```
    always@(stat)
    begin
        if(stat == 2'b10) begin
            $finish;
        end

        else if (stat == 2'b01) begin
            $finish;
        end

        else if (stat == 2'b11) begin
            $finish;
        end
    end

    always @(posedge clock)
    begin
        if (F_st == 0)
        begin
        F_prediction_pc <= f_prediction_pc;
        end
    end

    always #10 clock = ~clock;

    initial begin
        clock = 0;
        F_prediction_pc = 64'd0;
    end

    initial begin
        $dumpfile("dump.vcd");
        $dumpvars(0);
    end

    initial begin
        $monitor($time, "\n\nclock=%d\n Fetch_Reg:\t F_predicted_PC = %g\n\t\tFETCH:\t f_predicted_PC = %g\n\t\tD_Reg:\t D_in_code = %b D_in_fun = %b D_ra = %b D_rb = %b D_val_c = %g D_val_p
        = %g D_stat = %g\n\t\tDECODE:\t\td_val_a = %g d_val_b = %g d_src_b = %g d_src_a = %g\n\t\tE_Reg:\t\tE_icode = %b E_in_fun = %b E_val_a = %g E_val_b = %g E_val_c = %g E_dst_e = %b
        E_dst_m = %b E_src_a = %g E_src_b = %g E_stat = %g\n\t\tEXECUTE:\te_cnd = %b e_val_e = %g e_dst_e = %g\n\t\tMem_Reg:\t\tM_icode = %b M_cnd = %b M_val_a = %g M_val_e = %g M_dst_e = %b,
        M_dst_m = %b M_stat = %g\n\t\tMEMORY:\t\tm_valM = %g\n\t\tWb_Reg:\t\tWb_icode = %b Wb_val_e = %g Wb_val_m = %g Wb_dst_e = %b Wb_dst_m = %b Wb_stat = %g\n\t\tF_stall = %g D_st = %g
        D_bub = %g E_bub = %g set_cc = %g\n",clock,F_prediction_pc,f_prediction_pc,D_in_code,D_in_fun,D_ra,D_rb,D_val_c,D_val_p,D_stat,d_val_a,d_val_b,d_src_b,d_src_a,E_in_code,E_in_fun,
        E_val_a,E_val_b,E_val_c,E_dst_e,E_dst_m,E_src_a,E_src_b,E_stat,e_cnd,e_val_e,e_dst_e,M_in_code,M_cnd,M_val_a,M_val_e,M_dst_e,M_dst_m,M_stat,m_val_m,W_in_code,W_val_e,W_val_m,W_dst_e,
        W_dst_m,W_stat,F_st,D_st,D_bub,E_bub,set_cc);
    end

endmodule
```

TEST CASE1:

Instruction Code:

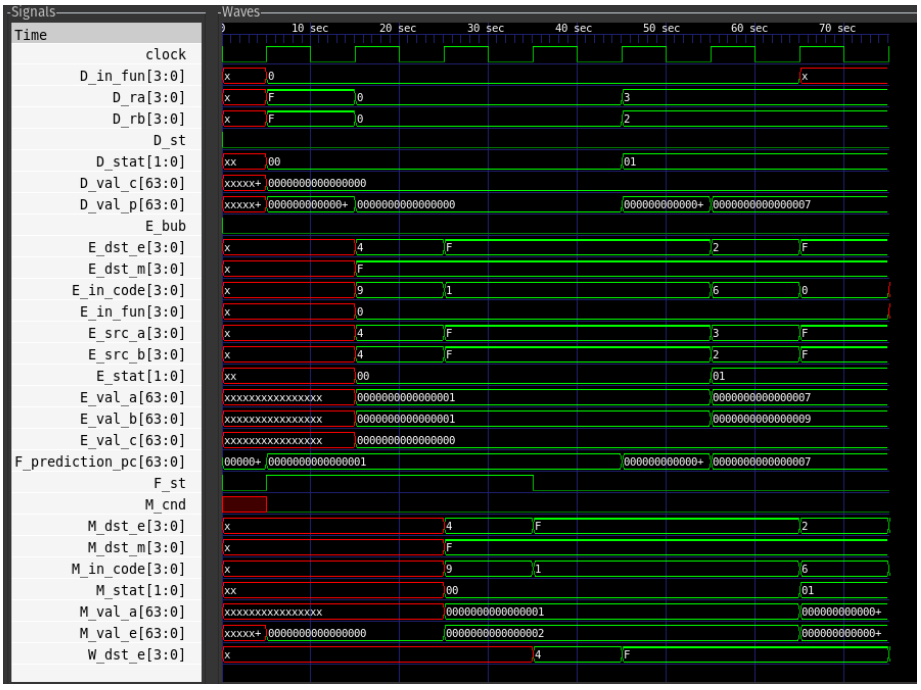


```
0000000000000000
0000000000000000
0000000000000009
0000000000000007
0000000000000001
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

reg_chunk before

```
// 0x00000000
0000000000000000
0000000000000000
0000000000000010
0000000000000007
0000000000000002
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

