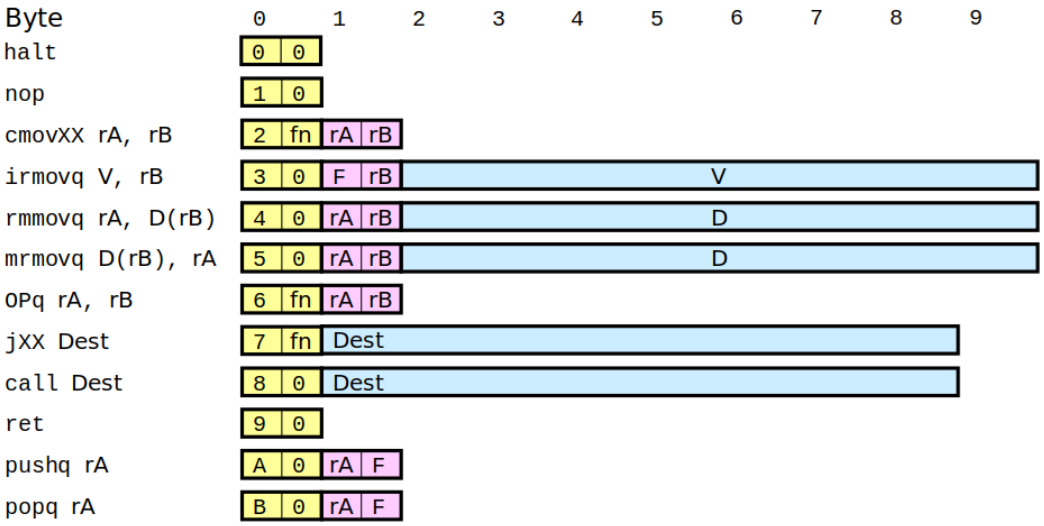
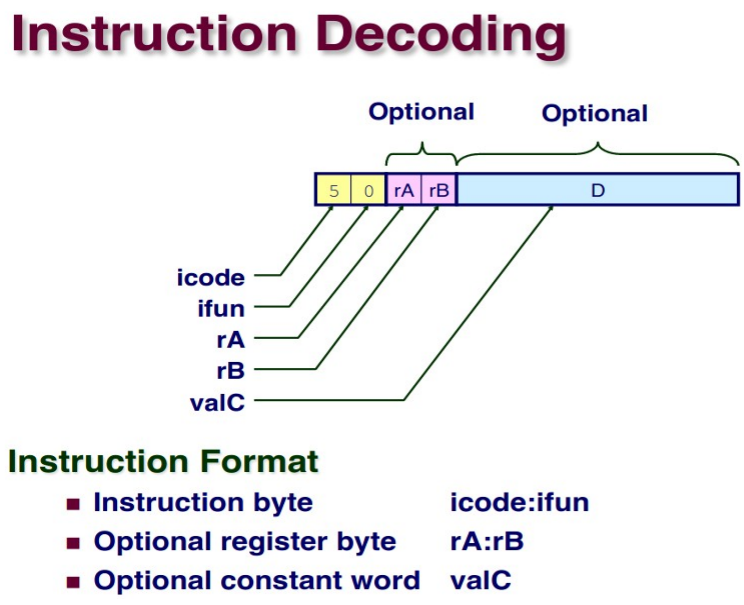
Intro to Processor Architecture

Y86 Processor Sequential Design Report

In-order to explain what type instructions are executed by the different modules of Y86-64 processor. Let's look at the instruction set of Y86-64 processor.

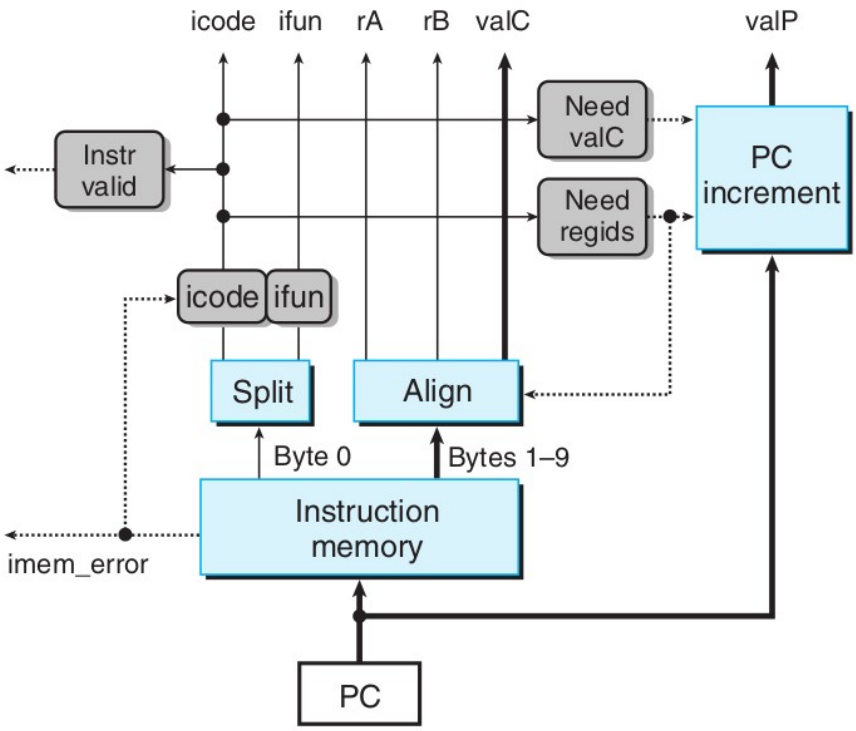


Let's see how each instruction is encoded.



---

1. FETCH STAGE



Architecture of Fetch

Now coming to fetch stage. I created a “input.txt file”, where the instructions are given called instruction bytes. The instruction is read through $readmemh function in fetch stage, the instruction length may vary depending on icode. The maximum length of an instruction is 10Bytes. Such instructions of 10Bytes can be given as input in straight as the Y86 supports a 16-bit address bus with a maximum of 65,536 bytes of addressable memory until it reaches out of limit.

If icode is 3 or 4 or 5 or 7 or 8, then the length of instruction is given by 10Bytes of information. In other cases of icode instruction length is 2Bytes except for ret or nop or halt instructions. Now let me explain the architecture of fetch.

The fetch stage reads the bytes of an instruction from memory, using the program counter (PC) as the memory address. Initially I set pc=0. From the instruction, initially split and align process happen where it extracts the two 4-bit portions of the instruction specifier byte (the first byte of instruction), referred to as icode (the instruction code) and ifun (the instruction function). It possibly fetches a register specifier byte, giving one or both of the 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. Now, that we have got the icode and ifun from the 1st byte of instruction, from this we assign the instruction to rest.

