Intro To Processor Architecture

5 stage Pipelined Y-86 Processor using Verilog

Bhaiya Vaibhaw Kumar

2019112021

MODULE DESCRIPTION AND ARCHITECTURE DESIGN

1. Pipe Selection and Fetch

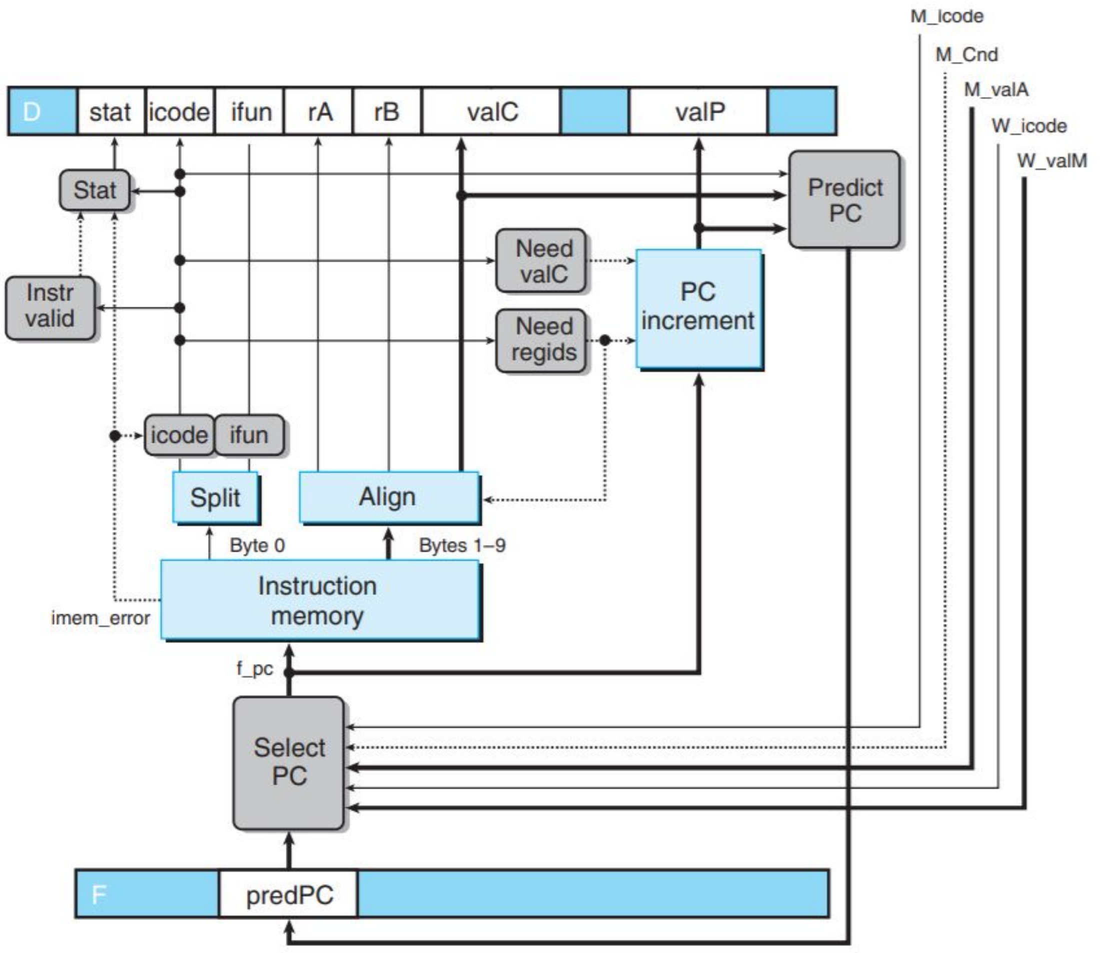


Figure 4.57 **PIPE PC selection and fetch logic.** Within the one cycle time limit, the processor can only predict the address of the next instruction.

The instruction memory reads bytes from an instruction by using the programme counter register as an address. valP, or the incremented programme counter, is computed by the PC incrementer. PC update- The new value of the program counter is selected to be either valP, the address of the next instruction, valC, the destination address specified by a call or jump instruction, or valM, the return address read from memory.

