https://w3.cs.jmu.edu/lam2mo/cs261_2017_08/files/y86-isa.pdf
https://cs.brown.edu/courses/cs033/docs/guides/x64_cheatsheet.pdf

PC

- holds current address of the current instruction being executed

Register File

%rsp - stack point - called by push, pop, call ret
Each register holds 64 bits

Machine code	assembly
0	%rax
1	%rcx
2	%rdx
3	%rbx
4	%rsp
5	%rbp
6	%rsi
7	%rdi
8	%r8
9	%r9
A	%r10
B	%r11
C	%r12
D	%r13
E	%r14
F	none

Condition Codes

Change based off of OPQ - ADD, SUB, AND, XOR

- Zero Flag (ZF) - true if most recent opq yields zero
- Sign Flag (SF) - true if most recent opq yields a negative
- Overflow Flag (OF) - true if most recent opq cause a two's-complement overflow (pos or neg)

Instruction Memory

- holds machine code

Data Memory

- holds program data

DO NOT WORRY ABOUT PROGRAM STATUS!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

Instruction Memory

- 13 instructions total
- Instruction code - icode
- Function code - ifun
- Do all of them except cmovXX

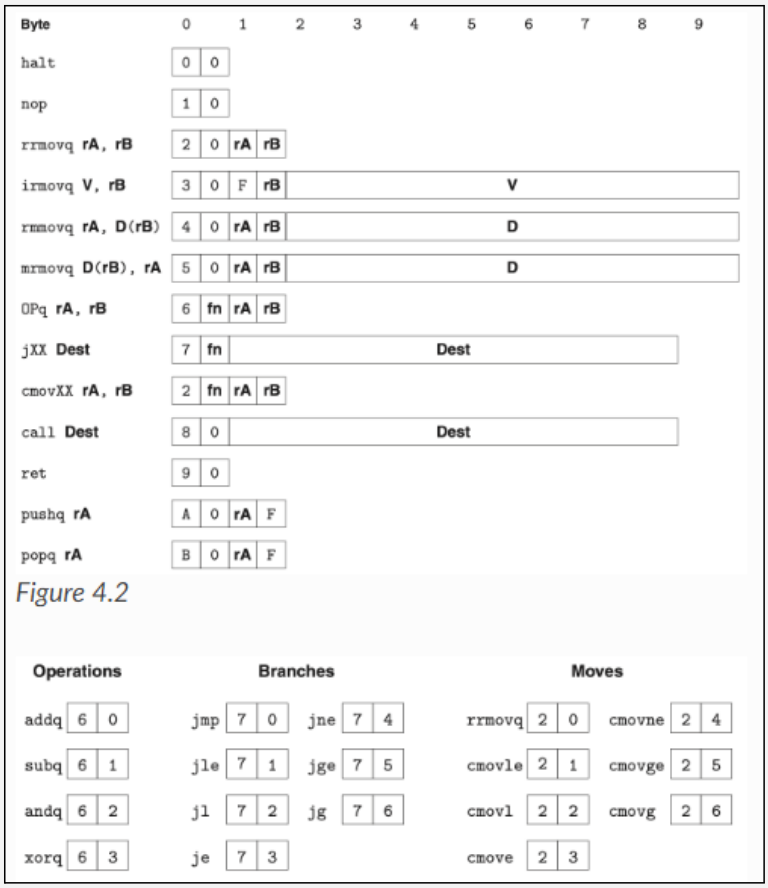


Figure 4.2

STAGE TABLE

Instruction	***Fetch***	***Decode***	***Execute***	***Memory***	***Write Back***	***PC Update***
halt	icode:ifun <- M1[PC] valP <- PC + 1	-	cpu.stat = HLT	-	-	PC <- 0
nop	icode:ifun <- M1[PC] valP <- PC + 1	-	-	-	-	PC <- valP
r r movq rA, rB	icode:ifun <- M1[PC]	valA <- R[rA]	valE <- 0 + valA	-	R[rB] <- valE	PC <- valP

