Y86-64 Sequential Processor Implementation

CSCE 312
Spring 2024

Thanks to Sungkeun Kim

Term project Overview

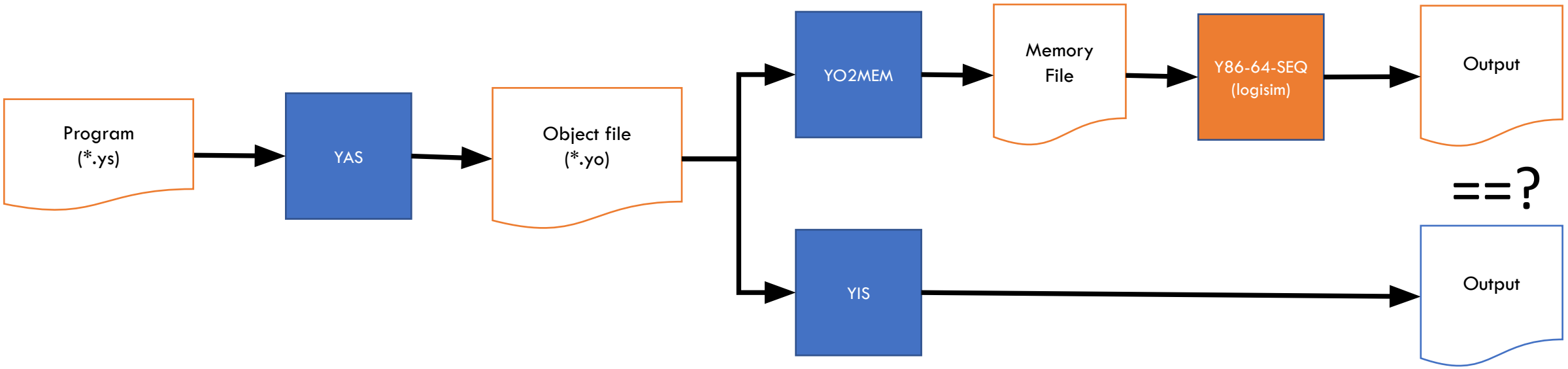
- Implement Y86 64-bit sequential processor (Y86-64-SEQ)
  - Bryant book 3rd edition - Chapter 4
  - The processor executes any assembly programs that YIS can execute
- Design tool: Logisim-Evolution
  - Version: v3.8.0

Details Regarding Project

- Extra credits(total 15, 5 in each) - only applicable to project
  - April 8 - Complete Fetch Stage
  - April 15 - Complete Decode & Execute Stage
  - April 22 - Complete Memory, Write-Back & PC Update Stage
- Probable Demo Date - April 30
  - Details will be announced later
- Check the demo & report guideline in Canvas

Let’s Get Started

Workflow


Program
(*.ys)
YAS
Object file
(*.yo)
YO2MEM
Memory
File
Y86-64-SEQ
(logisim)
Output
==?
YIS
Output

Programmer Visible State - Overview

- Programmer can read and modify some part of the processor state.
- We will use it for testing purpose.
  - Compare YIS's the visible states with your design

RF: Program registers

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

CC: Condition codes

ZF	SF	OF

Stat: Program status

PC

DMEM: Memory

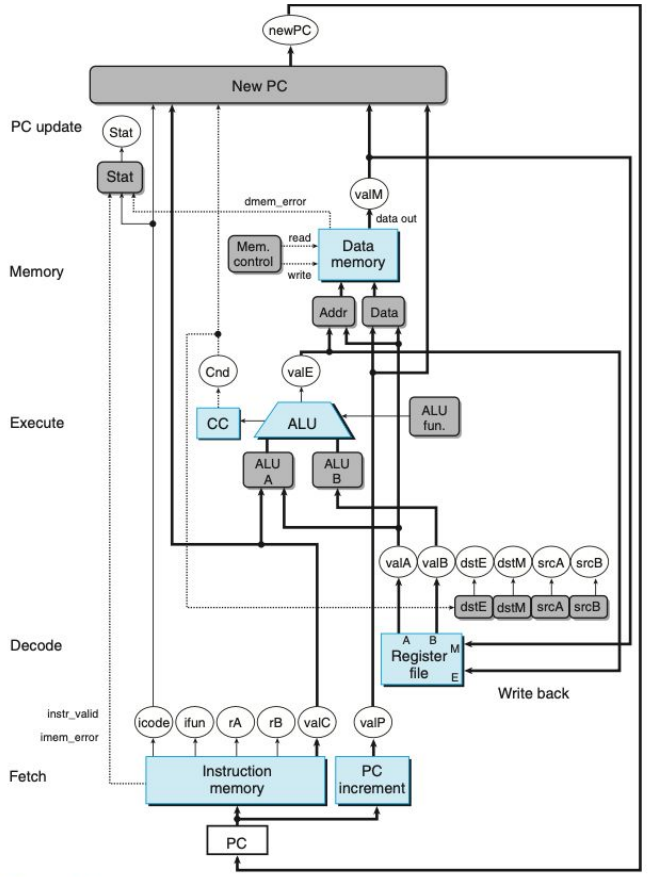


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Programmer Visible State – Program Counter

- Holds the address of the instruction currently being executed

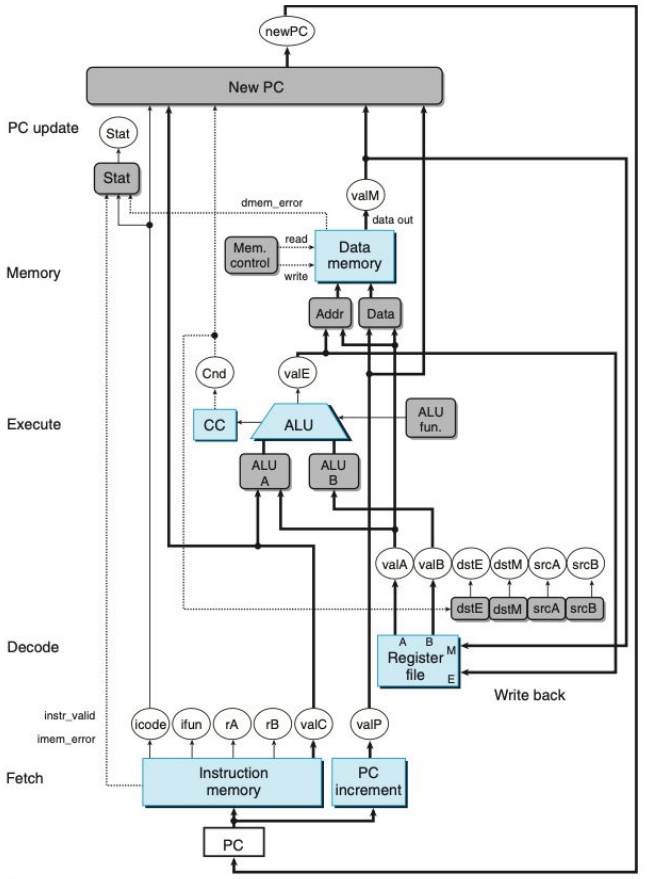


Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

Programmer Visible State – Register File

- 15 program register
- 64-bit word
- %rsp is used as a stack pointer by push, pop, call, ret

Number	Register name	Number	Register name
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	No register

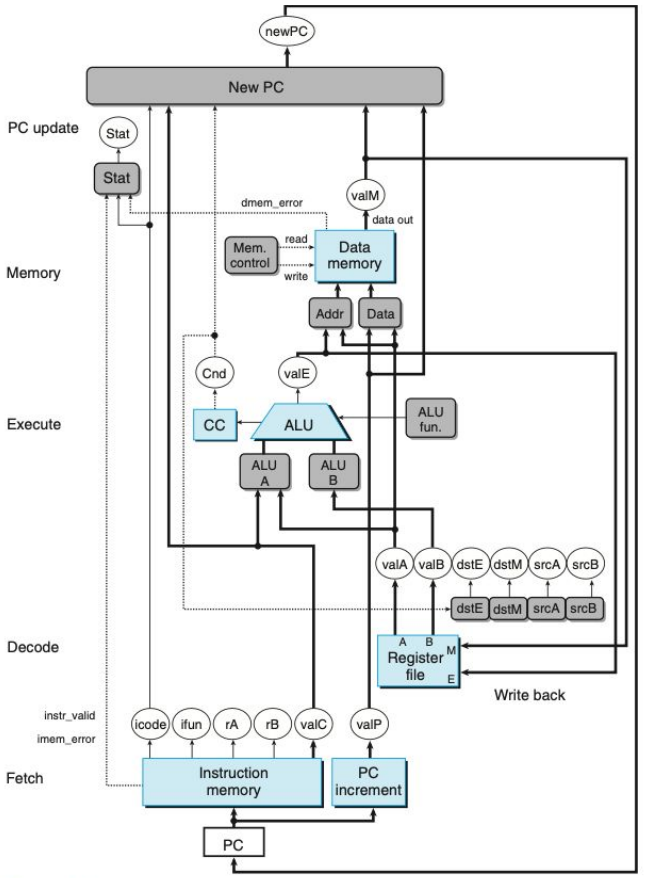


Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

Programmer Visible State – Condition Codes

- Stores the effect of the most recent arithmetic or logical instructions
  - ADD, SUB, AND, XOR
- Zero Flag (ZF)
  - The most recent operation yielded zero.
- Sign Flag (SF)
  - The most recent operation yielded a negative value.
- Overflow Flag (OF)
  - Caused a two's-complement overflow
    - Either negative or positive