So, Now to assign our instruction into rA, rB, valC and valP. Firstly, I will check if instruction is valid (instr_valid), i.e. if icode>11; then the instruction is not valid and processor stops executing that instruction and reach halt stage. And then check for memory error(imem_error), if instruction went out of memory through pc. i.e. if pc>65536 bytes. Then memory error occurs and again Y86-64 processor stops as it reaches halt stage. Now if the instruction is valid and there is no memory error. Then the instruction goes in search of the icode i.e. first four bits of the first byte of instruction. I have used case option available in verilog and searched for the icode of the given instruction.

We know that we have to assign valP=pc+length of the instruction. So if the icode is 3 or 4 or 5. I increased valP=pc+10,and accordingly I increased for rest cases of icode depending on the length.

Example instructions in txt file :

1st instruction: 60 42 (rest bytes not necessary as icode=6)
2nd instruction: 00 (halt case)

I have taken a 2D array (reg [7:0] memad[20:0])
to store multiple instructions. Where in each row 2bytes of information exist. I can increase column length to store more instructions. But for general purpose I set it's size to take only 2 instructions.

Now the instructions in txt file looks as:

```
0110 0000 //1st  byte
0100  0010 // 2nd byte and 1st instruction ends here
00            // 2nd instruction 1st byte:
```

Now that I know icode = memad[pc][7:4]; (i.e icode=6)
ifun = memad[pc][3:0]; (i.e ifun=0)

Now when icode reaches case 6, the following happens valP=pc+2(because of instruction set) and

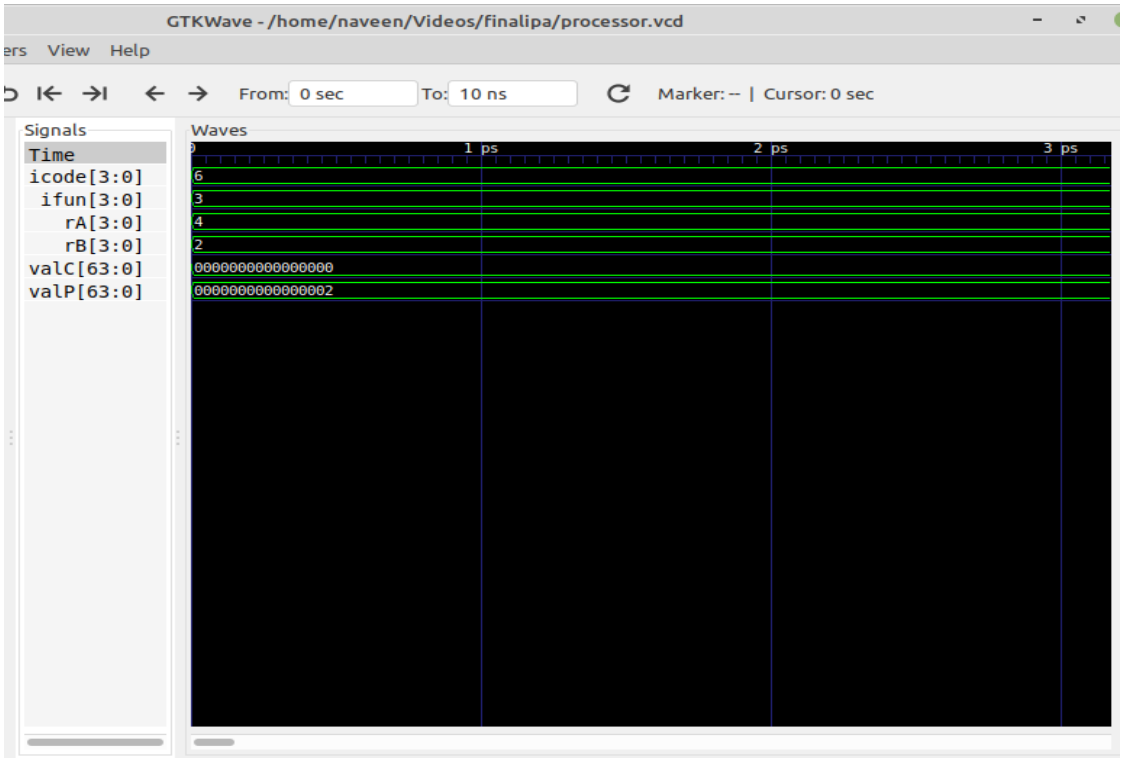
rA=memad[pc+1][7:4]; (i.e rA=4)
rB=memad[pc+1][3:0]; (i.e rB=2)

In this case there is no valC, i.e. the 8byte constant word. If I would have taken another instruction where constant exist, then valC takes it's value(clearly written in code).

After this the Y86-64 immediately tries to execute next instruction, as next instruction is halt. It stops. Let's verify the fetch on seeing through GTKwave.

```
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$ iverilog -o temp 1a.fetch_tb.v fetch.v
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$ vvp temp
WARNING: fetch.v:19: $readmemh: Standard inconsistency, following 1364-2005.
WARNING: fetch.v:19: $readmemh(input.txt): Not enough words in the file for the requested range [0:20].
VCD info: dumpfile fetch_tb.vcd opened for output.
icode= 6 ifun= 3 rA= 4 rB= 2 valc=                   0 valp=                   2
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$
```

All expected outputs have come, let's see on GTKwave:



Fetch stage outputs on GTKwave

So, the all the outputs have come correctly, So, we are done with FETCH stage.

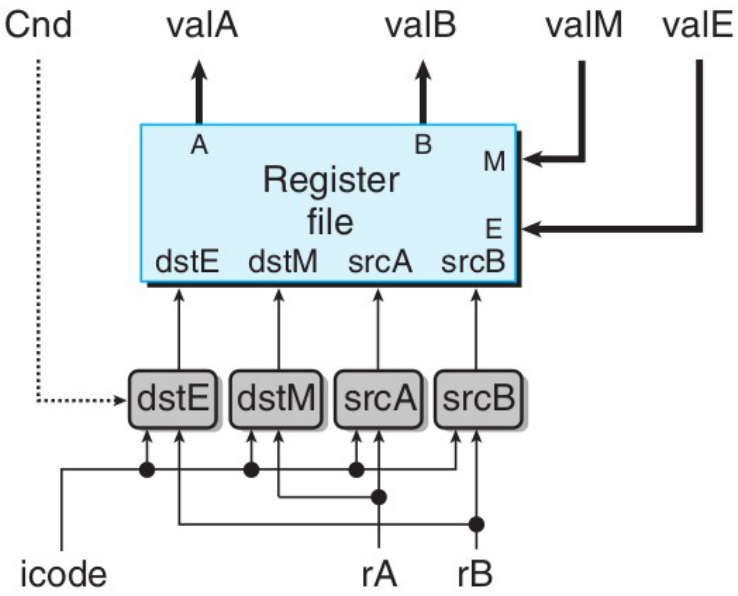
*__Instructions to run my code__ : I will upload the fetch.v, fetch_tb.v and input.txt in a folder named fetch. It is enough to run my codes on terminal placing these three in same folder.