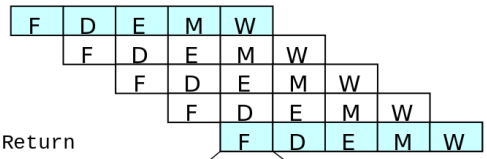
reg_chunk after



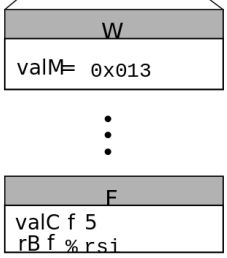
GTK WAVE

Correct Return Example

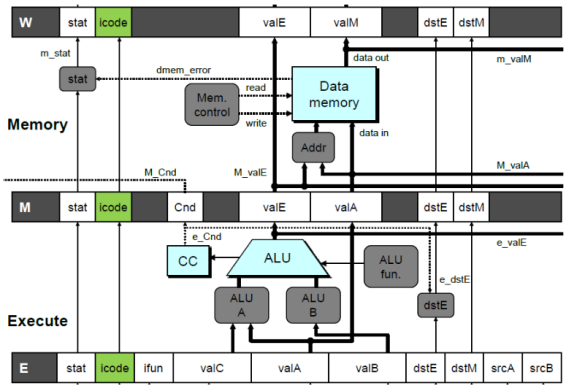
```
# demo- retb
0x026:     ret
           bubble
           bubble
           bubble
0x013:      irmovq$5,% rsi # Return
```



- As ret passes through pipeline, stall at fetch stage
  - While in decode, execute, and memory stage
- Inject bubble into decode stage
- Release stall when reach write-back stage



Detecting Return



Condition	Trigger
Processing ret	IRET in { D_icode, E_icode, M_icode }

Explanation: This is the instruction code for return, opq (add).

in_code = 9 => (return)

val_a = val_b = reg_chunk[rsp] = 1

val_e = 2, val_m = 4, PC=val_m = 4 (i.e instruction: 01100000)

i.e instruction at line 4(00100011) is skipped.

reg_chunk[3] = reg_chunk[4]+ reg_chunk[3] = 9+7 = 16.

TEST CASE2:

Instruction Code:

i.txt

```
01100011
00000000
01110100
00001101
00000000
00000000
00000000
00000000
00000000
00000000
00000000
01100000
00100011
00000000
```

common.txt

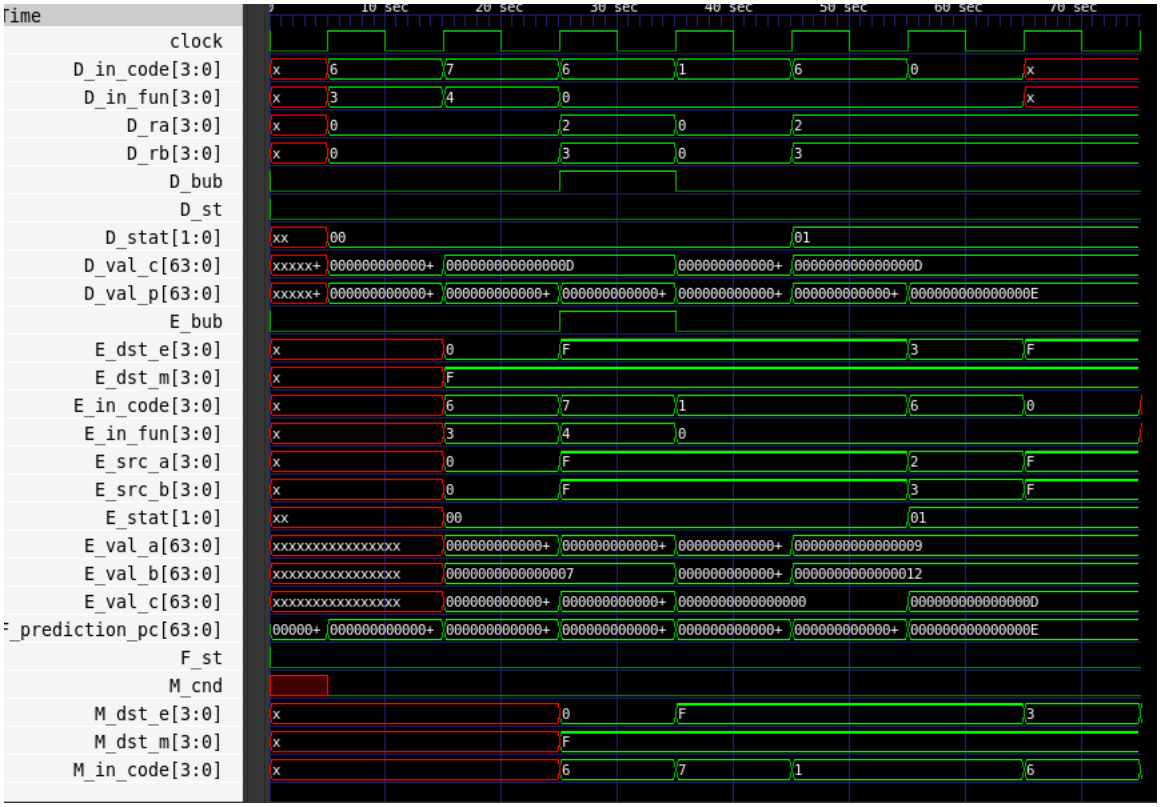
```
0000000000000007
0000000000000000
0000000000000009
0000000000000012
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

reg_chunk before

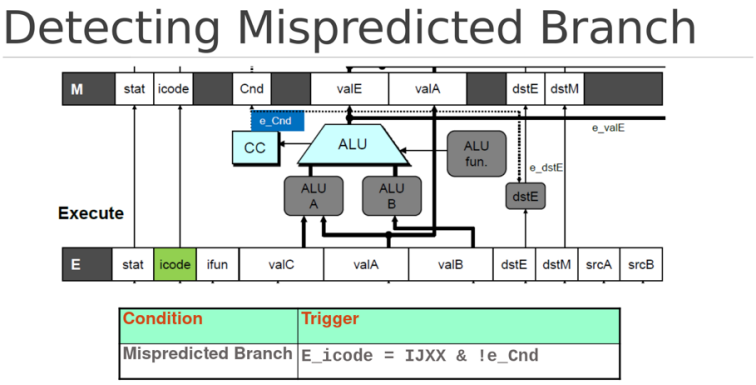
common.txt

