Intro to Processor Architecture Project Report

Shriya Dullur - 2020102006
Aishwaryabharathi Upadhyayula – 2020102060

Objective:

To build a modular processor architecture design based on the Y86-64 ISA using Verilog. Thoroughly test the design to satisfy all the specification requirements using simulations.

Sequential Processor:

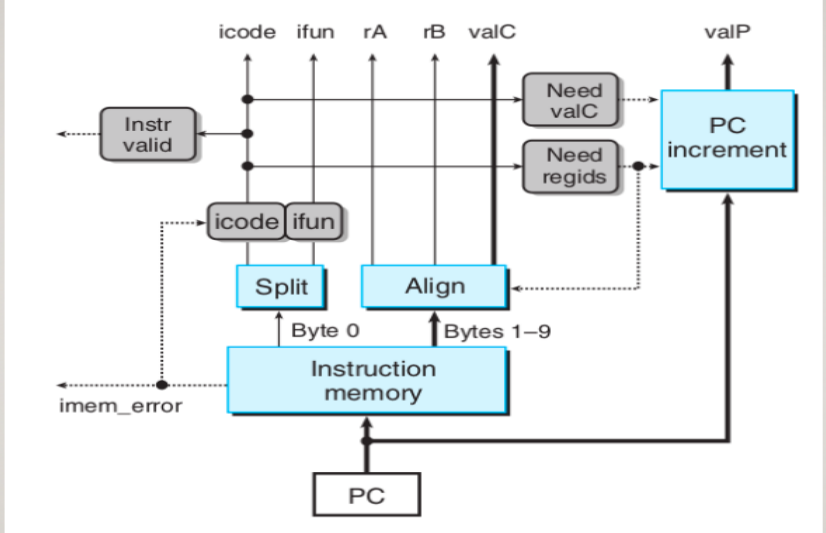
Firstly, we have implemented the Sequential Processor which has 6 modules or individual blocks, which are, Fetch, Decode, Write back, Execute, Memory and PC Update. Next, we will take a look at each of these modules individually.

FETCH MODULE :

The fetch stage reads the bytes of an instruction from instruction memory , using the program counter (PC) as the memory address. From the instruction it extracts the two 4-bit portions of the instruction specifier byte, icode (the instruction code) and ifun (the instruction function). It possibly fetches a register specifier byte, giving one or two register operand specifiers rA and rB. It also possibly fetches an 8-byte constant word valC. It computes valP to be the address of the instruction following the current one in sequential order. That is, valP equals the value of the PC plus the length of the fetched instruction. Here we can observe that valC value can't be directly obtained from reading the instruction memory as the bytes are in reverse order for which i did the extra processing required in the last few lines of the code. An extra variable f_c is included and reset to 0 initially when the FETCH is executed and later set to 1.

Parameters

- inputs: CLK, PC
- outputs: icode, ifun, rA, rB, valC
- Triggering Variable–PC (any change)



DECODE MODULE :

The decode stage reads up to two operands from the register file, giving values valA and/or valB. Initially we create 15 program registers Typically, it reads the registers designated by instruction fields rA and rB, but for some instructions it reads register %rsp. These values are first stored in the variables srcA and srcB and then written back to valA and valB variables.

WRITE-BACK MODULE :

The write-back stage is triggered with clock and writes up to two results {valE, valM} to the designated register file. (same as Decode module). Atmost two values can be written back to register file.

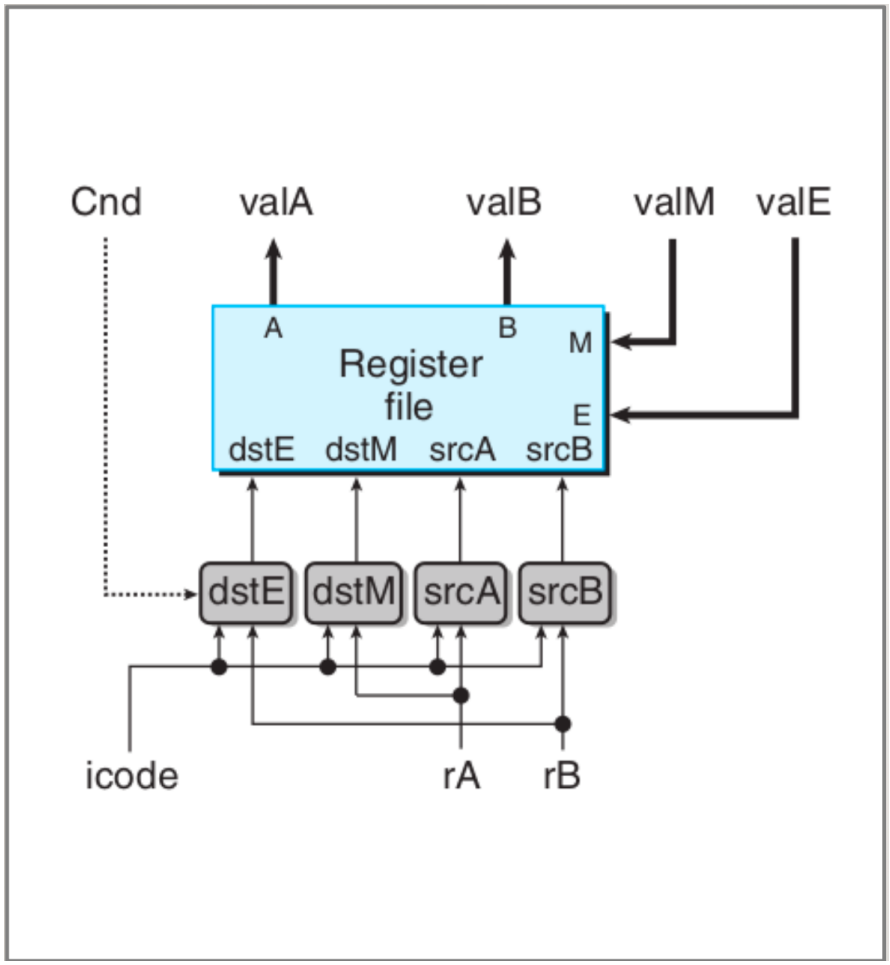
Parameters :

DECODE stage

- INPUTS: icode, ifun, rA, rB
- OUTPUTS: valA, valB

WRITE_BACK stage

- INPUTS: Cnd, valM,valE, R0,R1,R2,R3,R4,R5,R6,R7,R8,R9,R10,R11,R12,R13,R14;
  - OUTPUTS: Ro0,Ro1,Ro2,Ro3,Ro4,Ro5,Ro6,Ro7,Ro8,Ro9,Ro10,Ro11,Ro12,Ro13,Ro14



Number	Register name	Number	Register name
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	No register

EXECUTE MODULE

In the execute stage, the arithmetic/logic unit (ALU) either performs the operation specified by the instruction (according to the value of ifun), computes the effective address of a memory reference, or increments or decrements the stack pointer. We refer to the resulting value as valE. The condition codes are possibly set (only for icode == 6). For a conditional move instruction, the stage will evaluate the condition codes and move condition (given by ifun) and enable the updating of the destination

register only if the condition holds. Similarly, for a jump instruction, it determines whether the branch should be taken or not. This program is triggered after the DECODE stage using this variable.

Instructions :

- icode = 2 : The instruction is cmovXX . This instruction enables us to move the data from a register to other based on the condition codes set ie, zero flag(zf), signed flag(sf) and overflow flag(of). There are 7 possible values for ifun : rrmovq (0) , cmovle (1) , cmovl(2),cmove (3) , cmovne (4) , cmovge (5) , cmovg (6). Cnd condition code is set along with finding the value valE.
- icode = 3 : irmovq : Moves immediate value to the specified register.
- Icode = 4 : rmmovq : Copies the value from register to memory
- icode = 5 : mrmovq: Copies value stored in memory to register.
- Icode = 6 : Opq : For performing ALU operations like ADD,SUB,AND,XOR.
- icode = 7 : jXX Jump statements. Based on the conditional codes – zf,sf,of , a Condition Cnd is set and if the value is 1, then jump is performed at PC update or it remains the same.
- icode = 8 : call : A module/function is called
- icode = 9 : ret : Return statement.
- icode = A : pushq : A value is pushed to the top of the stack by decrementing the stack pointer.
- icode = B : popq : The top most value of stack is popped by incrementing the %rsp.

The value in valE is written back to memory or register only Cnd =1.

•For example, suppose we used one of the add instructions to perform the equivalent of the C assignment t = a+b, where variables , b, and t are integers. Then the condition codes would be set according to the following C expressions:

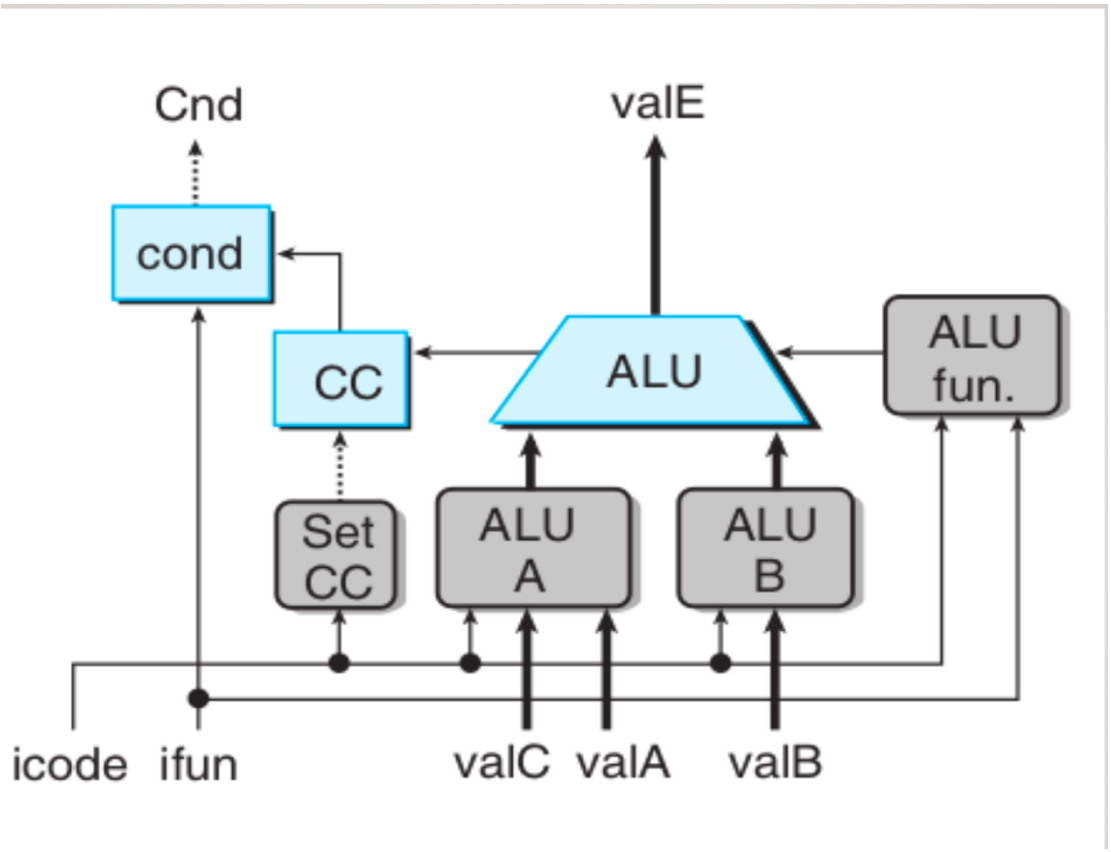
CF	(unsigned) t < (unsigned) a	Unsigned overflow
ZF	(t == 0)	Zero
SF	(t < 0)	Negative
OF	(a < 0 == b < 0) && (t < 0 != a < 0)	Signed overflow

