Computer Systems Organization (CS2.201)

PROCESSOR ARCHITECTURE DESIGN – SEQUENTIAL (SECTION 4.3)

Deepak Gangadharan
Computer Systems Group (CSG), IIIT Hyderabad

Slide Contents: Adapted from slides by Randal Bryant

Preliminaries

$$CPU\ time\ =\ Number\ of\ instructions \times Clocks\ per\ instruction\ (CPI) \times Clock\ cycle\ time$$

$$Clock\ rate = \frac{1}{Clock\ cycle\ time}$$

Factors affecting the above parameters:
Clock rate – hardware technology and organization
CPI – organization, ISA and compiler technology
Instruction count – ISA and compiler technology

Sequential Y86-64 Implementation

Sequential Y86-64 implementation

- Let us call the processor SEQ (for sequential processor)
- On each clock cycle, SEQ performs all the steps required to process a complete instruction
- Result: Very long cycle time and low clock rate
- Goal: Improve the sequential implementation by understanding the problems with it

Sequential Y86 Instruction Stages

Each instruction sequentially goes through following common stages:

1. Fetch
2. Decode
3. Execute
4. Memory
5. Write-back
6. PC update

The processor loops indefinitely, performing the functions in each stage unless any exception condition occurs.

Sequential Y86 Instruction Stages

Why common stages for all instructions?

- Very simple and uniform structure is important when designing hardware → To reduce the footprint of logic on the chip
- One way to minimize complexity is by sharing hardware as much as possible among instructions
- Cost of duplicating block of logic in hardware is much higher than the cost of having multiple copies of code in software

SEQ stages

Fetch
- Read instruction from instruction memory

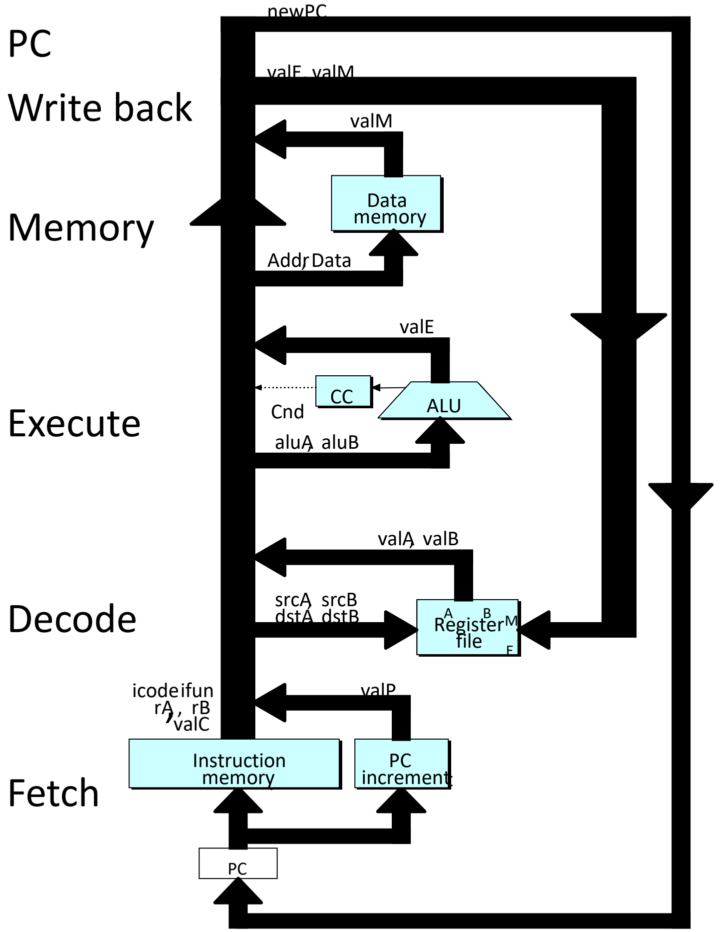
Decode
- Read program registers

Execute
- Compute value or address

Memory
- Read or write data

Write Back
- Write program registers

PC
- Update program counter



SEQ stages

Fetch:

- Reads bytes of an instruction from memory using the PC value as address → Extracts the two 4-bit portions of instruction specifier byte referred to as **icode** and **ifun**
- Possibly fetches the register specifier byte giving one or both of the register operand specifiers rA and rB
- Also possibly fetches an 8-byte constant word valC → Computes valP as the address of the next instruction in the sequence , i.e. valP = PC + length of fetched instruction

Decode:

- Reads up to to two operands from the register file giving values valA and/or valB
- For some instructions, it reads register %rsp

SEQ stages

Execute:

- ALU either performs operation given by ifun, computes effective address of a memory reference, or increments or decrements the stack pointer. Resulting value → valE
- Condition codes are possibly set
- For a jump instruction, tests condition code and branch condition (referred to by ifun) to determine if branch should be taken or not

SEQ stages

Memory:

- May read or write data from/to memory respectively. Value read referred to as valM.

