

# Introduction To Processor Architecture

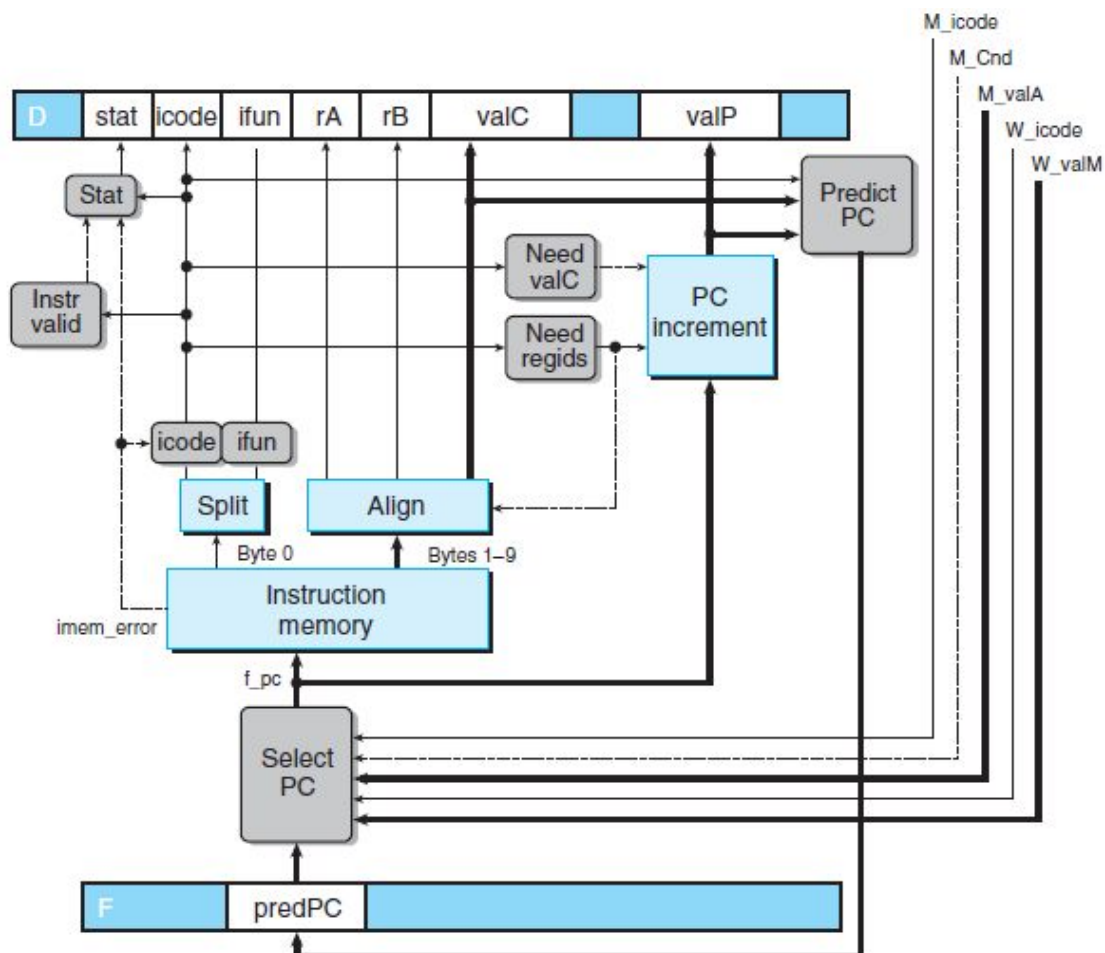
## Y86-64 Processor (5 Stage Pipeline)

Project


Chirag Shameek Sahu (2019102006)

### Architecture Diagram

#### PC Increment and Fetch



The fetch stage includes the instruction memory hardware unit. This unit reads 10 bytes from memory at a time, using the PC as the address of the first byte (byte 0). This byte is interpreted as the instruction byte and is split (by the unit labeled "Split") into two 4-bit quantities. The control logic blocks labeled "icode" and "ifun" then compute the instruction



and function codes as equaling either the values read from memory or, in the event that the instruction address is not valid (as indicated by the signal `imem_error`), the values corresponding to a nop instruction. Based on the value of `icode`, we can compute three 1-bit signals

- `instr_valid`. Does this byte correspond to a legal Y86-64 instruction? This signal is used to detect an illegal instruction.
- `need_regids`. Does this instruction include a register specifier byte?
- `need_valC`. Does this instruction include a constant word?