	rA:rB <- M1[PC + 1] valP <- PC + 2	valB <- R[rB]				
i r movq V, rB	icode:ifun <- M1[PC] rA:rB <- M1[PC + 1] valC <- M8[PC + 2] valP <- PC + 10	-	valE <- 0 + valC	-	R[rB] <- valE	PC <- valP
r m movq rA, D(rB)	icode:ifun <- M1[PC] rA:rB <- M1[PC + 1] valC <- M8[PC + 2] valP <- PC + 10	valA <- R[rA] valB <- R[rB]	valE <- valB + valC	M8[valE] <- valA	-	PC <- valP
m r movq D(rB), rA	icode:ifun <- M1[PC] rA:rB <- M1[PC + 1] valC <- M8[PC + 2] valP <- PC + 10	valB <- R[rB]	valE <- valB + valC	valM <- M8[valE]	R[rA] <- valM	PC <- valP
OPq rA, rB	icode:ifun <- M1[PC] rA:rB <- M1[PC + 1] valP <- PC + 2	valA <- R[rA] valB <- R[rB]	valE <- valB OP valA Set CC	-	R[rB] <- valE	PC <- valP
jXX Dest	icode:ifun <- M1[PC] valC <- M8[PC + 1] (true) valP <- PC + 9 (false)	-	Cnd <- cond(CC, ifun)	-	-	PC <- Cnd? valC : valP
call Dest	icode:ifun <- M1[PC] valC <- M8[PC + 1] valP <- PC + 9	valB <- R[rsp]	valE <- valB - 8	M8[valE] <- valP	R[rsp] <- valE	PC <- valC
ret	icode:ifun <- M1[PC]	valA <- R[rsp] valB <- R[rsp]	valE <- valB + 8	valM <- M8[valA]	R[rsp] <- valE	PC <- valM
pushq rA	icode:ifun <- M1[PC] rA:rB <- M1[PC + 1] valP <- PC + 2	valA <- R[rA] valB <- R[rsp]	valE <- valB - 8	M8[valE] <- valA	R[rsp] <- valE	PC <- valP
popq rA	icode:ifun <- M1[PC] rA:rB <- M1[PC + 1] valP <- PC + 2	valA <- R[rsp] valB <- R[rsp]	valE <- valB + 8	valM <- M8[valA]	R[rsp] <- valE R[rA] <- valM	PC <- valP

(ASSEMBLY CODE) .ys -> YAS -> generates .yo object file -> YIS -> simulates this .yo file and outputs

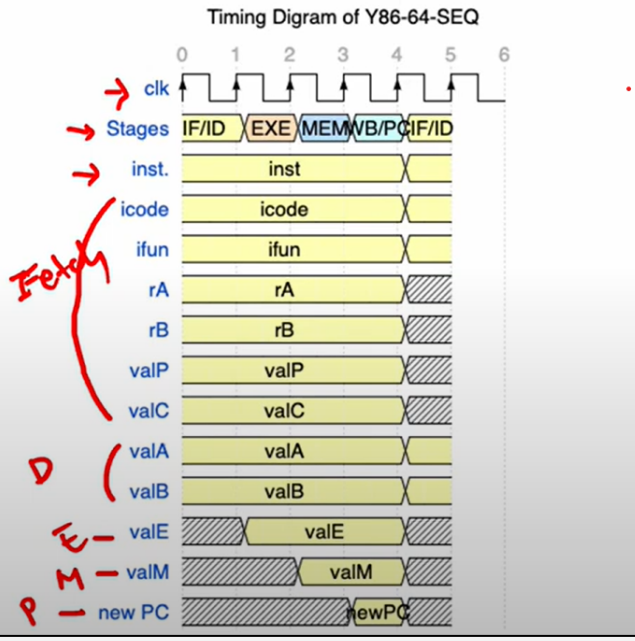
We will convert this .yo file into a logism memory file

Yo2mem: In a python environment, run the following command:

python yo2mem.py --yo <source_file_name>.yo --mem <memory_file_name>.mem

- Input to a certain module is not always available.
  - Ex) How many cycles to make input data to Write back?
- We need timings of inputs to each stage.
- Timing diagram depends on your design.
- Try to draw entire timing diagram in early stage.
  - Otherwise, you will have to revisit previous stages.