Write back:

- Writes up to two results to the register file

PC Update:

- PC is set to address of next instruction or valP

Executing Arithmetic/Logic Operation

`OPq rA, rB`

6	fn	rA	rB

Fetch
- Read 2 bytes

Decode
- Read operand registers

Execute
- Perform operation
- Set condition codes

Memory
- Do nothing

Write back
- Update register

PC Update
- Increment PC by 2

Stage Computation: Arithmetic/Logic Operations

	OPq rA, rB	
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valP $\leftarrow$ PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	Read operand A Read operand B
Execute	valE $\leftarrow$ valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] $\leftarrow$ valE	Write back result
PC update	PC $\leftarrow$ valP	Update PC

Executing rmmovq

`rmmovq rA, D(rB)`

4	0	rA	rB	D

Fetch
- Read 10 bytes

Decode
- Read operand registers

Execute
- Compute effective address

Memory
- Write to memory

Write back
- Do nothing

PC Update
- Increment PC by 10

Stage Computation: rmmovq

	`rmmovq` rA, D(rB)	
Fetch	icode:ifun $\leftarrow$ M_1[PC]	Read instruction byte
	rA:rB $\leftarrow$ M_1[PC+1]	Read register byte
	valC $\leftarrow$ M_8[PC+2]	Read displacement D
	valP $\leftarrow$ PC+10	Compute next PC
Decode	valA $\leftarrow$ R[rA]	Read operand A
	valB $\leftarrow$ R[rB]	Read operand B
Execute	valE $\leftarrow$ valB + valC	Compute effective address
Memory	M_8[valE] $\leftarrow$ valA	Write value to memory
Write back		
PC update	PC $\leftarrow$ valP	Update PC

- Use ALU for address computation

Executing popq

popq rA

B	0	rA	F

Fetch
- Read 2 bytes

Decode
- Read stack pointer

Execute
- Increment stack pointer by 8

Memory
- Read from old stack pointer

Write back
- Update stack pointer
- Write result to register

PC Update
- Increment PC by 2

Stage Computation: popq

	popq rA	
Fetch	icode:ifun $\leftarrow$ M_1[PC]	Read instruction byte
	rA:rB $\leftarrow$ M_1[PC+1]	Read register byte
	valP $\leftarrow$ PC+2	Compute next PC
Decode	valA $\leftarrow$ R[%rsp]	Read stack pointer
	valB $\leftarrow$ R[%rsp]	Read stack pointer
Execute	valE $\leftarrow$ valB + 8	Increment stack pointer
Memory	valM $\leftarrow$ M_8[valA]	Read from stack
Write back	R[%rsp] $\leftarrow$ valE	Update stack pointer
	R[rA] $\leftarrow$ valM	Write back result
PC update	PC $\leftarrow$ valP	Update PC

- Use ALU to increment stack pointer
- Must update two registers
  - Popped value
  - New stack pointer

Executing Conditional Moves

`cmovXX rA, rB`	2	fn	rA	rB

Fetch
- Read 2 bytes

Decode
- Read operand registers

Execute
- If !cnd, then set destination register to 0xF

Memory
- Do nothing

Write back
- Update register (or not)