ZF	Equal / zero
~ZF	Not equal / not zero
~(SF ^ OF) & ~ZF	Greater (signed >)
~(SF ^ OF)	Greater or equal (signed >=)
SF ^ OF	Less (signed <)
(SF ^ OF) | ZF	Less or equal (signed <=)

Parameters

EXECUTE stage

- inputs: icode, ifun, valA, valB, valC, valM, zf, sf, of
- OUTPUTS: Cnd, valE



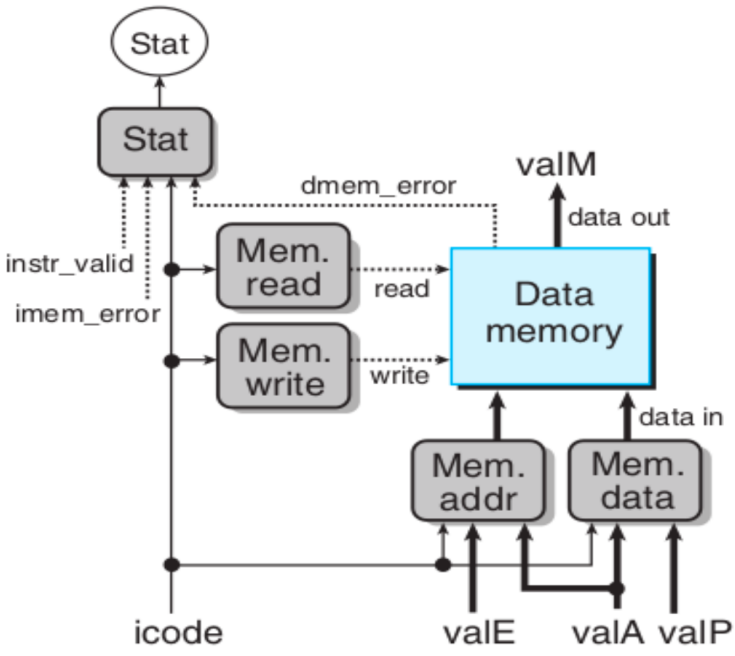
MEMORY MODULE :

The memory stage may write data to memory, or it may read data from memory. We refer to the value read as valM.

Parameters

MEMORY stage

- INPUTS: icode, valE,valA, valB, mem, valP
- OUTPUTS: valM, value



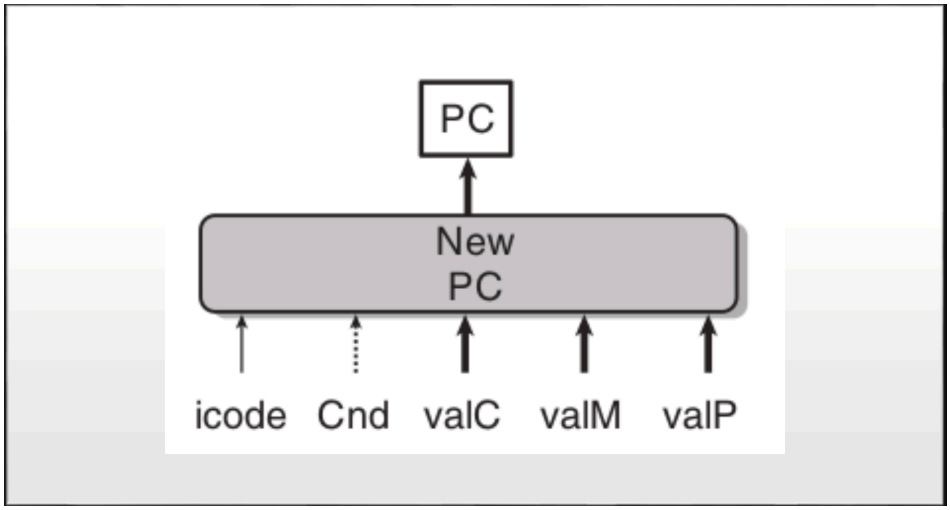
PC_UPDATE MODULE

The PC is set to the address of the next instruction. The new PC will be valC or valM or valP, depending on the instruction type and whether a conditional branch should be taken. This function is triggered by positive edge of clock and this is the last stage in our sequential implementation. If there are instructions left, PC would get updated triggering the fetch block and again the same process happens. Also, initially, pc is reset to zero in this module.

Parameters

PC_UPDATE stage

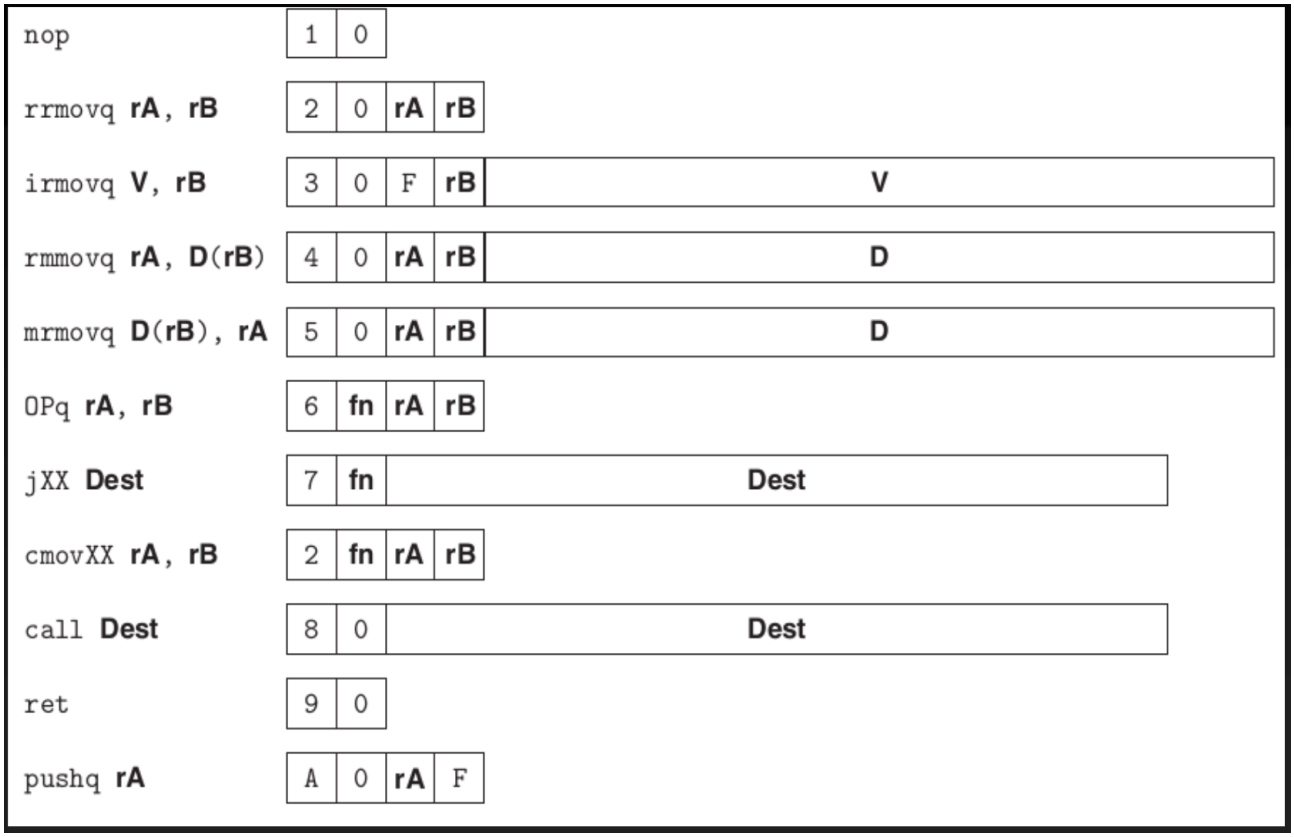
- INPUTS: icode,Cnd, valC, valM, valP
- OUTPUTS: PC

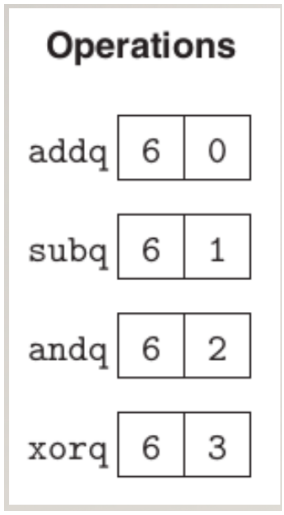


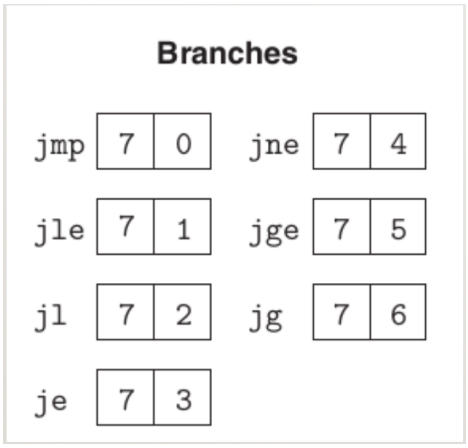
ERROR/STATUS CODES

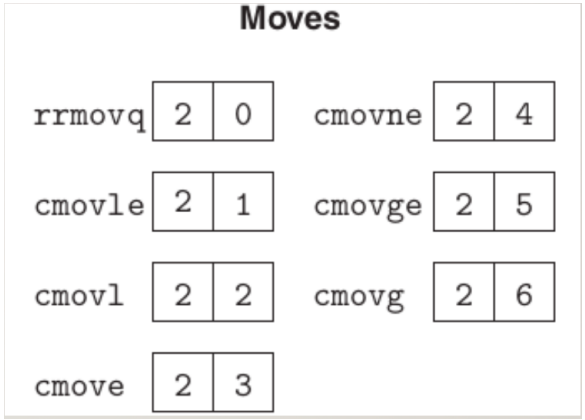
- Error/status codes in various stages are programmed as mentioned in the textbook.This includes inst_valid,