- Let's assume that
  - Every stage can finish its computation in a single cycle



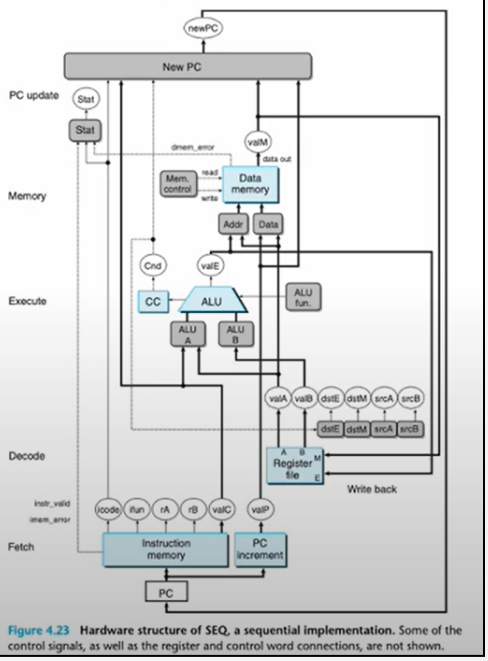
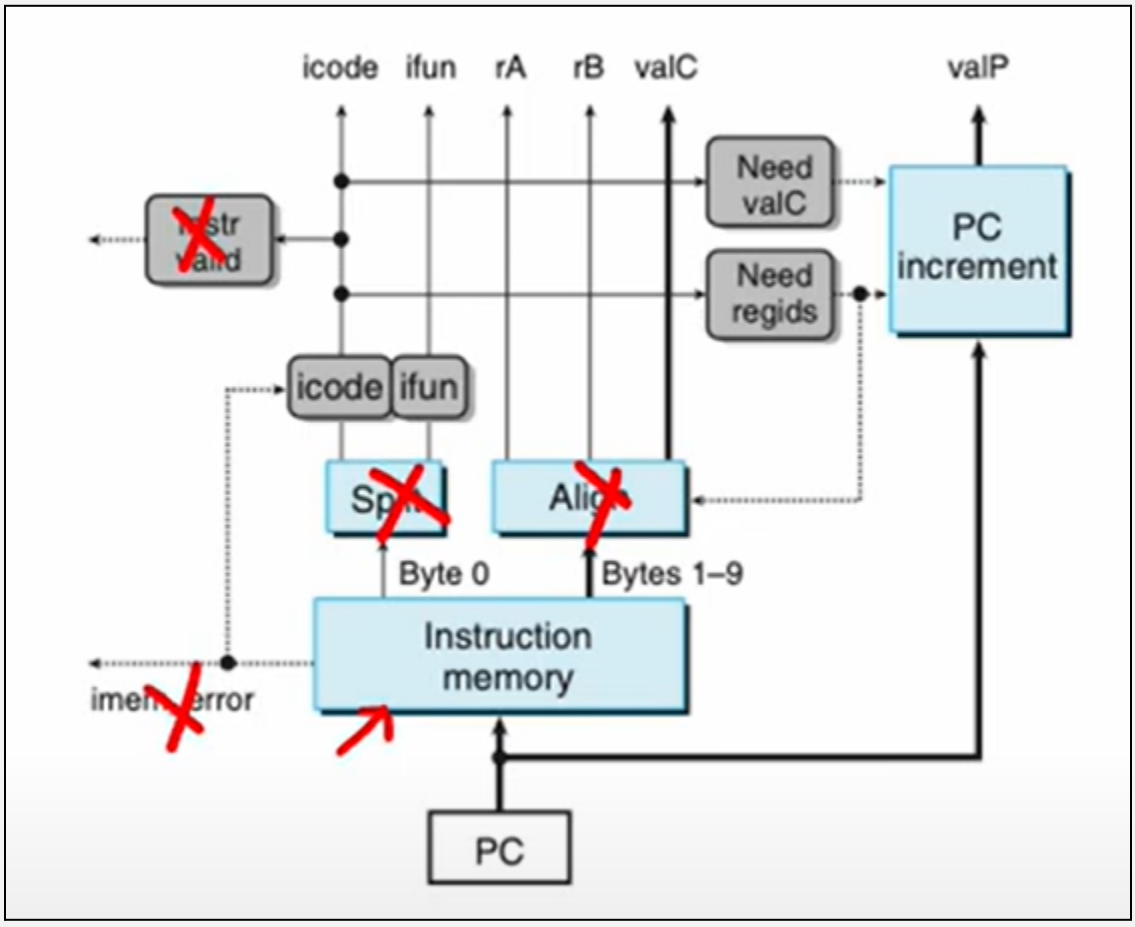
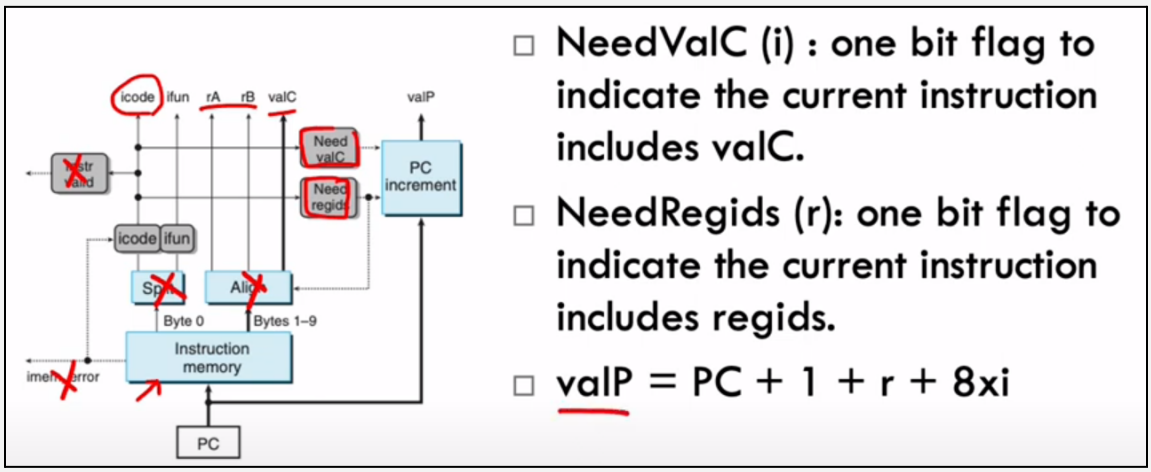