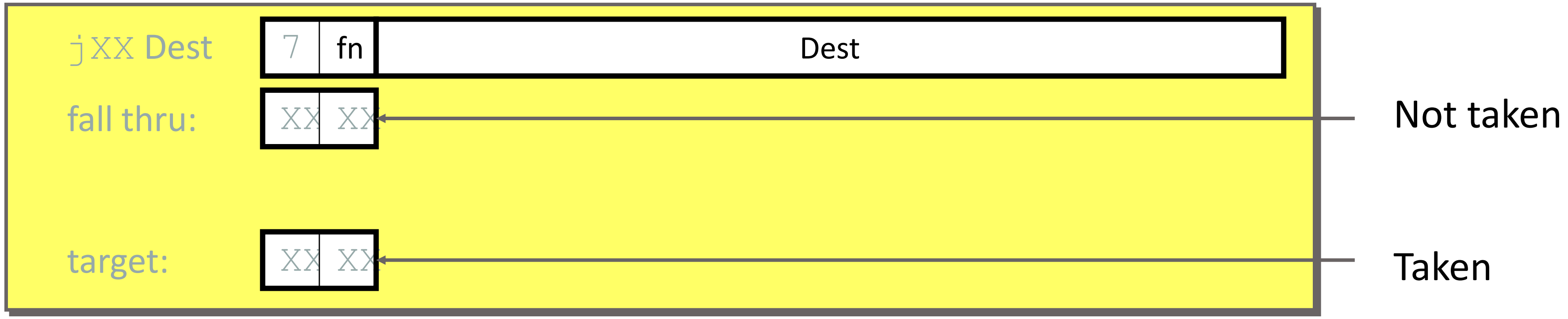
PC Update
- Increment PC by 2

Stage Computation: Cond. Move

	cmovXX rA, rB	
Fetch	icode:ifun $\leftarrow$ M_1[PC]	Read instruction byte
	rA:rB $\leftarrow$ M_1[PC+1]	Read register byte
	valP $\leftarrow$ PC+2	Compute next PC
Decode	valA $\leftarrow$ R[rA]	Read operand A
	valB $\leftarrow$ 0	
Execute	valE $\leftarrow$ valB + valA	Pass valA through ALU
	If ! Cond(CC,ifun) rB $\leftarrow$ 0xF	(Disable register update)
Memory		
Write back	R[rB] $\leftarrow$ valE	Write back result
PC update	PC $\leftarrow$ valP	Update PC

- Read register rA and pass through ALU
- Cancel move by setting destination register to 0xF
  - If condition codes & move condition indicate no move

Executing Jumps



Fetch
- Read 9 bytes
- Increment PC by 9

Decode
- Do nothing

Execute
- Determine whether to take branch based on jump condition and condition codes

Memory
- Do nothing

Write back
- Do nothing

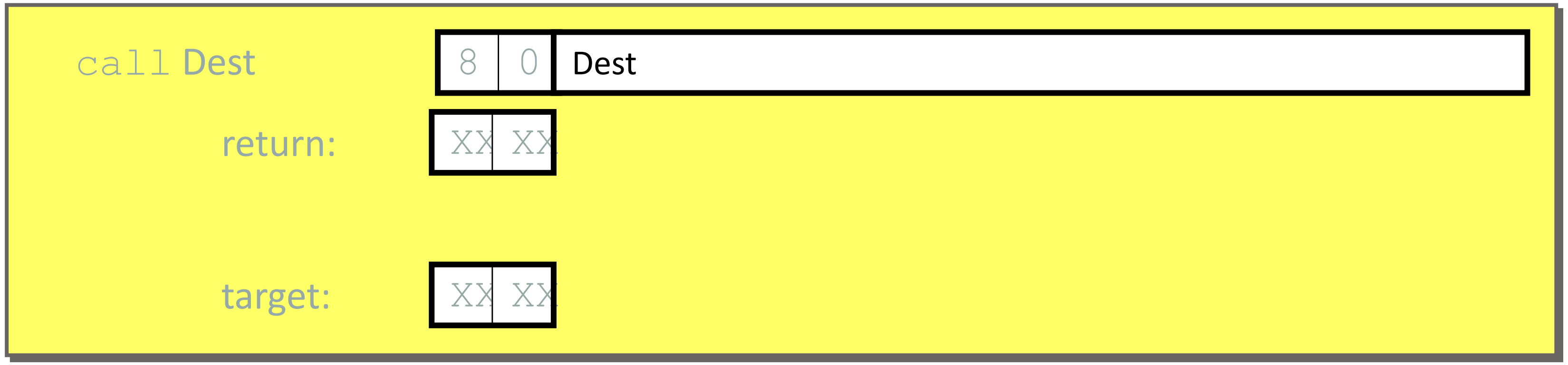
PC Update
- Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun $\leftarrow$ M_1[PC]	Read instruction byte
	valC $\leftarrow$ M_8[PC+1]	Read destination address
	valP $\leftarrow$ PC+9	Fall through address
Decode		
Execute	Cnd $\leftarrow$ Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC $\leftarrow$ Cnd ? valC : valP	Update PC

- Compute both addresses
- Choose based on setting of condition codes and branch condition