__*Instructions that Fetch supports :__ Fetch supports all instructions of the instruction set. This is basically nothing but it finds icode and ifun. With respect to icode it outputs rA, rB, valC,valP. For some instructions there may not be

valC as rA and rB don’t exist in when ret, nop and halt are used. But it’s not a poblem as valC goes to 0. Similarly valP can be 0 if halt is the first instruction as it doesn’t execute anything.

---

2. DECODE AND WRITEBACK STAGES



Architecture of Decode

Now coming to decode stage. The inputs at decode stage are icode, rA and rB. From which we get valA, valB. In the decode stage valM and valE are basically values that comes in when write back happens.

The decode stage reads up to two operands from the register file, giving values valA and/or valB. Typically, it reads the registers designated by instruction fields rA and rB, but for some instructions it reads register %rsp.

i.e valA=R[rA] and valB=R[rB]

Let me elaborate on this, basically rA and rB are registers. Initially, here in the decode stage, I created a register file and named it as regf. To create a register file, I took

reg [63:0] regf [15:0] (A 2D kind array)

8Bytes of information in a column and can go upto 15rows. where

I initialised the register file with some random decimal numbers. So conclusion is valA and valB are the values read from the registers rA and rB of the register file. In the register file, to be practical, I have taken all the decimal numbers in straight. i.e for register rA, valA is rA in decimal. Let me show that.

```
regf[0]=64'd0;
regf[1]=64'd1;
regf[2]=64'd2;
regf[3]=64'd3;
.    .    .
.    .    .
.    .    .
```

regf[15]=64'd15 . So, this would make situation better for us to understand or execute something. For any icode, valA=regf[rA] which is nothing but valA=rA(in decimal). If rA is 4, regf[4]=4, it continues this way. We can change this anytime we want. And it changes in write back condition.

And as decode executes all the instructions from the instruction set, there is a thing, that when icode is 10, then

valB=regf[4];

if icode is 11, then valB= regf[4]; these are specified instructions.

And if icode=1, both valA and valB are 0;

In the test bench of decode, I will given input for icode=6, rA=4, rB=2;
then as I have considered a pratically easy one, output valA and valB must equal rA and rB respectively.

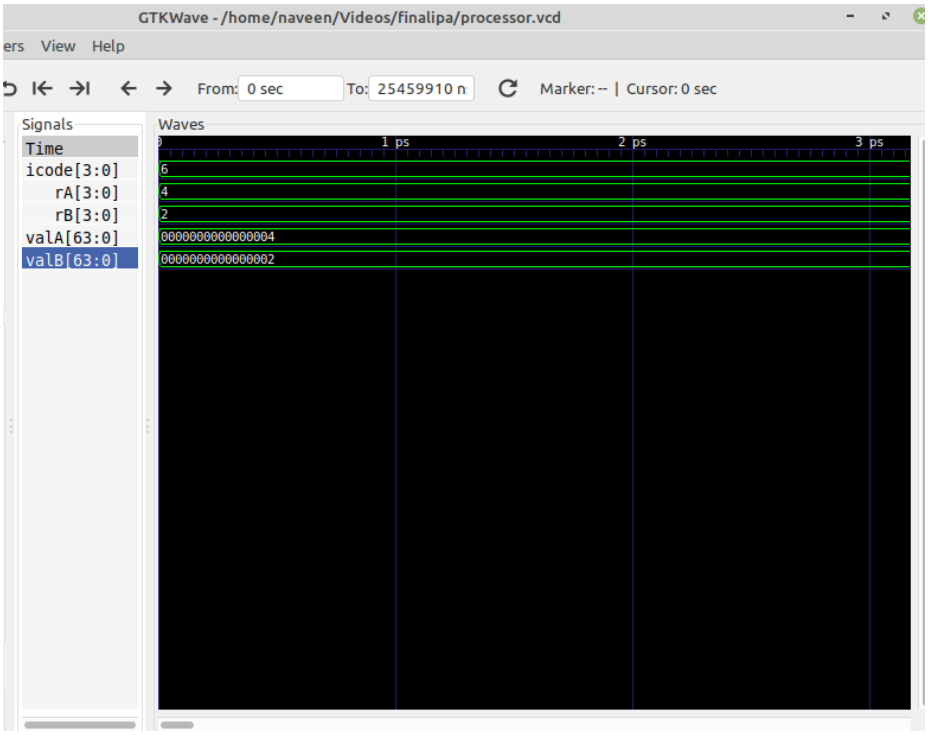
***Instructions that Decode supports :** Decode supports all instructions of the instruction set. Because we just want rA and rB of the particular instruction and nothing more than that. And then we will find the values of registers rA and rB from register file and use further if necessary. Special insructions are mentioned above. But the fact is decode performs all the instructions of the instruction set.

***Instructions to run my code** : I will upload the deocde.v and decode_tb.v in a folder named deocode. It is enough to run my codes on terminal placing these two codes in same folder.

Let’s verify the decode on seeing through GTKwave:

```
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/decode$ iverilog -o temp 2a.decode_tb.v decode.v
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/decode$ vvp temp
VCD info: dumpfile decode_tb.vcd opened for output.
icode= 6 rA= 4 rB= 2 valA=                   4 valB=                   2
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/decode$
```

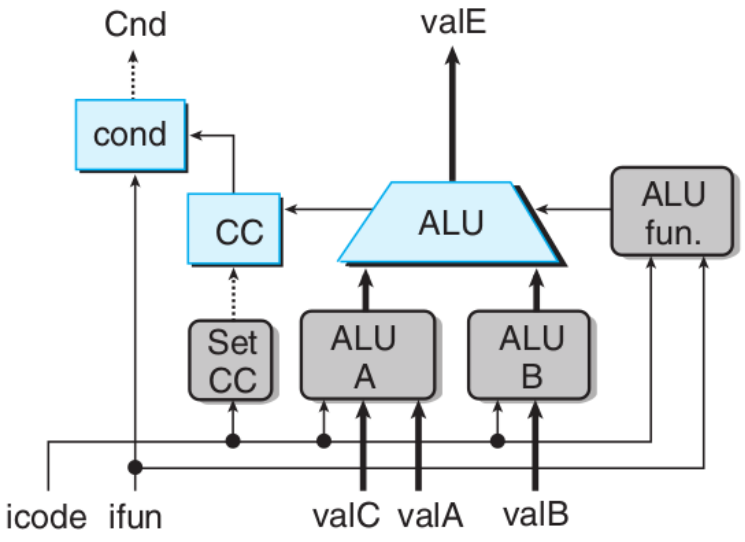
We can clearly see that valA and valB are equal to rA and rB. So, we got correct output.