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

FETCH UNIT:

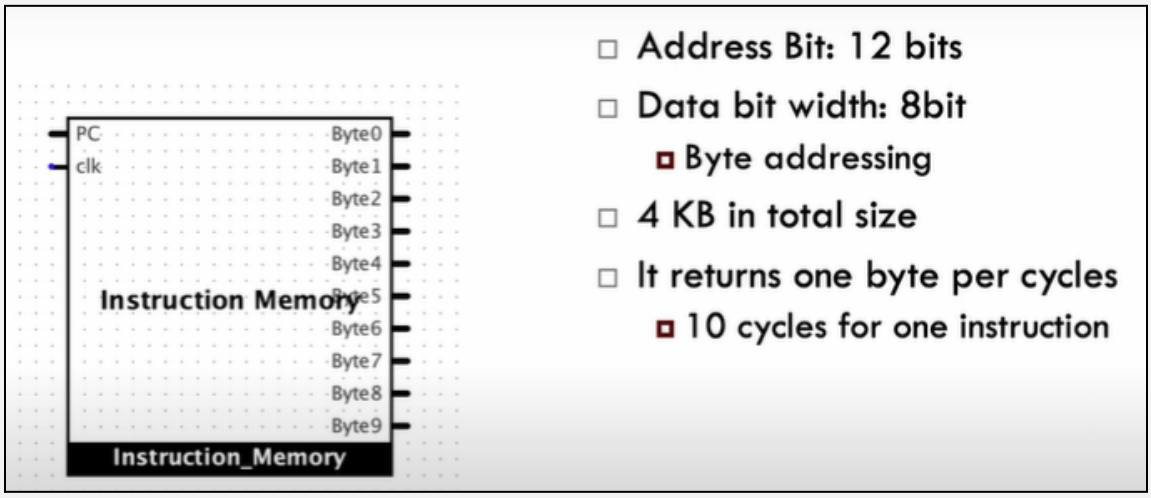
- Fetch an instruction (10 bytes) from memory using PC register
- Produce meaningful data/control signal from the instruction for later stages (icode, ifun, rA, rB, valC, valP)
- Calculate next PC value (valP)
- Blue boxes should contain registers whereas gray boxes are just control logic

- Red X's are things we can and should ignore (do not implement)
- 

- 

- We actually CANNOT IGNORE THE ALIGN BOX
- This is meant to take in the bytes 1-9 while the 0th bytes goes to icode and ifun.
- Align takes in 1-9 bytes and outputs rA, rB, and valC if necessary