```
// 0x00000000
0000000000000000
0000000000000000
0000000000000009
000000000000001b
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
```

reg_chunk after



GTK WAVE



Explanation: This is the instruction code for Jxx, Opq

in_code = 6(OPQ), f_in_code = 3(xor).

reg_chunk[0] = XOR(reg_chunk[0], reg_chunk[7]) = XOR(7,7)=0

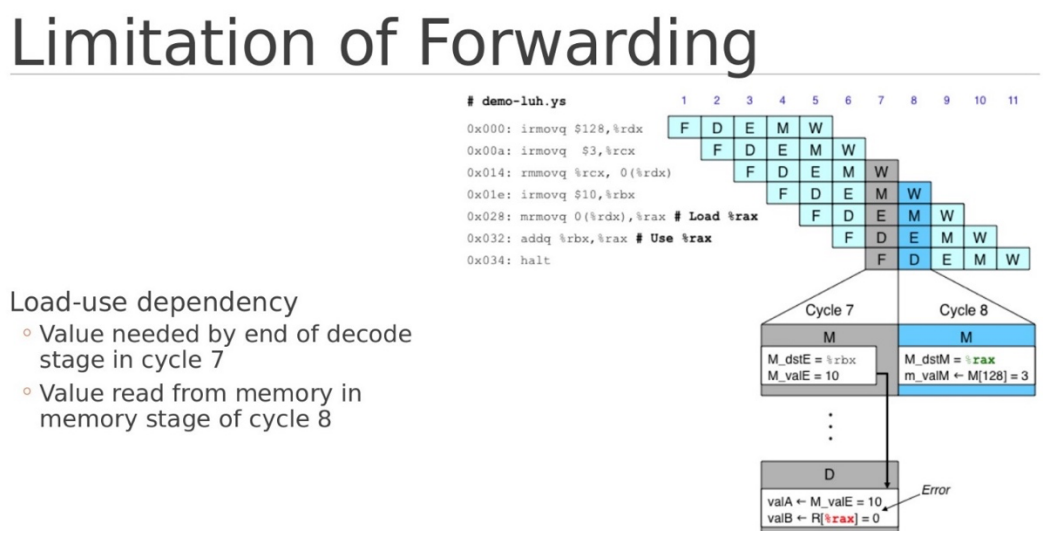
Now PC moves to instruction 01110100 => in_code=7(jxx), f_in_code = 4(jne). But since XOR(7,7) = 0. So Jump would not happen.

⇨ PC will move to line 9 i.e in_code=6(opq), f_in_code=0(add).

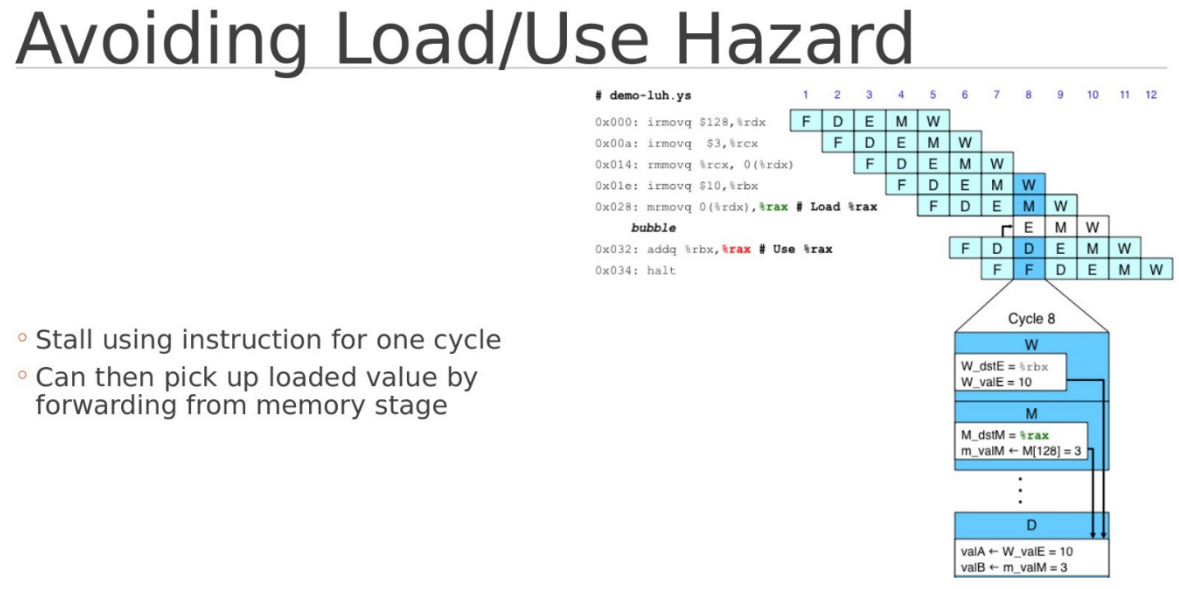
⇨ reg_chunk[3] = reg_chunk[2] + reg_chunk[3] = 9+18 = 27 (0x1b).

TEST CASE3:

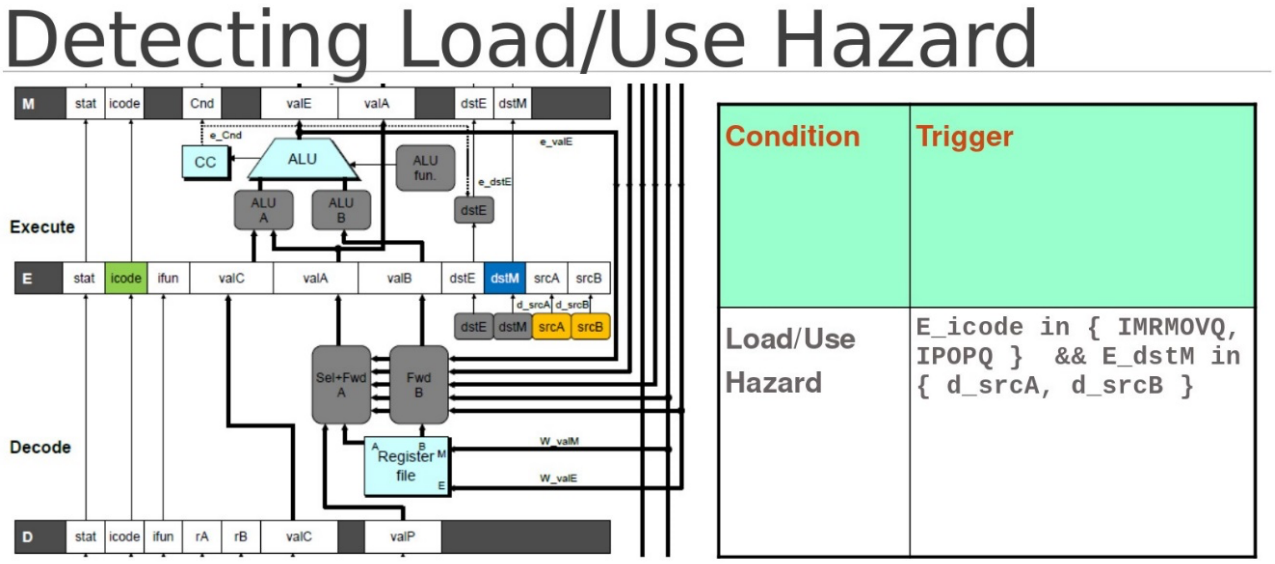
LOAD/USE HAZARD:



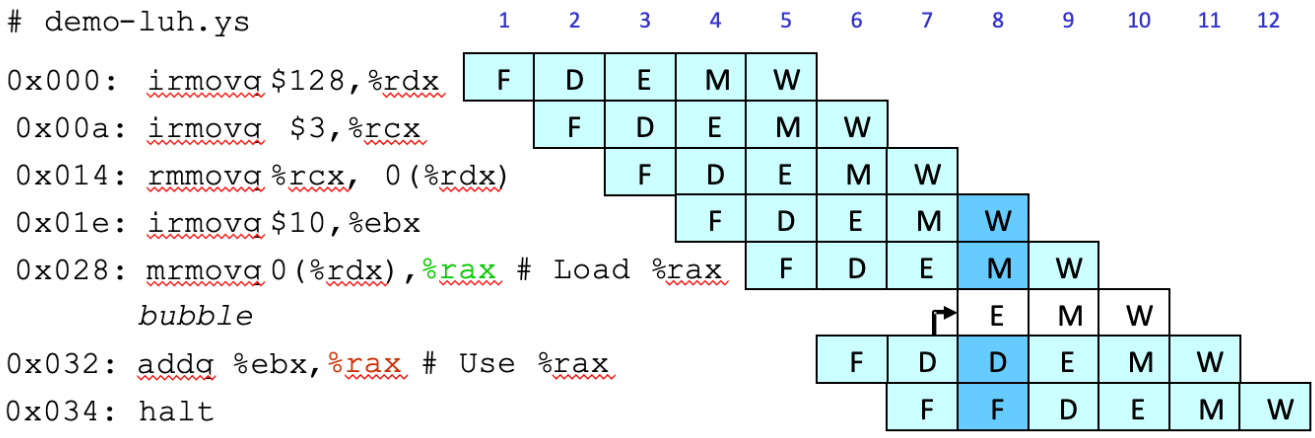
SOLUTION IMPLEMENTED IN CODE:



IDENTIFYING THIS HAZARD:



CODE IMPLEMENTED:



Instruction Code:

