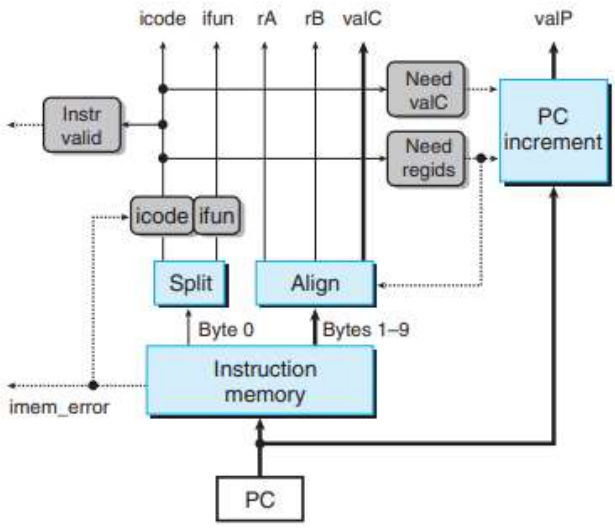
Project Report

Stage wise descriptions:

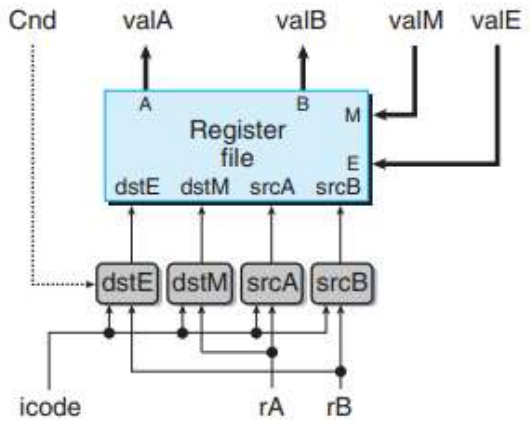
Fetch Stage

Every instruction in assembly language in encoded using a specific format. This format being either
[icode(4bit), ifun(4bit), rA(4bit), rB(4bit), const], or [icode(4bit), ifun(4bit), dest]. Both of these formats take up 10 bytes of data. The fetch stage takes in the instruction held by the Program Counter (PC) and splits it into its individual pieces. These individual pieces will be used for surther computations. Here, the next value held by the PC is also predicted.



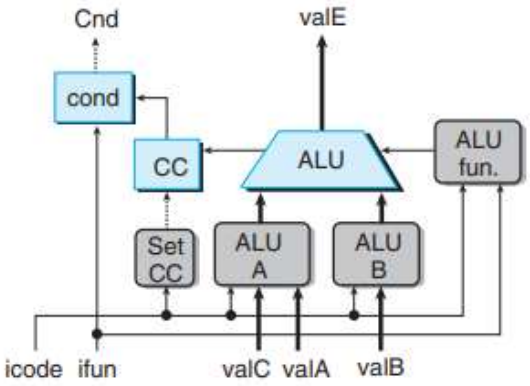
Decode Stage

The decode stage takes in the split instruction from the fetch stage and uses it to determine from which registers in the register file to read data. the required values valA and valB are sent to the next stage throught their corresponding read ports.



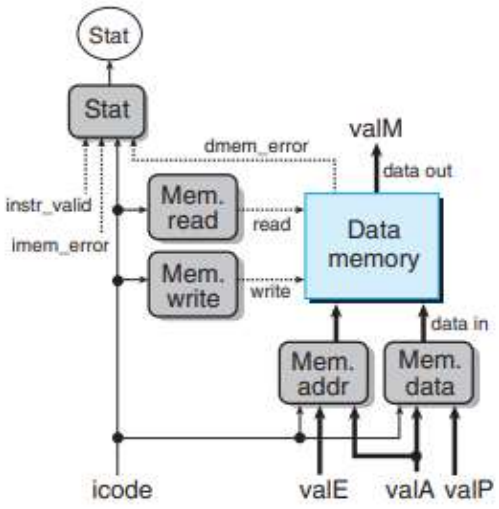
Execute stage

Now that the required data has been extracted in the decode stage, it can be sent for processing at the execute stage. The execute stage determines the condition codes and also provides the output of whatever arithmetic or logical operations need to be conducted.



Memory Stage

The memory stage determines the memory addresse from which to read or write data based off of the inputs

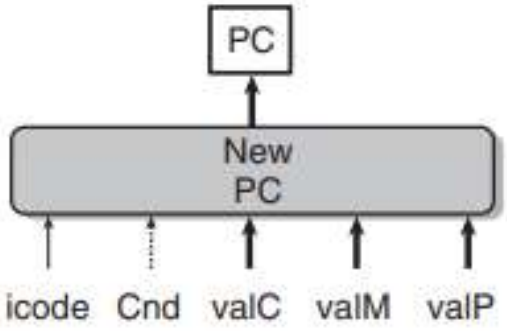


Writeback Stage

This stage writes back data to the register file. It is not necessary that data be written back at every stage.

