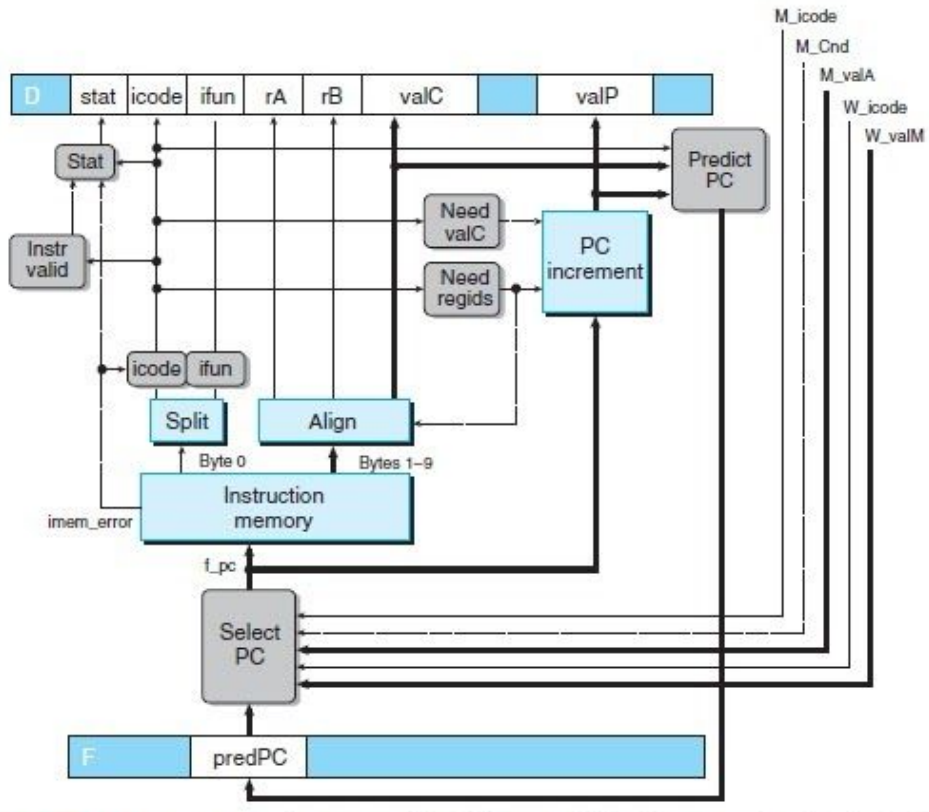
IPA Final Project

Y86 5-Stage Pipelined Processor (64-bit)

Theory for different modules -

1. FETCH STAGE -

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). The remaining 9 bytes read from the instruction memory encode some combination of the register specifier byte and the constant word.



Instruction Memory Block - The 0th byte is interpreted as the instruction byte and is split (by the **Split block**) 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 from Instr Valid block), the values corresponding to a nop instruction.

Input = 1 byte f_ibyte

Output = icode and ifun

The remaining 9 bytes are processed by the **Align block** 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. 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 Align block 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.

Input = 9 bytes f_ibytes and bool need regids

Output = f_rA, f_rB, f_valC

PC incremente Block - The PC incrementer block 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.

Input = f_pc , need regids and need valC

Output = ValP

Instr_valid - Checks if this byte corresponds to a legal Y86-64 instruction? This signal is used to detect an illegal instruction.

Need_regids - Checks if this instruction includes a register specifier byte?

-----> bool need_regids = icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ, IIRMOVQ, IRMMOVQ, IMRMOVQ };

Need_valC - Checks if this instruction includes a constant word?