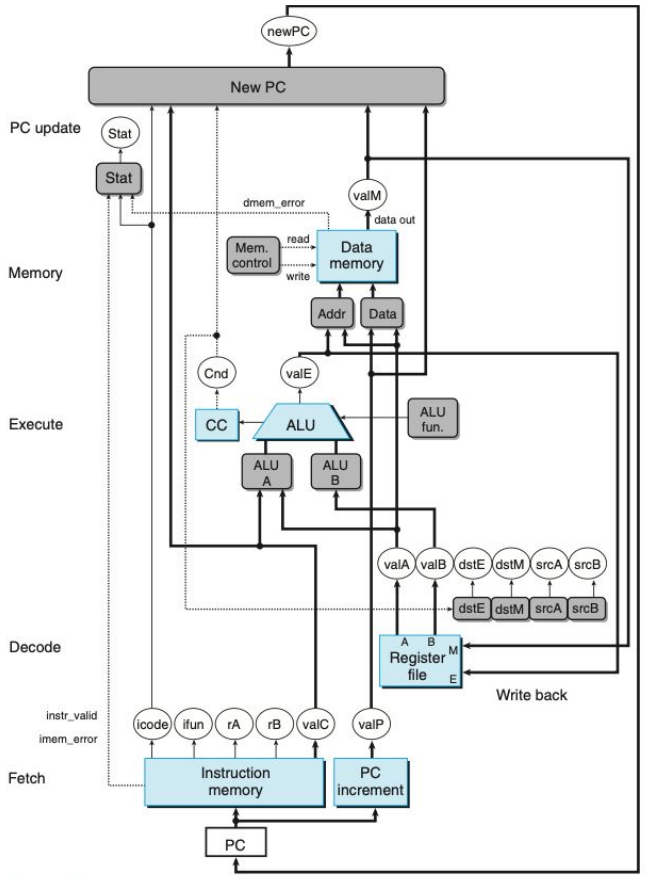


Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

Programmer Visible State – Memory

- Conceptually a large array of bytes
  - Holds both program (machine code) and data
- Instruction memory holds machine code
- Data memory holds program data

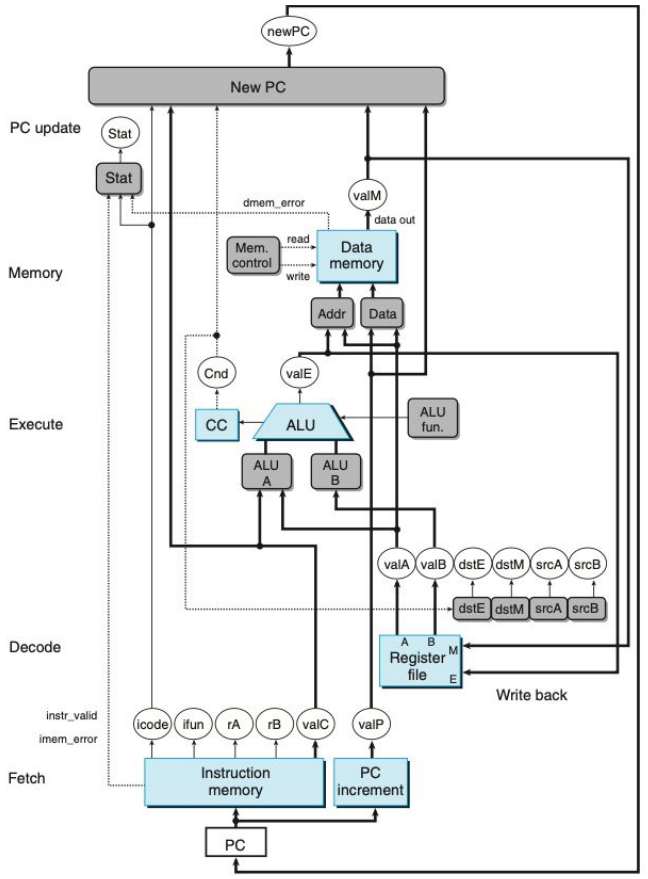


Figure 4.23 Hardware structure of SEQ, a sequential implementation. Some of the control signals, as well as the register and control word connections, are not shown.

Programmer Visible State – Program Status

- Indicates the overall state of program execution.
  - HLT (b0001)
    - Set if halt instruction encountered
  - INS (b0010)
    - Set if Invalid instruction encountered
  - IADR (b0100)
    - Set if Invalid Adrress encountered
  - DADR (b1000)
    - Set if Invalid Data memory address encountered
    - Both read/write signal are set
  - Can Happen more than one error.

Value	Name	Meaning
1	AOK	Normal operation
2	HLT	halt instruction encountered
3	ADR	Invalid address encountered
4	INS	Invalid instruction encountered

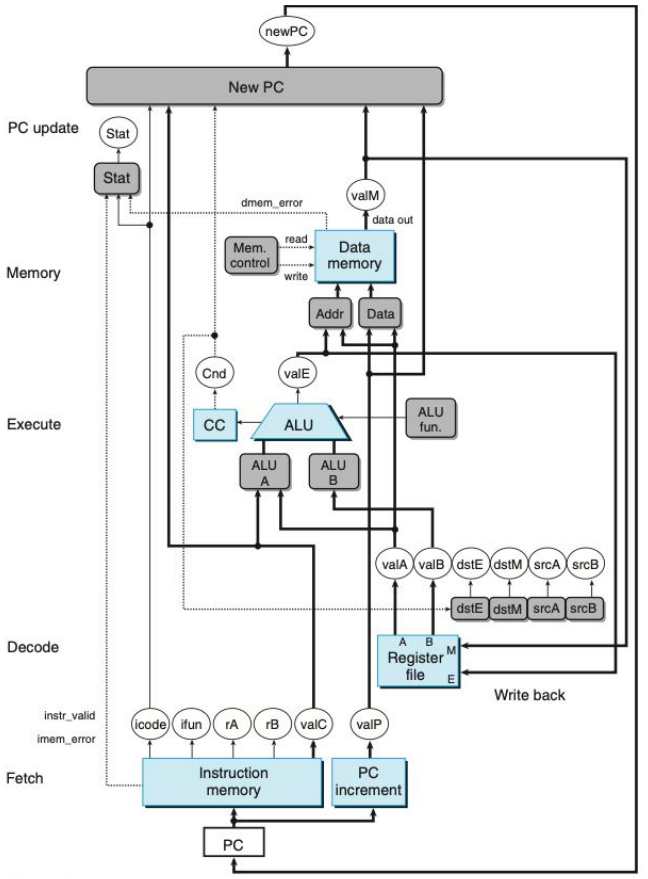


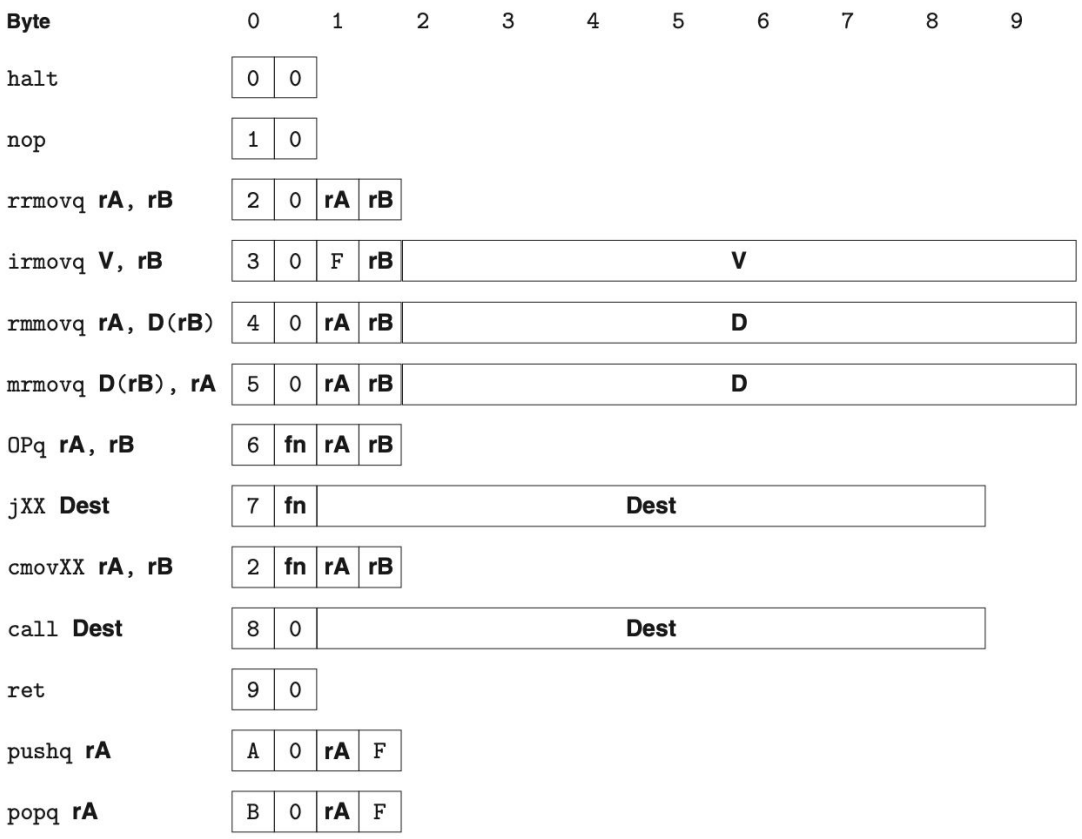
Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Instruction Set Architecture

Instruction Format

Operations			Branches				Moves				
addq	6	0	jmp	7	0	jne	7	4	rrmovq	2	0	cmovne	2	4
subq	6	1	jle	7	1	jge	7	5	cmovle	2	1	cmovge	2	5
andq	6	2	jl	7	2	jg	7	6	cmovl	2	2	cmovg	2	6
xorq	6	3	je	7	3				cmove	2	3			

- 13 Instructions in total
- Variable length
- First byte includes
  - Instruction code (icode)
  - Function code (ifun)
    - 38 different combination of icode and ifun



Organizing Processing into Stages

- Fetch
- Decode
- Execute
- Memory
- Write back
- PC Update

Computations on each stage

Stage	`rmmovq` rA, D(rB)	`mrmovq` D(rB), rA
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_8[PC + 2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_8[PC + 2] valP ← PC + 10
Decode	valA ← R[rA] valB ← R[rB]	valB ← R[rB]
Execute	valE ← valB + valC	valE ← valB + valC
Memory	M_8[valE] ← valA	valM ← M_8[valE]
Write back		R[rA] ← valM
PC update	PC ← valP	PC ← valP