```
00110000
11110010
10000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00110000
11110001
00000011
00000000
00000000
00000000
00000000
00000000
00000000
00000000
01000000
00010010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00110000
11110011
00001010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
01010000
00000010
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
01100000
00110000
00000000
```

common.txt

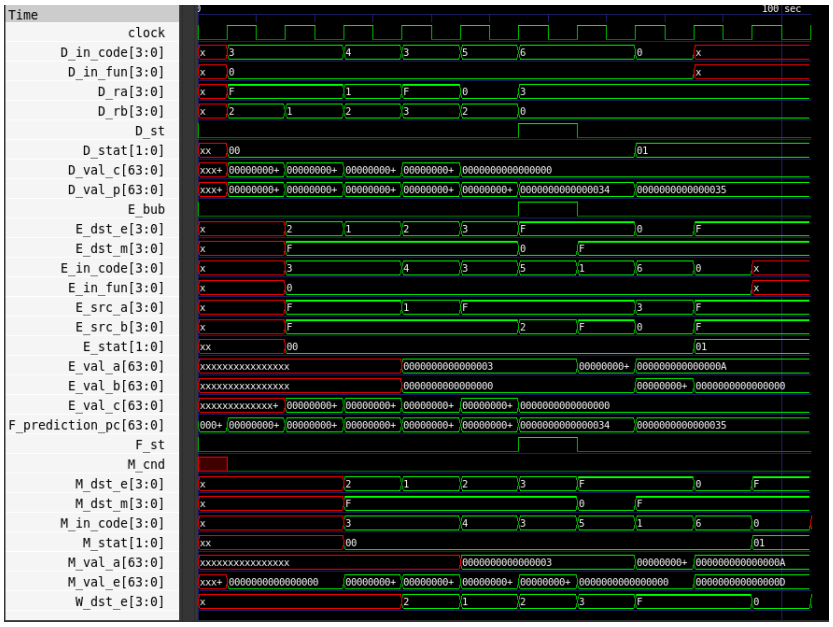
```
1   0000000000000000
2   0000000000000000
3   0000000000000000
4   0000000000000000
5   0000000000000000
6   0000000000000000
7   0000000000000000
8   0000000000000000
9   0000000000000000
10  0000000000000000
11  0000000000000000
12  0000000000000000
13  0000000000000000
14  0000000000000000
15  0000000000000000
```

Reg_chunk before

common.txt

