IPA Project Report

Konda Jayant Reddy (2019102010)

MODULE DESCRIPTIONS:

I have implemented a full pledged architecture implementation with 5 stage pipeline which includes support for eliminating pipeline hazards.

Tested and Verified for the code for finding HCF of 2 numbers

- Fetch:

Using the program counter register as an address, the instruction memory reads the bytes of an instruction. The PC incrementer computes valP, the incremented program counter.

- Decode:

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

- Execute:

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.

- ## Memory:

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.

- ## 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.

- ## 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.

ARCHITECTURE DIAGRAM:

1. **Fetch and Selection:**

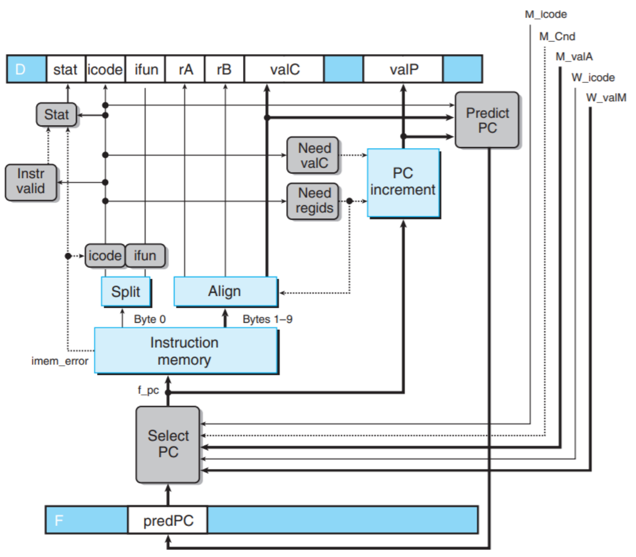


Figure 4.57 **PIPE PC selection and fetch logic.** Within the one cycle time limit, the processor can only predict the address of the next instruction.

2. **Execute State:**

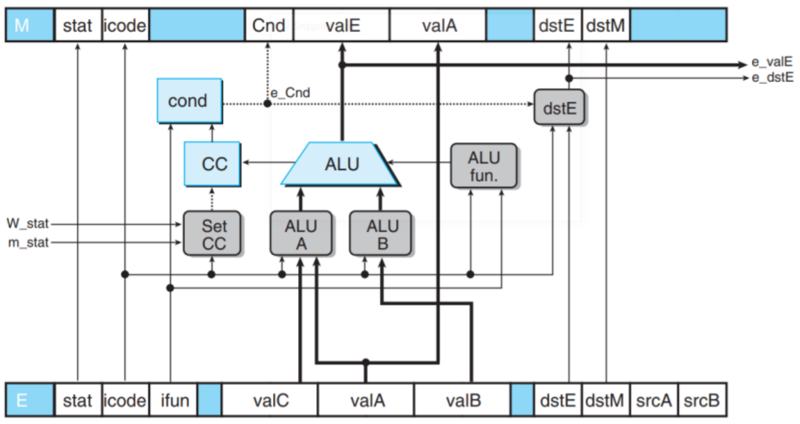


Figure 4.60 **PIPE execute stage logic.** This part of the design is very similar to the logic in the SEQ implementation.

3. Execute Memory Load Forward:

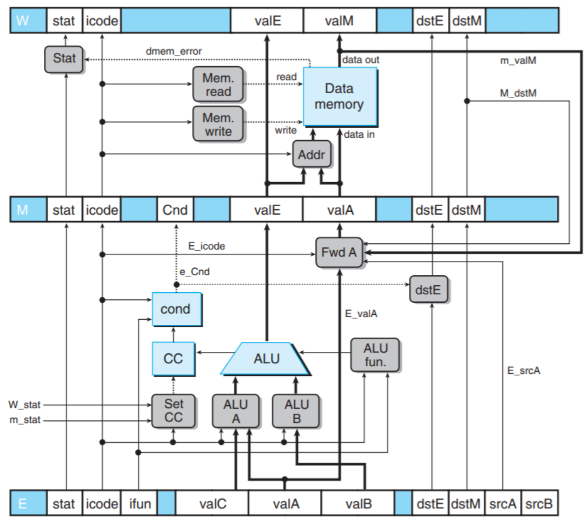


Figure 4.70 **Execute and memory stages capable of load forwarding.** By adding a bypass path from the memory output to the source of valA in pipeline register M, we can use forwarding rather than stalling for one form of load/use hazard. This is the subject of Problem 4.57.