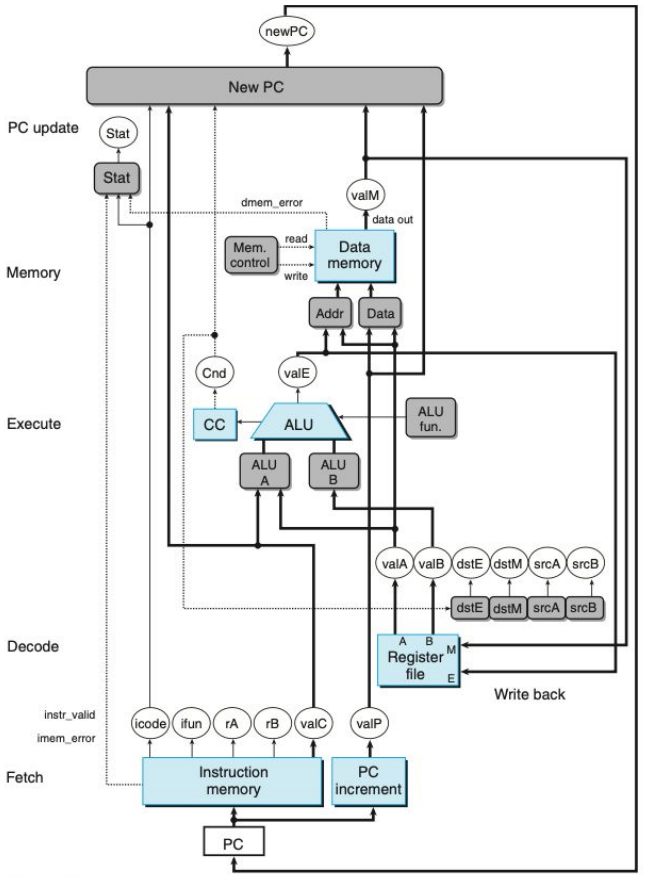


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Computations on each stage

- Recommend to review computations for all the instruction
  - Figure 4.18
  - Figure 4.19
  - Figure 4.20
  - Figure 4.21
  - Problem 4.17

Transformation of table

Stage	`rmmovq rA, D(rB)`	`mrmovq D(rB), rA`
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_8[PC + 2] valP ← PC + 10	icode:ifun ← M_1[PC] rA:rB ← M_1[PC + 1] valC ← M_8[PC + 2] valP ← PC + 10
Decode	valA ← R[rA] valB ← R[rB]	valB ← R[rB]
Execute	valE ← valB + valC	valE ← valB + valC
Memory	M_8[valE] ← valA	valM ← M_8[valE]
Write back		R[rA] ← valM
PC update	PC ← valP	PC ← valP

Instructions	FETCH	Instructions	Dec	Instructions	PC
rrmovq rA, rB		rrmovq rA, rB		rrmovq rA, rB	
irmovq V, rB		irmovq V, rB		irmovq V, rB	
rmmovq rA, D(rB)		rmmovq rA, D(rB)		rmmovq rA, D(rB)	
mrmovq D(rB), rA		mrmovq D(rB), rA		mrmovq D(rB), rA	
OPq rA, rB		OPq rA, rB		OPq rA, rB	
jXX Dest		jXX Dest		jXX Dest	
cmovXX rA, rB		cmovXX rA, rB		cmovXX rA, rB	
call Dest		call Dest		call Dest	
ret		ret		ret	
pushq rA		pushq rA		pushq rA	
pop rA		pop rA		pop rA	

Fetch Stage

Byte	0	1	2	3	4	5	6	7	8	9
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					

Instructions

rrmovq rA, rB

irmovq V, rB

rmmovq rA, D(rB)

icode:ifun ← M1[PC]
rA:rB ← M1[PC+1]
valC ← M8[PC + 2]
valP ← PC + 10

mrmovq D(rB), rA

icode:ifun ← M1[PC]
rA:rB ← M1[PC+1]
valC ← M8[PC + 2]
valP ← PC + 10

OPq rA, rB

jXX Dest

cmovXX rA, rB

call Dest

ret

pushq rA

pop rA

Fetch

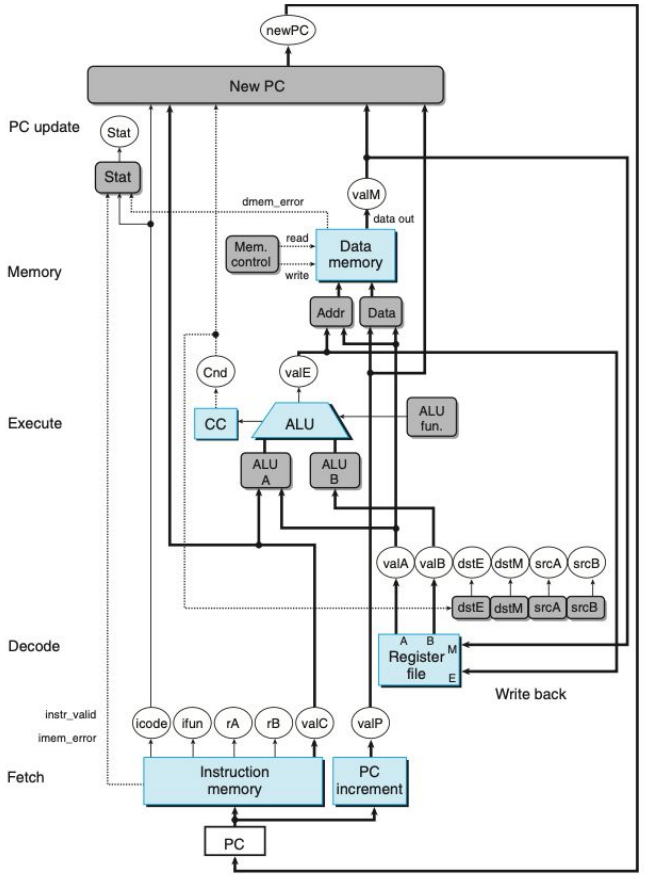


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Decode Stage

Byte	0	1	2	3	4	5	6	7	8	9
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					

Instructions	Decode
rrmovq rA, rB	
irmovq V, rB	
rmmovq rA, D(rB)	valA ⟵R[rA] valB ⟵R[rB]
mrmovq D(rB), rA	valB ⟵R[rB]
OPq rA, rB	
jXX Dest	
cmovXX rA, rB	
call Dest	
ret	
pushq rA	
pop rA	

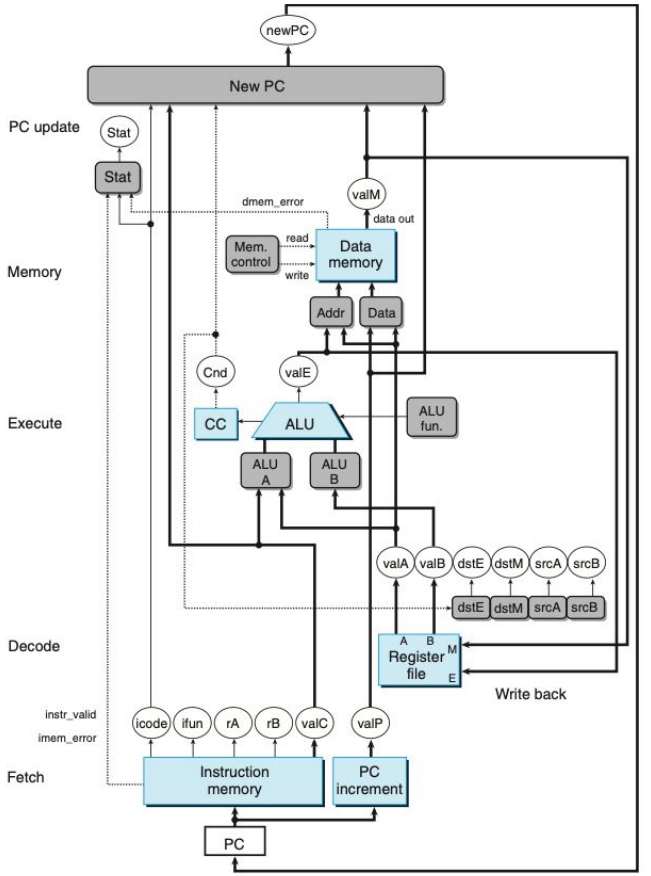


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Execute Stage

Byte	0	1	2	3	4	5	6	7	8	9
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					

Instructions	Execute
rrmovq rA, rB	
irmovq V, rB	
rmmovq rA, D(rB)	valE ⟵ valB + valC
mrmovq D(rB), rA	valE ⟵ valB + valC
OPq rA, rB	
jXX Dest	
cmovXX rA, rB	
call Dest	
ret	
pushq rA	
pop rA	

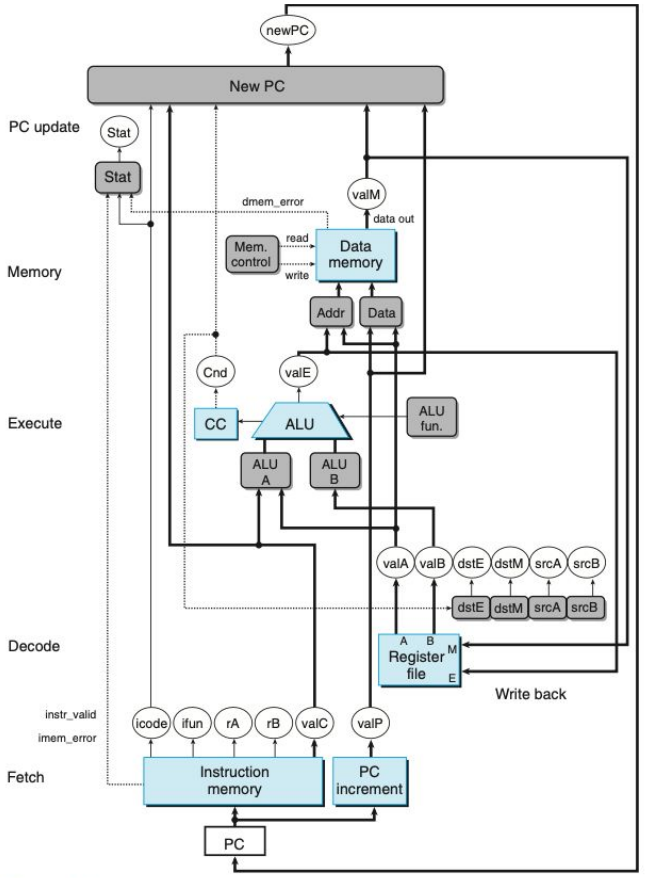


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Memory Stage

Byte	0	1	2	3	4	5	6	7	8	9
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					

Instructions	Memory
rrmovq rA, rB	
irmovq V, rB	
rmmovq rA, D(rB)	M_8[valE] ← valA
mrmovq D(rB), rA	valM ← M_8[valE]
OPq rA, rB	
jXX Dest	
cmovXX rA, rB	
call Dest	
ret	
pushq rA	
pop rA	