So, the all the outputs have come correctly, So, we are done with DECODE stage.

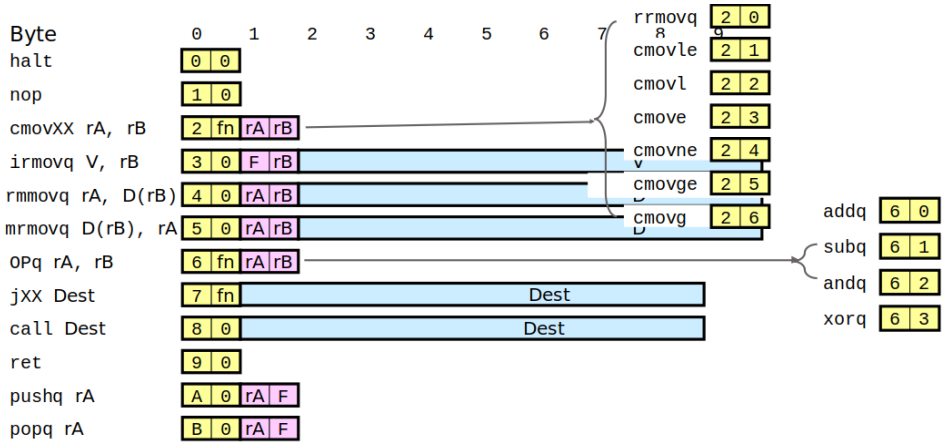
---

3. EXECUTE STAGE



Architecture of Execute

Now, 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. 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 or not the branch should be taken. Let's see the instruction set for this execute stage.



In execute logic, ALU implements 4 required function and generate 4 required functions. These functions specify zeroflag,sign flag and over flag. I took these functions as 0 initially and change according to the instruction. It also helps in conditional jump/move flag that I represented as Cnd.
I represent condition codes as:

cc[0]=0 //zeroflag
cc[1]=0 //signflag
cc[2]=0 //overflowflag, initially all are 0.

when result(that we get from computing values in rA and rB)=0.
result=0, ZF(zeroflag=1).
otherwise, ZF=0.

When result<0, SF(signflag=1). We use this to check if the number is negative.

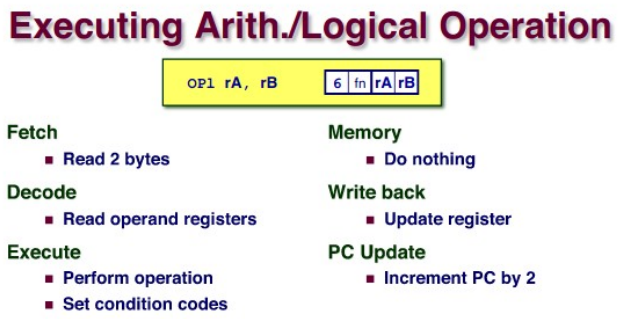
- If icode=2, then ifun is required to compute which instruction. i.e.

when I say that value from rA and rB is nothing but the value of register rA and rB in register file. if ifun=0, then just register to register move occurs, which we

call as unconditional move. And also as ifun changes, it computes if rA and rB are lessthan or equal(cmovle) and if just lessthan(cmovl). It also computes for if value in rA and rB are equal(cmove) or not equal(cmovne). And allow computes for greater than(cmovg) or equal(cmovgne). I would like to explain one instruction, lets consider icode=2 and ifun=1. Then the result is strictly less than or equal iff cc[0]==1 and cc[1]=1. Like this I computed for rest of the instructions as well. And at each instruction we check for cnd as well.

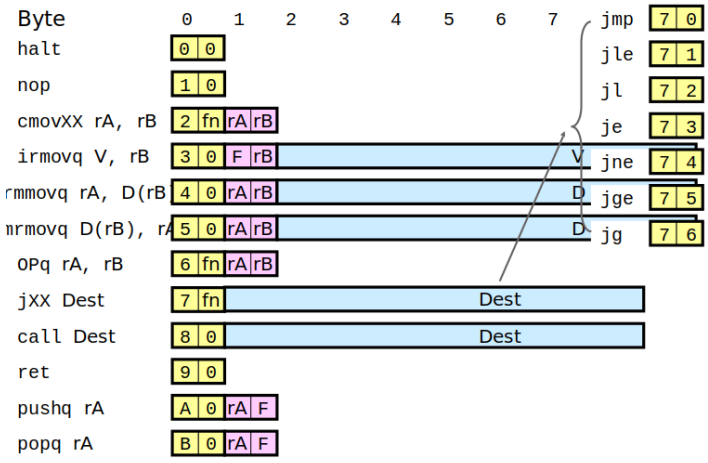
- If icode=4 or 5, then just Displacement occurs in the registers where changing of address happens.

- If icode=6, we call this instruction as arithmetic instruction. Here depending on ifun add/sub/and/or happens.



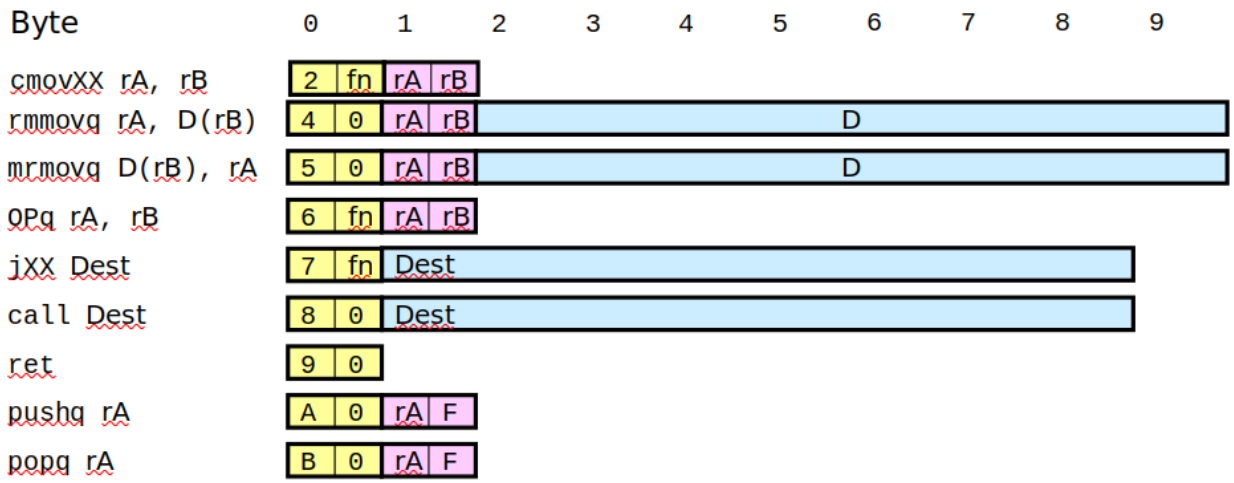
In this case, we will reset the condition codes. We will compute valE. If the result is positive when we add the values in registers rA an rB. We set the conditions if addition of two numbers results in negative. Which should never be. I have checked the valE, valA(value in register rA) and valB(value in register rB) through if conditions and made necessary condition codes.