2. Pipe Decode and Write-back

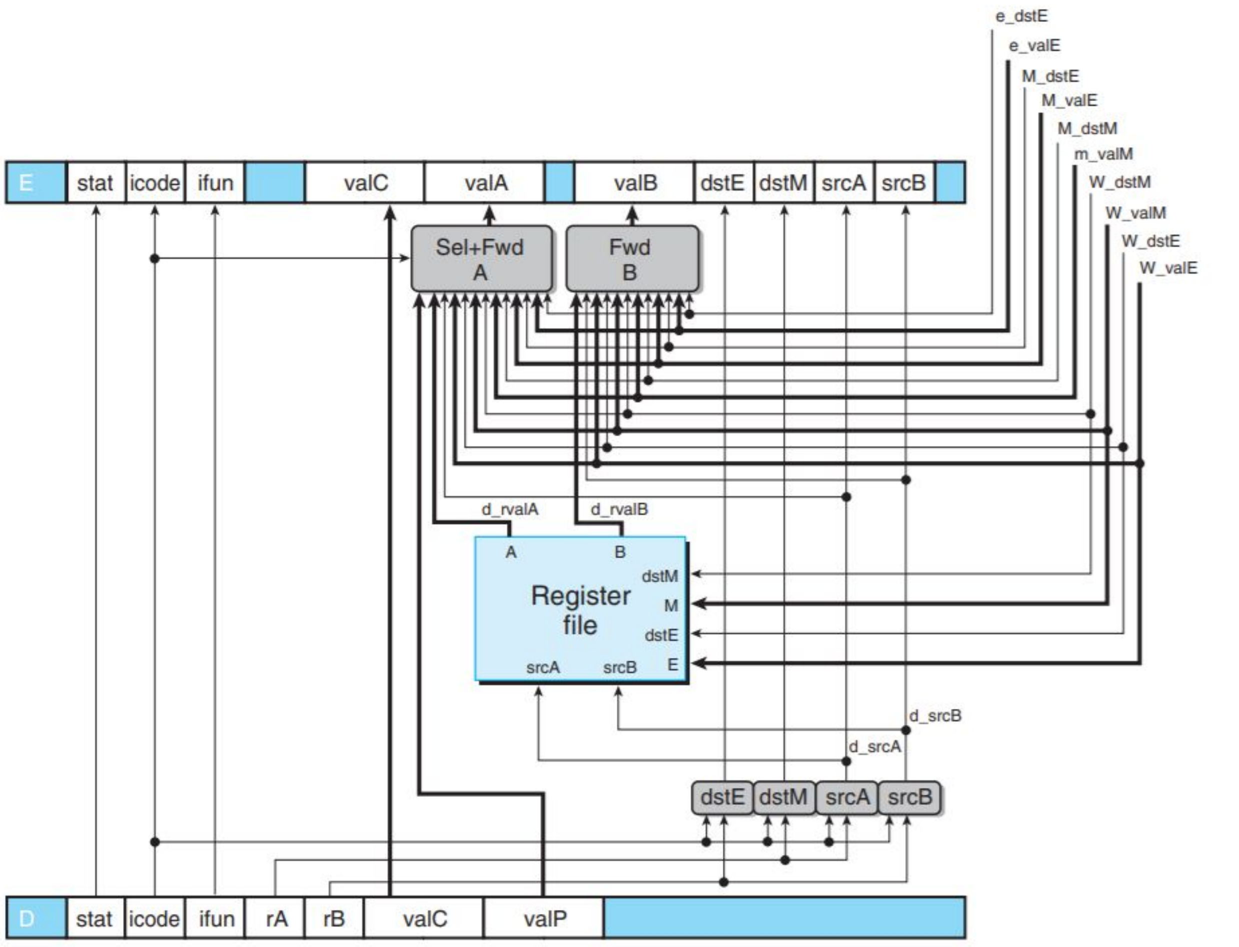


Figure 4.58 PIPE decode and write-back stage logic. No instruction requires both valP and the value read from register port A, and so these two can be merged to form the signal valA for later stages. The block labeled "Sel+Fwd A" performs this task and also implements the forwarding logic for source operand valA. 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.

Decode- The register file has two read ports, A and B, via which register values valA and valB are read simultaneously.

Write-back- The register file has two write ports. Port E is used to write values computed by the ALU, while port M is used to write values read from the data memory.