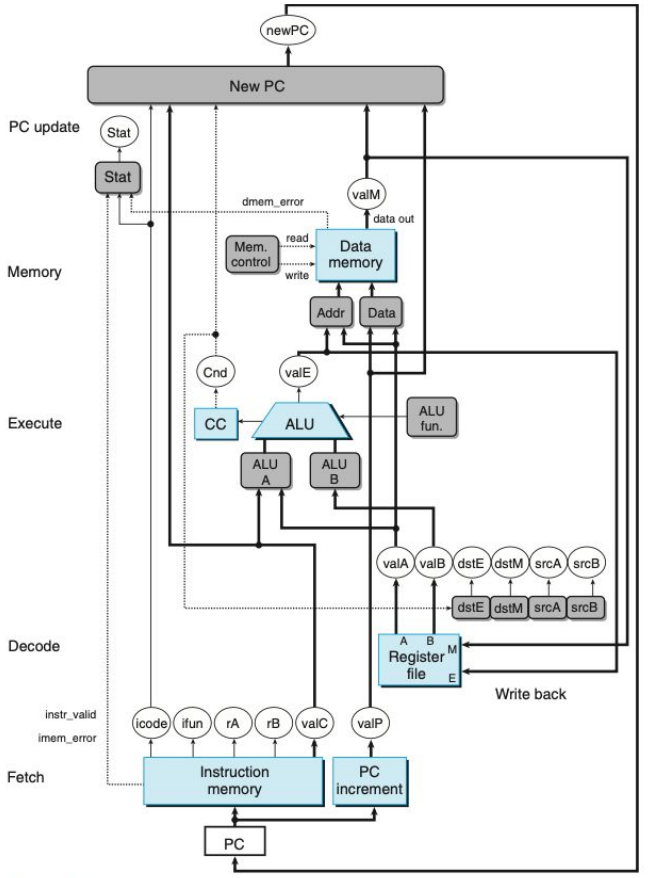


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Write back Stage

Byte	0	1	2	3	4	5	6	7	8	9
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					

Instructions	Write back
rrmovq rA, rB	
irmovq V, rB	
rmmovq rA, D(rB)	
mrmovq D(rB), rA	R[rA]⟵valM
OPq rA, rB	
jXX Dest	
cmovXX rA, rB	
call Dest	
ret	
pushq rA	
pop rA	

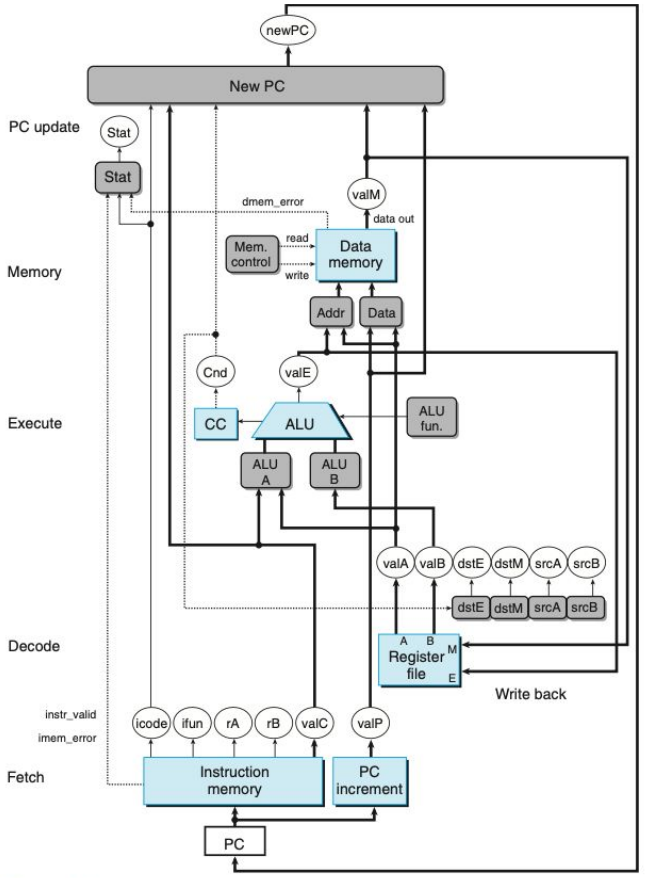


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

PC update Stage

Byte	0	1	2	3	4	5	6	7	8	9
rmmovq rA, D(rB)	4	0	rA	rB	D					
mrmovq D(rB), rA	5	0	rA	rB	D					

Instructions	PC update
rrmovq rA, rB	
irmovq V, rB	
rmmovq rA, D(rB)	PC←valP
mrmovq D(rB), rA	PC←valP
OPq rA, rB	
jXX Dest	
cmovXX rA, rB	
call Dest	
ret	
pushq rA	
pop rA	

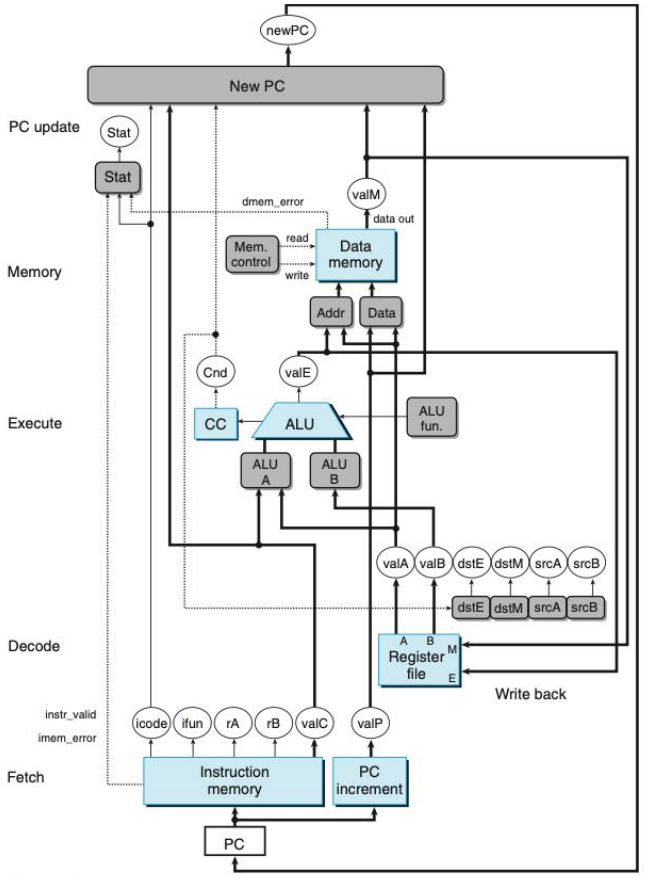


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Logisim Memory File Generation

What is logisim memory file?

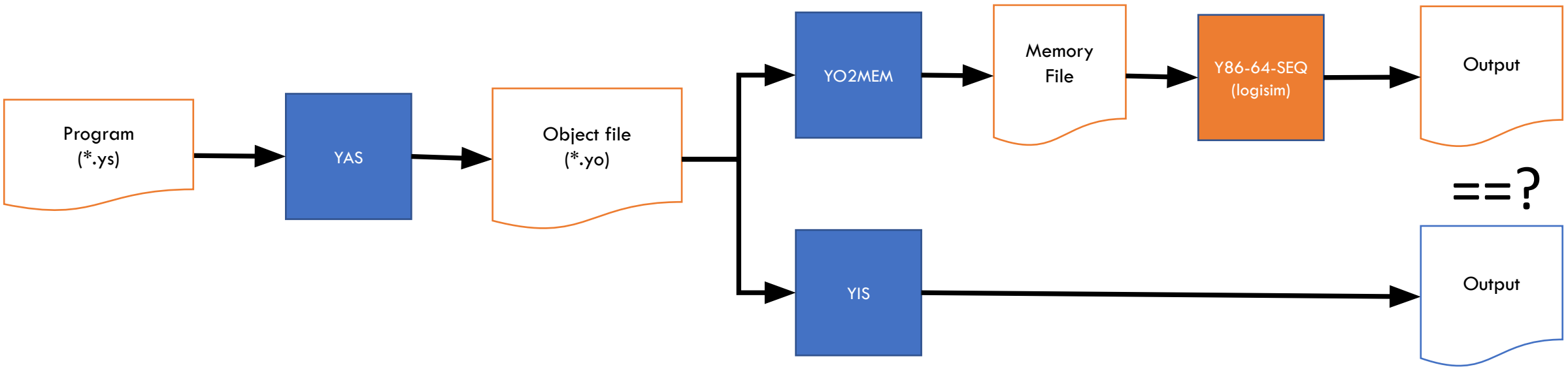
- The text file that contains memory data
- It follows format logisim understands

```
v3.0 hex words addressed
000: 30 f4 00 02 00 00 00 00 00 00
00a: 80 38 00 00 00 00 00 00 00
013: 00
018: 0d 00 0d 00 0d 00 00 00
020: c0 00 c0 00 c0 00 00 00
028: 00 0b 00 0b 00 0b 00 00
030: 00 a0 00 a0 00 a0 00 00
038: 30 f7 18 00 00 00 00 00 00 00
042: 30 f6 04 00 00 00 00 00 00 00
04c: 80 56 00 00 00 00 00 00 00
055: 90
056: 30 f8 08 00 00 00 00 00 00 00
060: 30 f9 01 00 00 00 00 00 00 00
06a: 63 00
06c: 62 66
06e: 70 87 00 00 00 00 00 00 00
077: 50 a7 00 00 00 00 00 00 00 00
081: 60 a0
083: 60 87
085: 61 96
087: 74 77 00 00 00 00 00 00 00
090: 90
```

Why do we need logisim file?