Executing `call`



Fetch

- Read 9 bytes
- Increment PC by 9

Decode

- Read stack pointer

Execute

- Decrement stack pointer by 8

Memory

- Write incremented PC to new value of stack pointer

Write back

- Update stack pointer

PC Update

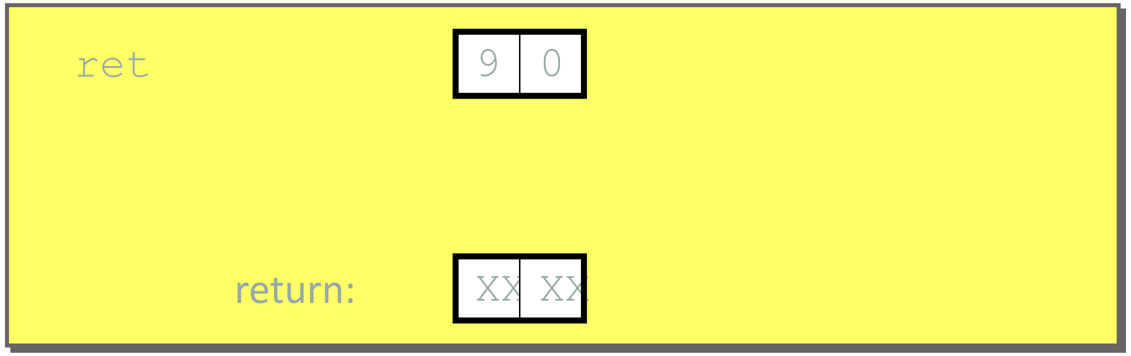
- Set PC to Dest

Stage Computation: `call`

	`call` Dest	
Fetch	icode:ifun $\leftarrow$ M_1[PC] valC $\leftarrow$ M_8[PC+1] valP $\leftarrow$ PC+9	Read instruction byte Read destination address Compute return point
Decode	valB $\leftarrow$ R[`%rsp`]	Read stack pointer
Execute	valE $\leftarrow$ valB + –8	Decrement stack pointer
Memory	M_8[valE] $\leftarrow$ valP	Write return value on stack
Write back	R[`%rsp`] $\leftarrow$ valE	Update stack pointer
PC update	PC $\leftarrow$ valC	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

Executing `ret`



Fetch
- Read 1 byte

Decode
- Read stack pointer

Execute
- Increment stack pointer by 8

Memory
- Read return address from old stack pointer

Write back
- Update stack pointer

PC Update
- Set PC to return address

Stage Computation: `ret`

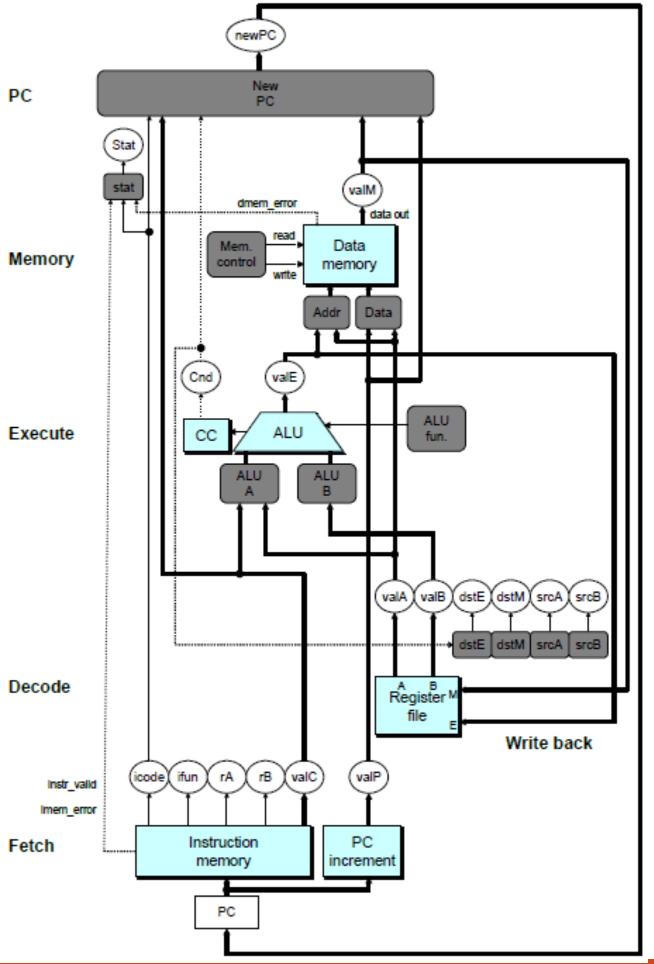
	ret	
Fetch	icode:ifun $\leftarrow$ M_1[PC]	Read instruction byte
Decode	valA $\leftarrow$ R[`%rsp`] valB $\leftarrow$ R[`%rsp`]	Read operand stack pointer Read operand stack pointer
Execute	valE $\leftarrow$ valB + 8	Increment stack pointer
Memory	valM $\leftarrow$ M_8[valA]	Read return address
Write back	R[`%rsp`] $\leftarrow$ valE	Update stack pointer
PC update	PC $\leftarrow$ valM	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