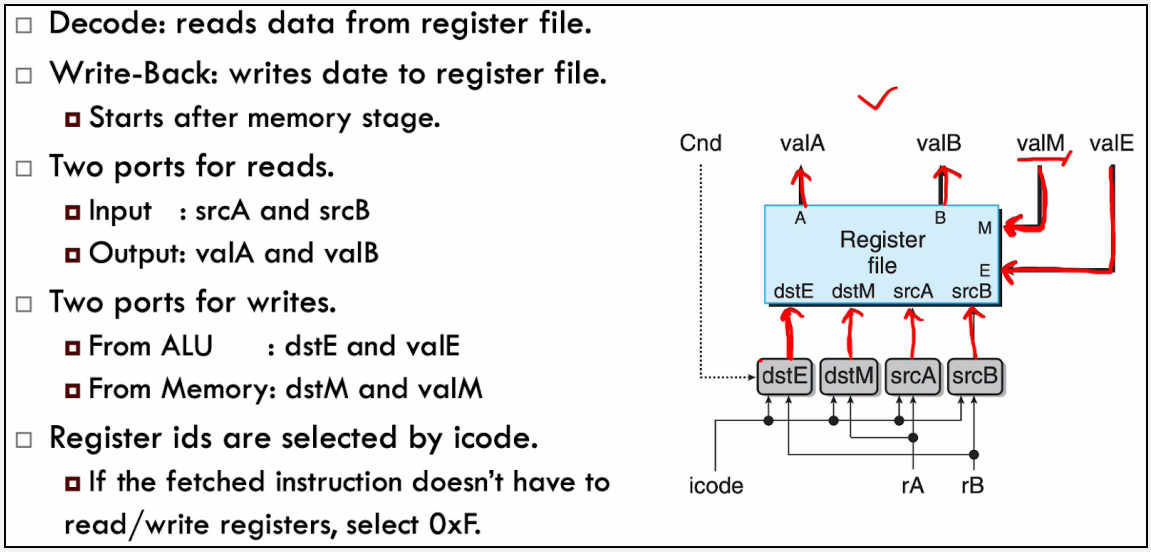
-



DECODE AND WRITE BACK:

- Share the same register file
- DECODE: - reads data
  - Inputs: srcA. srcB
  - Outputs: valA, valB
- WRITE BACK: - writes data
  - Input Address: dstE, dstM
  - Input Value:valE, valM
    - E is execute from ALU
    - M is from Memory
- Each input should have a controller to determine the right input based on icode

-

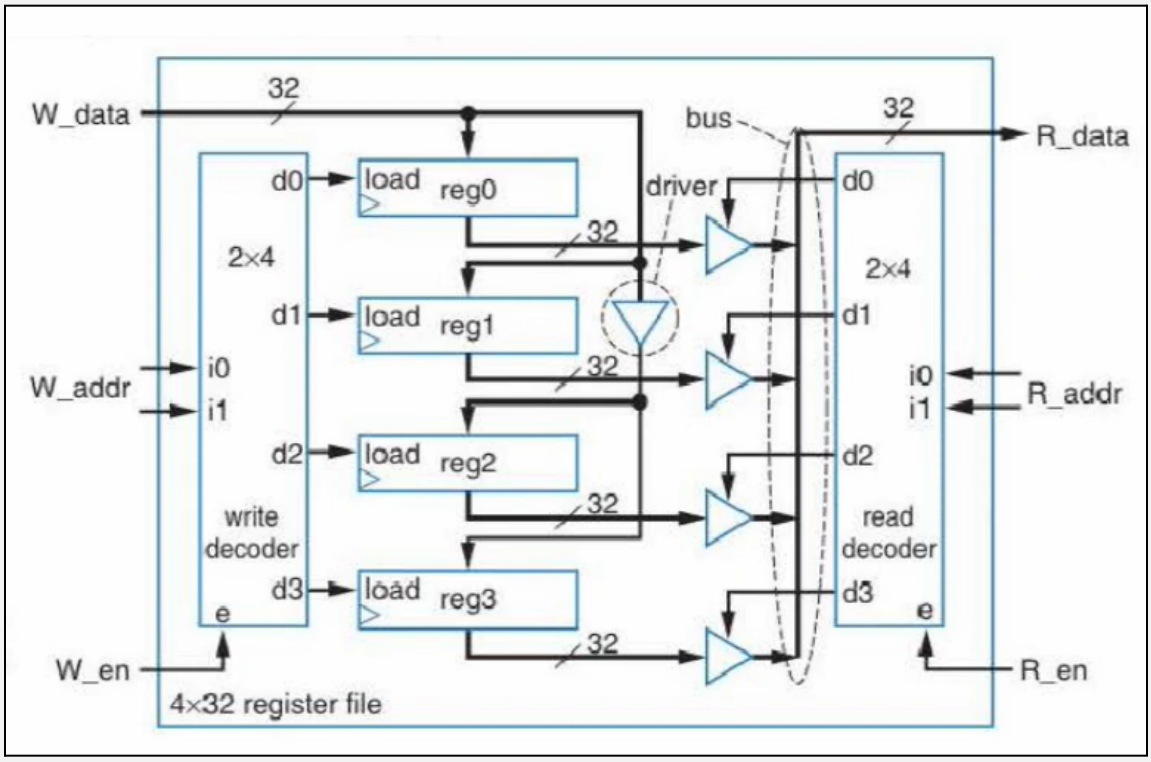


- ALL EMPTY SLOTS SHOULD CONTAIN 0xF - THIS IS BC our 16th register (f) should be a trash can