- We need test program loaded in memory.
  - We do not have Y86-64 assembler to generate executable.
  - We do not have operating system that loads the executable.
- We do have Y86-64 assembler (YAS) that generates .yo object file
  - YIS can simulates .yo and outputs the updates programmer visible state.
- We convert .yo file to logisim memory file so that your Y86-64 processor start execution of the program
- yo2mem script will be given

Workflow


Program
(*.ys)
YAS
Object file
(*.yo)
YO2MEM
Memory
File
Y86-64-SEQ
(logisim)
Output
==?
YIS
Output

Example: asum.yo to asum.mem

asum.yo

```
                        | # Execution begins at address 0
0x000:                  |       .pos 0
0x000: 30f40002000000000000 |       irmovq stack, %rsp      # Set up stack pointer
0x00a: 803800000000000000   |       call main               # Execute main program
0x013: 00                   |       halt                    # Terminate program
                        |
                        | # Array of 4 elements
0x018:                  |       .align 8
0x018: 0d000d000d000000     | array:        .quad 0x000d000d000d
0x020: c000c000c0000000     |       .quad 0x00c000c000c0
0x028: 000b000b000b0000     |       .quad 0x0b000b000b00
0x030: 00a000a000a00000     |       .quad 0xa000a000a000
                        |
0x038: 30f71800000000000000 | main: irmovq array,%rdi
0x042: 30f60400000000000000 |       irmovq $4,%rsi
0x04c: 805600000000000000   |       call sum                # sum(array, 4)
0x055: 90                   |       ret
                        |
                        | # long sum(long *start, long count)
                        | # start in %rdi, count in %rsi
0x056: 30f80800000000000000 | sum:  irmovq $8,%r8        # Constant 8
0x060: 30f90100000000000000 |       irmovq $1,%r9        # Constant 1
0x06a: 6300                 |       xorq %rax,%rax       # sum = 0
0x06c: 6266                 |       andq %rsi,%rsi       # Set CC
0x06e: 707000000000000000   |       jmp     test         # Goto test
0x077: 50a70000000000000000 | loop: mrmovq (%rdi),%r10   # Get *start
0x081: 60a0                 |       addq %r10,%rax       # Add to sum
0x083: 6087                 |       addq %r8,%rdi        # start++
0x085: 6196                 |       subq %r9,%rsi        # count--.  Set CC
0x087: 747700000000000000   | test: jne    loop          # Stop when 0
0x090: 90                   |       ret                  # Return
                        |
                        | # Stack starts here and grows to lower addresses
0x200:                  |       .pos 0x200
0x200:                  | stack:
```

yo2mem

```
v3.0 hex words addressed
000: 30 f4 00 02 00 00 00 00 00 00
00a: 80 38 00 00 00 00 00 00 00
013: 00
018: 0d 00 0d 00 0d 00 00 00
020: c0 00 c0 00 c0 00 00 00
028: 00 0b 00 0b 00 0b 00 00
030: 00 a0 00 a0 00 a0 00 00
038: 30 f7 18 00 00 00 00 00 00 00
042: 30 f6 04 00 00 00 00 00 00 00
04c: 80 56 00 00 00 00 00 00 00
055: 90
056: 30 f8 08 00 00 00 00 00 00 00
060: 30 f9 01 00 00 00 00 00 00 00
06a: 63 00
06c: 62 66
06e: 70 87 00 00 00 00 00 00 00
077: 50 a7 00 00 00 00 00 00 00 00
081: 60 a0
083: 60 87
085: 61 96
087: 74 77 00 00 00 00 00 00 00
090: 90
```

Timing of Each Stage

Keep the timing in your mind!

- 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.

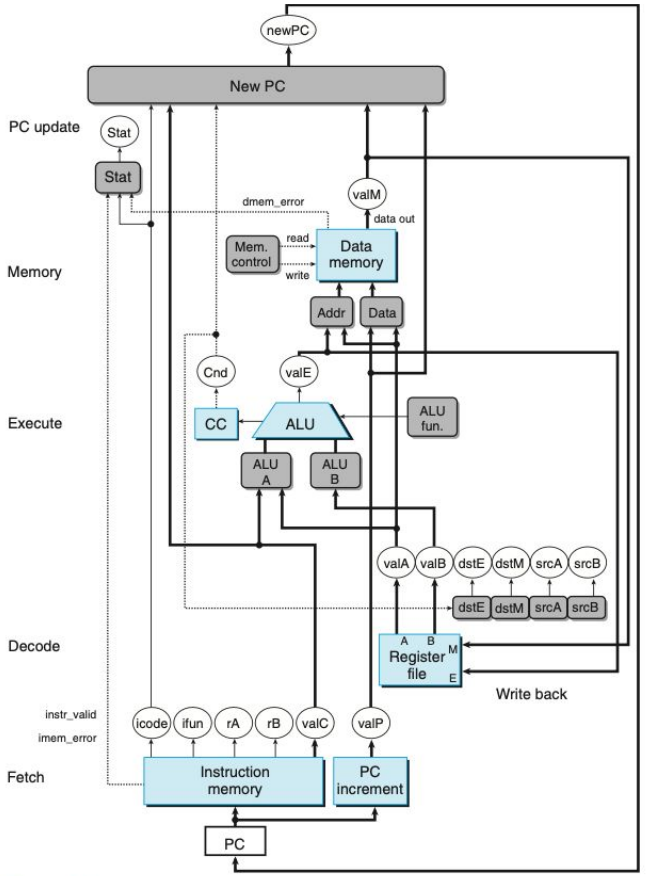
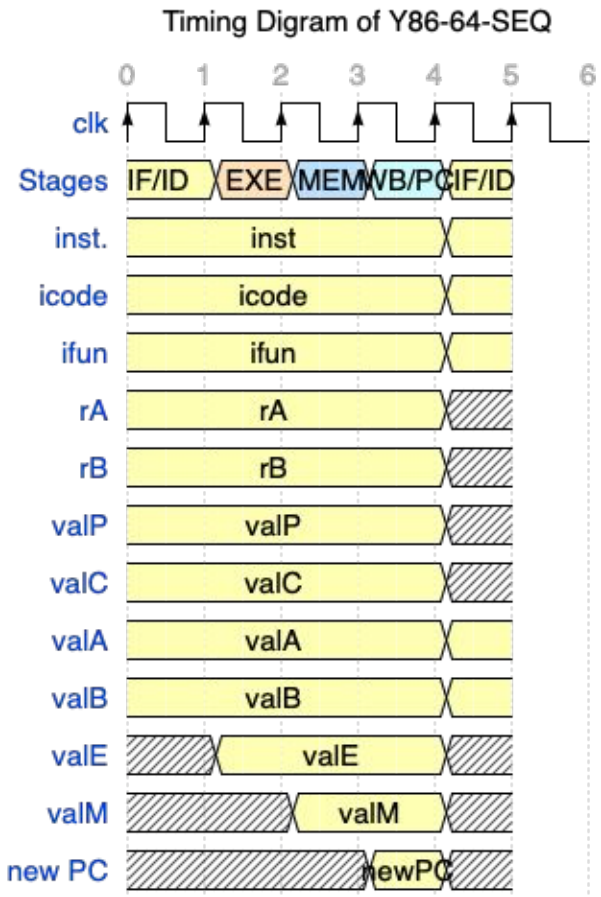


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Example

- Let's assume that, [COMPLICATED SO DO NOT FOLLOW THIS]
  - Every stage can finish its computation in a single cycle



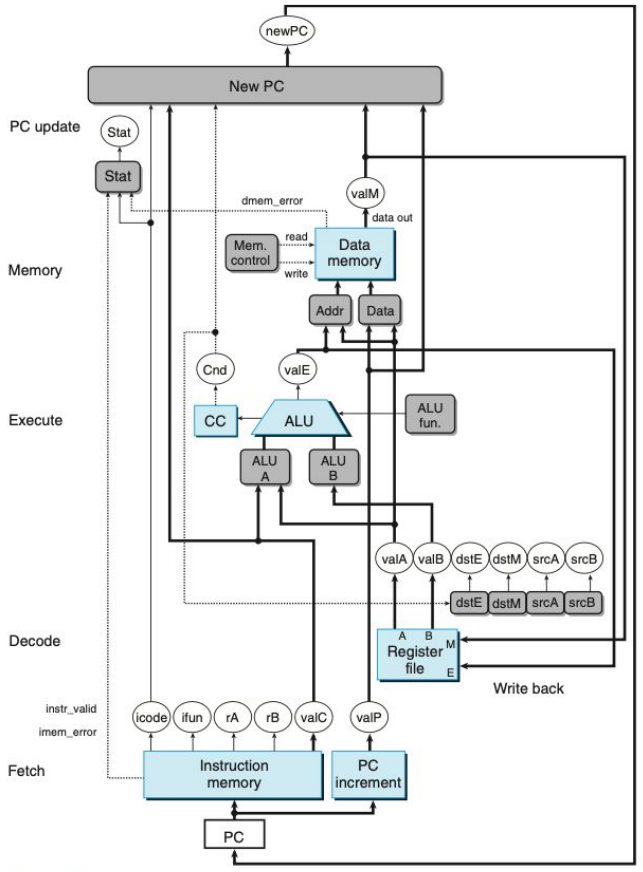


Figure 4.23 **Hardware structure of SEQ, a sequential implementation.** Some of the control signals, as well as the register and control word connections, are not shown.

Make your own timing diagram

- Useful website for drawing timing diagram
  - https://wavedrom.com

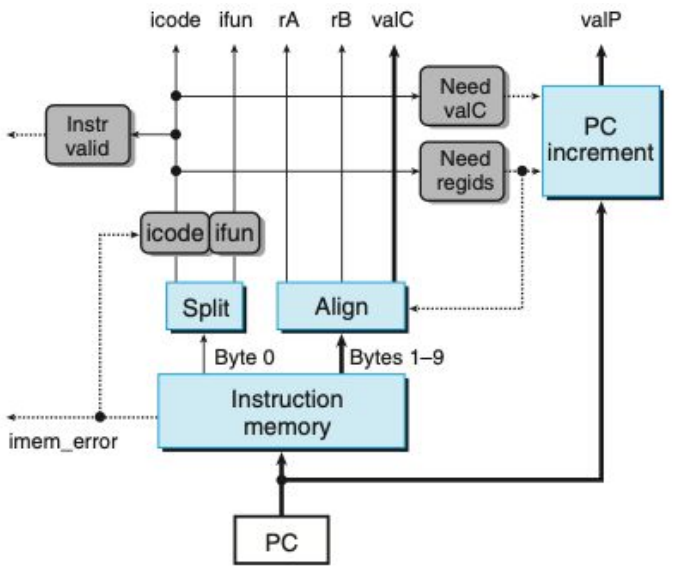
Fetch!!

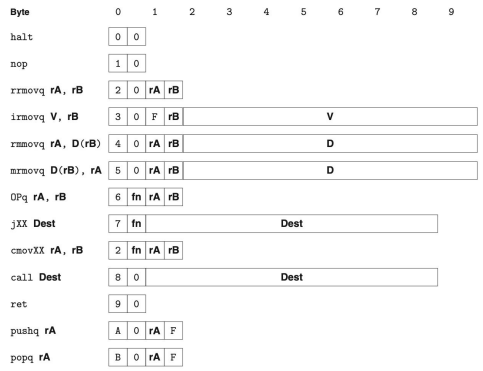
Fetch Unit

- 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
- Calculate a candidate of the next PC value (valP).
  - PC value is decided in PC Update stage.