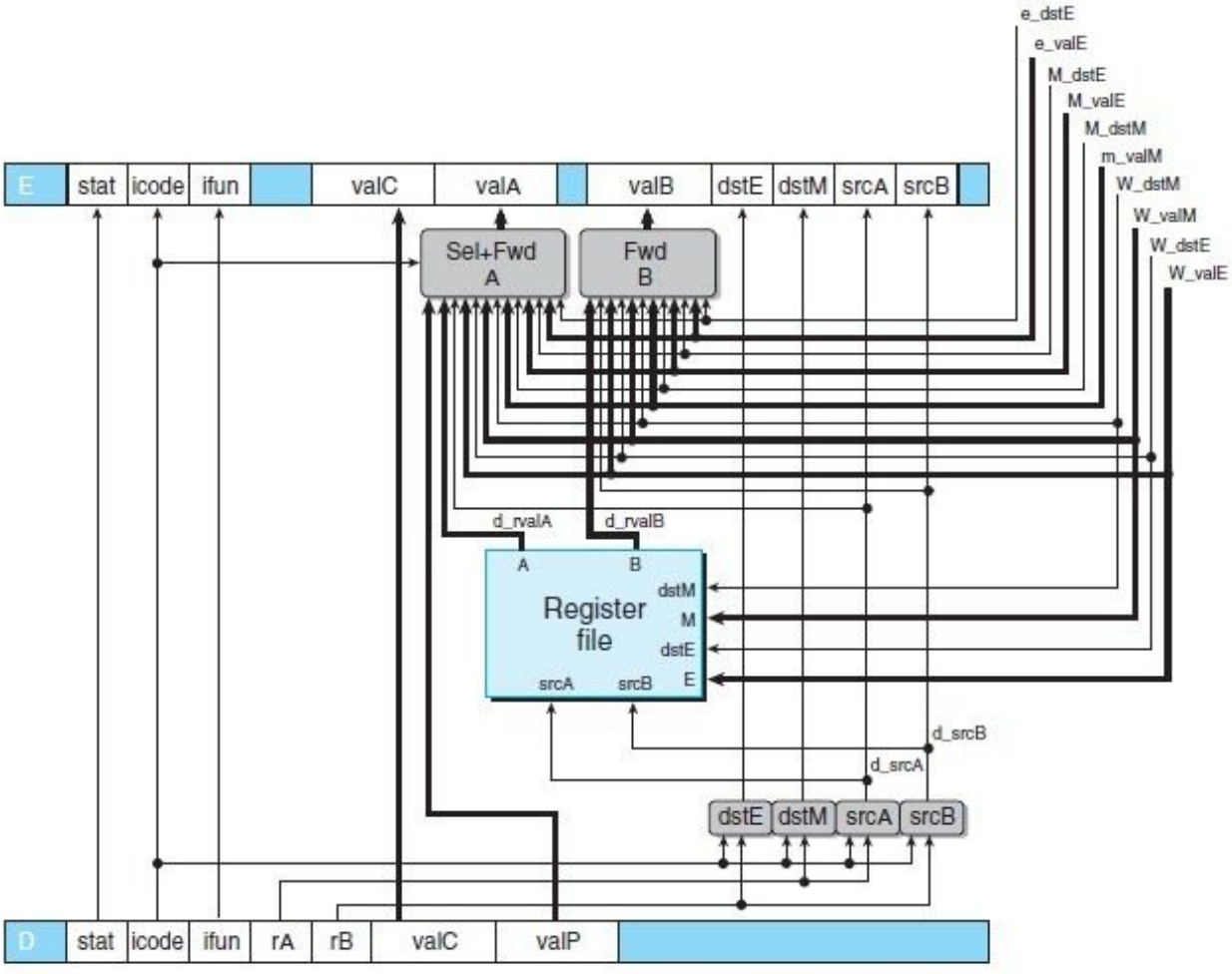
-----> bool need_valC = icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };

Select PC - The Select PC block chooses between three program counter sources. As a mispredicted branch enters the memory stage, the value of valP for this instruction (indicating the address of the following instruction) is read from pipeline register M (signal M_valA). When a ret instruction enters the write-back stage, the return address is read from pipeline register W (signal W_valM). All other cases use the predicted value of the PC, stored in pipeline register F (signal F_predPC).

Predict PC - The block can choose either valP (as computed by the PC incrementer) or valC (from the fetched instruction). This value is stored in pipeline register F as the predicted value of the program counter.