PC Update Stage

This stage predicts the next value of the PC



Stage wise computations:

Fetch Stage

The fetch stage computations are as follows:

1. nop:
icode:ifun <-- M1[PC]

2. cmov:
icode:ifun <-- M1[PC]
regA:regB <-- M1[PC+1]
valP <-- PC + 2

3. irmovq:
icode:ifun <-- M1[PC]
regA:regB <-- M1[PC+1]
valC <-- M8[PC+2]
valP <-- PC + 10

4. rmmovq:
icode:ifun <-- M1[PC]
regA:regB <-- M1[PC+1]
valC <-- M8[PC+2]
valP <-- PC + 10

5. mrmovq:
icode:ifun <-- M1[PC]
regA:regB <-- M1[PC+1]
valC <-- M8[PC+2]
valP <-- PC + 10

6. OPq:
icode:ifun <-- M1[PC]
regA:regB <-- M1[PC+1]
valP <-- PC + 2

7. jXX:
icode:ifun <-- M1[PC]

valC <-- M8[PC+1]
valP <-- PC + 9

8. call:
icode:ifun <-- M1[PC]
valC <-- M8[PC+1]
valP <-- PC + 9

9. ret:
icode:ifun <-- M1[PC]
valP <-- PC + 1

10. pushq:
icode:ifun <-- M1[PC]
regA:regB <-- M1[PC+1]
valP <-- PC + 2

11. popq:
icode:ifun <-- M1[PC]
regA:regB <-- M1[PC+1]
valP <-- PC + 2

Decode Stage
The decode computations are as follows:

1. cmov:
   valA <-- R[rA]
   valB <-- 0

2. irmovq:
   NONE

3. rmmovq:
   valA <-- R[rA]
   valB <-- R[rB]

4. mrmovq:
   valB <-- R[rB]

5. opq:
   valA <-- R[rA]
   valB <-- R[rB]

6. jXX:
   NONE

7. call:
   valB <--R[%rsp]

8. ret:
   valA <-- R[%rsp]
   valB <-- R[%rsp]

Execute Stage
The stage computations for the execute stage are as follows:

1.nop: NONE

2. cmov: valE <-- valA
set cnd

3. irmovq:
valE <-- valC

4. rmmovq:
valE <-- valB + valC

5. mrmovq :
valE <-- valB + valC

6. OPq:

valE <-- valA OP valB
set CC

7. jXX:
set cnd

8. call:
valE <-- valB - 8

9. ret:
valE <-- valB + 8

10. pushq:
valE <-- valB - 8

11. popq:
valE <-- valB + 8

Memory Stage
The stage computations for the memory stage are as follows:

1. nop: NONE

2. cmov: NONE

3. irmovq: NONE

4. rmmovq: NONE

5. mrmovq:
M8[valE] <-- valA

6. OPq: NONE

7. jXX: NONE

8. call:
M8[valE] <-- valP

9. ret:
valM <-- M8[valA]

10. pushq:
M8[valE] <-- valA

11. popq:
valM <-- M8[valA]

Writeback Stage
The stage computations for the writeback stage are as follows:

1. nop: NONE

2. cmov:
check cnd
R[regB] <-- valE

3. irmovq:
R[regB] <-- valE

4. rmmovq: NONE

5. mrmovq:
R[regA] <-- valM

6. OPq:
R[regB] <-- valE

7. jXX: NONE

8. call:
R[%rsp] <-- valE

9. ret:
R[%rsp] <-- valE

10. pushq:
R[%rsp] <-- valE

11. popq:
R[%rsp] <-- valE
R[regA] <-- valM

PC Update Stage
The stage computations for the PC update stage are as follows:

1. nop:
PC <-- valP

2. cmov:
PC <-- valP

3. irmovq:
PC <-- valP

4. rmmovq:
PC <-- valP

5. mrmovq:
PC <-- valP

6. OPq:
PC <-- valP

7. jXX:

check cnd
PC <-- valC : valP

8. call:
PC <-- valC

9. ret:
PC <-- valM

10. pushq:
PC <-- valP

11. popq:
PC <-- valP

Need for Pipelining

Pipelining enables parallel processing between stages to happen. Every stage has a register associated with it. At every clock cycle, the register holding the values from the previous cycle passes the value to its associated stage and takes in a new value. What this does is enables instruction movement to progress like a continuous service queue, rather than a one-by-one sequential setup. While the pipelining implementation in this project doesn't work, the registers and setup have been prepared accordingly.
To implement control logic, we can apply the control conditions (stall, bubble, etc) in an if statement in th registers themselves.