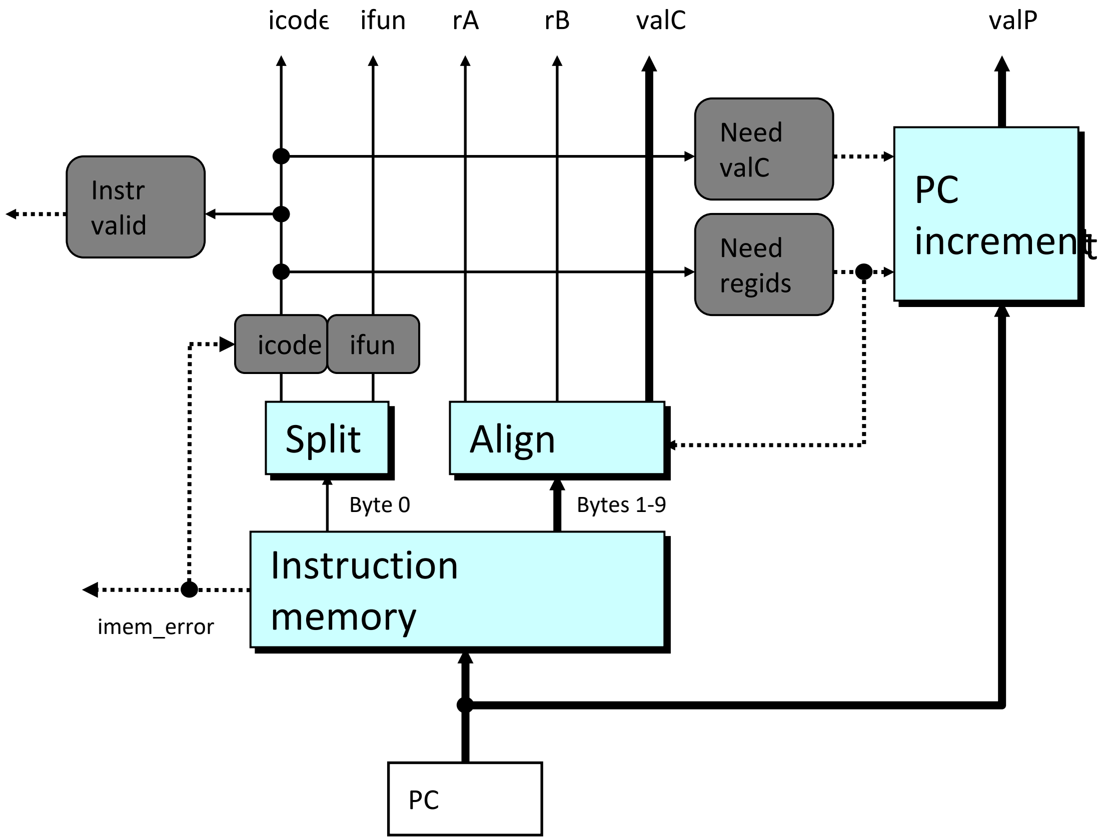
SEQ Hardware

Key

- Blue boxes: predesigned hardware blocks
  - E.g., memories, ALU
- Gray boxes: control logic
- White ovals: labels for signals
- Thick lines: 64-bit word values
- Thin lines: 4-8 bit values
- Dotted lines: 1-bit values