3. Pipe Execute

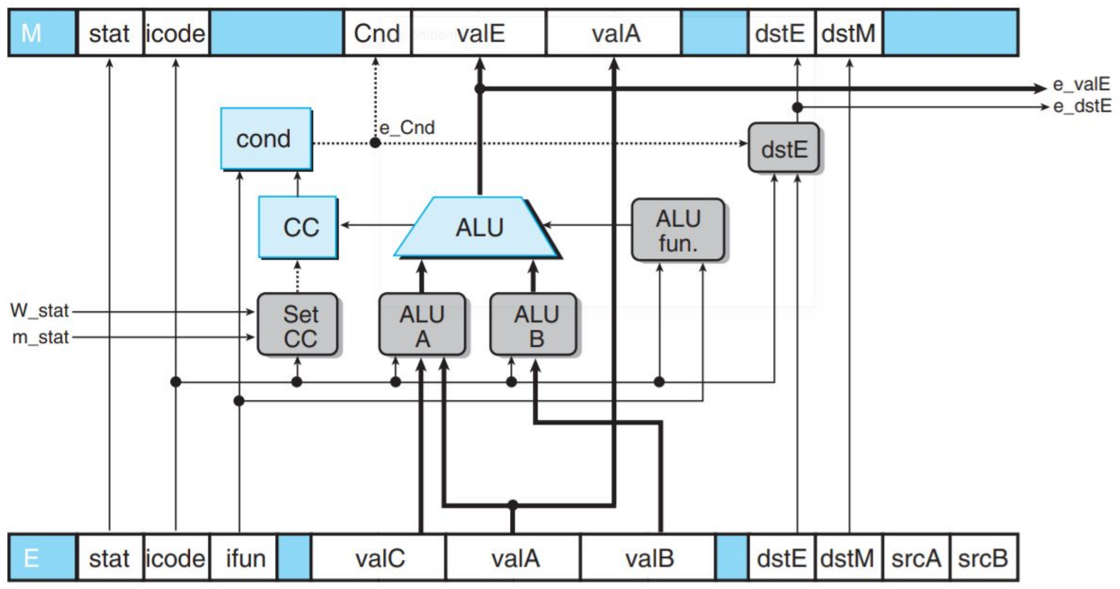


Figure 4.60 **PIPE execute stage logic.** This part of the design is very similar to the logic in the SEQ implementation.

The execute stage uses the arithmetic/logic (ALU) unit for different purposes according to the instruction type. For integer operations, it performs the specified operation. For other instructions, it serves as an adder to compute an incremented or decremented stack pointer, to compute an effective address, or simply to pass one of its inputs to its outputs by adding zero. The condition code register (CC) holds the three condition code bits. New values for the condition codes are computed by the ALU. When executing a conditional move instruction, the decision as to whether or not to update the destination register is computed based on the condition codes and move condition. Similarly, when executing a jump instruction, the branch signal Cnd is computed based on the condition codes and the jump type.