Instruction	***srcA*** rA / rsp / none	***srcB*** rB / rsp / none	***dstE*** rB / rsp / none	***dstM*** rA / rsp / none
halt				
nop				
r r movq rA, rB	rA	rB	rB	
i r movq V, rB			rB	
r m movq rA, D(rB)	rA	rB		
m r movq D(rB), rA		rB		rA
OPq rA, rB	rA	rB	rB	
jXX Dest				
call Dest		rsp	rsp	
ret	rsp	rsp	rsp	
pushq rA	rA	rsp	rsp	
popq rA	rsp	rsp	rsp	ra

-
- Implementing the Register File - note that this is a 4x32 reg file. I'm pretty sure ours should be 16x64
- Also, this is one read, one write
- Ours should be two read, two write

-

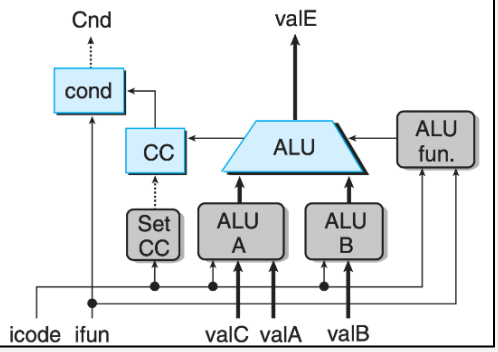


EXECUTE:

- ALU performs the operations (ADD, SUB, AND, XOR) based on icode and ifun

-

- Inputs:
  - ALU_fun: OP (0,1,2,3) ← ALU_fun(icode, ifun)
  - ALU_A: op1 ← (valC, **valA**, +8, -8, Z)
  - ALU_B: op2 ← (valB, 0, Z)
- Outputs: valE and ZF/SF/OF
  - valE: op2 OP op1
  - ZF/SF/OF



- Zero Flag (ZF)
  - The most recent operation yielded zero.
  - (t == 0)
- Sign Flag (SF)
  - The most recent operation yielded a negative value.
  - (t < 0)
- Overflow Flag (OF)
  - The most recent operation caused a two's-complement overflow.
    - Either negative or positive
  - (a < 0 == b < 0) && (t < 0 != a < 0)

- CC: stores the last Condition Code (ZF/SF/OF) - controlled by Set CC
- Set CC: only outputs 1 (update Condition) if icode is 6 (OPq)
- Condition: output 1 bit signal - used by jXX
- We only care about jmp, jle, jl, je, jn, jg, ge

Instruction		Synonym	Jump condition	Description
jmp	*Label*		1	Direct jump
jmp	**Operand*		1	Indirect jump
je	*Label*	jz	ZF	Equal / zero
jne	*Label*	jnz	~ZF	Not equal / not zero
js	*Label*		SF	Negative
jns	*Label*		~SF	Nonnegative
jg	*Label*	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge	*Label*	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl	*Label*	jnge	SF ^ OF	Less (signed <)
jle	*Label*	jng	(SF ^ OF) | ZF	Less or equal (signed <=)
ja	*Label*	jnbe	~CF & ~ZF	Above (unsigned >)
jae	*Label*	jnb	~CF	Above or equal (unsigned >=)
jb	*Label*	jnae	CF	Below (unsigned <)
jbe	*Label*	jna	CF | ZF	Below or equal (unsigned <=)

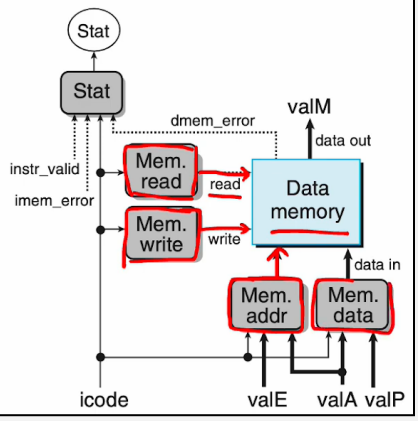
Instruction	***ALU A*** valA / valC / -8 / 8 / none	***ALU B*** valB / 0 / none	***alufun*** 0, 1, 2, 3	***set_cc*** 0 /1	***Condition*** (0 / 1/ none)
halt			0	0	
nop			0	0	