INSTRUCTIONS SUPPORTED BY THE PROCESSOR


nop 1 0
rrmovq rA, rB 2 0 rA rB
irmovq V, rB 3 0 F rB V
rmmovq rA, D(rB) 4 0 rA rB D
mrmovq D(rB), rA 5 0 rA rB D
OPq rA, rB 6 fn rA rB
jXX Dest 7 fn Dest
cmovXX rA, rB 2 fn rA rB
call Dest 8 0 Dest
ret 9 0
pushq rA A 0 rA F

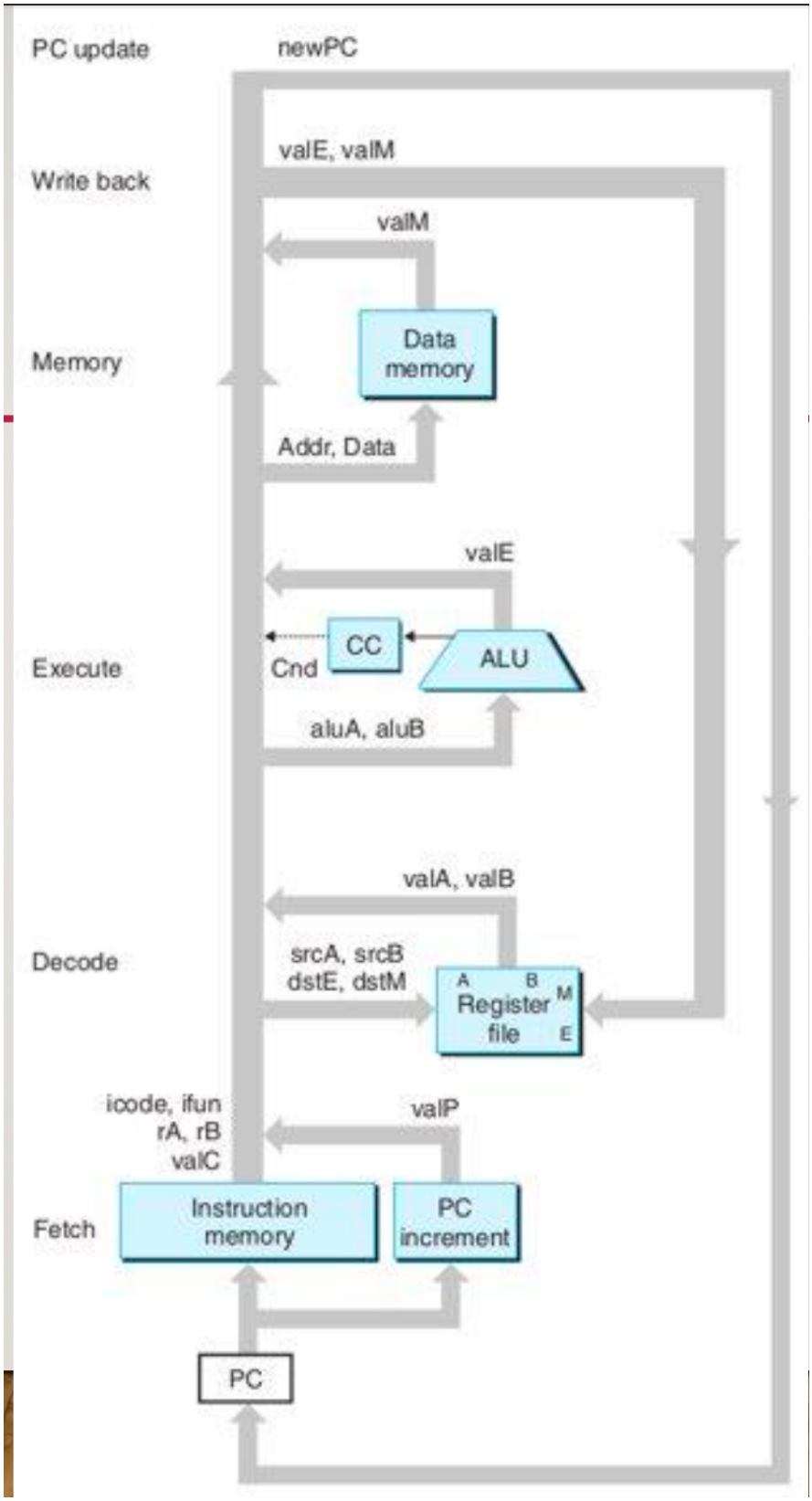

Operations
addq 6 0
subq 6 1
andq 6 2
xorq 6 3


Branches
jmp 7 0
jne 7 4
jle 7 1
jge 7 5
jl 7 2
jg 7 6
je 7 3


Moves
rrmovq 2 0
cmovne 2 4
cmovle 2 1
cmovge 2 5
cmovl 2 2
cmovg 2 6
cmove 2 3

Assembler

- This module is the final packaging unit of Y-86seq processor. This has all initial initiations of {modules, clock, variables}.
- Then all modules are given required variables. This module does not have any input/output. So, writing a testbench file did not seem viable.
- While giving inputs in the text file, each byte should be separated with a space or enter. Formatting of those input should follow Y-86 ISA scheme.



ALU

Consists of four operations – Addition (ADD) , Subtraction (SUB), AND and XOR for 64 bit data.

ADD

Adds two 64 bit numbers. Also finds out the overflow and underflow case.

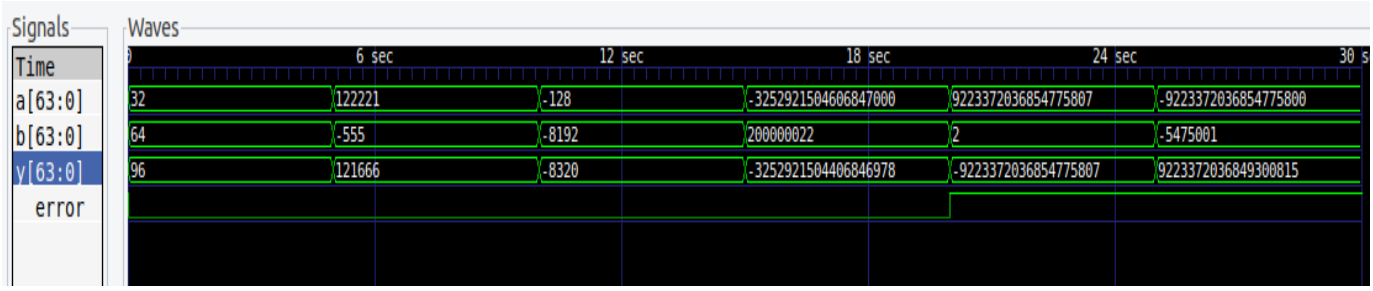
Inputs

```
 a = 32 ; b = 64 ;
#5 a = 122221 ; b = -555 ;
#5 a = -128 ; b = -8192 ;
#5 a = -3252921504606847000 ; b = 200000022;
#5 a = 9223372036854775807 ; b = 2 ; // a = maximum value a 64-bit signed integer can hold ; OVERFLOW CASE
#5 a = -9223372036854775800; b = -5475001 ; // UNDERFLOW CASE
```

Outputs

```
a = 0000000000000000000000000000000000000000000000000000000000100000
b = 0000000000000000000000000000000000000000000000000000000001000000
y = 0000000000000000000000000000000000000000000000000000000001100000

a = 0000000000000000000000000000000000000000000000001110111010110110 1
b = 1111111111111111111111111111111111111111111111111111111110111010101
y = 0000000000000000000000000000000000000000000000001110110110100001 0

a = 1111111111111111111111111111111111111111111111111111111110000000
b = 1111111111111111111111111111111111111111111111111000000000000000
y = 1111111111111111111111111111111111111111111111111011111110000000

a = 1101001011011011010011010010000001010011101101111111111111101000
b = 0000000000000000000000000000000000001011111010111100001000010110
y = 1101001011011011010011010010000001011111100110011100000111111110

Overflow has occurred for the inputs :
a = 0111111111111111111111111111111111111111111111111111111111111111
b = 0000000000000000000000000000000000000000000000000000000000000010
Please give valid inputs

Underflow has occurred for the inputs :
a = 1000000000000000000000000000000000000000000000000000000000001000
b = 1111111111111111111111111111111111111111101011000111010101000111
Please give valid inputs
```

GTKwave



SUB

Performs Signed Subtraction between two 64-bit numbers.

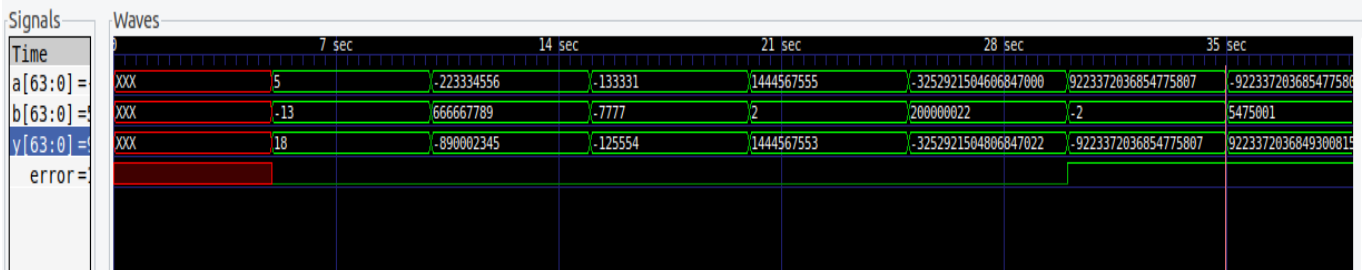
Inputs

```
#5 a = 5 ; b = -13 ;
#5 a = -223334556 ; b = 666667789 ;
#5 a = -133331 ; b = -7777 ;
#5 a = 1444567555 ; b = 2 ;
#5 a = -3252921504606847000 ; b = 200000022;
#5 a = 9223372036854775807 ; b = -2 ; // a = maximum value a 64-bit signed integer can hold ; OVERFLOW CASE
#5 a = -9223372036854775800; b = 5475001 ; // UNDERFLOW CASE
```

Outputs

```
a = 0000000000000000000000000000000000000000000000000000000000000101
b = 1111111111111111111111111111111111111111111111111111111111110011
y = 0000000000000000000000000000000000000000000000000000000000010010

a = 1111111111111111111111111111111111100101011000000101111101100100
b = 0000000000000000000000000000000000010011110111001000101100001101
y = 1111111111111111111111111111111111100101011110011101001000101011 1

a = 1111111111111111111111111111111111111111111111101111101110010110 1
b = 1111111111111111111111111111111111111111111111110000110011111
y = 1111111111111111111111111111111111111111111111100001010110001110

a = 0000000000000000000000000000000010101100001101001011010000000011
b = 0000000000000000000000000000000000000000000000000000000000000010
y = 0000000000000000000000000000000010101100001101001011010000000001

a = 1101001011011011010011010010000001010011101011011111111111101000
b = 0000000000000000000000000000000000010111110101111000010000010110
y = 1101001011011011010011010010000001000111110000100011110111010010

Overflow has occurred for the inputs :
a = 0111111111111111111111111111111111111111111111111111111111111111
b = 1111111111111111111111111111111111111111111111111111111111111110
Please give valid inputs

Underflow has occurred for the inputs :
a = 1000000000000000000000000000000000000000000000000000000000001000
b = 0000000000000000000000000000000000000000000010100111000101010111001
Please give valid inputs
```

GTKwave



AND

ANDs two 64-bit numbers without directly using the & operation.