The signals `instr_valid` and `imem_error` (generated when the instruction address is out of bounds) are used to generate the status code in the memory stage.

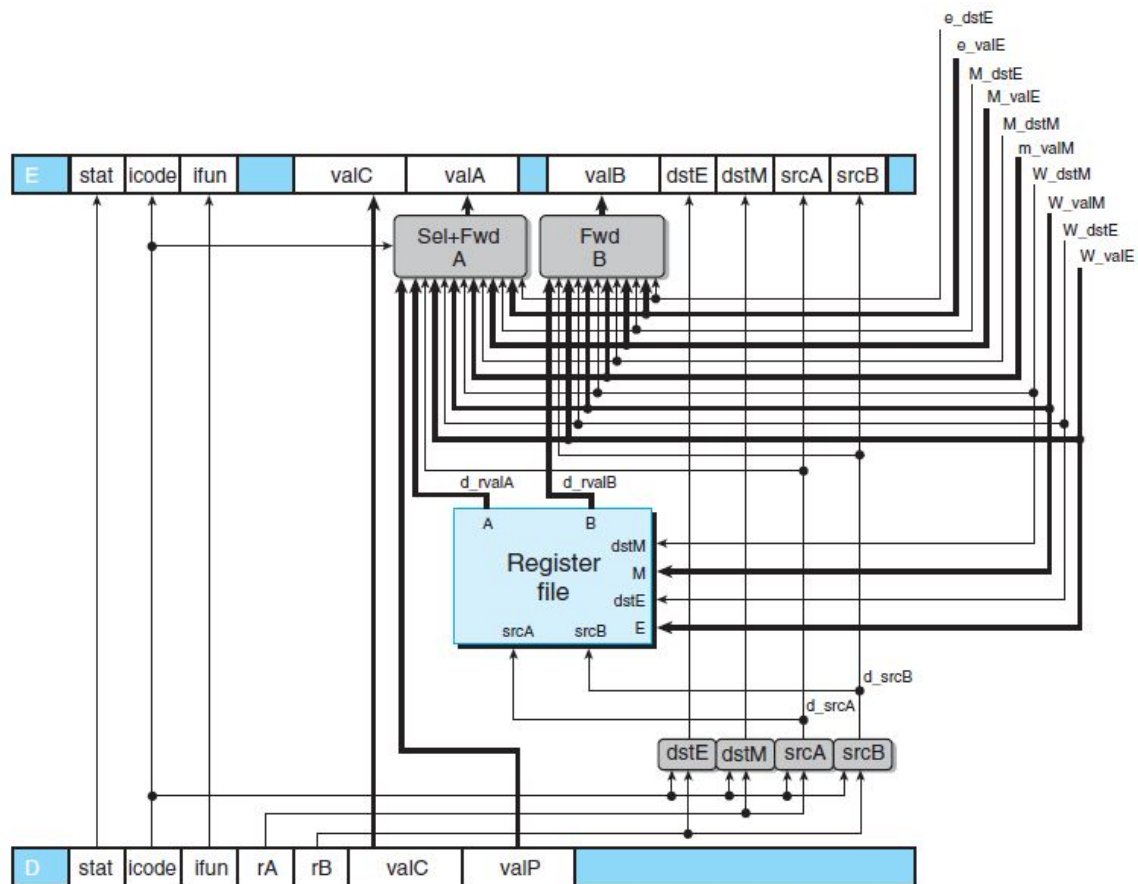
The remaining 9 bytes read from the instruction memory encode some combination of the register specifier byte and the constant word. These bytes are processed by the hardware unit labeled “Align” into the register fields and the constant word. Byte 1 is split into register specifiers `rA` and `rB` when the computed signal `need_regids` is 1. If `need_regids` is 0, both register specifiers are set to 0xF (RNONE), indicating there are no registers specified by this instruction. For any instruction having only one register operand, the other field of the register specifier byte will be 0xF (RNONE). Thus, we can assume that the signals `rA` and `rB` either encode registers we want to access or indicate that register access is not required. The unit labeled “Align” also generates the constant word `valC`. This will either be bytes 1–8 or bytes 2–9, depending on the value of signal `need_regids`.