Fetch Unit - details

Figure 4.27
SEQ fetch stage. Six bytes are read from the instruction memory using the PC as the starting address. From these bytes, we generate the different instruction fields. The PC increment block computes signal valP.

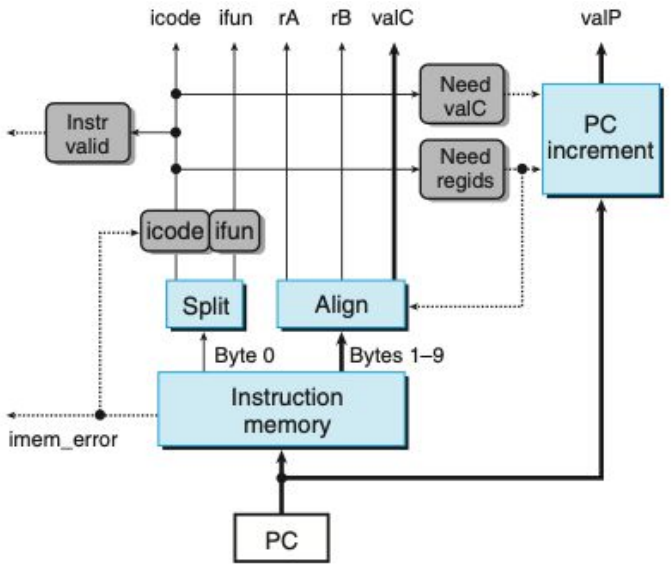




- NeedValC (i) : one bit flag to indicate the current instruction includes valC.
- NeedRegids (r): one bit flag to indicate the current instruction includes regids.
- valP = PC + 1 + r + 8xi

Fetch Unit - details

Figure 4.27
SEQ fetch stage. Six bytes are read from the instruction memory using the PC as the starting address. From these bytes, we generate the different instruction fields. The PC increment block computes signal valP.



Byte	0	1	2	3	4	5	6	7	8	9
`halt`	0	0								
`nop`	1	0								
`rrmovq` **rA, rB**	2	0	**rA**	**rB**						
`irmovq` **V, rB**	3	0	F	**rB**	**V**					
`rmmovq` **rA, D(rB)**	4	0	**rA**	**rB**	**D**					
`mrmovq` **D(rB), rA**	5	0	**rA**	**rB**	**D**					
`OPq` **rA, rB**	6	**fn**	**rA**	**rB**						
`jXX` **Dest**	7	**fn**	**Dest**							
`cmovXX` **rA, rB**	2	**fn**	**rA**	**rB**						
`call` **Dest**	8	0	**Dest**							
`ret`	9	0								
`pushq` **rA**	A	0	**rA**	F						
`popq` **rA**	B	0	**rA**	F						

Possible implementation approach

- Read 10 bytes from memory respective to PC address
  - You can consider using Counter to keep track of cycle number
- Get icode and ifun from byte 0
- Based on icode, generate NeedValC, NeedRegids, InstrValid
- Generate rA, rB, valC
- If any of rA or rB is not present, return 0xf
- Compute next PC candidate value

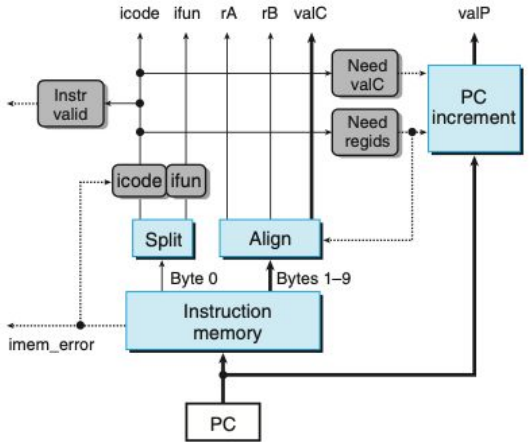
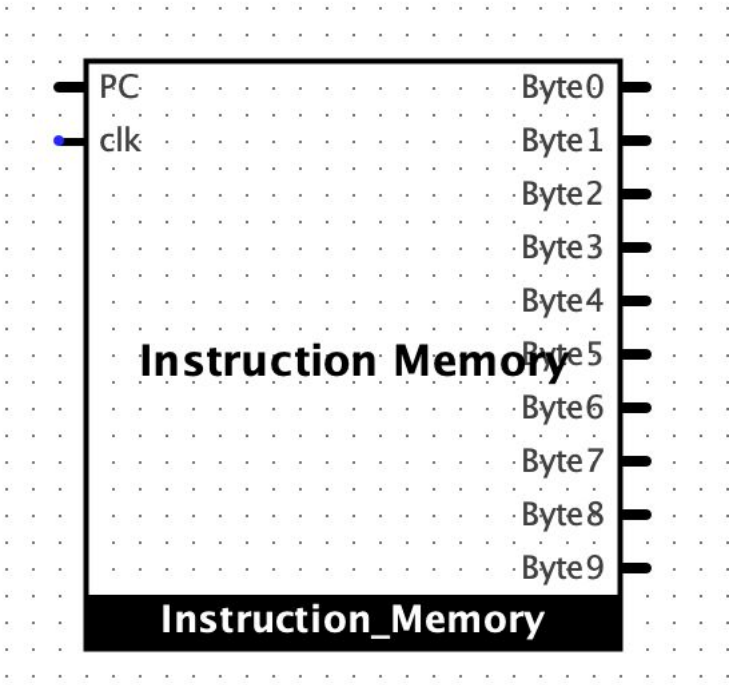


Figure 4.27
SEQ fetch stage. Six bytes are read from the instruction memory using the PC as the starting address. From these bytes, we generate the different instruction fields. The PC increment block computes signal valP.

Instruction Memory



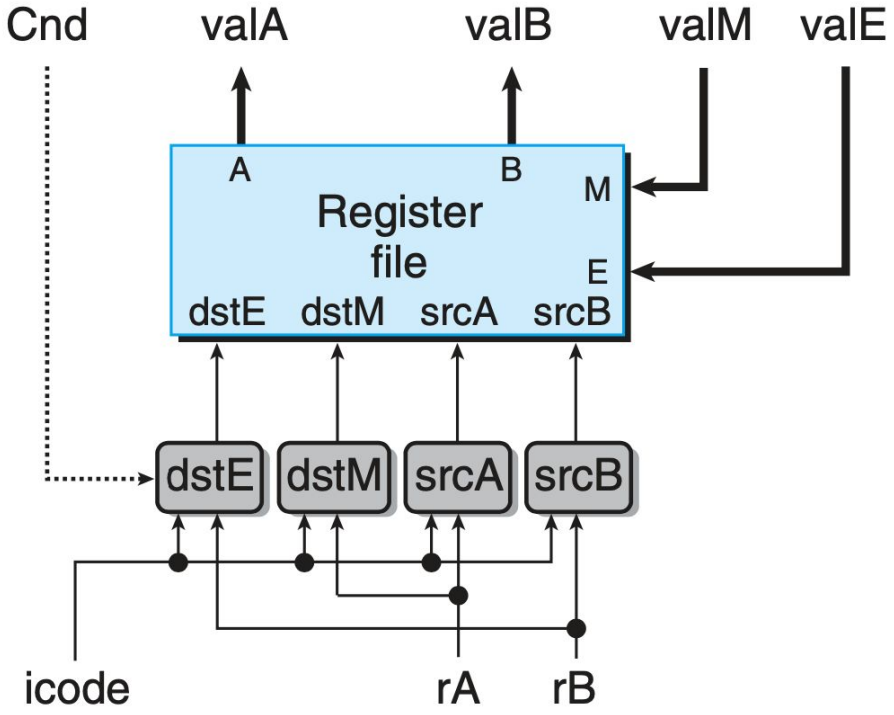
- Address Bit: 12 bits
- Data bit width: 8bit
  - Byte addressing
- 4 KB in total size
- It returns one byte per cycles
  - 10 cycles for one instruction

Decode and Write-Back!!

Decode and Write-Back

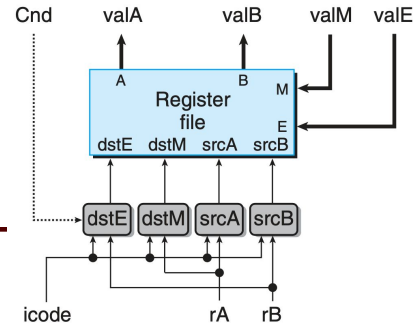
Number	Register name	Number	Register name
0	%rax	8	%r8
1	%rcx	9	%r9
2	%rdx	A	%r10
3	%rbx	B	%r11
4	%rsp	C	%r12
5	%rbp	D	%r13
6	%rsi	E	%r14
7	%rdi	F	No register

- Decode: reads data from register file.
- Write-Back: writes date to register file.
  - Starts after memory stage.
- Two ports for reads.
  - Input : srcA and srcB
  - Output: valA and valB
- Two ports for writes.
  - From ALU : dstE and valE
  - From Memory: dstM and valM
- Register ids are selected by icode.
  - If the fetched instruction doesn't have to read/write registers, select 0xF.



Stage	`rmmovq` rA, D(rB)	`mrmovq` D(rB), rA
Fetch	icode:ifun $\leftarrow$ M_1[PC]	icode:ifun $\leftarrow$ M_1[PC]
	rA:rB $\leftarrow$ M_1[PC + 1]	rA:rB $\leftarrow$ M_1[PC + 1]
	valC $\leftarrow$ M_8[PC + 2]	valC $\leftarrow$ M_8[PC + 2]
	valP $\leftarrow$ PC + 10	valP $\leftarrow$ PC + 10
Decode	valA $\leftarrow$ R[rA]	
	valB $\leftarrow$ R[rB]	valB $\leftarrow$ R[rB]
Execute	valE $\leftarrow$ valB + valC	valE $\leftarrow$ valB + valC
Memory	M_8[valE] $\leftarrow$ valA	valM $\leftarrow$ M_8[valE]
Write back		
		R[rA] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ valP

Circuits to generate src/dst registers.