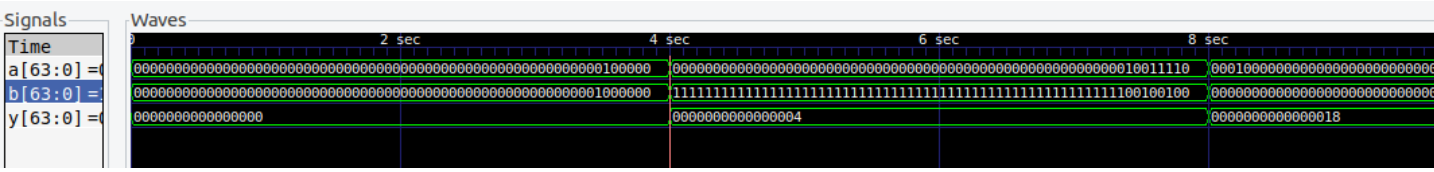
Inputs

```
a = 32 ; b = 64 ;
#4 a = 158 ; b = -220;
#4 a = 1152921504606847000 ; b = 5475001 ;
#4 a = -3252921504606847000 ; b = 200000022;
```

Output

```
a = 0000000000000000000000000000000000000000000000000000000000100000
b = 0000000000000000000000000000000000000000000000000000000001000000
y = 0000000000000000000000000000000000000000000000000000000000000000

a = 0000000000000000000000000000000000000000000000000000000010011110
b = 1111111111111111111111111111111111111111111111111111111100100100
y = 0000000000000000000000000000000000000000000000000000000000000100

a = 0001000000000000000000000000000000000000000000000000000000011000
b = 0000000000000000000000000000000000000000010100111000101010111001
y = 0000000000000000000000000000000000000000000000000000000000011000

a = 1101001011011011010011010010000001010011101011011111111111101000
b = 0000000000000000000000000000000000000010111110101111000010000010110
y = 0000000000000000000000000000000000000001110101001110000010000000000
```

Gtkwave

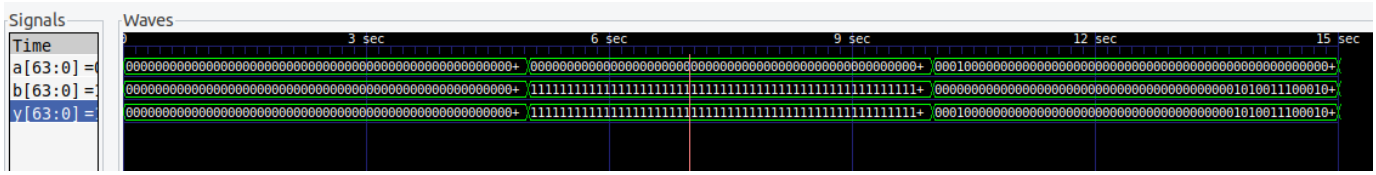


XOR

Outputs for same inputs as AND
Performs XOR operation

```
a = 0000000000000000000000000000000000000000000000000000000000100000
b = 0000000000000000000000000000000000000000000000000000000001000000
y = 0000000000000000000000000000000000000000000000000000000001100000

a = 0000000000000000000000000000000000000000000000000000000010011110
b = 1111111111111111111111111111111111111111111111111111111100100100
y = 1111111111111111111111111111111111111111111111111111111110111010

a = 0001000000000000000000000000000000000000000000000000000000011000
b = 0000000000000000000000000000000000000000010100111000101010111001
y = 0001000000000000000000000000000000000000010100111000101010100001

a = 1101001011011011010011010010000001010011101011011111111111101000
b = 0000000000000000000000000000000000000010111110101111000010000010110
y = 1101001011011011010011010010000001011000010001100011110111111110
```

Gtkwave



ALU

Takes inputs and the controls and calculates the required value.

Input

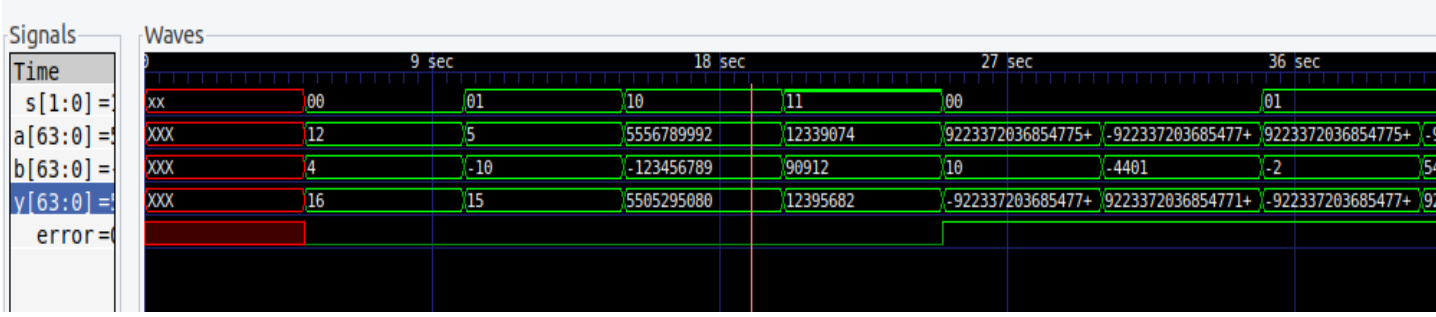
```
#5 s = 2'b00 ; a = 12; b = 4 ;
#5 s = 2'b01 ; a = 5 ; b = -10;
#5 s = 2'b10 ; a = 5556789992 ; b = -123456789 ;
#5 s = 2'b11 ; a = 12339074 ; b = 90912;
#5 s = 2'b00 ; a = 9223372036854775803 ; b = 10;     // OVERFLOW CASE
#5 s = 2'b00 ; a = -9223372036854775800; b = -4401; // UNDERFLOW CASE
#5 s = 2'b01 ; a = 9223372036854775807 ; b = -2 ;   // OVERFLOW CASE
#5 s = 2'b01 ; a = -9223372036854775800; b = 5471 ; // UNDERFLOW CASE
```

Output

```
Control = 00
Inputs :
a = 00000000000000000000000000000000000000000000000000000000000001100
b = 00000000000000000000000000000000000000000000000000000000000000100
Output :
    00000000000000000000000000000000000000000000000000000000000010000

Control = 01
Inputs :
a = 00000000000000000000000000000000000000000000000000000000000000101
b = 11111111111111111111111111111111111111111111111111111111111110110
Output :
    00000000000000000000000000000000000000000000000000000000000001111

Control = 10
Inputs :
a = 00000000000000000000000000000001010010110011010111100010111101000
b = 11111111111111111111111111111111111100010100100001100101110101 1
Output :
    00000000000000000000000000000001010010000010010000100010111101000

Control = 11
Inputs :
a = 00000000000000000000000000000000000000000101111000100011110000010
b = 00000000000000000000000000000000000000000000000101100011001000000
Output :
    00000000000000000000000000000000000000000101111010010010010100010

Control = 00
Overflow has occurred for the inputs :
a = 01111111111111111111111111111111111111111111111111111111111111011
b = 00000000000000000000000000000000000000000000000000000000000001010
Please give valid inputs
```

```
Control = 00
Underflow has occurred for the inputs :
a = 1000000000000000000000000000000000000000000000000000000000001000
b = 1111111111111111111111111111111111111111111111111011011001111
Please give valid inputs

Control = 01
Overflow has occurred for the inputs :
a = 0111111111111111111111111111111111111111111111111111111111111111
b = 1111111111111111111111111111111111111111111111111111111111111110
Please give valid inputs

Control = 01
Underflow has occurred for the inputs :
a = 1000000000000000000000000000000000000000000000000000000000001000
b = 0000000000000000000000000000000000000000000000001010101011111
Please give valid inputs
```

Gtkwave



SEQ design

Fetch module

Fetch stage finds out icode, ifun, rA,rB,valC (if present, x otherwise) and compute valP value. ValP = program counter + length of instruction. The following is the output.

For the Instruction 06 00 which means, OPq with addition .

```
PC=                  0
instr=0110000001000010110011011010101110001001011001110100000001000101001000110000000 1
icode= 0110
ifun=0000
rA=0100
rB=0010
valC=                x
valP=               16
```

Decode module

In this stage, the values present in register file with indexes rA and rB should be extracted into valA and valB respectively.