The PC incrementer hardware unit generates the signal `valP`, based on the current value of the PC, and the two signals `need_regids` and `need_valC`. For PC value `p`, `need_regids` value `r`, and `need_valC` value `i`, the incrementer generates the value  $p+1+r+8i$ .

Unlike in SEQ, we must split the computation of the instruction status into two parts. In the fetch stage, we can test for a memory error due to an out-of-range instruction address, and we can detect an illegal instruction or a halt instruction.

Detecting an invalid data address must be deferred to the memory stage.

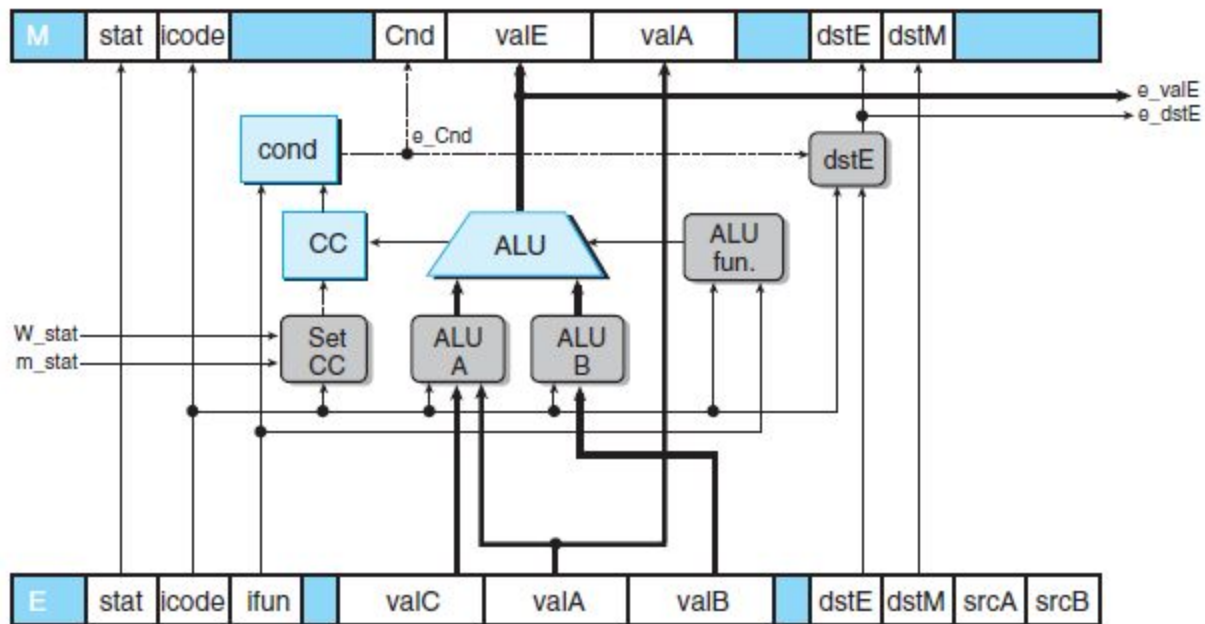
## Decode and Write Back



The register file has four ports. It supports up to two simultaneous reads (on ports A and B) and two simultaneous writes (on ports E and M). Each port has both an address connection and a data connection, where the address connection is a register ID, and the data connection is a set of 64 wires serving as either an output word (for a read port) or an input word (for a write port) of the register file. The two read ports have address inputs `srcA` and `srcB`, while the two write ports have address inputs `dstE` and `dstM`. The special identifier 0xF (RNONE) on an address port indicates that no register should be accessed.

The four blocks `dstE`, `dstM`, `srcA`, `srcB`, generate the four different register IDs for the register file, based on the instruction code `icode`, the register specifiers `rA` and `rB`, and possibly the condition signal `Cnd` computed in the execute stage. Register ID `srcA` indicates which register should be read to generate `valA`. Register ID `dstE` indicates the destination register for write port E, where the computed value `valE` is stored. But here, the register IDs supplied to the write ports come from the write-back stage (signals `W_dstE` and `W_dstM`), and not the one coming from the decode stage because we want the writes to occur to the destination registers specified by the instruction in the write-back stage.

