Intro to Processor Architecture

5-Stage Pipelined Y86 Processor

Tested and Verified for the code which finds HCF of two Numbers

Abhishek Chawla

10th March, 2021

Module Descriptions and Architecture Diagram

1. Fetch Logic and PC Selection

There are four modules in this stage:

a) Instruction Memory
b) Split
c) Align
d) PC increment

a) Instruction Memory:

In this block, we are reading all the instructions from memory. It's output enters both the split and align module. The output will consist of 10 bytes from which 1 byte goes into split module and the other 9 bytes go into align module.

b) Split:

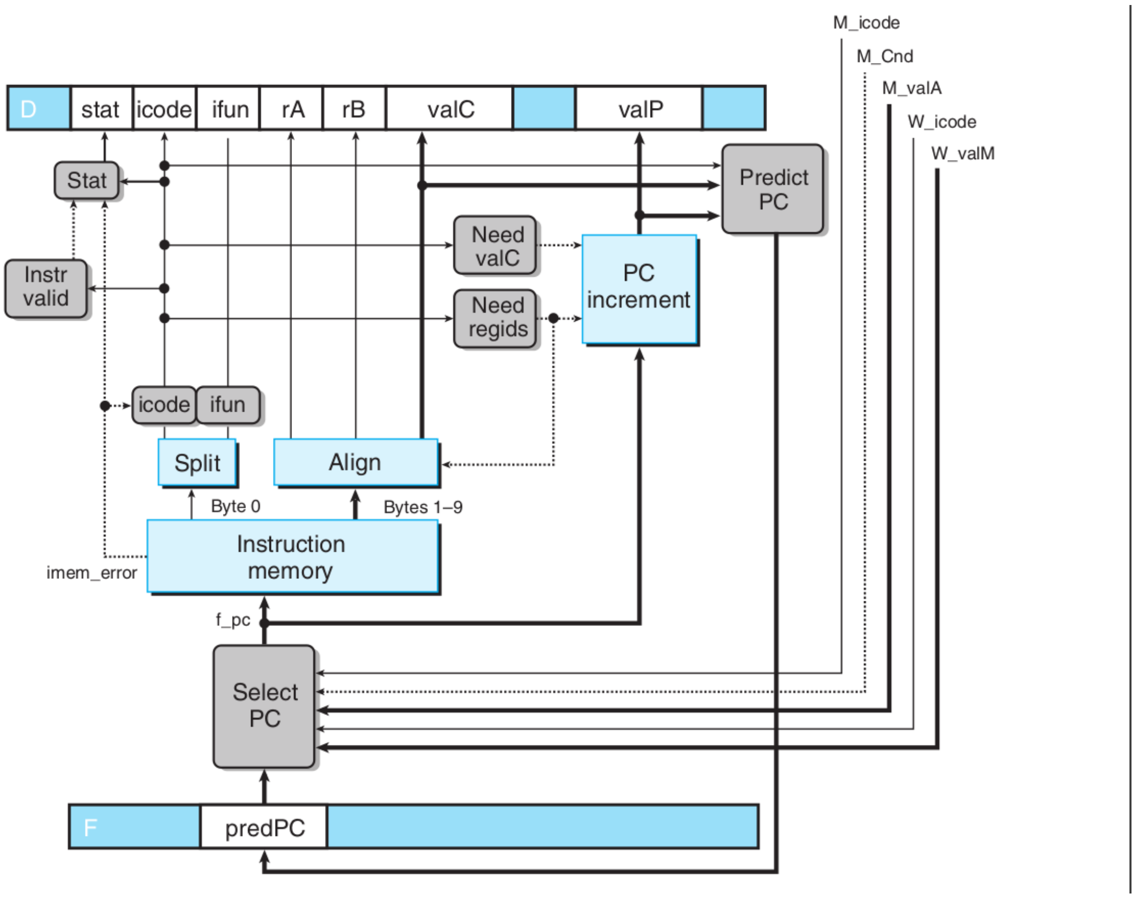
In this, the first byte is interpreted as the instruction byte and is split (by the unit labeled "Split") into two 4-bit quantities (icode and ifun). Icode and ifun are the outputs of this module.The control logic blocks labeled "icode" and "ifun" then compute the instruction and function codes as equaling 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 (shown as dashed lines in the figure given below).

c) Align:

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.

d) PC Increment:

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.