The testbench is written in such a way that if there is no change in valA, valB ie, if any instruction does not use both rA,rB , then the value from previous is stored.

```
 0icode =  6, rA =  0 , rB =  2 , valA =                    1 , valB =                    3
10icode =  8, rA =  0 , rB =  2 , valA =                    1 , valB =                    5
20icode = 11, rA =  0 , rB =  2 , valA =                    5 , valB =                    5
30icode =  5, rA =  1 , rB =  7 , valA =                    5 , valB =                    8
40icode =  9, rA =  1 , rB =  7 , valA =                    5 , valB =                    5
```

Execute module

ValA , valB,valC passed from decode and fetch stages are used to find out the value of the output valE. The condition codes are also set at execute stage depending on the instruction and values obtained. For a given icode ie, the instruction, ifun value is used (if any) to find out the exact operation and using the valA, valB,valC and stack pointer register , we can find out the output from execute .
For the test bench,

```
initial begin
$monitor($time," icode = %d ifun = %d\nvalA = %d \nvalB = %d\nvalC = %d \nvalE = %d\n Cnd=%d\n zf = %d of  = %d sf =
 %d",icode,ifun,valA,valB,valC,valE,Cnd,zf,of,sf);
icode = 4'd3 ; valC = 64'd2 ; clk = 1;
#10 icode =4'd4 ;valB = 64'd9;
#10 icode = 4'd9;
#10 icode = 4'd6 ; ifun = 4'd1; valA = 64'd1 ;valB = 64'd1 ;
#10 valA = -10 ; valB = 20 ;
#10 ifun = 4'd2 ; valA = 1; valB = 2;
#10 icode = 4'd2 ;valA = 7 ;ifun = 4 ;
#10 icode = 4'd7 ; ifun = 2 ;
#10 ifun = 3 ;
#10 icode = 4'd6 ; ifun = 4'd0; valA = 9223372036854775807 ; valB = 2 ;
#10 $finish ;
```

Outputs are

```
              0 icode =  3 ifun =  x
valA =                  x
valB =                  x
valC =                  2
valE =                  2
Cnd=0
zf = 0 of  = 0 sf = 0
             10 icode =  4 ifun =  x
valA =                  x
valB =                  9
valC =                  2
valE =                 11
Cnd=0
zf = 0 of  = 0 sf = 0
             20 icode =  9 ifun =  x
valA =                  x
valB =                  9
valC =                  2
```

```
valE =              17
Cnd=0
zf = 0 of  = 0 sf = 0
            30 icode =  6 ifun =  1
valA =              1
valB =              1
valC =              2
valE =              0
Cnd=0
zf = 1 of  = 0 sf = 0
            40 icode =  6 ifun =  1
valA =             -10
valB =              20
valC =              2
valE =             -30
Cnd=0
zf = 0 of  = 0 sf = 1
            50 icode =  6 ifun =  2
valA =              1
valB =              2
valC =              2
valE =              0
Cnd=0
zf = 1 of  = 0 sf = 0
            60 icode =  2 ifun =  4
valA =              7
valB =              2
valC =              2
valE =              7
Cnd=1
zf = 0 of  = 0 sf = 0
            70 icode =  7 ifun =  2
valA =              7
valB =              2
valC =              2
valE =              7
Cnd=0
zf = 0 of  = 0 sf = 0
            80 icode =  7 ifun =  3
valA =              7
valB =              2
valC =              2
```

valE = 7
Cnd=0
zf = 0 of = 0 sf = 0
90 icode = 6 ifun = 0
valA = 9223372036854775807
valB = 2
valC = 2
valE = -9223372036854775807
Cnd=0
zf = 0 of = 1 sf = 1

Memory module

In the memory stage, values are read from memory / written back into memory depending on the instruction .

Writeback stage

The values from execute and memory stages are written back into the register file used in decode stage. Maximum of two values can be written back. Considers condition codes set in the execute stage to write back. Writes back into the register file only if the condition codes of jXX, cmovXX are set to 1.

In decode and writeback, as same register file is used, we need to pass the inputs of register file to both the stages from the main assembler.

For the following test bench inputs, the outputs are shown.

```
icode = 4'd6 ; rA = 0; rB = 1 ; valM = 0 ;valE = 19 ;
#10 icode = 4'd8 ;
#10 icode = 4'hB; valM= 13 ;
#10 icode = 4'd1 ;
#10 icode = 4'd2 ; Cnd = 0 ;rB = 5 ; valE = 7 ;
#10 Cnd = 1;
```

Outputs are :
ValM is the output of memory stage
ValE is the output of execute stage
icode is the instruction
rA and rB are the register operands
R0,R1,R7 are the values in the register file (input)
Ro0,Ro1,Ro4,Ro5 are the values in modified register (output)

Outputs for the inputs :

```
                0 icode =  6, Cnd = x
valM=             0
valE =           19
rA =  0 , rB =  1,
R0=             0
R1 =             1
R7 =             7
Ro0 =            x
Ro1 =            x
Ro4 =            x
Ro5 =            x

              10 icode =  8, Cnd = x
valM=             0
valE =           19
rA =  0 , rB =  1,
R0=             0
R1 =             1
R7 =             7
Ro0 =            0
Ro1 =            1
Ro4 =           19
Ro5 =            5

              20 icode = 11, Cnd = x
valM=            13
valE =           19
rA =  0 , rB =  1,
R0=             0
R1 =             1
R7 =             7
Ro0 =            0
Ro1 =            1
Ro4 =           19
Ro5 =            5

              30 icode =  1, Cnd = x
valM=            13
valE =           19
rA =  0 , rB =  1,
```

```
R0=              0
R1 =             1
R7 =             7
Ro0 =            0
Ro1 =            1
Ro4 =            4
Ro5 =            5

          40 icode =  2, Cnd = 0
valM=            13
valE =           7
rA =  0 , rB =  5,
R0=              0
R1 =             1
R7 =             7
Ro0 =            0
Ro1 =            1
Ro4 =            4
Ro5 =            5

          50 icode =  2, Cnd = 1
valM=            13
valE =           7
rA =  0 , rB =  5,
R0=              0
R1 =             1
R7 =             7
Ro0 =            0
Ro1 =            1
Ro4 =            4
Ro5 =            7
```

Here, we can notice, the values every alternate clock cycle, because the write back updates at negative edge of clock and the change can be seen in next clock cycle. Clock negates for every 10 ns

PC update stage

	OPq rA, rB	
PC update	PC ← valP	Update PC
	rmmovq rA, D(rB)	
PC update	PC ← valP	Update PC
	popq rA	
PC update	PC ← valP	Update PC
	jXX Dest	
PC update	PC ← Cnd ? valC : valP	Update PC
	call Dest	
PC update	PC ← valC	Set PC to destination
	ret	
PC update	PC ← valM	Set PC to return address

For the inputs, outputs are

```
icode = 6 ; valP = 5 ;
#10 icode = 7 ; Cnd = 1; valC = 13 ;
#10 Cnd = 0 ;
#10 icode = 9 ; valM = 4;
#10 icode = 3; valP = 0;
```

```
icode =  6 , Cnd = x, valP =                5, valC =                x, valM =                x, PC =                5
icode =  7 , Cnd = 1, valP =                5, valC =               13, valM =                x, PC =               13
icode =  7 , Cnd = 0, valP =                5, valC =               13, valM =                x, PC =                5
icode =  9 , Cnd = 0, valP =                5, valC =               13, valM =                4, PC =                4
icode =  3 , Cnd = 0, valP =                0, valC =               13, valM =                4, PC =                0
```

Status codes

The following are the status codes used in Processor design.

Value	Name	Meaning
1	AOK	Normal operation
2	HLT	halt instruction encountered
3	ADR	Invalid address encountered
4	INS	Invalid instruction encountered

We implemented the processor with codes AOK, HLT and INS

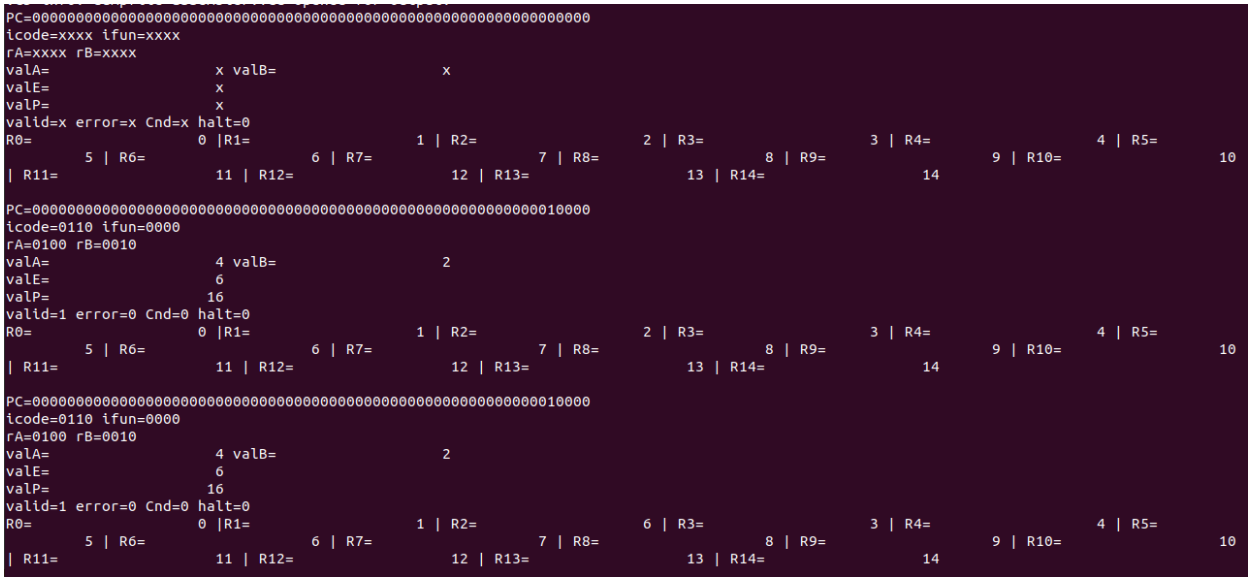
Overall Sequential design

Example :

For the following ,
0110000001000010110011011010101110001001011001110100000001000101001
0001100000001

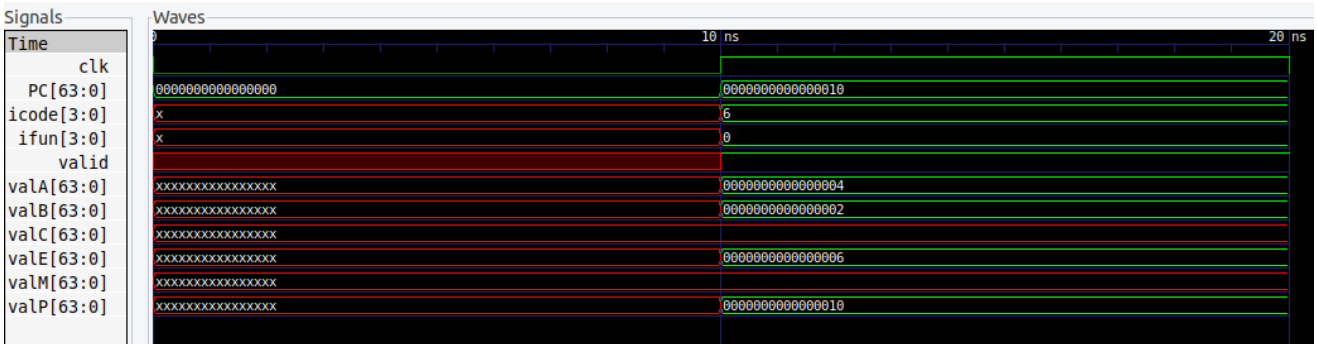
Expected output for first instruction is
icode = 0110
ifun = 0000
rA = 0100
rB = 0010
valC = xx
valA = R[rA] = 4
valB = R[rB] = 2
valE = valA + valB = 6
ValP = 2 bytes
PC = 0(initial PC) + valP = 2 bytes = 16 bits
Contition codes : zf = 0 ; sf = 0 ; of = 0
Cnd = 0 (for jXX and cmovXX)
Valid = 1 (instruction validity)
hlt= ins=error = 0