Instructions	Decode	Write back
rrmovq rA, rB		
irmovq V, rB		
rmmovq rA, D(rB)	valA □ R[rA] valB □ R[rB]	N/A
mrmovq D(rB), rA	valB □ R[rB]	R[rA] □ valM
OPq rA, rB		
jXX Dest		
cmovXX rA, rB		
call Dest		
ret		
pushq rA		
pop rA		

Instructions	srcA rA/RRSP/RNONE	srcB rB/RRSP/RNONE	dstE rB/RRSP/RNONE	dstM rA/RNONE
halt				
nop				
rrmovq rA, rB				
irmovq V, rB				
rmmovq rA, D(rB)	rA	rB	0xF	0xF
mrmovq D(rB), rA	0xF	rB	0xF	rA
OPq rA, rB				
jXX Dest				
cmovXX rA, rB				
call Dest				
ret				
pushq rA				
popq rA				

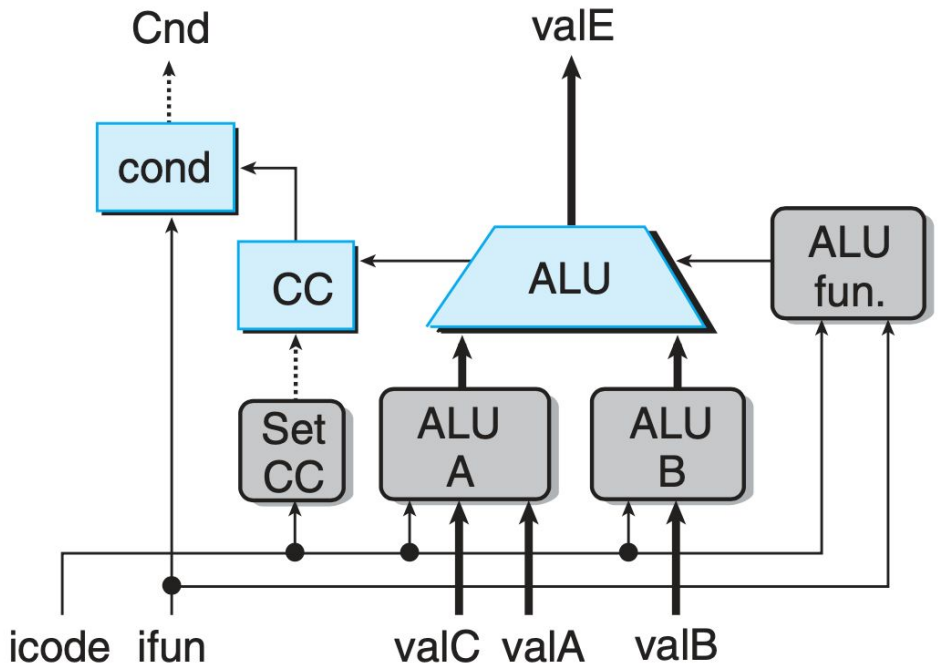
Execute!!

Units in Execution Stage

Operations			Branches				Moves					
addq	6	0	jmp	7	0	jne	7	4	rrmovq	2	0	cmovne	2	4
subq	6	1	jle	7	1	jge	7	5	cmovle	2	1	cmovge	2	5
andq	6	2	jl	7	2	jg	7	6	cmovl	2	2	cmovg	2	6
xorq	6	3	je	7	3				cmove	2	3			

Figure 4.3 Function codes for Y86-64 instruction set. The code specifies a particular integer operation, branch condition, or data transfer condition. These instructions are shown as `OPq`, `jXX`, and `cmovXX` in Figure 4.2.

- ALU: Performs the operations (ADD, SUB, AND, XOR)
- 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



ZF/SF/OF

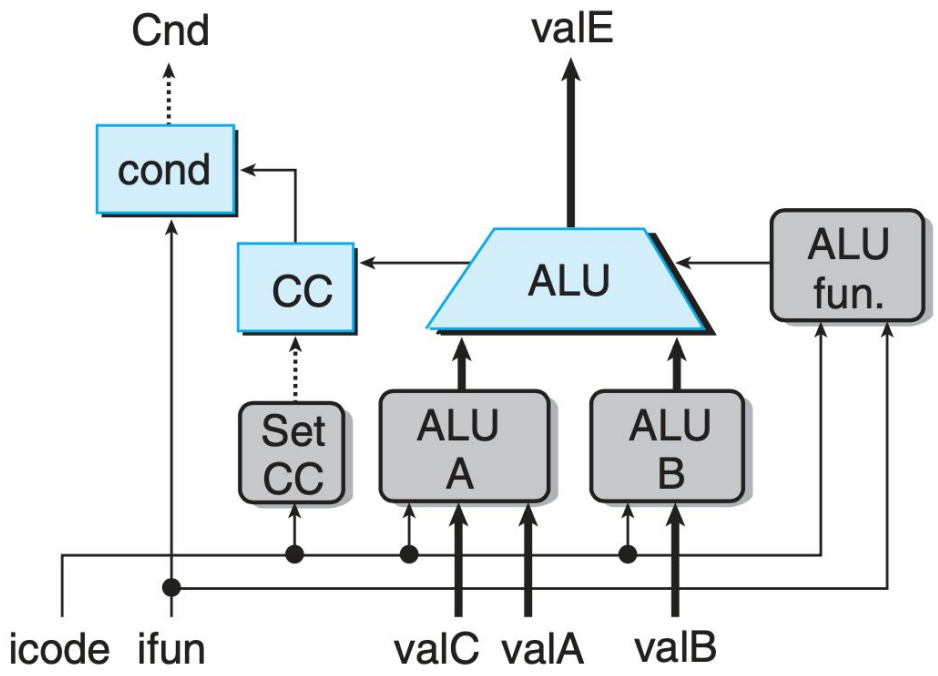
- For t = a + b
- Zero Flag (ZF)
  - The most recent operation yielded zero.
  - (t == 0)
- Sign Flag (SF)
  - The most recent operation yielded a negative value.
  - (t < 0)
- Overflow Flag (OF)
  - The most recent operation caused a two's-complement overflow.
    - Either negative or positive
  - (a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

CC module

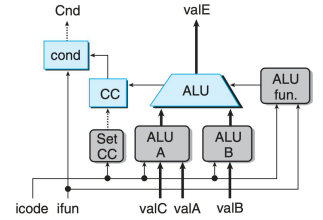
Operations			Branches				Moves					
addq	6	0	jmp	7	0	jne	7	4	rrmovq	2	0	cmovne	2	4
subq	6	1	jle	7	1	jge	7	5	cmovle	2	1	cmovge	2	5
andq	6	2	jl	7	2	jg	7	6	cmovl	2	2	cmovg	2	6
xorq	6	3	je	7	3				cmove	2	3			

Figure 4.3 **Function codes for Y86-64 instruction set.** The code specifies a particular integer operation, branch condition, or data transfer condition. These instructions are shown as `OPq`, `jXX`, and `cmovXX` in Figure 4.2.

- CC: stores the last Condition Code (ZF/SF/OF)
  - Controlled by Set CC
- SetCC
  - If the current icode is 6, outputs 1 to update C
- Cond: outputs 1 bit signal
  - Used by jXX in NewPC, cmovXX in WriteBack



Cond module



Operations			Branches				Moves					
addq	6	0	jmp	7	0	jne	7	4	rrmovq	2	0	cmovne	2	4
subq	6	1	jle	7	1	jge	7	5	cmovle	2	1	cmovge	2	5
andq	6	2	jl	7	2	jg	7	6	cmovl	2	2	cmovg	2	6
xorq	6	3	je	7	3				cmove	2	3			

Figure 4.3 **Function codes for Y86-64 instruction set.** The code specifies a particular integer operation, branch condition, or data transfer condition. These instructions are shown as `OPq`, `jXX`, and `cmovXX` in Figure 4.2.

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 <=)

Figure 3.15 **The jump instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

Instruction		Synonym	Move condition	Description
`cmove`	*S*, *R*	`cmovz`	`ZF`	Equal / zero
`cmovne`	*S*, *R*	`cmovnz`	`~ZF`	Not equal / not zero
`cmovs`	*S*, *R*		`SF`	Negative
`cmovns`	*S*, *R*		`~SF`	Nonnegative
`cmovg`	*S*, *R*	`cmovnle`	`~(SF ^ OF) & ~ZF`	Greater (signed >)
`cmovge`	*S*, *R*	`cmovnl`	`~(SF ^ OF)`	Greater or equal (signed >=)
`cmovl`	*S*, *R*	`cmovnge`	`SF ^ OF`	Less (signed <)
`cmovle`	*S*, *R*	`cmovng`	`(SF ^ OF) | ZF`	Less or equal (signed <=)
`cmova`	*S*, *R*	`cmovnbe`	`~CF & ~ZF`	Above (unsigned >)
`cmovae`	*S*, *R*	`cmovnb`	`~CF`	Above or equal (Unsigned >=)
`cmovb`	*S*, *R*	`cmovnae`	`CF`	Below (unsigned <)
`cmovbe`	*S*, *R*	`cmovna`	`CF | ZF`	Below or equal (unsigned <=)

Figure 3.18 **The conditional move instructions.** These instructions copy the source value S to its destination R when the move condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

ALU A and ALU B

Operations			Branches				Moves					
addq	6	0	jmp	7	0	jne	7	4	rrmovq	2	0	cmovne	2	4
subq	6	1	jle	7	1	jge	7	5	cmovle	2	1	cmovge	2	5
andq	6	2	jl	7	2	jg	7	6	cmovl	2	2	cmovg	2	6
xorq	6	3	je	7	3				cmove	2	3			

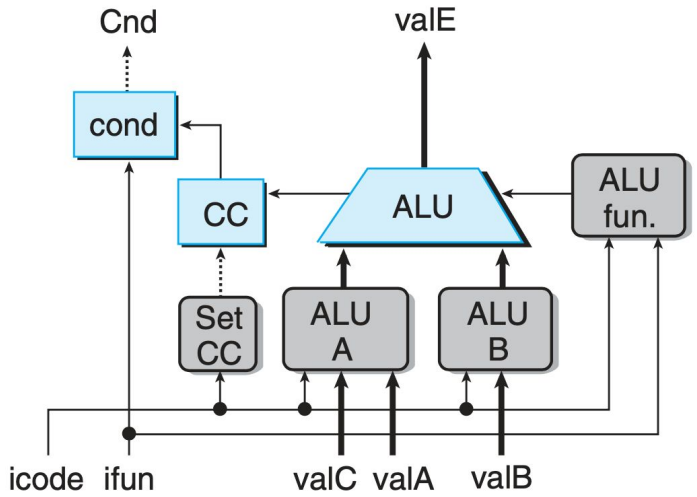
Figure 4.3 Function codes for Y86-64 instruction set. The code specifies a particular integer operation, branch condition, or data transfer condition. These instructions are shown as `OPq`, `jXX`, and `cmovXX` in Figure 4.2.