2. DECODE and WRITE-BACK STAGE -

There are four ports in the registry file. It can handle up to two reads (on ports A and B) and two writes at the same time (on ports E and M). Each port has an address connection and a data connection, with the address connection being a register ID and the data connection being a collection of 64 wires serving as either a read port's output word or a write port's input word.



Register File - The four blocks generate the four different register IDs for the register file, based on the instruction code icode, the register specifiersrA 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. The register signal srcB indicates which register should be read to generate the signal valB.

HCL code for srcA and srcB -

-----> word srcA = [icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ } : rA; icode in { IPOPQ, IRET } : RRSP; 1 : RNONE; # Don't need register];

-----> word srcB = [icode in { IOPQ, IRMMOVQ, IMRMOVQ } : rB; icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP; 1 : RNONE; # Don't need register];

Register ID dstE indicates the destination register for write port E, where the computed value valE is stored. Register ID dstM indicates the destination register for write port M, where valM, the value read from memory, is stored.

HCL code for dstE and dstM -

-----> word dstE = [icode in { IRRMOVQ } : rB; icode in { IIRMOVQ, IOPQ} : rB; icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP; 1 : RNONE; # Don't write any register];

-----> word dstM = [icode in { IMRMOVQ, IPOPQ } : rA; 1 : RNONE; # Don't write any register];

Input = dstE, valE, dstM, valM, srcA, srcB, reset, clock

Output = ValA, ValB, rax, rcx, rdx, rbx, rsp, rbp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14

* For pipeline we implement Decode and Writeback stages together

The blocks labeled dstE, dstM, srcA, and srcB are very similar to their counterparts in the implementation of SEQ. The register IDs supplied to the write ports come from the write-back stage (signals W_dstE and W_dstM), rather than from the decode stage. This is because we want the writes to occur to the destination registers specified by the instruction in the write-back stage.

The block added here is the "Sel+Fwd A" and "Fwd B"

Sel + Fwd A block - The block labeled “Sel+Fwd A” serves two roles. It merges the valP signal into the valA signal for later stages in order to reduce the amount of state in the pipeline register. It also implements the forwarding logic for source operand valA. The merging of signals valA and valP exploits the fact that only the call and jump instructions need the value of valP in later stages, and these instructions do not need the value read from the A port of the register file. This selection is controlled by the icode signal for this stage. When signal D_icode matches the instruction code for either call or jXX, this block should select D_valP as its output.

Fwd B block - The block labeled “Fwd B” implements the forwarding logic for source operand valB.

The register write locations are specified by the dstE and dstM signals from the write-back stage rather than from the decode stage, since it is writing the results of the instruction currently in the write-back stage.

HCL code for d_valA and d_valB-

-----> word d_valA = [D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC

d_srcA == e_dstE : e_valE; # Forward valE from execute

d_srcA == M_dstM : m_valM; # Forward valM from memory

d_srcA == M_dstE : M_valE; # Forward valE from memory

d_srcA == W_dstM : W_valM; # Forward valM from write back

d_srcA == W_dstE : W_valE; # Forward valE from write back

1 : d_rvalA; # Use value read from register file];