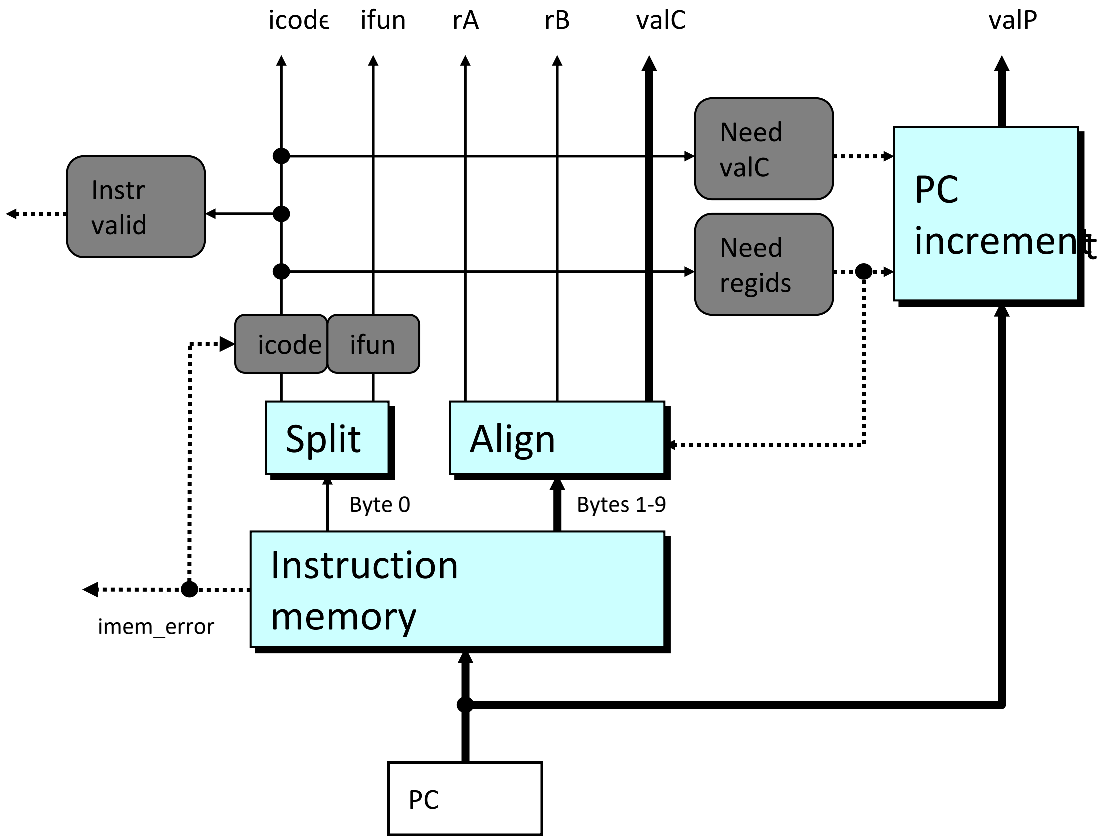
Fetch Logic



Predefined Blocks

- PC: Register containing PC
- Instruction memory: Read 10 bytes (PC to PC+9)
  - Signal invalid address
- Split: Divide instruction byte into icode and ifun
- Align: Get fields for rA, rB, and valC

Fetch Logic



Control Logic

- Instr. Valid: Is this instruction valid?
- icode, ifun: Generate no-op if invalid address
- Need regids: Does this instruction have a register byte?
- Need valC: Does this instruction have a constant word?

Decode Logic

Register File

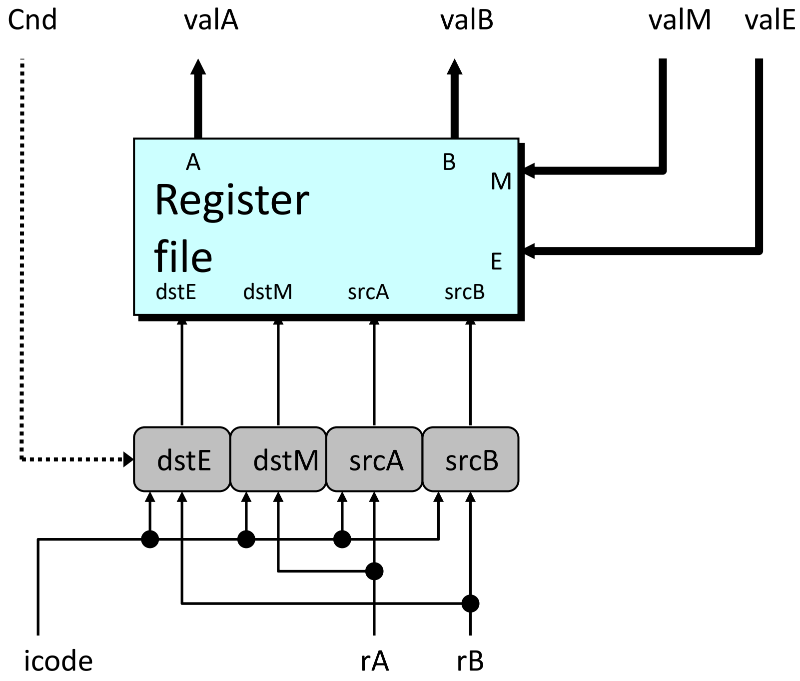
- Read ports A, B
- Write ports E, M
- Addresses are register IDs or 15 (0xF) (no access)

Control Logic

- srcA, srcB: read port addresses
- dstE, dstM: write port addresses

Signals

- Cnd: Indicate whether or not to perform conditional move
  - Computed in Execute stage