```
1   // 0x00000000
2   000000000000000d
3   0000000000000003
4   0000000000000080
5   000000000000000a
6   0000000000000000
7   0000000000000000
8   0000000000000000
9   0000000000000000
10  0000000000000000
11  0000000000000000
12  0000000000000000
13  0000000000000000
14  0000000000000000
15  0000000000000000
16  0000000000000000
```

reg_chunk after



GTK WAVE

PROCESSOR FEATURES:

Processor Freq.

$$f = \frac{1}{T_{clk}} Hz$$

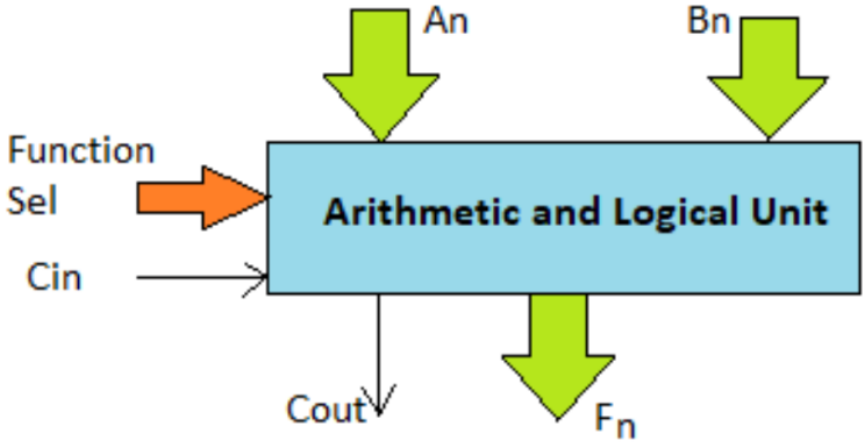
In this implementation we have taken $T_{clk} = 1ns$.

$$\Longrightarrow f = \frac{1}{1\times10^{-9}} Hz = 10^{9} Hz = 1GHz.$$

Memory Initialization:

The data memory and instruction memory are separated. The instruction memory has 256-32 bits words, that is 1024 bytes. This is equivalent to 1kB of instruction memory. While the data memory had 1024-64 bits which is 8kB od data memory.

ALU DESIGN:



An arithmetic-logic unit is the part of CPU that carries out arithmetic and logical operations on the operands in computer instruction words. We designed an ALU that performs four different arithmetic and logical operations on 64-bit operands: ADD, SUB, AND, XOR. This ALU takes two 64-bit operands and the operations to be performed as its inputs and returns a 64-bit output and an overflow bit (in case of ADD and SUB). We have created three different modules for the specified operations (ADD and SUB are performed
using the same module). A separate test bench has been written to test the functioning of each module separately.

ALU CONTROL AND OUTPUT DISPLAY:-

ALU as we know displays one of the four of the AND, ADD, SUB & XOR operations depending on select lines or control values:

1) Control 0 - ADD x and y
2) Control 1 – Subtract y from x
3) Control 2 – AND x and y
4) Control 3 – XOR x and y

Every time an input is given, all the four operations add, subtract, AND, XOR are computed and based on the control signal the required value is given as output.

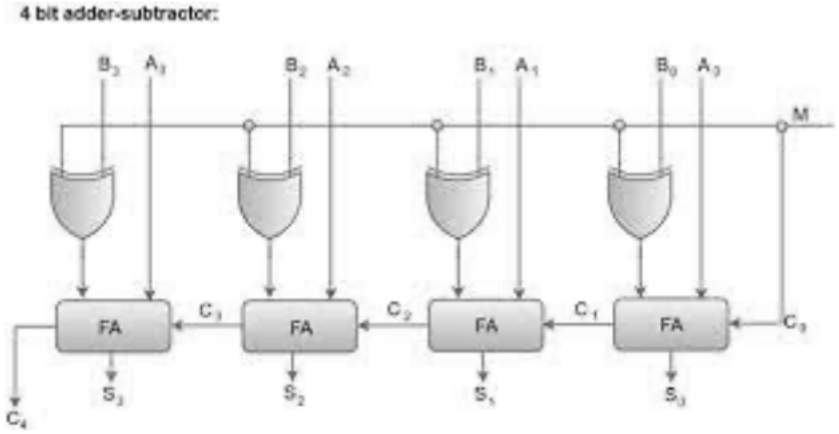
ALU BLOCK DETAILS ALONG WITH CODE,TESTBENCHES CONCEPTS AND RESULTS:

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