4. Pipe Decode and Write-Back:

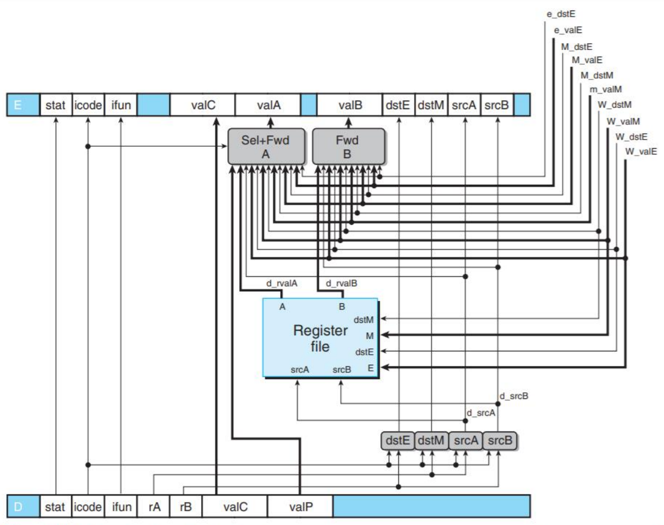


Figure 4.58 **PIPE decode and write-back stage logic.** No instruction requires both valP and the value read from register port A, and so these two can be merged to form the signal valA for later stages. The block labeled "Sel+Fwd A" performs this task and also implements the forwarding logic for source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

5. **Pipe Memory:**

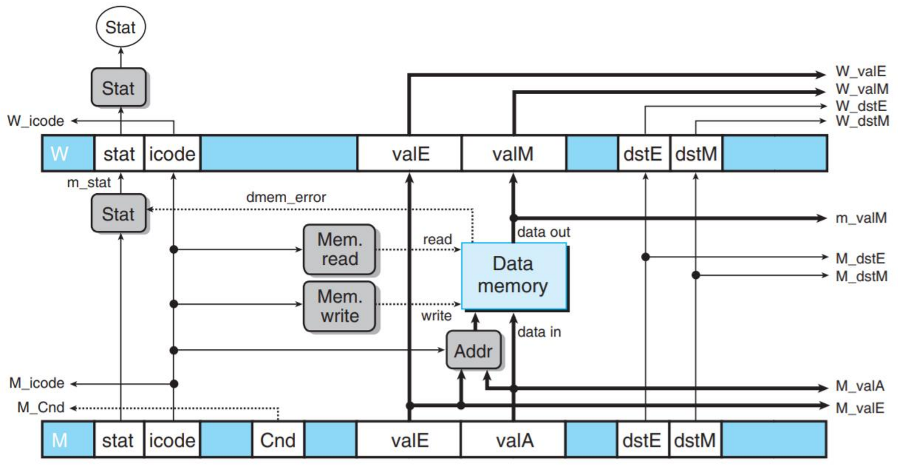


Figure 4.61 **PIPE memory stage logic.** Many of the signals from pipeline registers M and W are passed down to earlier stages to provide write-back results, instruction addresses, and forwarded results.

INSTRUCTIONS SUPPORTED BY THE PROCESSOR:

halt

nop

rrmovq

irmovq

rmmovq

mrmovq

OPq

jXX

cmovXX

call

ret

pushq

popq

ASSEMBLY CODE:

C++ Code:

```
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;
}
```

Assembly Code:

```
rmovq 58, %rax
    irmovq 98, %rbx

    clock:
    rrmovq %rbx, %rcx
    subq %rax, %rcx
    jg .swap
    jl .repsub
    halt

    repsub:
    subq %rax, %rbx
    jmp .check

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

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

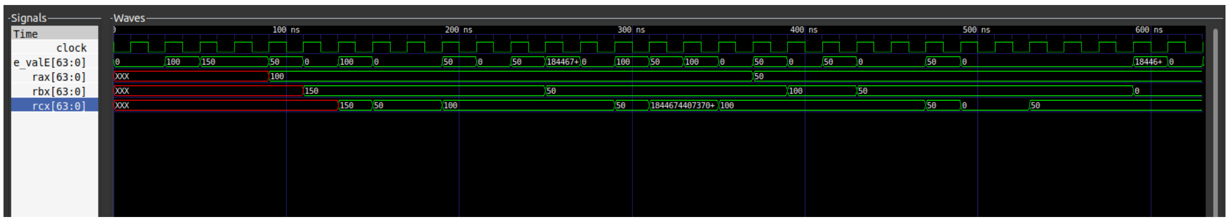
00

00

00

2B

GTKWAVE OUTPUT:



Inputs: All parameters are named according to their meaning

The inputs given are 100 and 150.

INSTRUCTIONS TO RUN THE CODE:

All files correspond to iverilog on the Linux Terminal. The code can be runned with the respective testbench.

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 y86_processor.v y86_processor_tb.v
```

5. To run:

```
./a.out
```

6. To show output in gtkwave:

```
gtkwave y86_processor_tb.vcd &
```