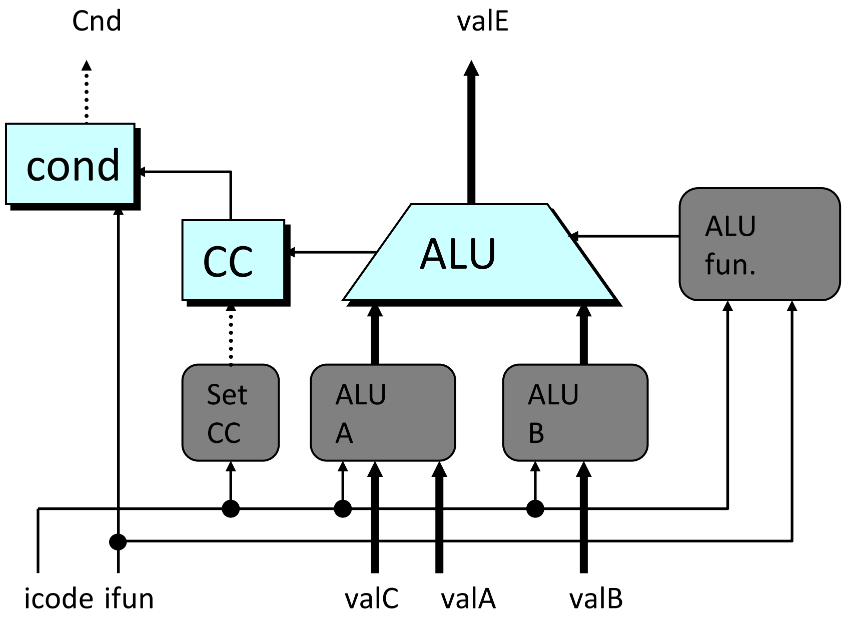
Execute Logic

Units

- ALU
  - Implements 4 required functions
  - Generates condition code values
- CC
  - Register with 3 condition code bits
- cond
  - Computes conditional jump/move flag

Control Logic

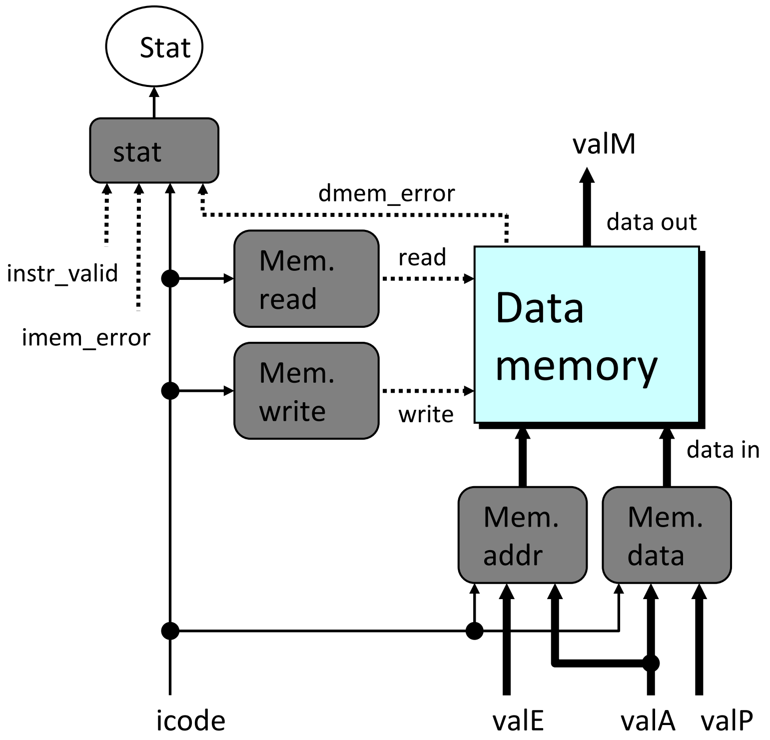
- Set CC: Should condition code register be loaded?
- ALU A: Input A to ALU
- ALU B: Input B to ALU
- ALU fun: What function should ALU compute?



Memory Logic

Memory
- Reads or writes memory word

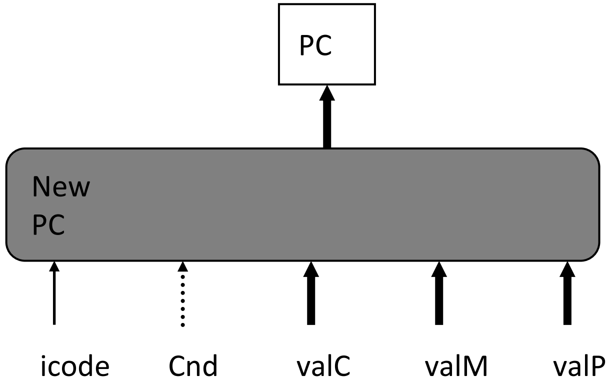
Control Logic
- stat: What is instruction status?
- Mem. read: should word be read?
- Mem. write: should word be written?
- Mem. addr.: Select address
- Mem. data.: Select data