- If icode=7, then jump instructions occur. These jump occurs according to the change in ifun.



- For icode 8, call instruction happens. For icode 9, ret occurs.
- For icode 10 and 11. push and pop occurs. When pop occurs.

***Instructions that Execute supports:**



Instructions supported by Execute stage

```
int aluA = [
        icode in { IRRMOVQ, IOPQ } : valA;
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
        icode in { ICALL, IPUSHQ } : -8;
        icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
```

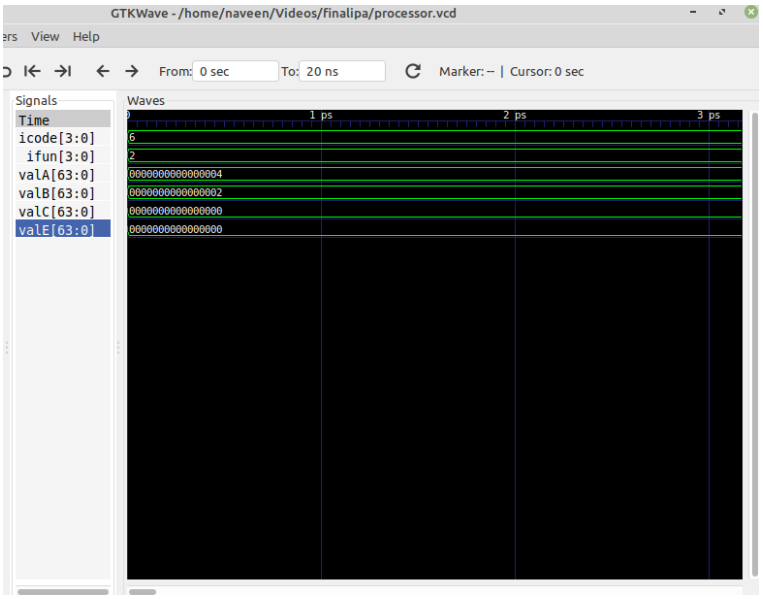
So, execute performs the above instructions. Therefore the input instruction comes into execute stage if it's icode is 2 or 5 or 6 or 7 or 8 or 9 or 10 or 11.

***Instructions to run my code** : I will upload the execute.v and execute_tb.v in a folder named execute. It is enough to run my codes on terminal placing these two codes in same folder.

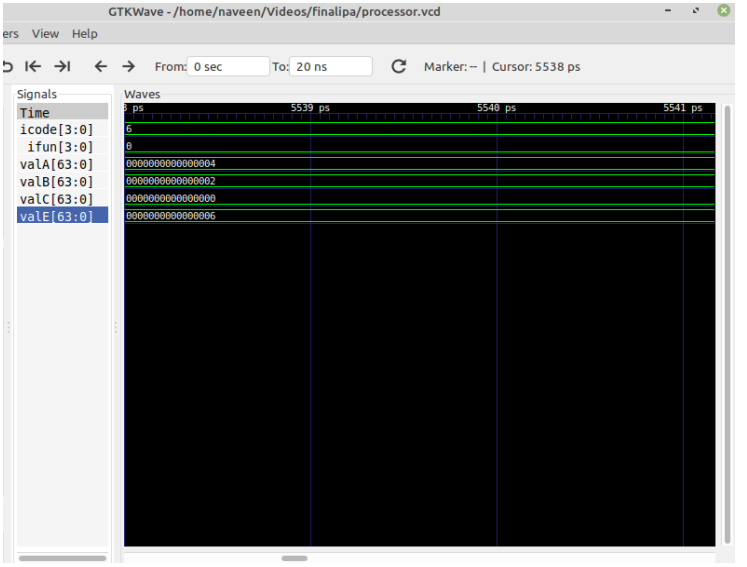
Example : when icode=6, ifun=2. In this instruction the execute stage does computes &(and operation) value in register rA and rB. Where I considered rA=4, rB=2. Therefore valE=0. Similarly when I changed ifun, it added and the valE is 6.

```
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$ iverilog -o temp 3a.execute_tb.v execute.v
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$ vvp temp
VCD info: dumpfile execute_tb.vcd opened for output.
valE=                   0
icode= 6 ifun= 2 valC=                   0 valA=                   4 valB=                   2 valE=                   0 cnd=x
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$ iverilog -o temp 3a.execute_tb.v execute.v
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$ vvp temp
VCD info: dumpfile execute_tb.vcd opened for output.
valE=                   6
icode= 6 ifun= 0 valC=                   0 valA=                   4 valB=                   2 valE=                   6 cnd=x
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$
```

Let’s verify on GTKwave:



Output when ifun=2(does &)

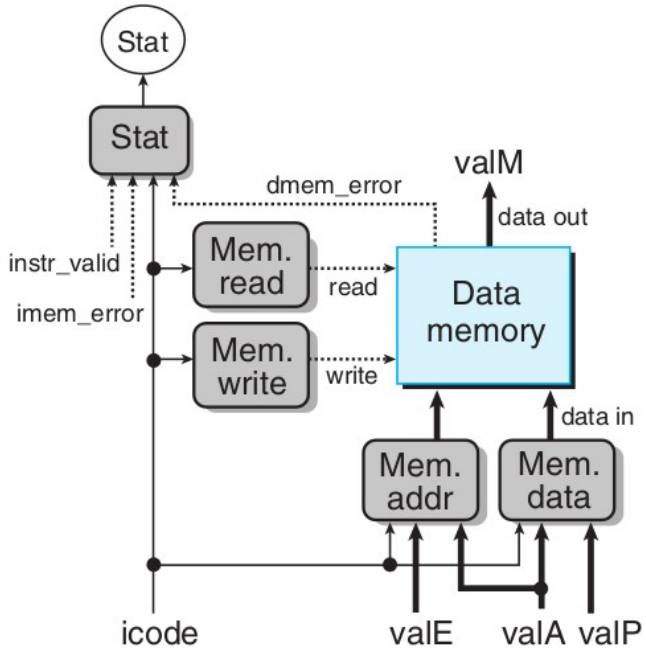


Output when ifun=0(adds)

All the outputs have come as expected. The valE changed from 0 to 6 i.e after first instruction. So, we are done with EXECUTE stage.