4. Pipe Memory

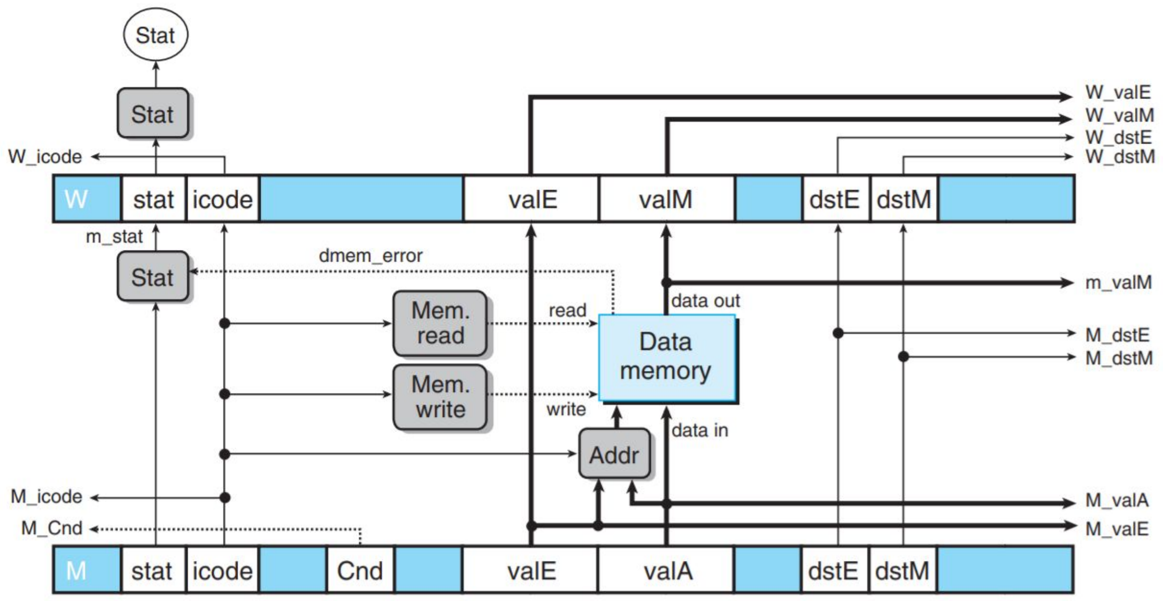


Figure 4.61 PIPE memory stage logic. Many of the signals from pipeline registers M and W are passed down to earlier stages to provide write-back results, instruction addresses, and forwarded results.

The data memory reads or writes a word of memory when executing a memory instruction. The instruction and data memories access the same memory locations, but for different purposes.

5. Execute and Memory stages capable of Load forwarding

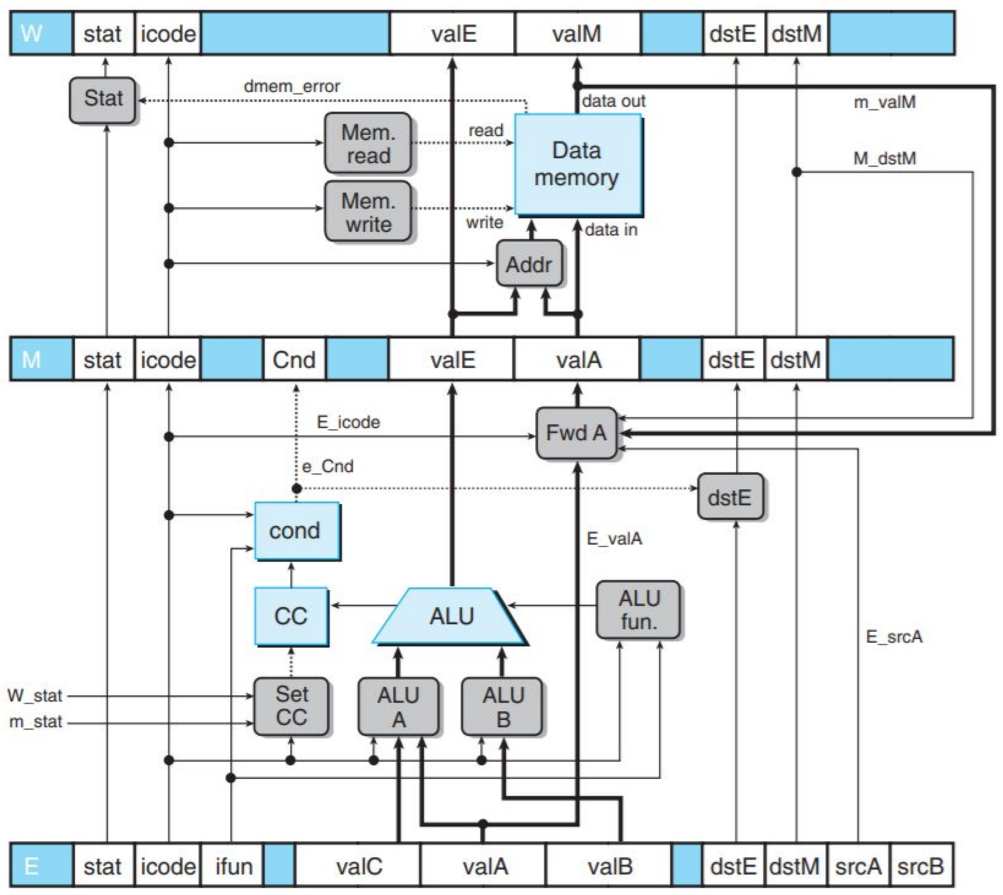


Figure 4.70 **Execute and memory stages capable of load forwarding.** By adding a bypass path from the memory output to the source of valA in pipeline register M, we can use forwarding rather than stalling for one form of load/use hazard. This is the subject of Problem 4.57.