Obtained output



Here , when t = 0, all the values are set to xx
After the first clock cycle, the instruction is read and the values are calculated at the positive edge and the values can be seen. But writeback updates values at the

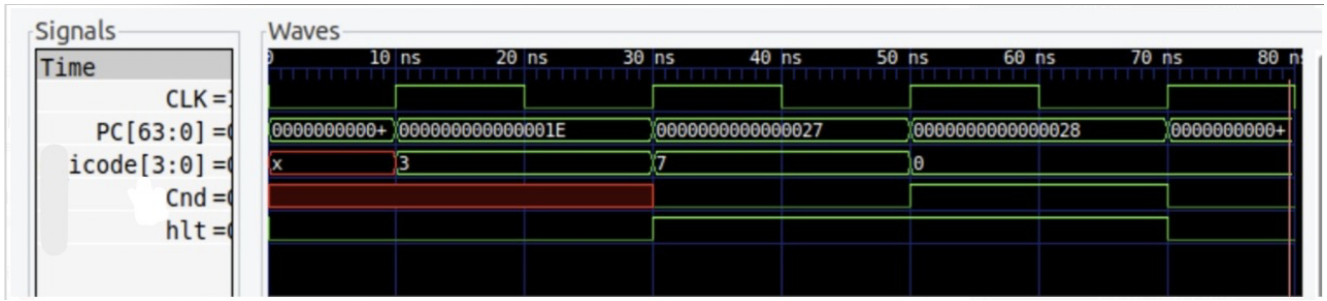
negative edge of clock , so we can notice the change in register file in the next clock cycle.



For multiple instructions,
Eg : irmovq -> jXX -> halt (from instruction memory)

```
VCD info: dumpfile assembler.vcd opened for output.
PC=                    20 CLK=0 icode= 3 halt=0
PC=                    30 CLK=1 icode= 7 halt=0
PC=                    30 CLK=0 icode= 7 halt=0
PC=                    39 CLK=1 icode= 0 halt=1
PC=                    39 CLK=0 icode= 0 halt=1
PC=                    40 CLK=1 icode= 0 halt=1
PC=                    40 CLK=0 icode= 0 halt=1
PC=                    41 CLK=1 icode= x halt=0
PC=                    41 CLK=0 icode= x halt=0
```

Gtkwave for the above instructions



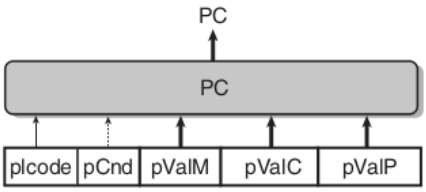
Pipelined Y86-64 Architecture

In Pipelining, multiple instructions are overlapped and different instructions are executed in different stages. Pipeline is divided into stages and are connected to each other. A key feature of pipelining is that it increases the throughput of the system, but it may increase latency. Each stage has combinational logic.
In pipelining, the outputs of a stage will be loaded as inputs to next stage when the positive edge of clock arises.

Pipelining implementations

Rearranging the Computation Stages

In Sequential design, the PC update is at the end, but in pipelining, the PC update stage comes to the beginning of the clock cycle. The updated PC value is sent to fetch stage which is calculated from the values passed from different stages .

Pipeline and Sequential 

have very similar hardware except, we insert registers between stages of pipelining. These registers stop the signals from one stage from flowing into the next stage and affecting the processing happening there.

The five registers are :

- **F** : Holds a predicted Program counter value.
- **D** : present between Decode and Fetch stages.It holds information about the most recently fetched instruction for processing by the decode stage.
- **E** : Present 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** : Present 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** : Present 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.

There is no register holding values for PC update.

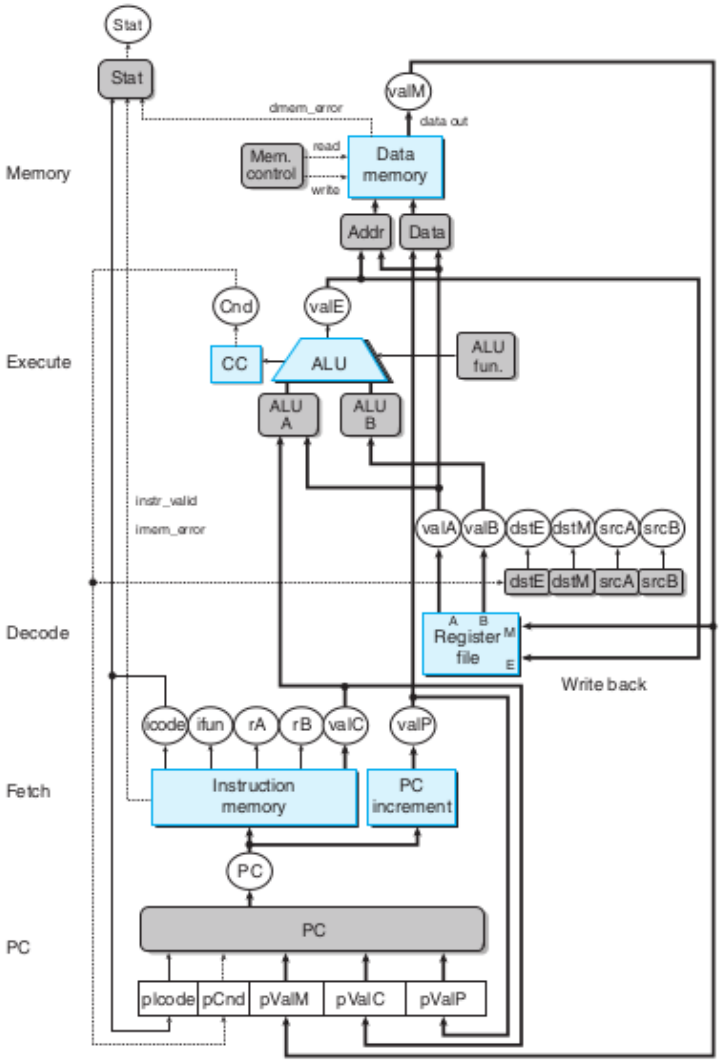
The PC computation at the beginning of the clock cycle is suitable for Pipeline design.

SEQ+ hardware :

Rearranging and Relabelling signals

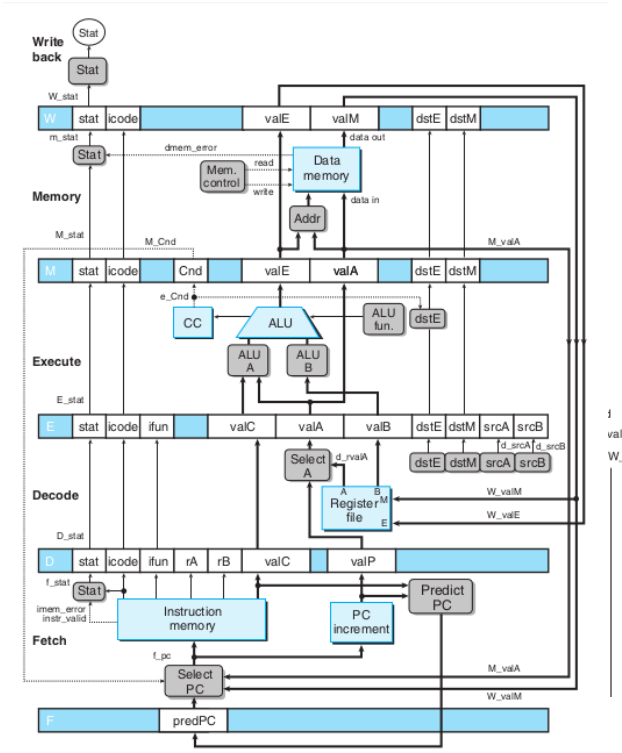
In pipelining, the instruction have to pass through each stage one by one and in every stage, the values needed for executing the instruction will be relabelled. If the icode passes out from execute stage, then it is relabelled as E_icode. Similarly for all stages, f_icode,e_valE etc. value is simply held

If the then Capital letter in a stage, prefix otherwise, if is used to is computed at a the value small letter is used. stage,



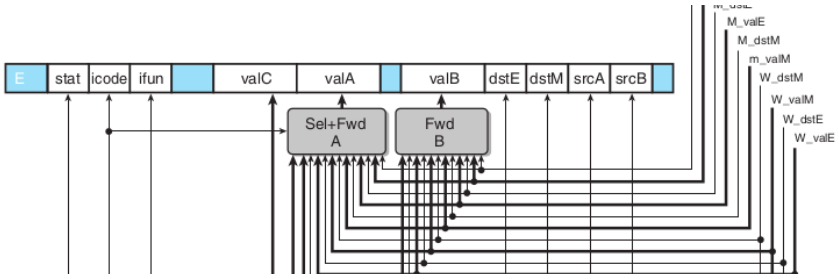
Architecture

The above picture shows the Hardware design structure of Pipeline

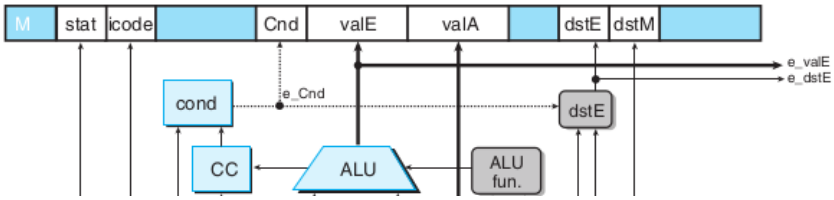


PC Prediction and Fetch

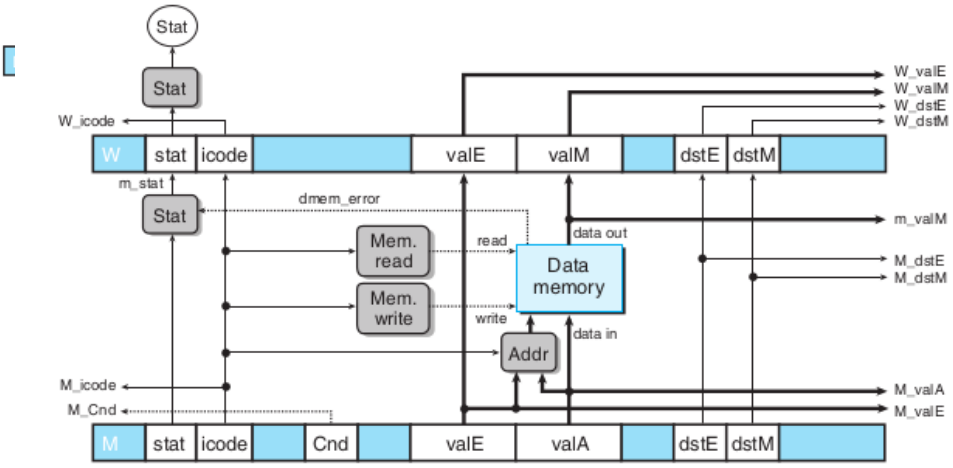
Decode and Writeback stage



Execute stage



Memory stage



Instructions supported by the architecture

- halt
- nop
- cmovxx
- irmovq
- rmmovq
- mrmovq
- Opq
- jXX
- call
- ret
- pushq
- popq

General Pipeline Control logic:

Load/use hazards:
The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.
Processing ret:
The pipeline must stall until the ret instruction reaches the write-back stage.
Mispredicted branches:
By the time the branch logic detects that a jump should not have been taken, several instructions at the branch target will have started down the pipeline. These instructions must be canceled, and fetching should begin at the instruction following the jump instruction.
Exceptions:
When an instruction causes an exception, we want to disable the updating of the programmer-visible state by later instructions and halt execution once the excepting instruction reaches the write-back stage.