-----> word d_valB = [d_srcB == e_dstE : e_valE; # Forward valE from execute

d_srcB == M_dstM : m_valM; # Forward valM from memory

d_srcB == M_dstE : M_valE; # Forward valE from memory

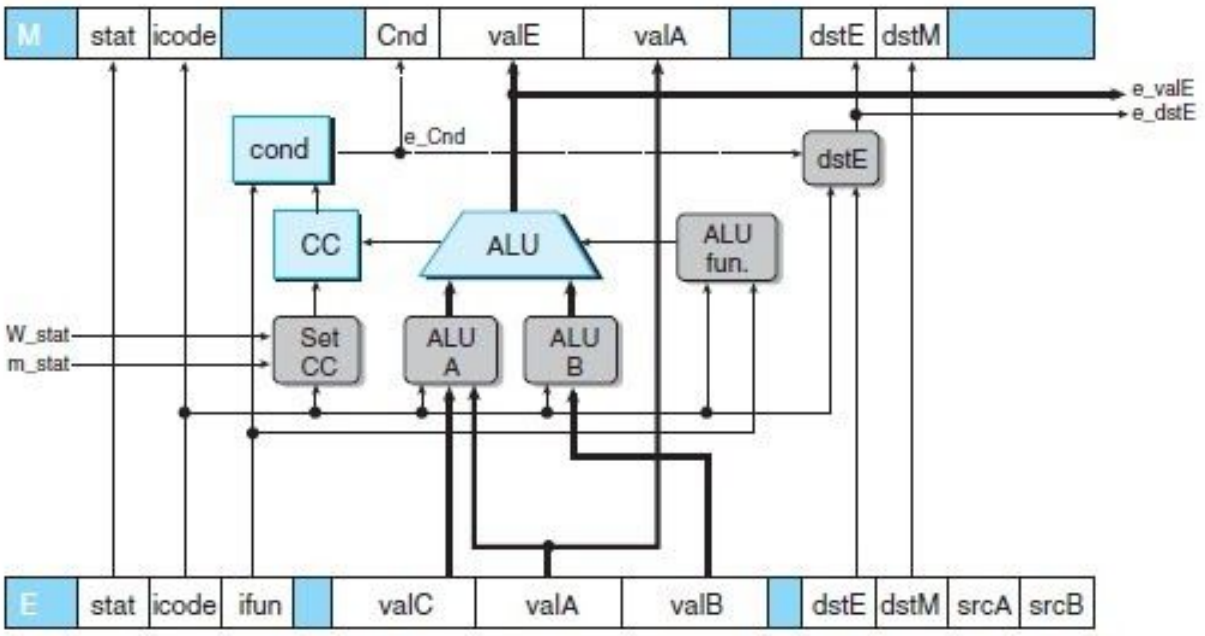
d_srcB == W_dstM : W_valM; # Forward valM from write back

d_srcB == W_dstE : W_valE; # Forward valE from write back

1 : d_rvalB; # Use value read from register file];

3. EXECUTE STAGE -

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.



ALU - The value of aluA can be valA, valC, or either −8 or +8, depending on the instruction type. We can therefore express the behavior of the control block that generates aluA.The ALU either performs the operation for an integer operation instruction or acts as an adder.

HCL code for aluA and aluB -

-----> word 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];

------> word aluB = [icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL, IPUSHQ, IRET, IPOPQ } : valB;

icode in { IRRMOVQ, IIRMOVQ } : 0;

Other instructions don't need ALU];

Input = aluA, aluB,alufun

Output = valE, new_cc

For more understanding of ALU refer
https://github.com/ProcessorArch2020/alu-assignment-pragya919/blob/main/IPA_assign_1.pdf

Condition Code block - Our ALU generates the three signals on which the condition codes are based—zero, sign, and overflow—every time it operates. We also have "**Set CC**," which determines whether or not to update the condition codes, has signals m_stat and W_stat as inputs.

-----> bool set_cc = E_icode == IOPQ && # State changes only during normal operation

!m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };

Input = new_cc, set_cc, reset,clock

Output = cc

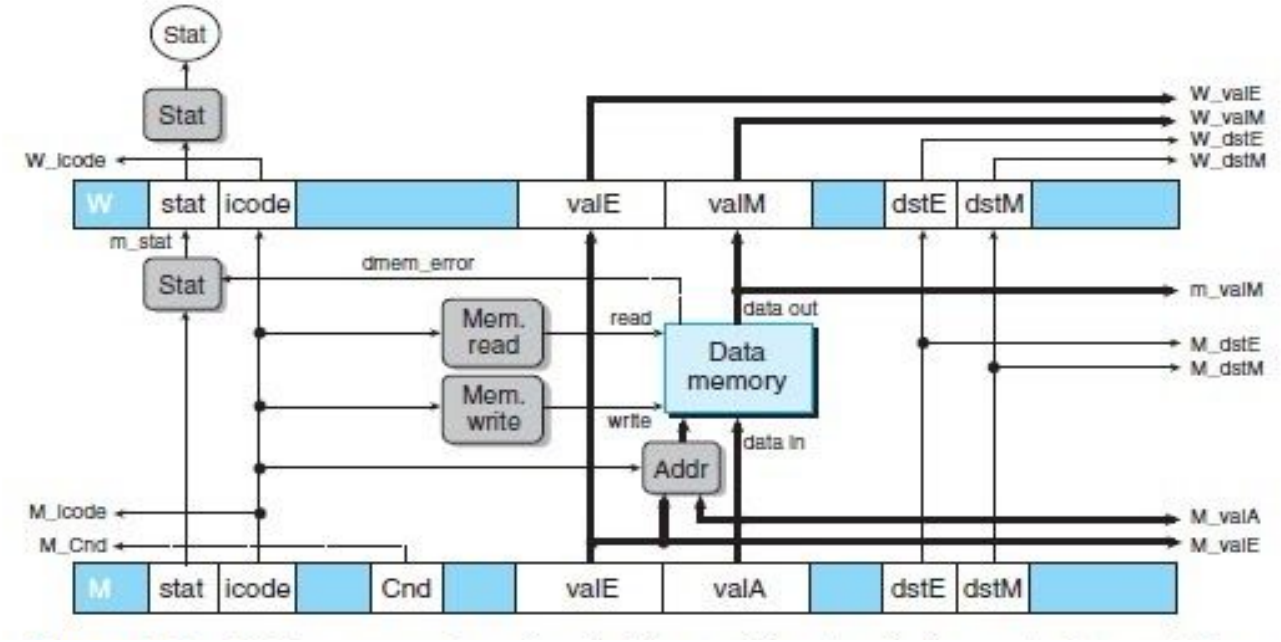
Cond block - . It generates the Cnd signal used both for the setting of dstE with conditional moves and in the next PC logic for conditional branches.

Input = ifun, cc

Output = Cnd

4. MEMORY STAGE -

The memory stage has the task of either reading or writing program data.



Data Memory - 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 want to set the control signal mem_read only for instructions that read data from memory.

Input = mem_addr, M_valA, mem_read, mem_write

Output = mem_data, dmem_error

HCL code for mem_data and mem_write

-----> word mem_data = [# Value from register icode in { IRMMOVQ, IPUSHQ } : valA; # Return PC

icode == ICALL : valP; # Default: Don't write anything];

------> bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };

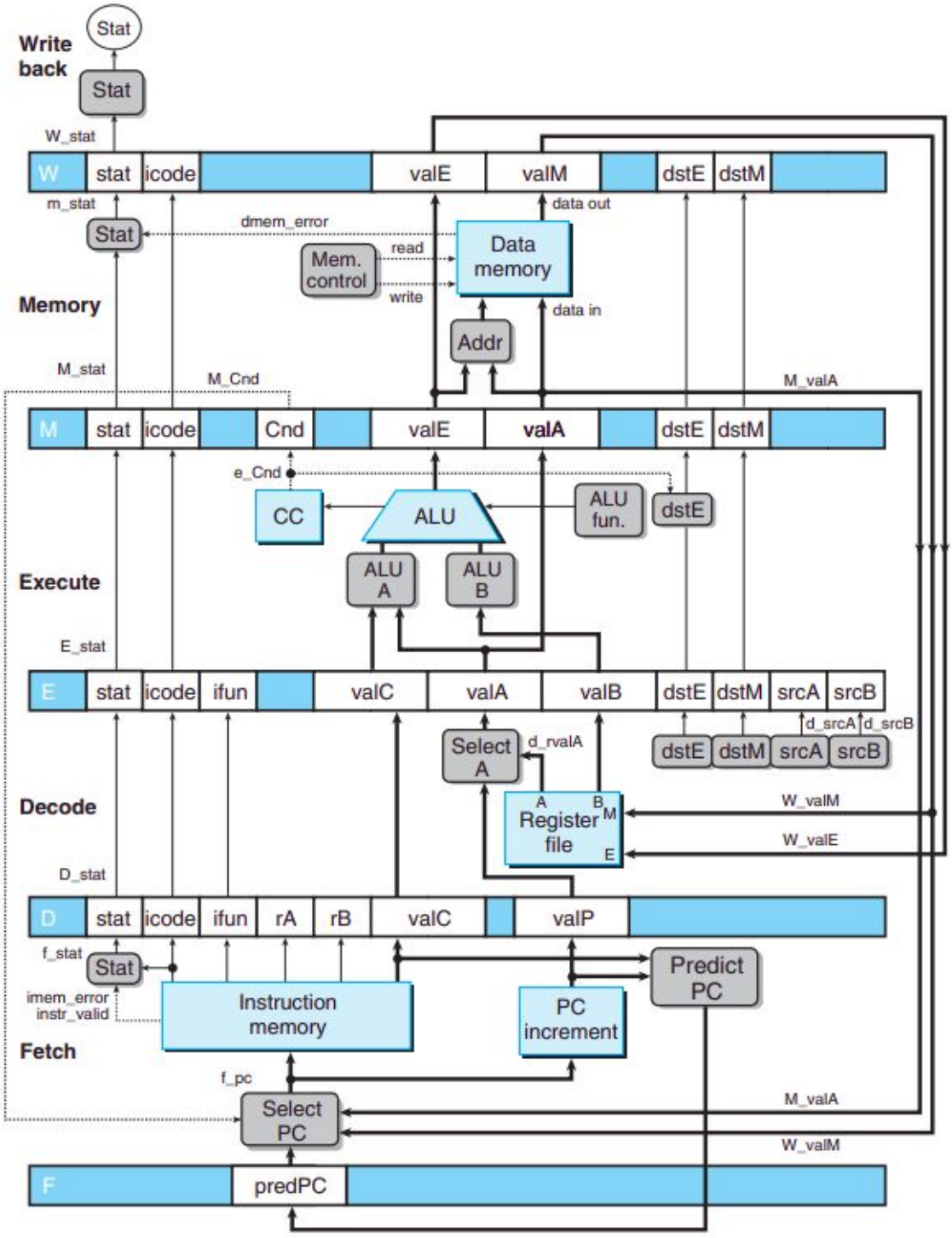
Stat - One small part of the write-back stage remains. The overall processor status Stat is computed by a block based on the status value in pipeline register W. Since pipeline register W holds the state of the most recently completed instruction, it is natural to use this value as an indication of the overall processor status. The only special case to consider is when there is a bubble in the write-back stage. This is part of normal operation, and so we want the status code to be AOK for this case as well.

HCL code for STAT -

-----> word Stat = [W_stat == SBUB : SAOK;

1 : W_stat;];

FULL FLOWCHART -



Instructions Working -

1. halt
2. nop
3. rrmovq
4. irmovq
5. rmmovq
6. mrmovq
7. OPq
8. jXX
9. cmovXX
10. call
11. ret
12. pushq
13. popq

Processor features -

64-bit Y86 5-staged pipelined processor architecture

Clock pulse has period of 10 and hence the frequency is 1/10 = 0.1

Instruction memory has size of 8*1024

Data memory has size of 64*8192

Example to show GTKWave output -

1.**HCF of 2 numbers**. rA has 100 and rB is 35. HCF is 5. The program runs properly and shows all the wire values!