Instructions	Execute
rrmovq rA, rB	
irmovq V, rB	
rmmovq rA, D(rB)	valE □ valB + valC
mrmovq D(rB), rA	valE □ valB + valC
OPq rA, rB	valE □ valB OP valA
jXX Dest	Cnd □ Cond (CC, ifun)
cmovXX rA, rB	
call Dest	
ret	
pushq rA	
pop rA	

Instructions	ALU_A valA/valC/-8/8/X	ALU_B valB/0/X(don't care)	alufun 0,1,2,3	set_cc 1/0	Cnd (0/1/x)
halt					
nop					
rrmovq rA, rB					
irmovq V, rB					
rmmovq rA, D(rB)	valC	valB	0	0	x
mrmovq D(rB), rA	valC	valB	0	0	x
OPq rA, rB	valA	valB	ifun	1	x
jXX Dest	x	x	0	0	Cond(CC, ifun)
cmovXX rA, rB					
call Dest					
ret					
pushq rA					
popq rA					

Possible implementation approach

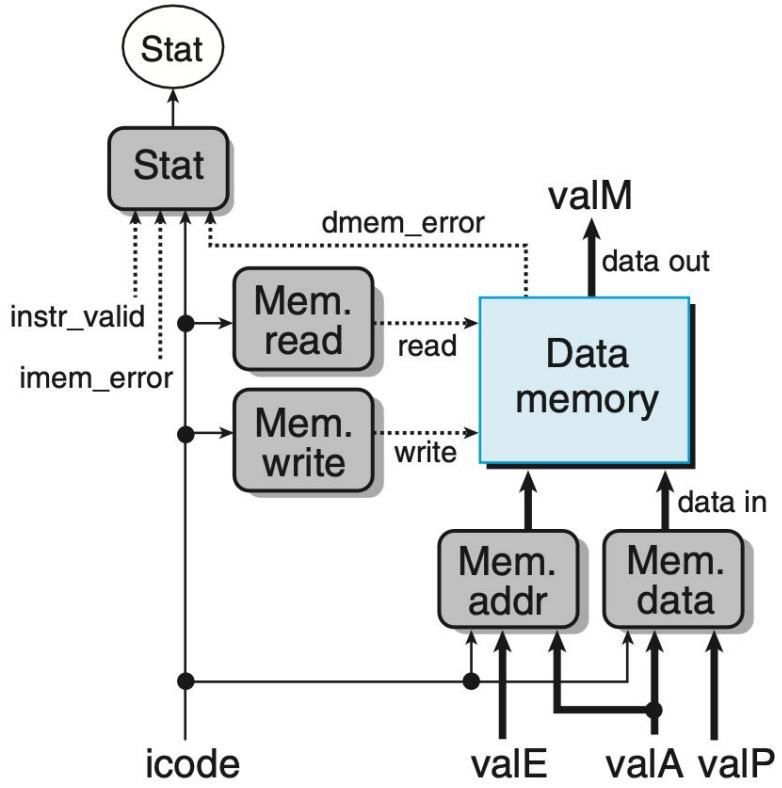
- Based on icode, figure out ALU A and ALU B values
- If icode=6(OPq), choose operations to do based on ifunc, otherwise default operation is addition
- Execute the operation and output will be valE
- If icode=6(OPq), set the CC and update the cond module



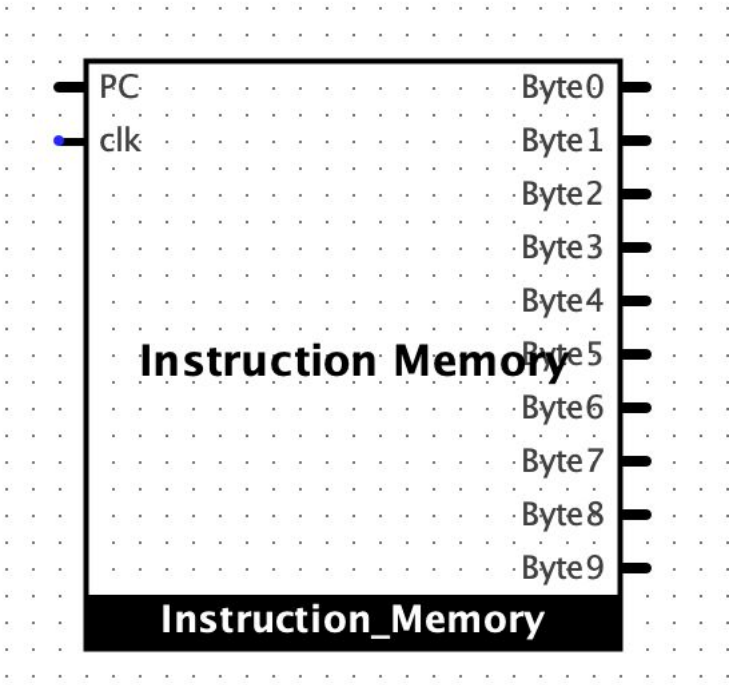
Memory!!

Units in Memory Stage

- 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



Data Memory



- Same with Instruction Memory
  - Address Bit: 12 bits
  - Data bit width: 8bit
  - Byte addressing
  - 4 KB in total size
- It returns one byte per cycles
  - 8 cycles for quad word (8 bytes)
- You can use the same logisim memory file for both instruction and data memory.

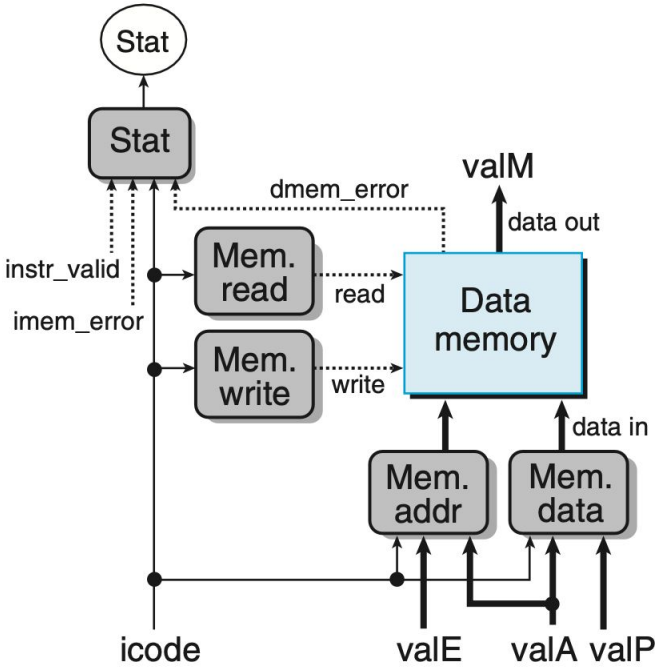
Mem read/write/addr/data

Instructions	Memory
rrmovq rA, rB	
irmovq V, rB	
rmmovq rA, D(rB)	M_8[valE] □ valA
mrmovq D(rB), rA	valM □ M_8[valE]
OPq rA, rB	
jXX Dest	
cmovXX rA, rB	
call Dest	
ret	
pushq rA	
pop rA	

Instructions	Mem. read 1: read	Mem. Write 1: write	Mem. Addr valE/valA	Mem. Data valA/valP
halt				
nop				
rrmovq rA, rB				
irmovq V, rB				
rmmovq rA, D(rB)	0	1	valE	valA
mrmovq D(rB), rA	1	0	valE	X
OPq rA, rB				
jXX Dest				
cmovXX rA, rB				
call Dest				
ret				
pushq rA				
popq rA				

Possible implementation approach

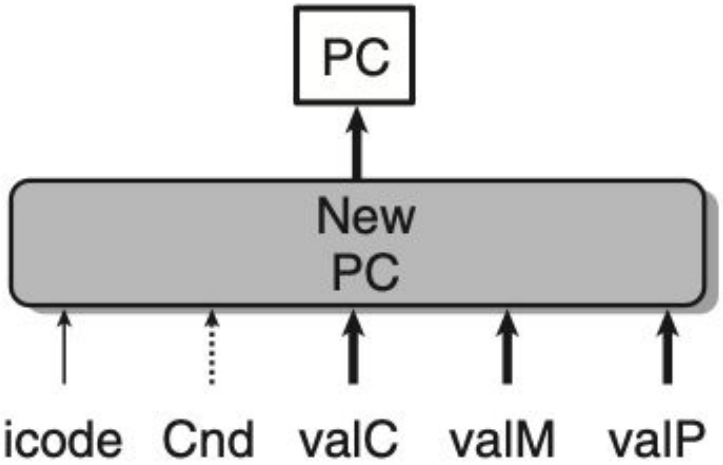
- Based on icode, decide whether read or write
- Based on icode, figure out memAddr, memData
- If read, fetch 8 bytes from memory and store in registers and output valM
  - Take 8 cycles for read, may consider using counter
  - Take care of timing for memory stage
- If write, store 8 bytes of memData in memory
  - Take 8 cycles for write, may consider using counter
  - Take care of timing for memory stage
- Update the statistics



PC Update!!

PC update stage

- Selects next PC among valC, valM, and valP
  - Based on Cnd signal
- Write back starts at the same timing with PC update



PC Selection table

Instructions	PC update
rrmovq rA, rB	
irmovq V, rB	
rmmovq rA, D(rB)	PC □ valP
mrmovq D(rB), rA	PC □ valP
OPq rA, rB	
jXX Dest	PC □ Cnd ? valC : valP
cmovXX rA, rB	
call Dest	
ret	PC □ valM
pushq rA	
pop rA	

Instructions	PC (valC, valM, valP)
halt	
nop	
rrmovq rA, rB	
irmovq V, rB	
rmmovq rA, D(rB)	valP
mrmovq D(rB), rA	valP
OPq rA, rB	
jXX Dest	valC/valP
cmovXX rA, rB	
call Dest	
ret	valM
pushq rA	
popq rA	

We are Done!!

Thank You!