## Execute



The execute stage includes the arithmetic/logic unit (ALU). This unit performs the operation add, subtract, and, or exclusive-or on inputs aluA and aluB based on the setting of the alufun signal. These data and control signals are generated by three control blocks. The ALU output becomes the signal valE. The ALU computation for each instruction is shown as the first step in the execute stage. The operands are listed with aluB first, followed by aluA to make sure the subq instruction subtracts valA from valB. The value of aluA can be valA, valC, or either -8 or +8, depending on the instruction type.

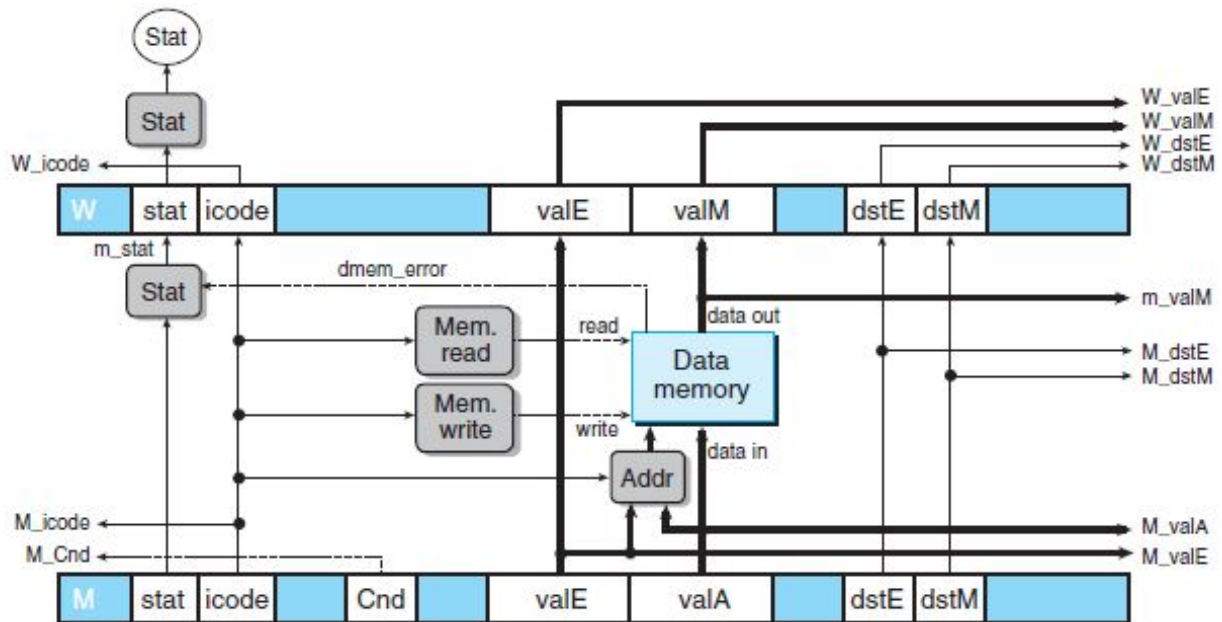
The execute stage also includes the condition code register. Our ALU generates the three signals on which the condition codes are based—zero, sign, and overflow—every time it operates. However, we only want to set the condition codes when an OPq instruction is executed. We therefore generate a signal set\_cc that controls whether or not the condition code register should be updated.

The hardware unit labeled “cond” uses a combination of the condition codes and the function code to determine whether a conditional branch or data transfer should take place. It generates the Cnd signal used both for the setting of dstE with conditional moves and in the next PC logic for conditional branches. For other instructions, the Cnd signal may be set to either 1 or 0, depending on the instruction’s function code and the setting of the condition codes.

The signals e\_valE and e\_dstE are directed toward the decode stage as one of the forwarding sources. One difference is that the logic labeled “Set CC,” which determines whether or not to update the condition codes, has signals m\_stat and W\_stat as inputs.