```
module XOR_operation (
  input [63:0] val1_xor,
  input [63:0] val2_xor,
  output [63:0] result_xor
//   output [63:0] additional_check
);
  genvar iterator_xor;


    generate for (iterator_xor = 0; iterator_xor <= 63; iterator_xor = iterator_xor + 1) begin
      xor(result_xor[iterator_xor], val1_xor[iterator_xor], val2_xor[iterator_xor]);
    end
  endgenerate
```

```
Test Case 1:
Input val1_xor: 0101101001011010010110100101101001011010010110100101101001011010
Input val2_xor: 1111111111111111111111111111111111111111111111111111111111111111
Result: 1010010110100101101001011010010110100101101001011010010110100101
Test Case 2:
Input val1_xor: 0000000000000000000000000000000000000000000000000000000000000000
Input val2_xor: 1111111111111111111111111111111111111111111111111111111111111111
Result: 1111111111111111111111111111111111111111111111111111111111111111
Test Case 3:
Input val1_xor: 0101101001011010010110100101101001011010010110100101101001011010
Input val2_xor: 0000000000000000000000000000000000000000000000000000000000000000
Result: 0101101001011010010110100101101001011010010110100101101001011010
Test Case 4:
Input val1_xor: 0000000000000000000000000000000000000000000000000000000000000000
Input val2_xor: 0000000000000000000000000000000000000000000000000000000000000000
Result: 0000000000000000000000000000000000000000000000000000000000000000
Test Case 5:
Input val1_xor: 1111111111111111111111111111111111111111111111111111111111111111
Input val2_xor: 1111111111111111111111111111111111111111111111111111111111111111
Result: 0000000000000000000000000000000000000000000000000000000000000000
```

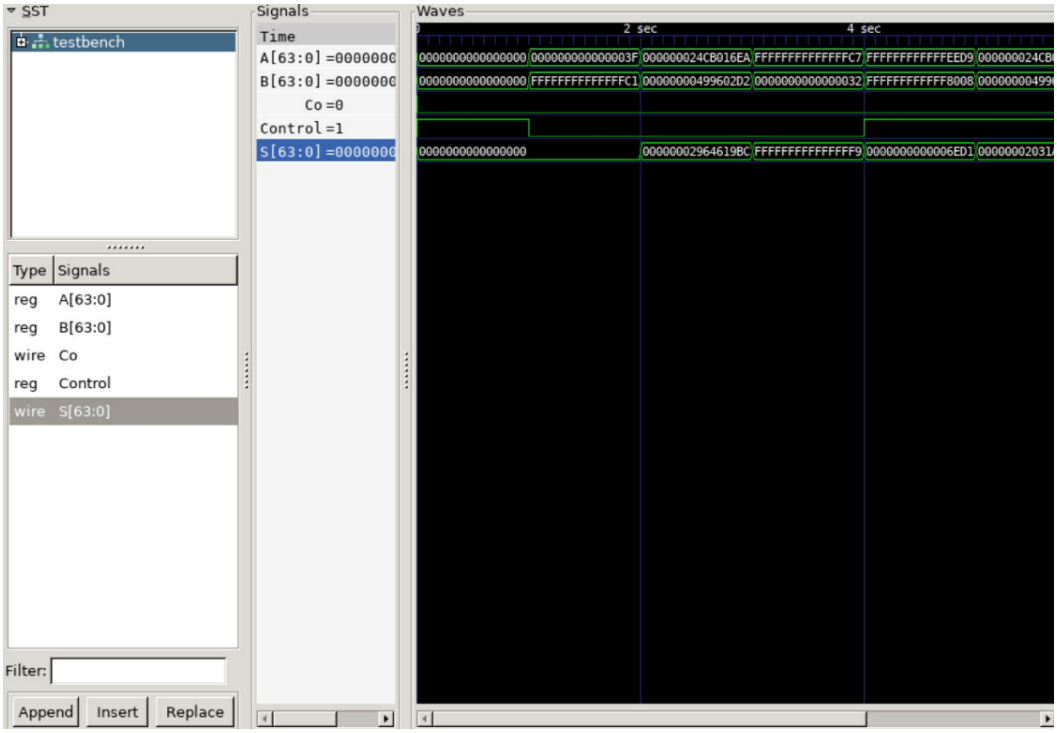
THE ADDER AND SUBTRACTOR MODULE (CONTROL VALUE=0 FOR ADDITION AND 1 FOR SUBTRACTION):




```verilog
`include "fullAdder.v"

module sum(A, B, Control, S, Co);

    input [63:0] A, B;
    input Control;

    output [63:0] S;
    output Co;

    wire [64:0] Cin;
    assign Cin[0] = Control;

    genvar i;
    generate for(i = 0; i <= 63; i = i+1)
        begin
            add_sub x1(A[i], B[i], Cin[i], Control, S[i], Cin[i+1]);
        end
    endgenerate

    xor(Co, Cin[64], Cin[63]);

endmodule
```

```
pingu@ubuntu ~/p/i/A/a/ALU (ALU)> iverilog -o out.vvp test_finalAddr.v finalAddSub.v
pingu@ubuntu ~/p/i/A/a/ALU (ALU)> vvp out.vvp
VCD info: dumpfile dump.vcd opened for output.
A = 00000000000000000000000000000000000000000000000000000000000000000000
B = 00000000000000000000000000000000000000000000000000000000000000000000
Control = 1
S = 00000000000000000000000000000000000000000000000000000000000000000000
Co = 0