---

4. MEMORY STAGE



Architecture of Memory

Coming to memory stage. The memory stage may write data to memory, or it may read data from memory. We refer to the value read as valM. Here in memory stage we take input from icode, valE(from execute), valA(from decode) and valP(from fetch). And assign these values to memory address and memory data accordingly:

mem_addr =

icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE; //icode is in 3 or 10 or 8 or 5. Then mem_add=valE.

icode in { IPOPQ, IRET } : valA; // icode= 9 or 11. mem_add=valE.

mem_data =

icode in { IRMMOVQ, IPUSHQ } : valA;

icode == ICALL : valP;

Similarly we will find mem_read and mem_write to finally assign a value to valM.

So,

bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET }; //icode=1, if it exists in any of these instructions

bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL }; //icode=1, if it exists in any of these instructions

Now I have mem_addr, mem_data,mem_read and mem_write. Now I will assign according to if mem_write is 1 or mem_data is 1.

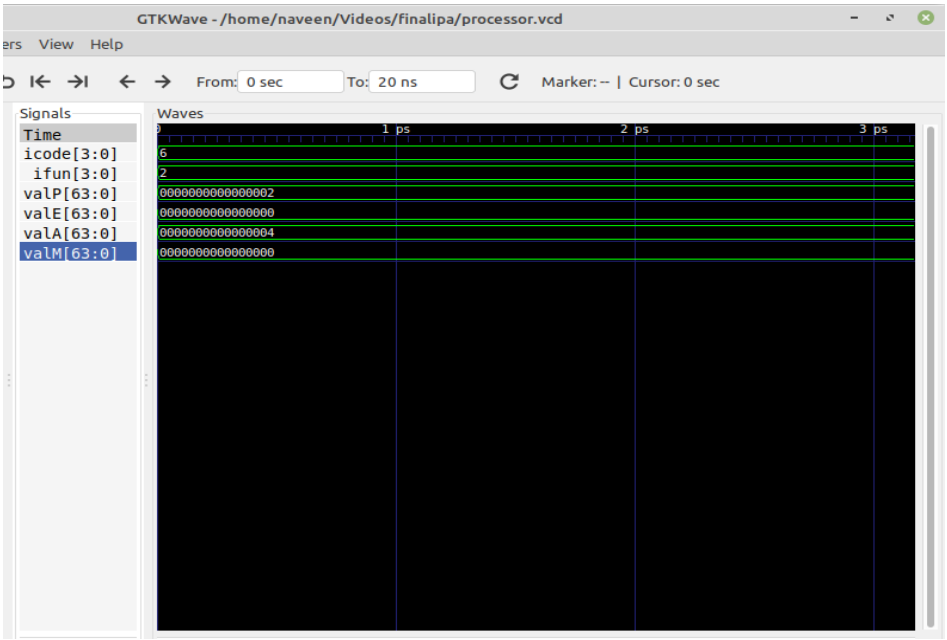
So,

if mem_read=1, then valM=mem_addr.
If mem_write=1, then mem_addr = mem_data.

And also in this stage I declared the memory as 2D kind array like
reg [63:0]M[0:1023].
So, Basically we dmem_error, when the size of memory is greater than 1023.

```
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$ vvp temp
VCD info: dumpfile memory_tb.vcd opened for output.
icode= 6 valA=                   4 valE=                   8 valM=                   z3
naveen@naveen-VivoBook-15-ASUS-Laptop-X507UF:~/Videos/finalipa$
```

Let’s verify through GTKwave:



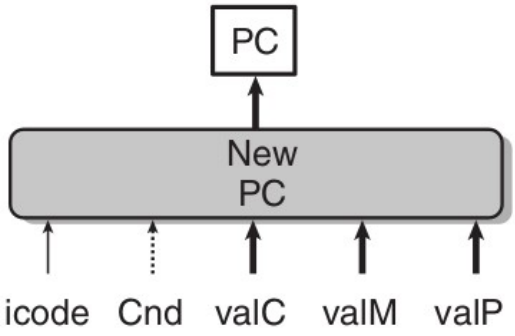
As we didn’t work with 10byte instruction. I gave icode to be 6. therefore we can clearly see the expected output valM that is 0. So, we are done with MEMORY stage.

Instruction that memory stage: supports RMMOVQ, PUSHQ, CALL, MRMOVQ, POPQ, IRET.

*__Instructions to run my code__ : I will upload the memory.v and memory_tb.v in a folder named deocode. It is enough to run my codes on terminal placing these two codes in same folder.