Instruction memory - code.mem file

Assembly code -

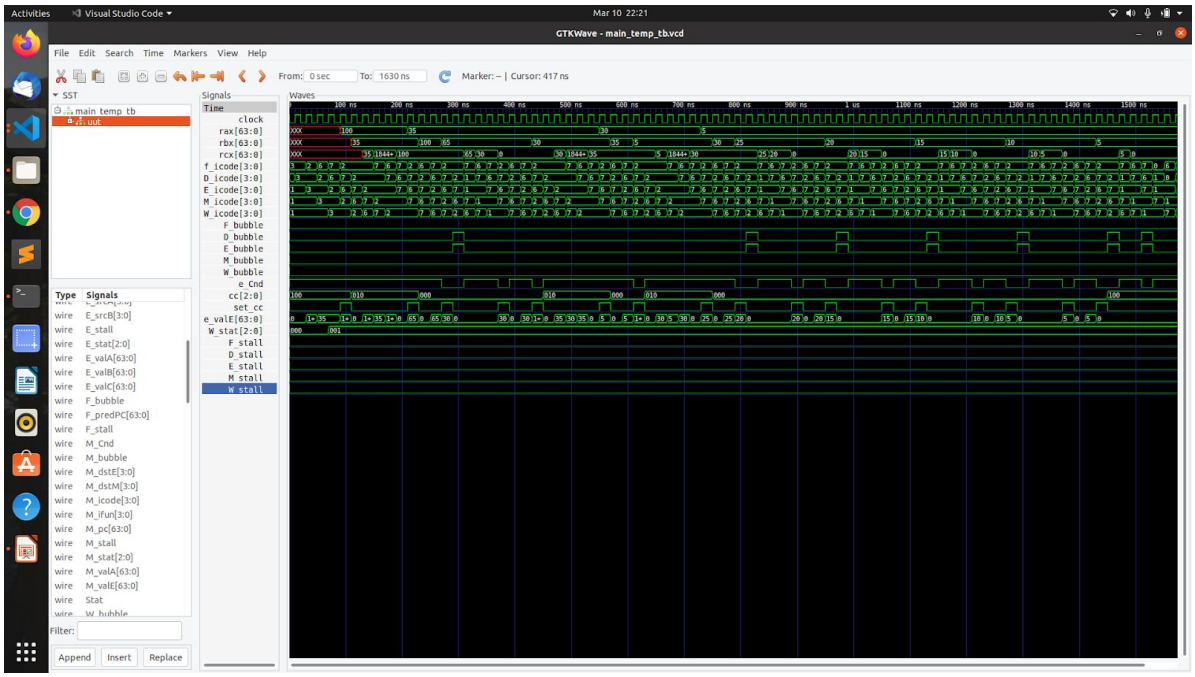
```
rmovq 64 %rax
irmovq 23 %rbx

Loop1:
rrmovq %rbx %rcx
subq %rax, %rcx
jg .Loop3
jl .Loop2
halt

Loop2:
subq %rax, %rbx
jmp .Loop1

Loop3:
rrmovq %rax, %rcx
rrmovq %rbx, %rax
rrmovq %rbx, %rbx
jmp .Loop2
```

Output -



Explanation - The GTKWave output shows the clock pulse. rA has 100 and rB has 35 which decrements to finally give the HCF as 5 by recursive method. This is followed by icode wires in all the stages. It is clearly visible that the stages work in a pipelined format. Next is bubble variable for all stages followed by Cnd (branch conditions), cc(condition codes), set_cc. Then is e_valE which shows the execute stage output values at all times. W_stat is the status which halts the program when it is 010. At the end we have stall variable for all stages which is 0 here. HCF = 5 is visible in rB as well as in e_valE.

C++ code -

```
#include <iostream>
using namespace std;
int gcd(int a, int b)
  { return gcd(a-b, b);}
int main()
{int a = 100, b = 35;
    cout<<"GCD of "<<a<<" and "<<b<<" is "<<gcd(a, b);}
```

2. The code stores 100 in rB and 10 in rA and then performs repeated subtraction while executing a conditional jump. Thus the value of rB keeps on decreasing till it is equal to 0.