A = 00000000000000000000000000000000000000000000000000000000000000111111
B = 11111111111111111111111111111111111111111111111111111111111111000001
Control = 0
S = 00000000000000000000000000000000000000000000000000000000000000000000
Co = 0

A = 00000000000000000000000000000001001001100101100000001011011101010
B = 00000000000000000000000000000001001001100101100000001011010010
Control = 0
S = 00000000000000000000000000000010100101100100011000011001101111100
Co = 0

A = 11111111111111111111111111111111111111111111111111111111111000111
B = 00000000000000000000000000000000000000000000000000000000000110010
Control = 0
S = 11111111111111111111111111111111111111111111111111111111111111001
-7 = 11111111111111111111111111001
Co = 0

A = 11111111111111111111111111111111111111111111111111110111011011001
B = 11111111111111111111111111111111111111111111111111100000000001000
Control = 1
S = 00000000000000000000000000000000000000000000000000110111011010001
Co = 0

A = 00000000000000000000000000000001001001100101100000001011011101010
B = 00000000000000000000000000000001001001100101100000001011010010
Control = 1
S = 00000000000000000000000000000010000000110001101000010100000011000
Co = 0
```

GTK WAVE OF ADD/SUB TEST BENCH:



ALU:

```
module add_sub(A, B, Cin, Control, S, Co);
    input A, B, Cin, Control;

    output S, Co;

    wire a_xor_b, a_and_b, and2, ctrl;

    xor(ctrl, Control, B);

    xor(a_xor_b, A, ctrl);
    xor(S, a_xor_b, Cin);
    and(a_and_b, A, ctrl);
    and(and2, a_xor_b, Cin);
    or(Co, a_and_b, and2);
endmodule

module XOR_operation (
  input [63:0] val1_xor,
  input [63:0] val2_xor,
  output [63:0] result_xor
//   output [63:0] additional_check
);
  genvar iterator_xor;


    generate for (iterator_xor = 0; iterator_xor <= 63; iterator_xor = iterator_xor + 1) begin
     xor(result_xor[iterator_xor], val1_xor[iterator_xor], val2_xor[iterator_xor]);
    end
  endgenerate


//   assign additional_check = val1_xor ^ val2_xor;
endmodule

module AND_operation (
  input [63:0] val1,
  input [63:0] val2,
  output [63:0] result
//   output [63:0] additional_check
);
  genvar iterator_and;


    generate  for (iterator_and = 0; iterator_and <= 63; iterator_and = iterator_and + 1) begin
      and(result[iterator_and], val1[iterator_and], val2[iterator_and]);
    end
  endgenerate


//   assign additional_check = val1 & val2;

endmodule

module sum(A, B, Control, S, Co);

    input [63:0] A, B;
    input Control;

    output [63:0] S;
    output Co;

    wire [64:0] Cin;
    assign Cin[0] = Control;

```

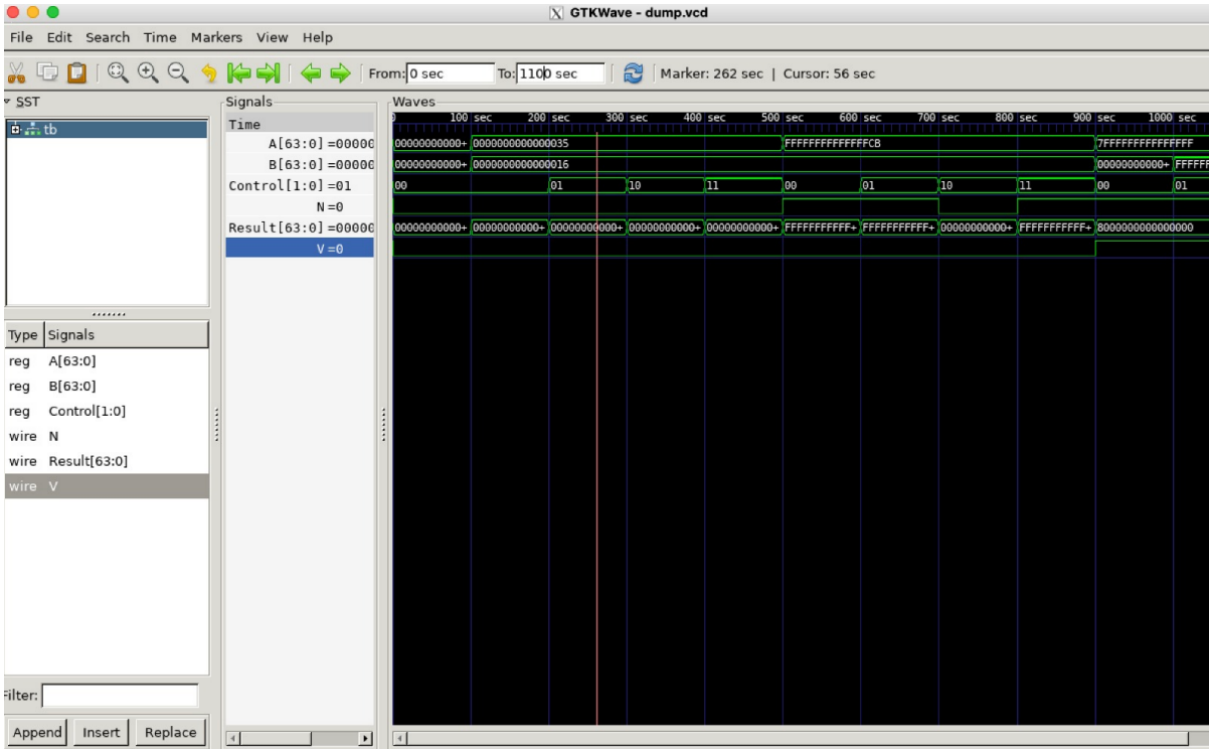
```
    genvar i;
    generate for(i = 0; i <= 63; i = i+1)
        begin
            add_sub x1(A[i], B[i], Cin[i], Control, S[i], Cin[i+1]);
        end
    endgenerate