INSTRUCTIONS SUPPORTED BY THE PROCESSOR

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

Encoded Instructions:

30

F0

00

00

00

00

00

00

00

38

30

F3

00

00

00

00

00

00

00

62

20

31

61

01

72

00

00

00

00

00

00

00

36

76

00

00

00

00

00

00

00

2B

00

61

03

70

00

00

00

00

00

00

00

14

20

01

20

30

20

13

70

00

00

00

00

00

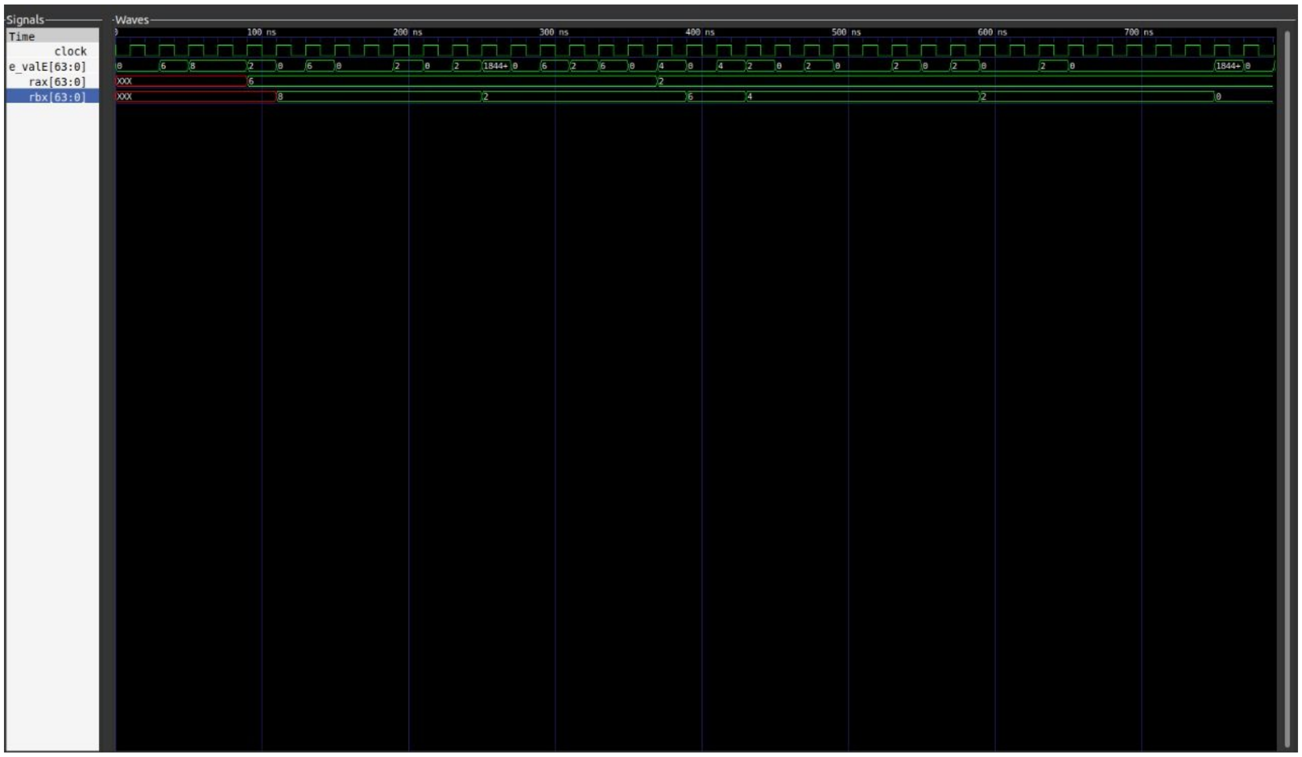
00

00

2B

GTKWAVE OUTPUTS

The code runs properly and was verified on HCF.



Inputs are 6 and 8. And the parameters are as named in the report.

INSTRUCTIONS TO RUN THE CODE

1. Install verilog through Terminal:
```
sudo apt-get install verilog
```
2. Install Gtkwave using this:
```
sudo apt-get install gtkwave
```
3. Compile the code using this command:
```
iverilog IPA_FINAL.v IPA_FINAL_TESTBENCH.v
```
4. Run the code using:
```
./a.out
```
6. To get the output in gtkwave:
```
gtkwave IPA_FINAL_TESTBENCH.vcd
```

On the Linux Terminal, all files correspond to iverilog. The code can be tested using the appropriate testbench.