---

5. PC UPDATE

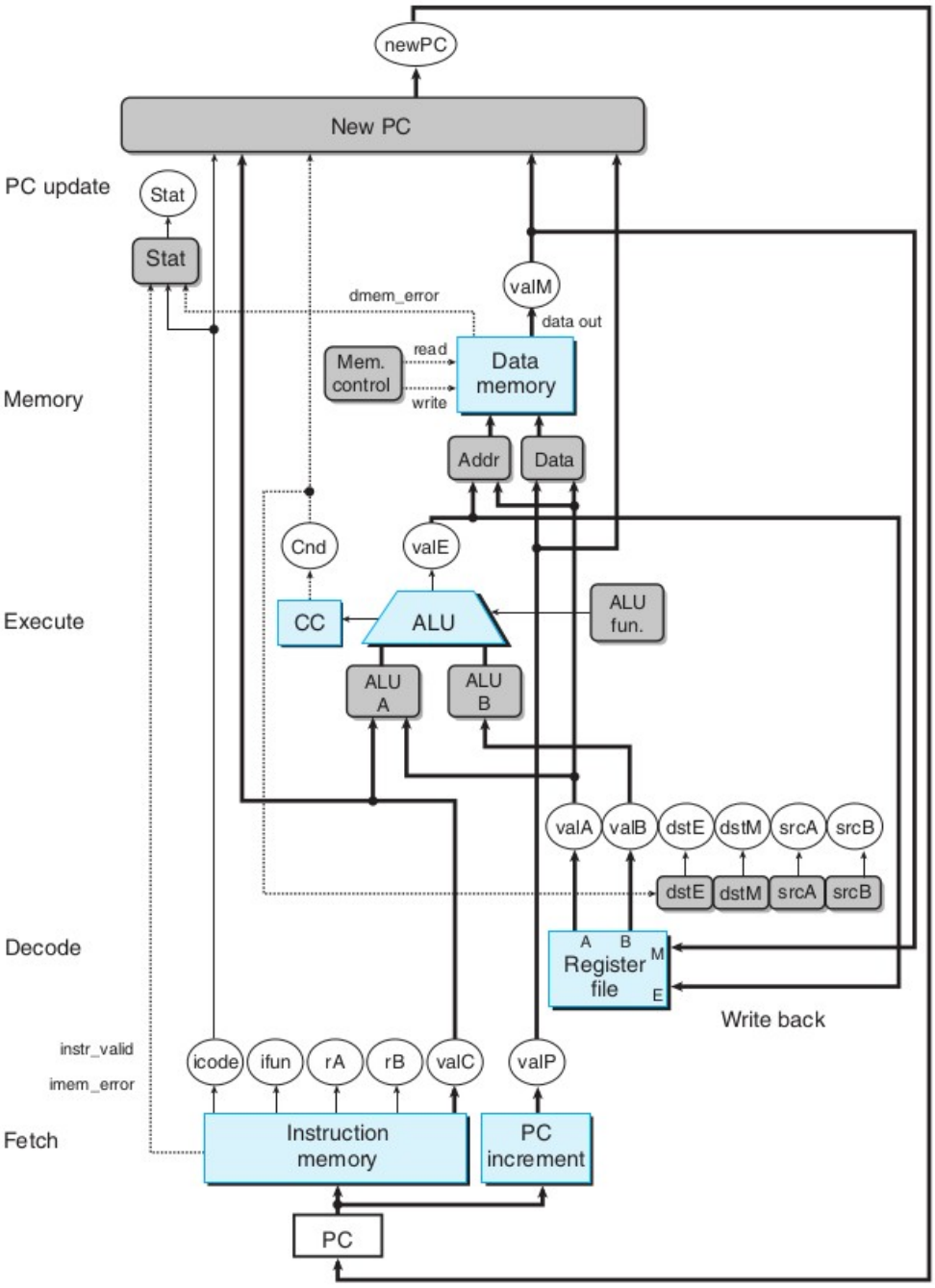


Architecture of PC_Update

Now coming to pc_update. The PC is set to the address of the next instruction. Initially we consider pc to be zero. In the pc update, we input clk as well. So, that the processor executes at the positive edge of clk everytime when the instruction changes. An instruction is identified by it's icode and then according to the icode, the instruction is performed. I used case icode to update pc after every instruction.

---

6. SEQUENTIAL Y86 PROCESSOR



Architecute of Sequential Y86 Processor

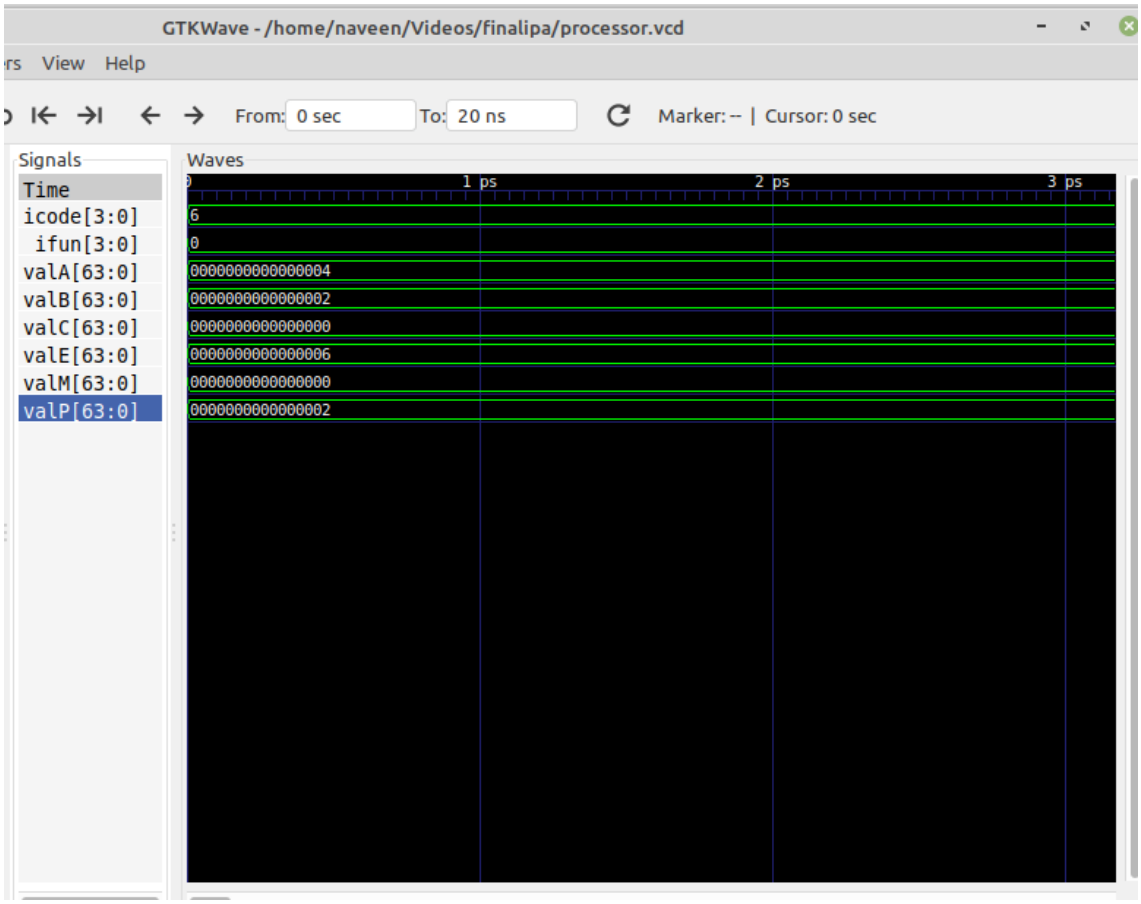
Here in the final stage, I implemented the whole processor by calling the rest of the modules. That is I included all the 5 modules through include function. And here I checked the processor working condition by giving 4 instructions at same time.