r r movq rA, rB	valA	0	0	0	
i r movq V, rB	valC	0	0	0	
r m movq rA, D(rB)	valC	valB	0	0	
m r movq D(rB), rA	valC	valB	0	0	
OPq rA, rB	valA	valB	ifun	1	
jXX Dest			0	0	Cond(CC, ifun)
call Dest	-8	valB	1	0	
ret	8	valB	0	0	
pushq rA	-8	valB	1	0	
popq rA	8	valB	0	0	

MEMORY:

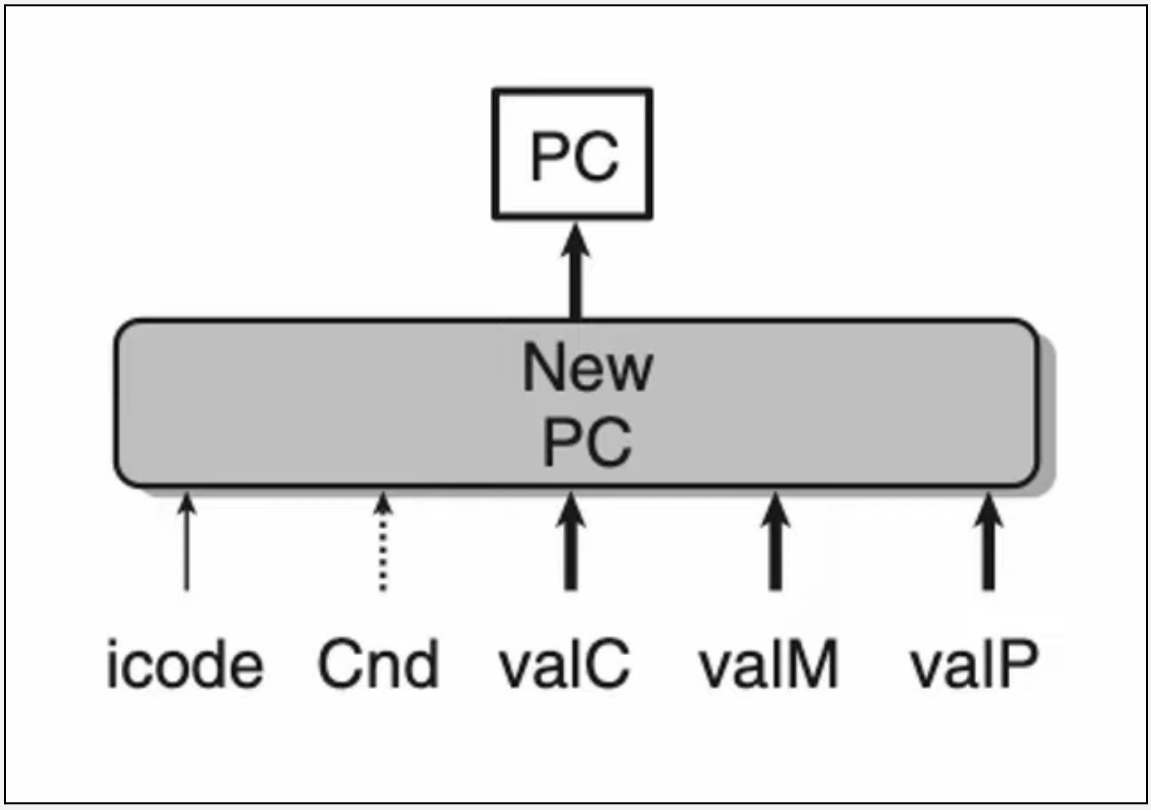
Instruction	***Memory Read*** 1: Read	***Memory Write*** 1: Write	***Memory Address*** valE /valA	***Memory Data*** valA / valP
halt				
nop				
r r movq rA, rB				
i r movq V, rB				
r m movq rA, D(rB)		1	valE	valA
m r movq D(rB), rA	1		valE	
OPq rA, rB				
jXX Dest				
call Dest		1	valE	valP
ret	1		valA	
pushq rA		1	valE	valA
popq rA	1		valA	

-
  - Data Memory: Read/Write data to/from RAM
    - Inputs: addr, data, read, write
    - Outputs: valM
    - Refer to Bryant Book 4.3 (Figure 4.30)
  - Mem read/write
    - Generate read/write control signal
  - Mem addr/data
    - Selects correct address and data line