Testing the Pipelining Code:

The sequential processor is already ready, minor changes have to be made to it apart from adding the code for the 5 registers to obtain the pipelined processor.

Now, we shall take a look at the program structure of each of the five registers.

Fetch Register:

In this register, we take the input as the clk and the predicted PC value and the output that we get will be the f_predPC value.

```
input CLK,input [63:0] predPC, output reg [63:0] f_predPC
```

At the positive edge of the clock, f_predPC takes the value of the predPC.

```
f_predPC <= predPC;
```

Decode Register:

The outputs of the Fetch block are fed as the inputs to this register. The inputs to this register include,

```
input CLK, input [2:0] F_status, input [3:0] F_icode, input [3:0] F_ifun, input [3:0]
F_rA, input [3:0] F_rB, input [63:0] F_valC, input [63:0] F_valP
```

The outputs that we get will be,

```
output reg [2:0] d_status, output reg [3:0] d_icode, output reg [3:0] d_ifun, output reg
[3:0] d_rA, output reg [3:0] d_rB, output reg [63:0] d_valC, output reg [63:0] d_valP
```

At the positive edge of the clock, each of the inputs get transferred to another set of decode variables,

```
d_status<=F_status;
d_rA<=F_rA;
d_rB<=F_rB;
d_valC<=F_valC;
d_icode<=F_icode;
d_ifun<=F_ifun;
d_valP<=F_valP;
```

Execute Register:

This register takes the outputs of the decode stage as the inputs of this registers including,

```
input CLK, input [2:0] D_status, input [3:0] D_icode, input [3:0] D_ifun, input [3:0]
D_rA, input [3:0] D_rB, input [63:0] D_valC, input [63:0] D_valP, input [63:0] D_valA,
input [63:0] D_valB
```

The outputs of this register are,

```
output reg [2:0]  e_status, output reg [3:0]  e_icode, output reg [3:0]  e_ifun, output reg [3:0]  e_rA, output reg [3:0]  e_rB, output reg [63:0] e_valC, output reg [63:0] e_valP, output reg [63:0] e_valA, output reg [63:0] e_valB
```

On the positive edge of the clock, the input variables get transferred to the output variables using a non-blocking assignment,

```
e_icode <= D_icode;
e_rB <= D_rB;
e_valA <= D_valA;
e_ifun <= D_ifun;
e_valB <= D_valB;
e_valC <= D_valC;
e_status <= D_status;
e_valP <= D_valP;
e_rA <= D_rA;
```

Memory Register:

The inputs to this register are the outputs of the Execute block, including,

```
input CLK, input [2:0] E_status, input [3:0] E_icode, input [3:0] E_rA, input [3:0] E_rB, input [63:0] E_valC, input [63:0] E_valP, input [63:0] E_valA, input [63:0] E_valB, input E_Cnd, input [63:0] E_valE
```

The outputs of this register are,

```
output reg [2:0] m_status, output reg [3:0] m_icode, output reg [3:0] m_rA, output reg [3:0] m_rB, output reg [63:0] m_valC, output reg [63:0] m_valP, output reg [63:0] m_valA, output reg [63:0] m_valB, output reg m_Cnd, output reg [63:0] m_valE
```

The inputs variables are transferred to the output variables above at the rising edge of the clock using non-blocking assignments.

```
m_status <= E_status;
m_icode <= E_icode;
m_rA <= E_rA;
m_rB <= E_rB;
m_valC <= E_valC;
m_valP <= E_valP;
m_valA <= E_valA;
m_valB <= E_valB;
m_Cnd <= E_Cnd;
m_valE <= E_valE;
```

Write back Register:

The inputs of the write back register are the outputs of the memory register. The inputs of this register are:

```
input clk,M_Cnd, input [2:0] M_status, input [3:0] M_icode,M_rA,M_rB, input [63:0]
M_valC,M_valP,M_valA,M_valB,M_valE,M_valM
```

The outputs of this register are:

```
output reg [2:0] w_status,output reg [3:0] w_icode,w_rA,w_rB,output reg
[63:0]w_valC,w_valP,w_valA,w_valB,output reg w_Cnd,output reg [63:0]w_valE,w_valM
```

The input values are transferred to the outputs using non-blocking assignments at the rising edge of the clock.

```
w_status    <=   M_status;
    w_icode    <=   M_icode;
    w_rA       <=   M_rA;
    w_rB       <=   M_rB;
    w_valA     <=   M_valA;
    w_valB     <=   M_valB;
    w_valC     <=   M_valC;
    w_valE     <=   M_valE;
    w_valM     <=   M_valM;
    w_valP     <=   M_valP;
    w_Cnd      <=   M_Cnd;
```

Using the 5 registers and the 5 stages, we can run our processor by finally writing an assembler unit which calls all the variables and calls all the functions in a synchronized manner by using a clock.

The instructions that we have used for testing the processor are:

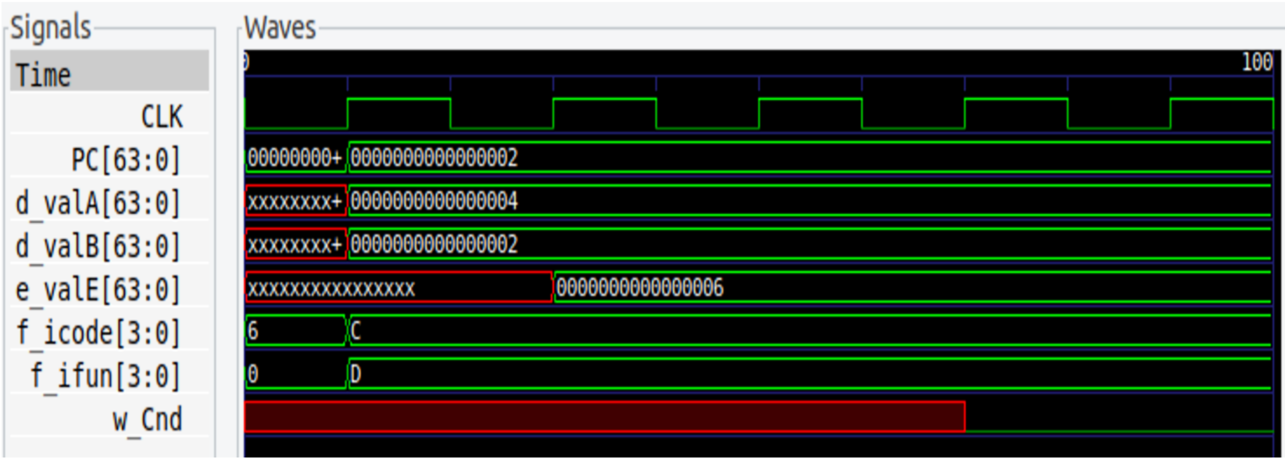
- Addq
- Subq

Addq:

The output values PC , CLK, icode, ifun, valA, valB, valE obtained in the terminal for the instruction Addq are as follows:

```
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ iverilog assembler_pipe.v
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ ./a.out
VCD info: dumpfile assembler.vcd opened for output.
PC=                    0 clk=0 icode= 6 ifun= x valA=                    x valB=                    x valE=                    x
PC=                    2 clk=1 icode=12 ifun= 0 valA=                    x valB=                    x valE=                    x
PC=                    2 clk=0 icode=12 ifun= 0 valA=                    x valB=                    x valE=                    x
PC=                    2 clk=1 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    6
PC=                    2 clk=0 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    6
PC=                    2 clk=1 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    6
PC=                    2 clk=0 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    6
PC=                    2 clk=1 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    6
PC=                    2 clk=0 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    6
PC=                    2 clk=1 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    6
PC=                    2 clk=0 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    6
```

The gtkwave waveforms for the Addq instruction are as follows:

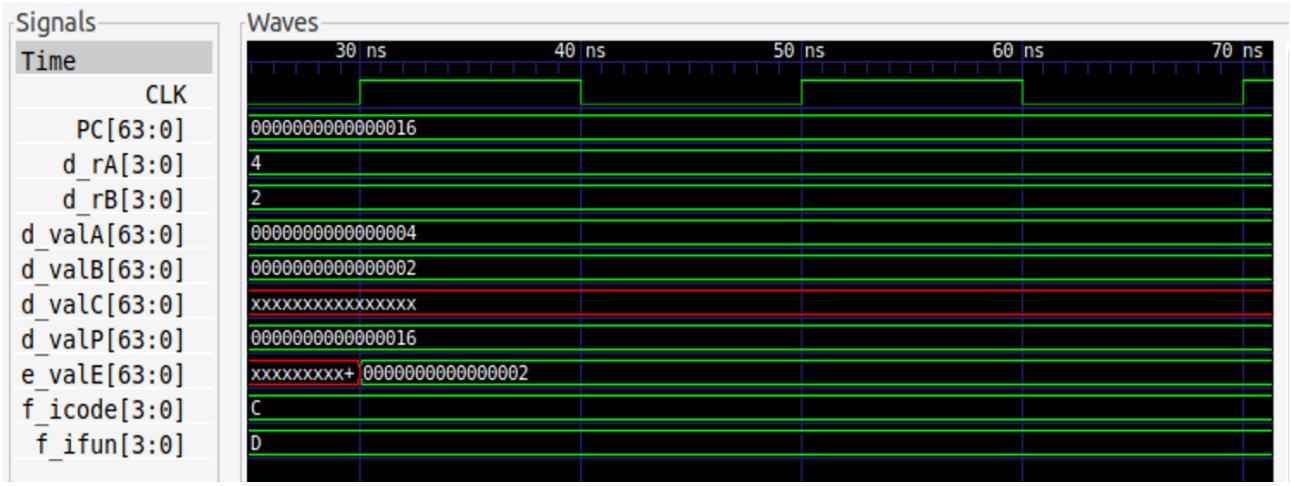


subq:

The output values PC , CLK, icode, ifun, valA, valB, valE obtained in the terminal for the instruction Subq are as follows:

```
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ iverilog assembler_pipe.v
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ ./a.out
VCD info: dumpfile assembler.vcd opened for output.
PC=                   20 clk=0 icode= 6 ifun= x valA=                    x valB=                    x valE=                    x
PC=                   22 clk=1 icode=12 ifun= 1 valA=                    x valB=                    x valE=                    x
PC=                   22 clk=0 icode=12 ifun= 1 valA=                    x valB=                    x valE=                    x
PC=                   22 clk=1 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    2
PC=                   22 clk=0 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    2
PC=                   22 clk=1 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    2
PC=                   22 clk=0 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    2
PC=                   22 clk=1 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    2
PC=                   22 clk=0 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    2
PC=                   22 clk=1 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    2
PC=                   22 clk=0 icode=12 ifun=13 valA=                    4 valB=                    2 valE=                    2
```

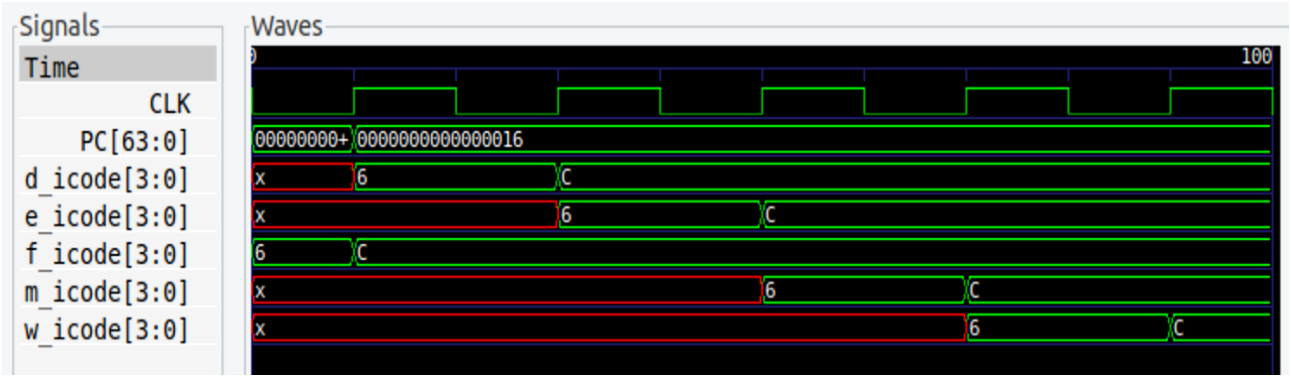
The gtkwave waveforms for the Subq instruction are as follows:



We can check if our processor is working in a pipelined fashion by checking the icode values in different stages.

```
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ iverilog assembler_pipe.v
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ ./a.out
VCD info: dumpfile assembler.vcd opened for output.
CLK=0 PC=                   20 f_icode= 6 d_icode= x e_icode= x m_icode= x w_icode= x
CLK=1 PC=                   22 f_icode=12 d_icode= 6 e_icode= x m_icode= x w_icode= x
CLK=0 PC=                   22 f_icode=12 d_icode= 6 e_icode= x m_icode= x w_icode= x
CLK=1 PC=                   22 f_icode=12 d_icode=12 e_icode= 6 m_icode= x w_icode= x
CLK=0 PC=                   22 f_icode=12 d_icode=12 e_icode= 6 m_icode= x w_icode= x
CLK=1 PC=                   22 f_icode=12 d_icode=12 e_icode=12 m_icode= 6 w_icode= x
CLK=0 PC=                   22 f_icode=12 d_icode=12 e_icode=12 m_icode= 6 w_icode= x
CLK=1 PC=                   22 f_icode=12 d_icode=12 e_icode=12 m_icode=12 w_icode= 6
CLK=0 PC=                   22 f_icode=12 d_icode=12 e_icode=12 m_icode=12 w_icode= 6
CLK=1 PC=                   22 f_icode=12 d_icode=12 e_icode=12 m_icode=12 w_icode=12
CLK=0 PC=                   22 f_icode=12 d_icode=12 e_icode=12 m_icode=12 w_icode=12
```

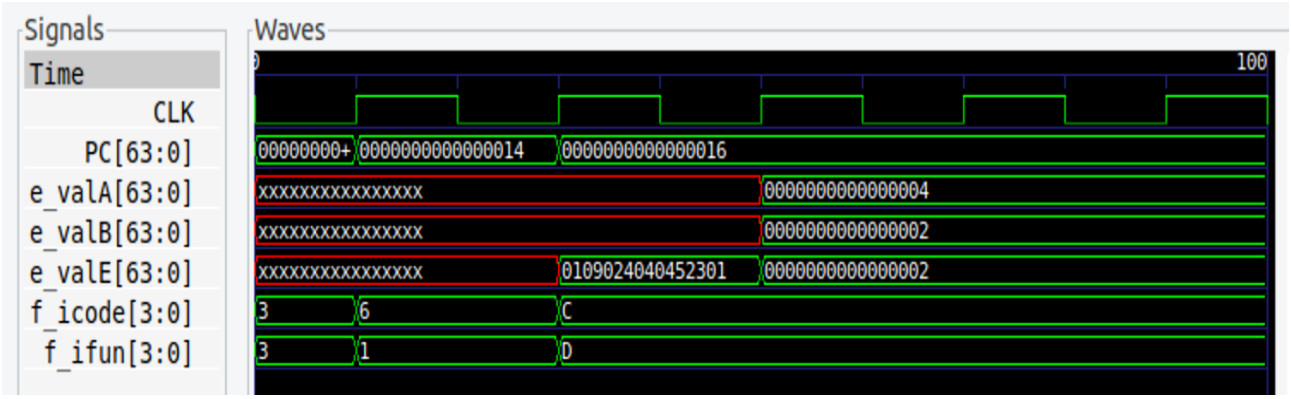
The gtkwave waveforms that we get for the above values are:



The output values PC , CLK, icode, ifun, valA, valB, valE obtained in the terminal for the instruction irmovq are as follows:

```
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ iverilog assembler_pipe.v
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ ./a.out
VCD info: dumpfile assembler.vcd opened for output.
PC=                10 clk=0 icode= 3 ifun= x valA=                x valB=                x valE=                x
PC=                20 clk=1 icode= 6 ifun= 3 valA=                x valB=                x valE=                x
PC=                20 clk=0 icode= 6 ifun= 3 valA=                x valB=                x valE=                x
PC=                22 clk=1 icode=12 ifun= 1 valA=                x valB=                x valE=   74593343807759105
PC=                22 clk=0 icode=12 ifun= 1 valA=                x valB=                x valE=   74593343807759105
PC=                22 clk=1 icode=12 ifun=13 valA=                4 valB=                2 valE=                2
PC=                22 clk=0 icode=12 ifun=13 valA=                4 valB=                2 valE=                2
PC=                22 clk=1 icode=12 ifun=13 valA=                4 valB=                2 valE=                2
PC=                22 clk=0 icode=12 ifun=13 valA=                4 valB=                2 valE=                2
PC=                22 clk=1 icode=12 ifun=13 valA=                4 valB=                2 valE=                2
PC=                22 clk=0 icode=12 ifun=13 valA=                4 valB=                2 valE=                2
```

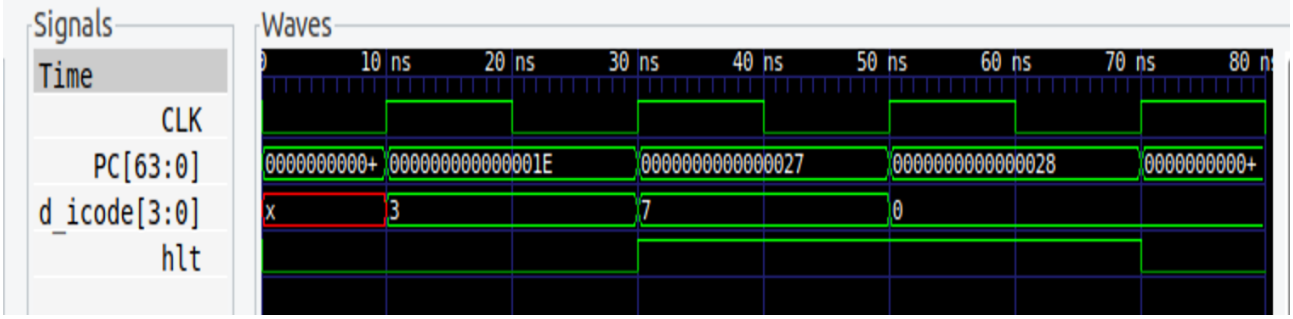
The gtkwave waveforms that we get for the above values are:



The output values obtained for the instructions irmovq, jmp and halt are as follows:

```
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ iverilog assembler_pipe.v
aishwarya@ubuntu:~/Downloads/Pipeline_V/pipelining$ ./a.out
VCD info: dumpfile assembler.vcd opened for output.
PC=                20 CLK=0 icode= 3 halt=0
PC=                30 CLK=1 icode= 7 halt=0
PC=                30 CLK=0 icode= 7 halt=0
PC=                39 CLK=1 icode= 0 halt=1
PC=                39 CLK=0 icode= 0 halt=1
PC=                40 CLK=1 icode= 0 halt=1
PC=                40 CLK=0 icode= 0 halt=1
PC=                41 CLK=1 icode= x halt=0
PC=                41 CLK=0 icode= x halt=0
```

The gtkwave forms that we get for the above output values are:



Instructions to Run the Code:

1. Download all the files
2. Navigate to sequential or pipeline
3. Run the commands :
   - **SEQUENTIAL**
     - iverilog assembler.v
     - vvp a.out
   - **PIPELINE**
     - iverilog assembler_pipe.v
     - vvp a.out

Challenges encountered:

- It was difficult to come up with the logic behind design but with in-depth knowledge of Verilog coding and processor design, we were able to design a sequential processor as well as a pipelined processor.
- It was difficult to synchronise the modules with clock, but with proper understanding of the subject, we could overcome this issue.
- Implementing the functions with clock correctly was really challenging.

References :

Computer_Systems_A_Programmers_Perspective_3rd_Ed_Randal_E._Bryant_David_R._OHallaron