The PC selection logic 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):

```
word f_pc = [

# Mispredicted branch. Fetch at incremented PC

M_icode == IJXX && !M_Cnd : M_valA;

# Completion of RET instruction

W_icode == IRET : W_valM;

# Default: Use predicted value of PC

1 : F_predPC;

];
```

The PC prediction logic chooses valC for the fetched instruction when it is either a call or a jump, and valP otherwise:

```
word f_predPC = [

f_icode in { IJXX, ICALL } : f_valC;

1 : f_valP;

];
```

The logic blocks labeled “Instr valid,” “Need regids,” and “Need valC” are the same as for SEQ, with appropriately named source signals. 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.

2. Decode and Write-Back

These two stages (decode and write-back) are combined because they both access the register file. 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.

Register ID dstE indicates the destination register for write port E, where the computed value valE is stored. Observe that 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.

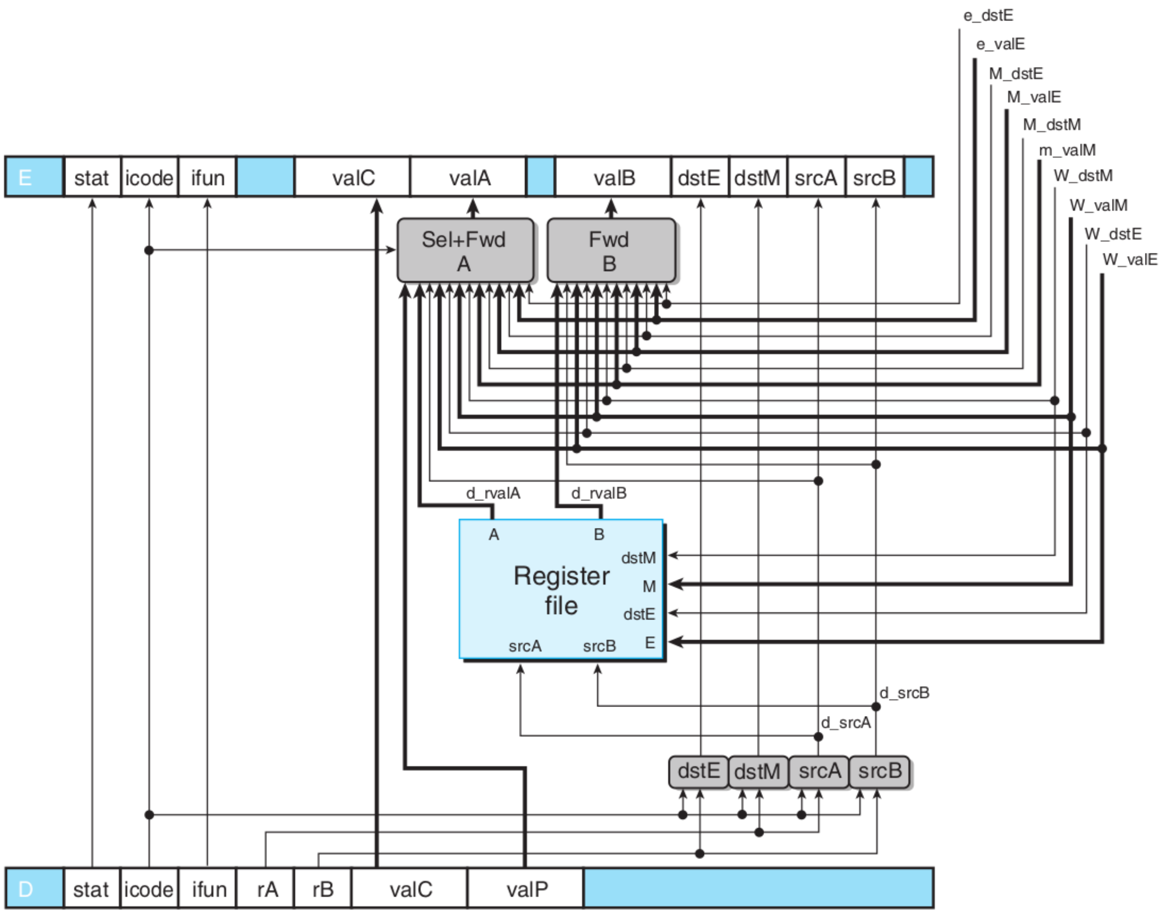
Most of the complexity of this stage is associated with the forwarding logic. The block labeled “Sel+Fwd A” serves two roles. It merges the valP signal into the valA signal for later stages in order to reduce the amount of state in the pipeline register. It also implements the forwarding logic for source 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.