PC UPDATE:

- New PC selects the next PC among valC, valM, and valP based on Cnd signal
- Write back starts at the same time with PC update

- 

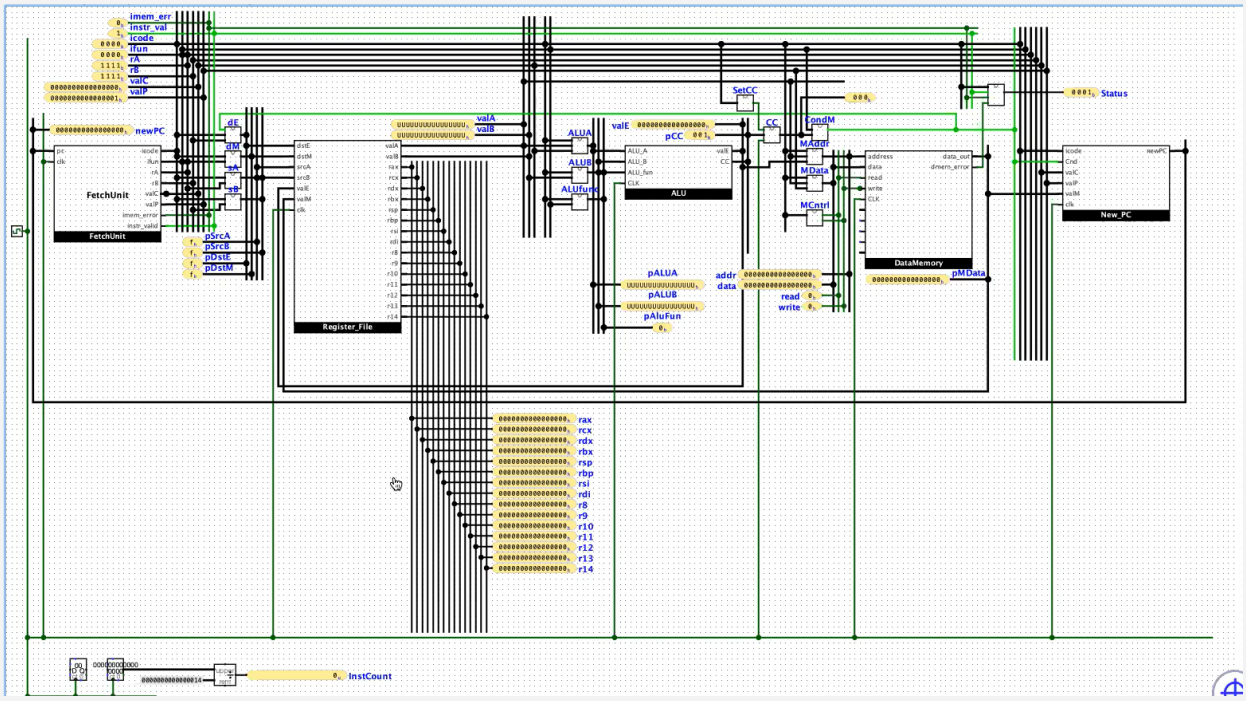
Instruction	***PC Update*** valC, valP, valM

halt	0
nop	valP
r r movq rA, rB	valP
i r movq V, rB	valP
r m movq rA, D(rB)	valP
m r movq D(rB), rA	valP
OPq rA, rB	valP
jXX Dest	valC / valP
call Dest	valC
ret	valM
pushq rA	valP
popq rA	valP

EXAMPLE
0x016: 30f4800000000000000

PC = 0x016
icode:ifun = M1[0x016] = 3:0 = irmovq
rA:rB = M1[0x017] = f:4 = 15:4
valC = M8[0x018] = …000080

- Imagine 8000000000000000 into groups of two = abcdefgh
- Then, reading it will become hgfedcba


imem_err
instr_val
icode
ifun
rA
rB
valC
valP
newPC
FetchUnit
dE
dM
sA
sB
pSrcA
pSrcB
pDstE
pDstM
Register_File
valA
valB
ALUA
ALUB
ALUfunE
ALU
valE
pCC
SetCC
CC
CondM
MAddr
MData
MCntrl
DataMemory
pMData
addr
data
read
write
pALUA
pALUB
pAluFun
Status
New_PC
rax
rcx
rdx
rbx
rsp
rbp
rsi
rdi
r8
r9
r10
r11
r12
r13
r14
InstCount