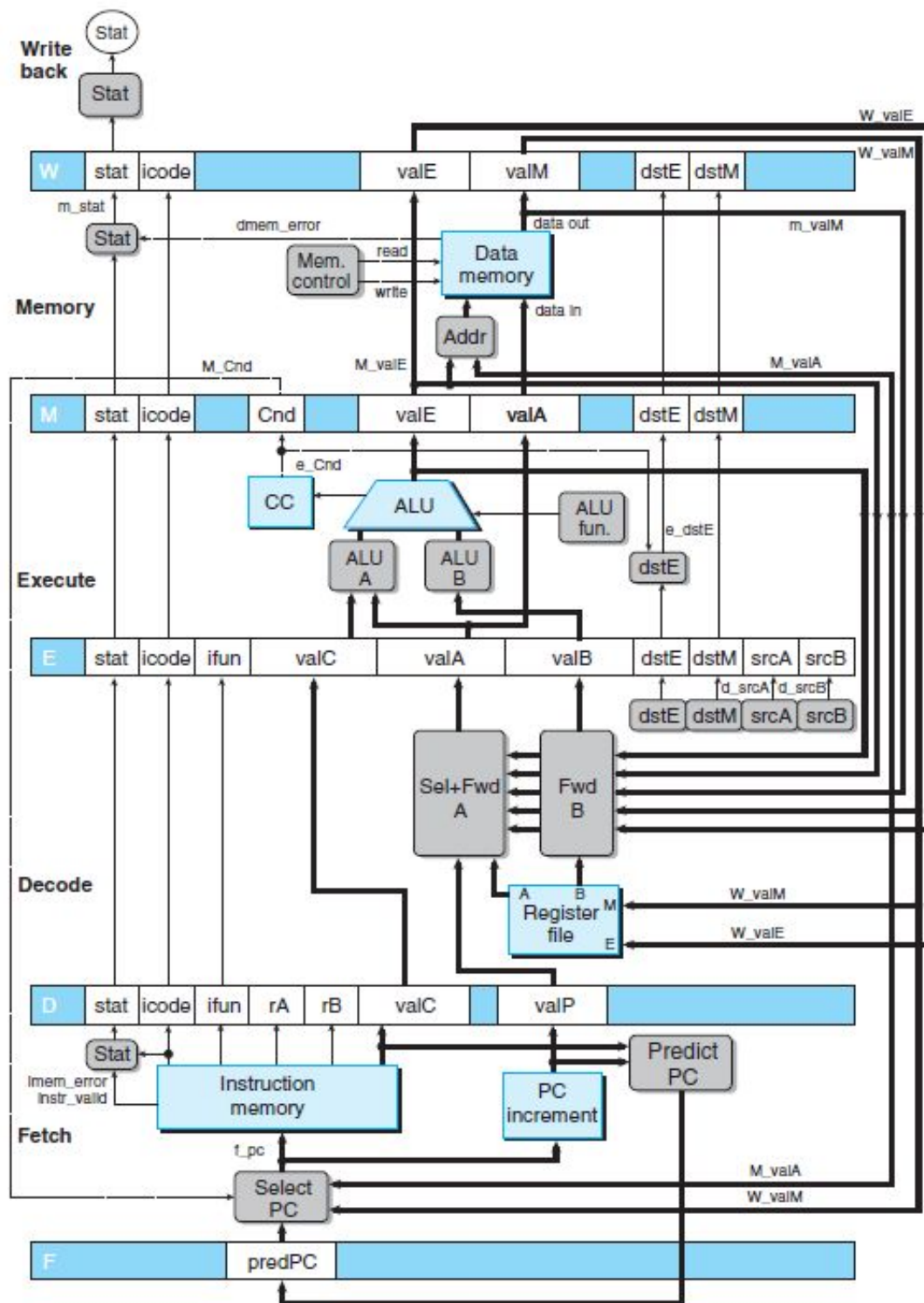
These signals are used to detect cases where an instruction causing an exception is passing through later pipeline stages, and therefore any updating of the condition codes should be suppressed.

## Memory



The memory stage has the task of either reading or writing program data. Two control blocks generate the values for the memory address and the memory input data (for write operations). Two other blocks generate the control signals indicating whether to perform a read or a write operation. When a read operation is performed, the data memory generates the value *valM*. We set the control signal *mem\_read* only for instructions that read data from memory. Many of the values in pipeline registers and M and W are supplied to other parts of the circuit as part of the forwarding and pipeline control logic.