Instruction memory -

10

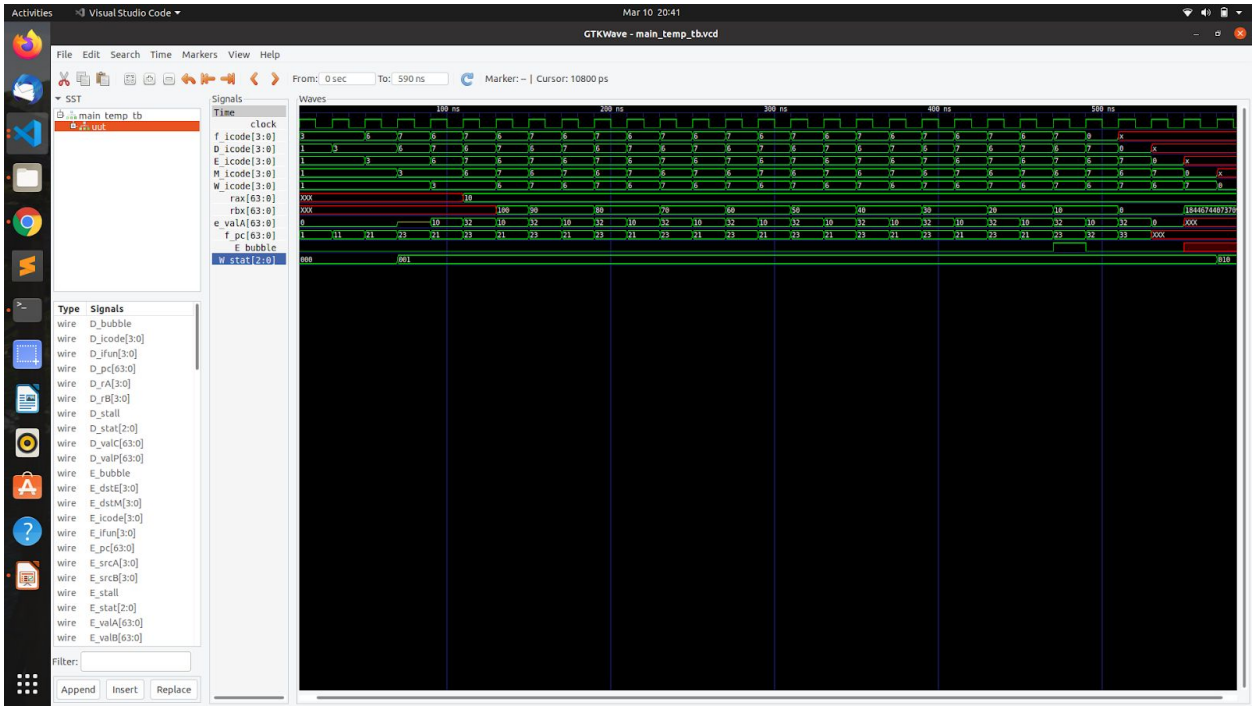
30 F0 00 00 00 00 00 00 00 0A

30 F3 00 00 00 00 00 00 00 64

61 03

74 00 00 00 00 00 00 00 15 00

Output -



Explanation - The GTKWave output shows the clock pulse followed by icode wires in all the stages. It is clearly visible that the stages work in a pipelined format . Further we can see the values of rA and rB where rA remains 10 throughout and rB decreases by 10 every time to go from 100, 90, 80.....10,0.

Further we can see e_valE value and E_bubble. At the end we can see the value of W_stat which as soon as goes to 010, the program terminates.

This was just a simple and random example to check the implementation of the processor.

Hence proved that the processor runs well for conditional jumps, loops and all the instructions mentioned above etc.

* How to run the code -

The files are as follows -

1. main_temp.v - Main processor file
2. main_temp_tb.v - Testbench for the same
3. fetch.v - Fetch logic
4. decode.v - Decode + Writeback logic
5. memory.v - Memory Logic
6. .mem files for instruction memory

FOR CONVENIENCE, EVERY BLOCK HAS BEEN LABELLED IN THE CODE WITH A COMMENT

Run the following commands -

1. iverilog main_temp_tb.v main_temp.v
2. ./a.out
3. gtkwave main_temp_tb.vcd