Instructions in txt file are as follows:

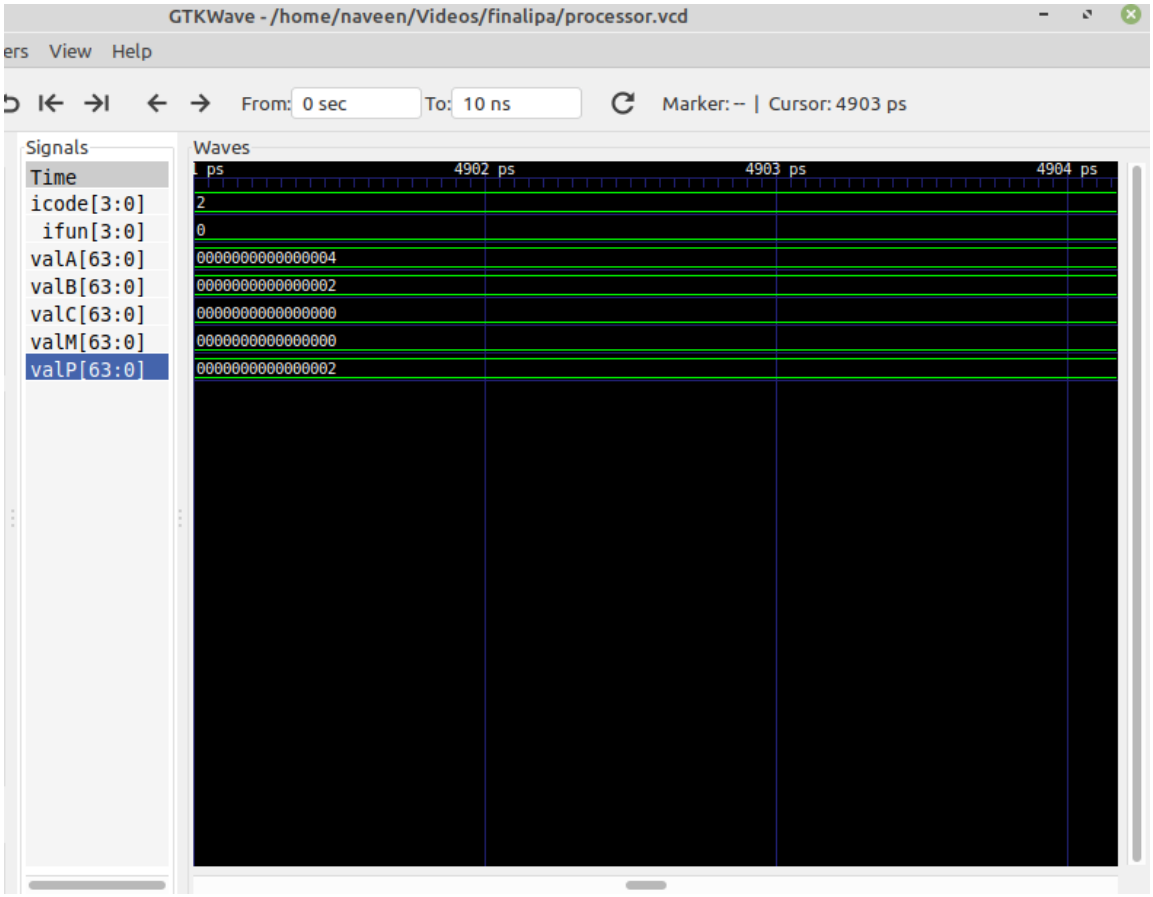
60 42
20 42
4042cdab896745230100
00

Now, let’s run the processor with the above inputs.

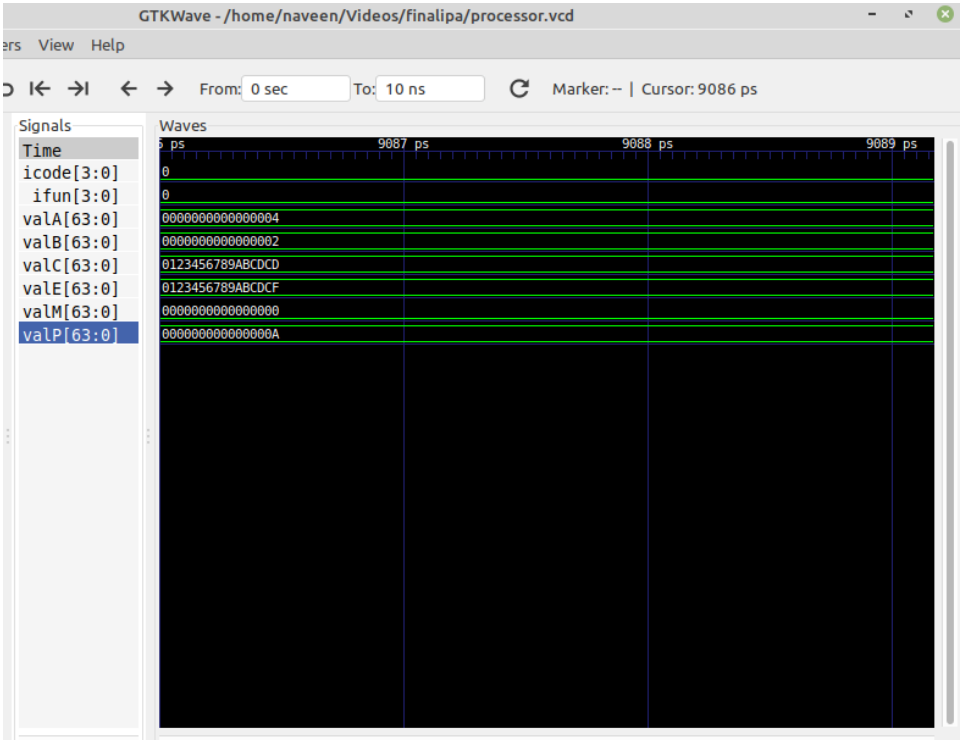
1st instruction output



2nd instruction output



3rd instruction output



And final instruction is halt. So the processor stops and donot return anything.

So, all the outputs on GTKwave are expected and are correct. So, we are done with PROCESSOR state. Next is implementing GCD.

```
#include <stdio.h>

// Recursive function to return gcd of a and b
int gcd(int a, int b)
{
    // Everything divides 0
    if (a == 0)
       return b;
    if (b == 0)
       return a;

    // base case
    if (a == b)
        return a;

    // a is greater
    if (a > b)
        return gcd(a-b, b);
    return gcd(a, b-a);
}


int main()
{
    int a = 98, b = 56;
    printf("GCD of %d and %d is %d ", a, b, gcd(a, b));
    return 0;
}

c code in Assembly.
```