    xor(Co, Cin[64], Cin[63]);

endmodule


// ============================== ALU Implementation=====================================


module ALU(A, B, Control, Result, flag);

    input [63:0] A, B;
    input [1:0] Control;

    output reg [63:0] Result;
    output wire [2:0] flag;

    wire [63:0] and_result, xor_result, add_result, sub_result;
    wire Cout_1, Cout_2;
    reg V; //overflow
    reg N; //Negitive flag

    AND_operation x1(A, B, and_result);
    XOR_operation x2(A, B, xor_result);
    sum x3(A, B, 1'b0, add_result, Cout_1);
    sum x4(A, B, 1'b1, sub_result, Cout_2);

    always @*
    begin
        if(Control == 2'b00)
        begin
            Result = add_result;
            V = Cout_1;
        end
        else if(Control == 2'b01)
        begin
            Result = sub_result;
            V = Cout_2;
        end
        else if(Control == 2'b10)
        begin
            Result = and_result;
            V = 1'b0;
        end
        else if(Control == 2'b11)
        begin
            Result = xor_result;
            V = 1'b0;
        end
        N = Result[63];
    end
    assign flag[0] = V; //overflow flag
    assign flag[1] = (N==1); //negative flag
    assign flag[2] = (Result==64'b0); //zero flag
endmodule
```

Output of ALU testbench

```
pingu@ubuntu ~/p/i/A/a/ALU (ALU)> vvp out.vvp
VCD info: dumpfile dump.vcd opened for output.
A: 00000000000000000000000000000000000000000000000000000000000000000000
B: 00000000000000000000000000000000000000000000000000000000000000000000
Control: 00
Result: 00000000000000000000000000000000000000000000000000000000000000000000
negative flag: 0
overflow flag: 0

A: 00000000000000000000000000000000000000000000000000000000000000110101
B: 00000000000000000000000000000000000000000000000000000000000000010110
Control: 00
Result: 00000000000000000000000000000000000000000000000000000000000001001011
negative flag: 0
overflow flag: 0

A: 00000000000000000000000000000000000000000000000000000000000000110101
B: 00000000000000000000000000000000000000000000000000000000000000010110
Control: 01
Result: 00000000000000000000000000000000000000000000000000000000000000011111
negative flag: 0
overflow flag: 0

A: 00000000000000000000000000000000000000000000000000000000000000110101
B: 00000000000000000000000000000000000000000000000000000000000000010110
Control: 10
Result: 00000000000000000000000000000000000000000000000000000000000000010100
negative flag: 0
overflow flag: 0

A: 00000000000000000000000000000000000000000000000000000000000000110101
B: 00000000000000000000000000000000000000000000000000000000000000010110
Control: 11
Result: 00000000000000000000000000000000000000000000000000000000000000100011
negative flag: 0
overflow flag: 0
```

```
A: 11111111111111111111111111111111111111111111111111111111111111001011
B: 00000000000000000000000000000000000000000000000000000000000000010110
Control: 00
Result: 11111111111111111111111111111111111111111111111111111111111100001
negative flag: 1
overflow flag: 0

A: 11111111111111111111111111111111111111111111111111111111111111001011
B: 00000000000000000000000000000000000000000000000000000000000000010110
Control: 01
Result: 11111111111111111111111111111111111111111111111111111111110110101
negative flag: 1
overflow flag: 0

A: 11111111111111111111111111111111111111111111111111111111111111001011
B: 00000000000000000000000000000000000000000000000000000000000000010110
Control: 10
Result: 00000000000000000000000000000000000000000000000000000000000000000010
negative flag: 0
overflow flag: 0

A: 11111111111111111111111111111111111111111111111111111111111111001011
B: 00000000000000000000000000000000000000000000000000000000000000010110
Control: 11
Result: 11111111111111111111111111111111111111111111111111111111111011101
negative flag: 1
overflow flag: 0
```

```
A: 01111111111111111111111111111111111111111111111111111111111111111
B: 00000000000000000000000000000000000000000000000000000000000000001
Control: 00
Result: 10000000000000000000000000000000000000000000000000000000000000000
negative flag: 1
overflow flag: 1

A: 01111111111111111111111111111111111111111111111111111111111111111
B: 11111111111111111111111111111111111111111111111111111111111111111
Control: 01
Result: 10000000000000000000000000000000000000000000000000000000000000000
negative flag: 1
overflow flag: 1
```



GTK WAVE OUTPUT

CHALLENGES FACED:

- While making pipeline architecture we faced issue in implementing stall and data forwarding. So, for that we had to refer to the theory and get a better understanding of these topics to be able to implement them.
- Another issue we faced during the pipeline part was in branch prediction. We forgot to add a condition to predict PC for jxx. And a similar mistake also happened in execute since input and output variable names for the execute block while assigning input to the Memory block I input wrong variable to it.