## Overall Implementation of Y-86 Processor (5 Stage Pipeline)



## Supported Instructions

I have implemented the 5 stage pipelined version of Y86-64. This processor supports the following instructions:

Instruction	What does it do?
IHALT	Code for halt instruction
INOP	Code for nop instruction
IRRMOVQ	Code for rrmovq (register to memory) instruction
IIRMOVQ	Code for irmovq (immediate to register) instruction
IRMMOVQ	Code for rmmovq (register to memory) instruction
IMRMOVQ	Code for mrmovq (memory to register) instruction
IOPL	Code for integer operation instructions
IJXX	Code for jump instructions
ICALL	Code for call instruction
IRET	Code for ret instruction
IPUSHQ	Code for pushq instruction
IPOPQ	Code for popq instruction



Some of the above instructions also have sub-instructions that can be performed.

- IOPL

1. ADDQ - Performs addition of 2 numbers
2. SUBQ - Performs subtraction between 2 numbers
3. ANDQ - Performs LOGICAL AND of 2 numbers
4. XORQ - Performs LOGICAL XOR of 2 numbers

- IJXX

1. C\_YES - Jumps without condition
2. C\_LE - Jumps if it is less than or equal to
3. C\_L - Jumps if it is less than
4. C\_E - Jumps if it is equal to
5. C\_NE - Jumps if it is not equal to
6. C\_GE - Jumps if it is greater than or equal to
7. C\_G - Jumps if it is greater than



## C++ Code and Its Assembly Language Code

- C++ code for HCF: Code from Geeks For Geeks

```
#include<bits/stdc++.h>
using namespace std;

int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

int main()
{
    int a=56,b=24;
    cout<<gcd(56,24)<<endl;
}
```

- Assembly Language

```
rmovq 26, %rax
irmovq 38, %rbx

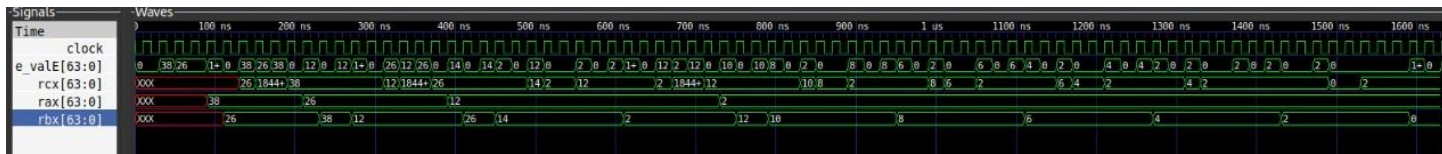
f1:
rrmovq %rbx, %rcx
subq %rax, %rcx
jg .swap
jl .repeatsub
halt

repeatsub:
subq %rax, %rbx
jmp .f1

swap:
rrmovq %rax, %rcx
rrmovq %rbx, %rax
rrmovq %rbx, %rbx
jmp .repeatsub
```

## Output (Using GTKWave)

The 2 numbers which were given as input are 26 to register %rax and 38 to register %rbx. In the end we see that the number 2 is in the register %rbx, which is the HCF.



## Processor Feature

The program runs properly on all instructions. It has been tested on running HCF. There has been no issue in running the codes on the processor.

## Instruction To Run

The files included in the submission are

- fetch.v - this consists of the fetch and PC incremter and Instruction Memory module.
- decode.v - this consists of the decode module. It also performs the corresponding write-back also.
- execute.v - this consists of the entire execute module (including ALU and CC).
- Y86.v - this is the overall implementation of the pipeline processor.
- Y86\_tb.v is the test bench of the Y86-64 processor.
- instructionencoded.mem - is the encoded instruction.

In order to run the code we have to download all the codes and store in the same folder.

To compile the code, write the following instruction on the terminal

```
iverilog -o Y86_tb Y86_tb.v Y86.v
```

Then to run the code, type

```
./a.out
```

To see the GTKwave output we type the instruction

```
gtkwave processor_tb.vcd
```