There are five different forwarding sources, each with a data word and a destination register ID: (see the following table)

Data word	Register ID	Source description
e_valE	e_dstE	ALU output
m_valM	M_dstM	Memory output
M_valE	M_dstE	Pending write to port E in memory stage

W_valM	W_dstM	Pending write to port M in write-back stage
W_valE	W_dstE	Pending write to port E in write-back stage



If none of the forwarding conditions hold, the block should select d_rvalA, the value read from register port A, as its output. Putting all of this together, we get the following HCL description for the new value of valA for pipeline register E:

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

];
```

The priority given to the five forwarding sources in the above HCL code is very important. This priority is determined in the HCL code by the order in which the five destination register IDs are tested. If any order other than the one shown were chosen, the pipeline would behave incorrectly for some programs.

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:

```
word Stat = [

W_stat == SBUB : SAOK;

1 : W_stat;

];
```

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. These data and control signals are generated by three control blocks, as diagrammed in Figure below. The ALU output becomes the signal valE.

The operands are listed with aluB first, followed by aluA to make sure the subq instruction subtracts valA from valB. We can see that the value of aluA can be valA, valC, or either −8 or +8, depending on the instruction type.

Looking at the operations performed by the ALU in the execute stage, we can see that it is mostly used as an adder. For the OPq instructions, however, we want it to use the operation encoded in the ifun field of the instruction. 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:

```
bool set_cc = icode in { IOPQ };
```

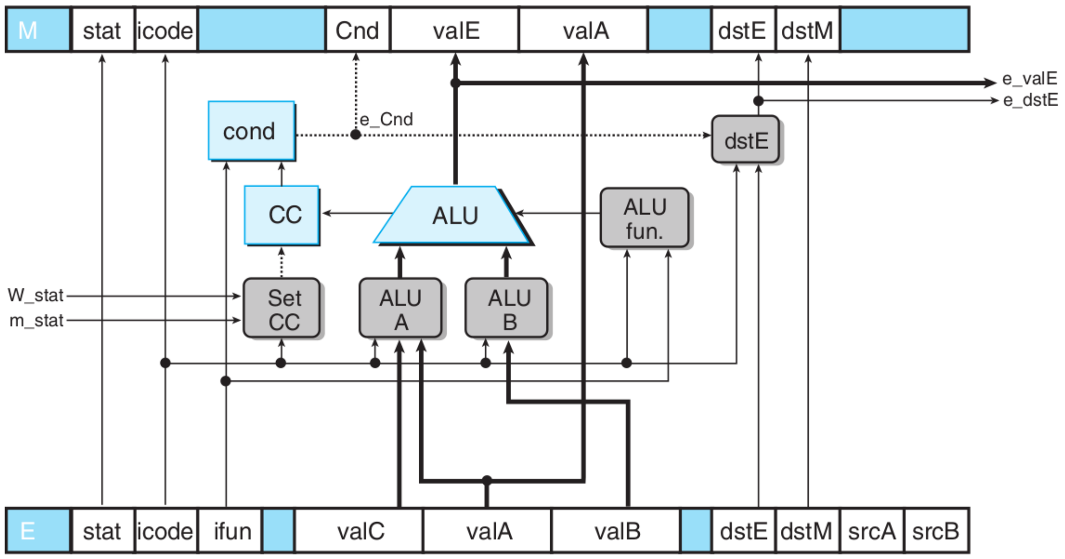
Looking at the operations performed by the ALU in the execute stage, we can see that it is mostly used as an adder. For the OPq instructions, however, we want it to use the operation encoded in the ifun field of the instruction. We can therefore write the HCL description for the ALU control as follows:

```
word alufun = [

icode == IOPQ : ifun;

1 : ALUADD;

];
```

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, but it will be ignored by the control logic.

We can see the signals e_valE and e_dstE directed toward the decode stage as one of the forwarding sources. One difference from sequential in pipelined 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



4. Memory Stage

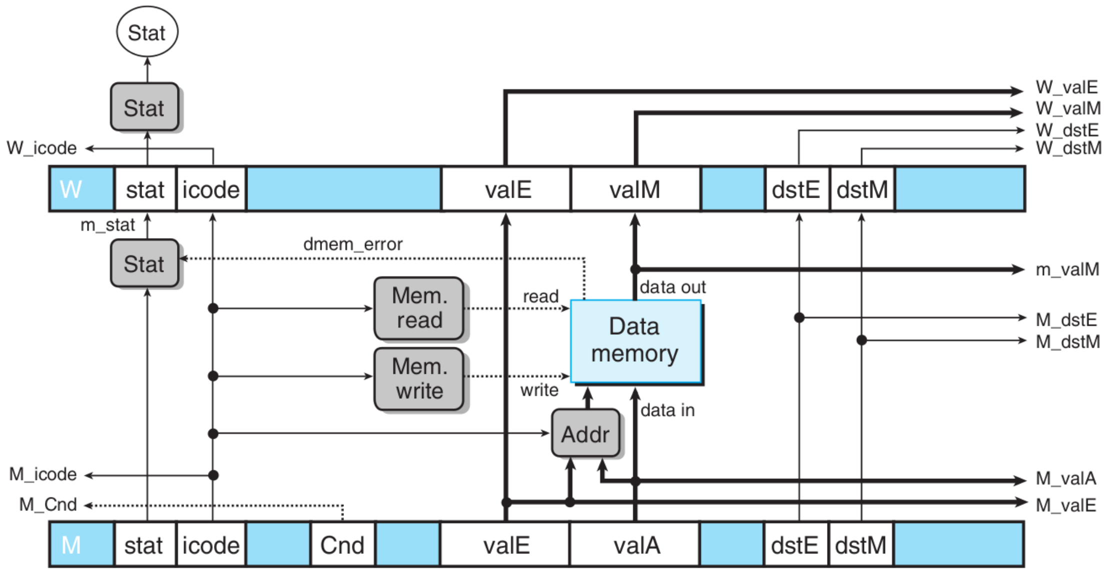
The memory stage has the task of either reading or writing program data. As shown in Figure below, two control blocks generate the values for the memory address and the memory input data (for write operations). Two other blocks generate the control signals indicating whether to perform a read or a write operation. When a read operation is performed, the data memory generates the value valM.

The address for memory reads and writes is always valE or valA.

We want to set the control signal mem_read only for instructions that read data from memory, as expressed by the following HCL code:

```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
```

A final function for the memory stage is to compute the status code Stat resulting from the instruction execution according to the values of icode, imem_error, and instr_valid generated in the fetch stage and the signal dmem_error generated by the data memory.



5. Pipeline Control Logic

We are now ready to complete our design for PIPE by creating the pipeline control logic. This logic must handle the following four control cases for which other mechanisms, such as data forwarding and branch prediction, do not suffice:

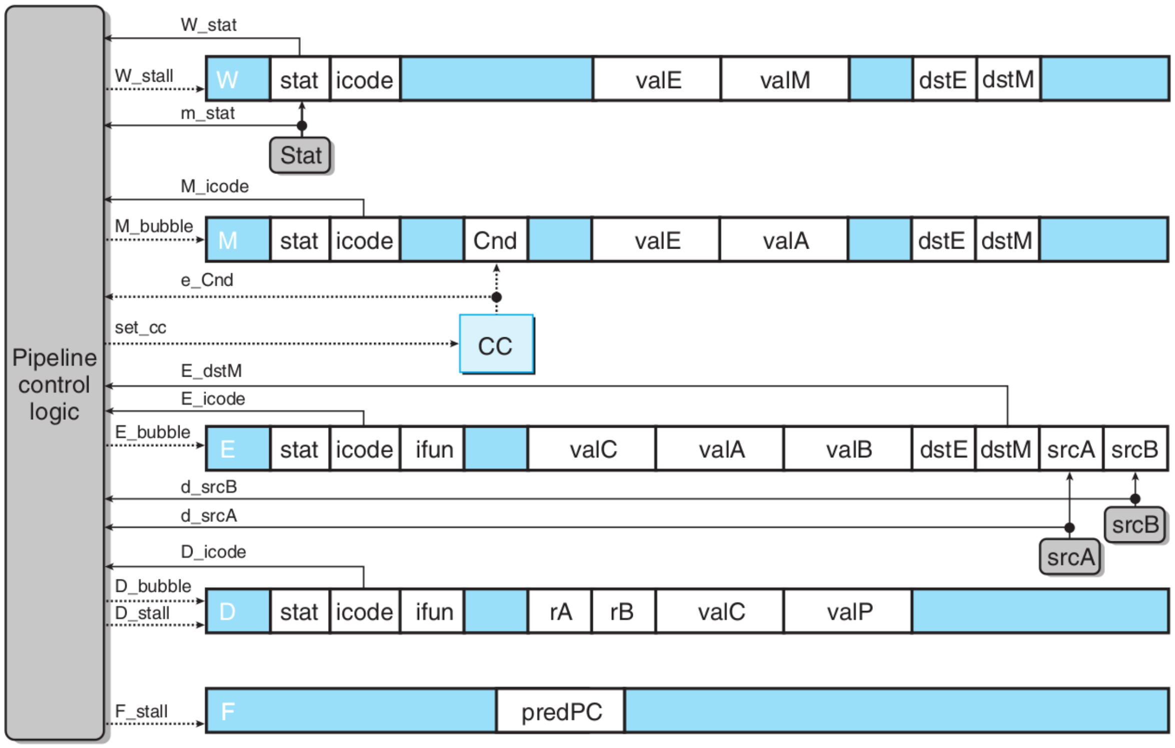
Load/use hazard:. The pipeline must stall for one cycle between an instruction that reads a value from memory and an instruction that uses this value.

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.

Implementation:



Based on signals from the pipeline registers and pipeline stages, the control logic generates stall and bubble control signals for the pipeline registers and also determines whether the condition code registers should be updated.

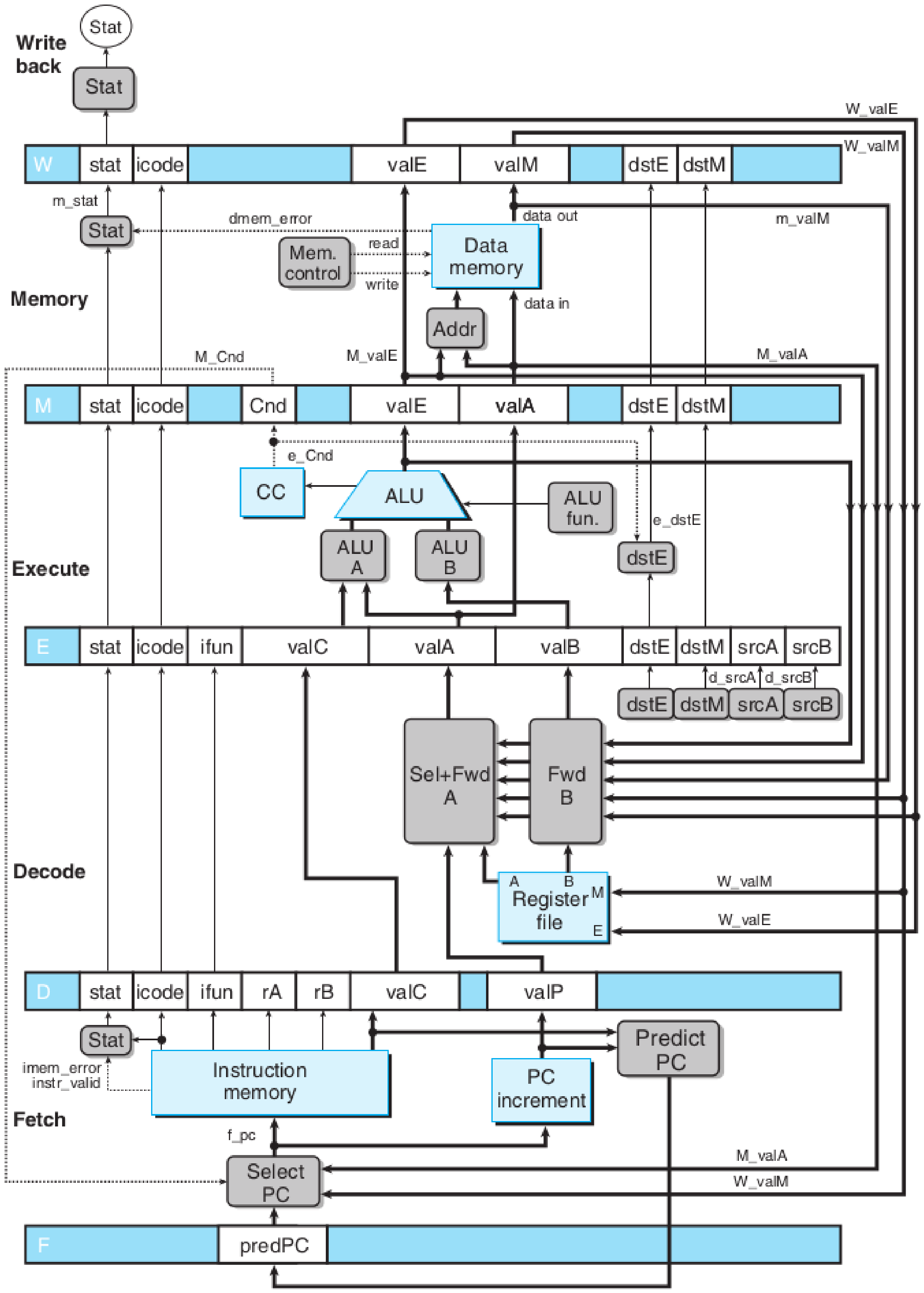
Pipeline register F must be stalled for either a load/use hazard or a retinstruction:

```
bool F_stall =

# Conditions for a load/use hazard

E_icode in { IMRMOVQ, IPOPQ } &&

E_dstM in { d_srcA, d_srcB } ||

# Stalling at fetch while ret passes through pipeline

IRET in { D_icode, E_icode, M_icode };
```

Pipeline register D must be set to bubble for a mispredicted branch or a ret instruction. As the analysis in the preceding section shows, however, it should not inject a bubble when there is a load/use hazard in combination with a ret instruction:

```
bool D_bubble =

# Mispredicted branch

(E_icode == IJXX && !e_Cnd) ||

# Stalling at fetch while ret passes through pipeline

# but not condition for a load/use hazard

!(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&

IRET in { D_icode, E_icode, M_icode };
```

COMPLETE PIPELINED PROCESSOR:


Stat
Write back
Stat
W_valE
W_valM
W
stat
icode
valE
valM
dstE
dstM
m_stat
Stat
dmem_error
data out
m_valM
Mem. control
read
write
Data memory
data in
Memory
Addr
M_Cnd
M_valE
M_valA
M
stat
icode
Cnd
valE
valA
dstE
dstM
e_Cnd
CC
ALU
ALU fun.
e_dstE
ALU A
ALU B
dstE
Execute
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
d_srcA
d_srcB
dstE
dstM
srcA
srcB
Sel+Fwd A
Fwd B
Decode
A
B
M
Register file
E
W_valM
W_valE
D
stat
icode
ifun
rA
rB
valC
valP
Stat
Predict PC
imem_error
instr_valid
Instruction memory
PC increment
Fetch
f_pc
M_valA
W_valM
Select PC
F
predPC

Instructions Supported By Processor:

- halt
- nop
- rrmovq
- irmovq
- rmmovq
- mrmovq
- OPq
- jXX
- cmovXX
- call
- ret
- pushq
- popq

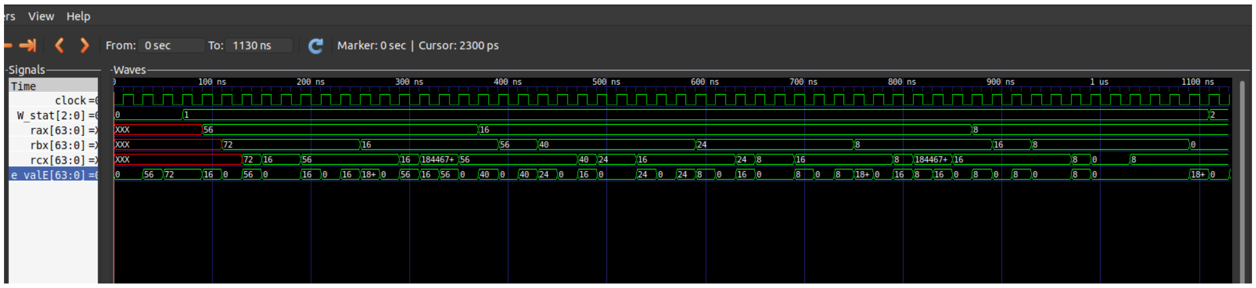
HCF Code in C++ :

```
#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=72;

   cout<<gcd(56,72)<<endl;

}
```

HCF Code in Assembly Language:

```
rmovq 56, %rax

  irmovq 72, %rbx

  fun1:

  rrmovq %rbx, %rcx

  subq %rax, %rcx

  jg .fun3

  jl .fun2

  halt


  fun2:

  subq %rax, %rbx

  jmp .fun1


  fun3:

  rrmovq %rax, %rcx

  rrmovq %rbx, %rax

  rrmovq %rbx, %rbx

  jmp .fun2
```

GTKWave Output: (for HCF)



This finds the gcd of 56 and 72 which comes out to be 8 and is stored in rbx.

All the parameters are by their names (standard).

Instructions to run my code:

1. Download the codes folder to your machine.

2. To install verilog: (in Terminal)

```
sudo apt-get install verilog
```

3. To install Gtkwave:

```
sudo apt-get install gtkwave
```

4. To compile the code:

```
iverilog processor.v processor_tb.v
```

5. To run:

```
./a.out
```

6. To show output in gtkwave:

```
gtkwave processor_tb.vcd
```