PC Update Logic

New PC

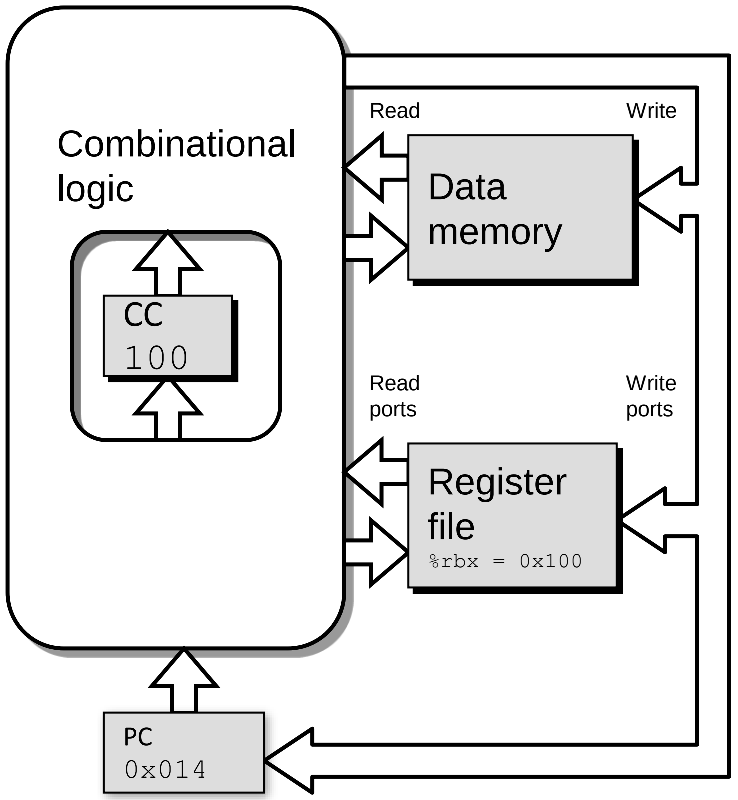
- Select next value of PC



PC Update

	OPq rA, rB	
PC update	PC ← valP	Update PC
	`rmmovq` rA, D(rB)	
PC update	PC ← valP	Update PC
	`popq` rA	
PC update	PC ← valP	Update PC
	jXX Dest	
PC update	PC ← Cnd ? valC : valP	Update PC
	`call` Dest	
PC update	PC ← valC	Set PC to destination
	`ret`	
PC update	PC ← valM	Set PC to return address

SEQ Operation

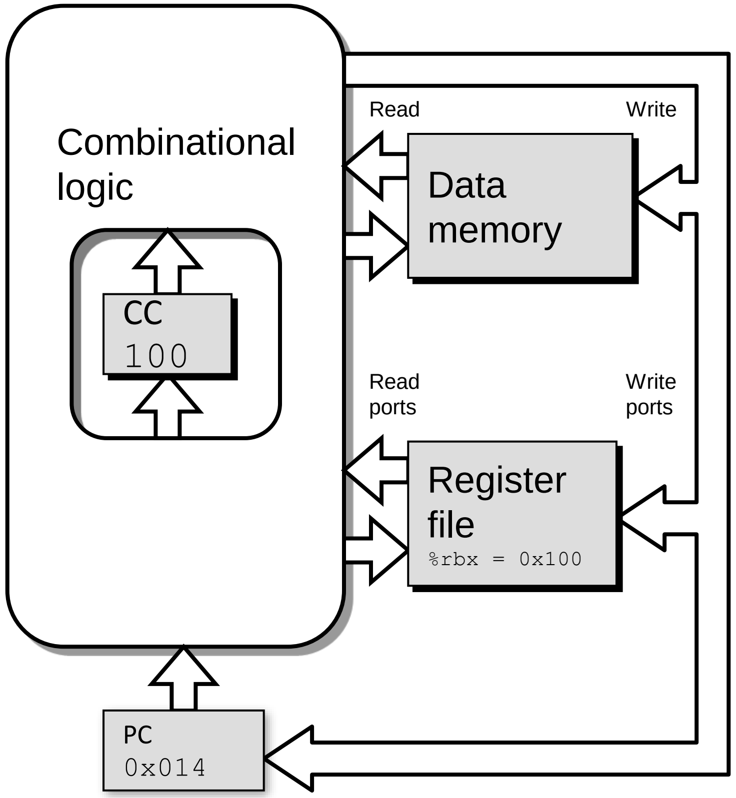


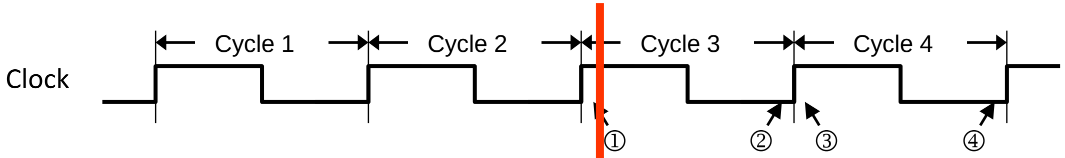
State
- PC register
- Cond. Code register
- Data memory
- Register file

All updated as clock rises

Combinational Logic
- ALU
- Control logic
- Memory reads
  - Instruction memory
  - Register file
  - Data memory

SEQ Operation #2

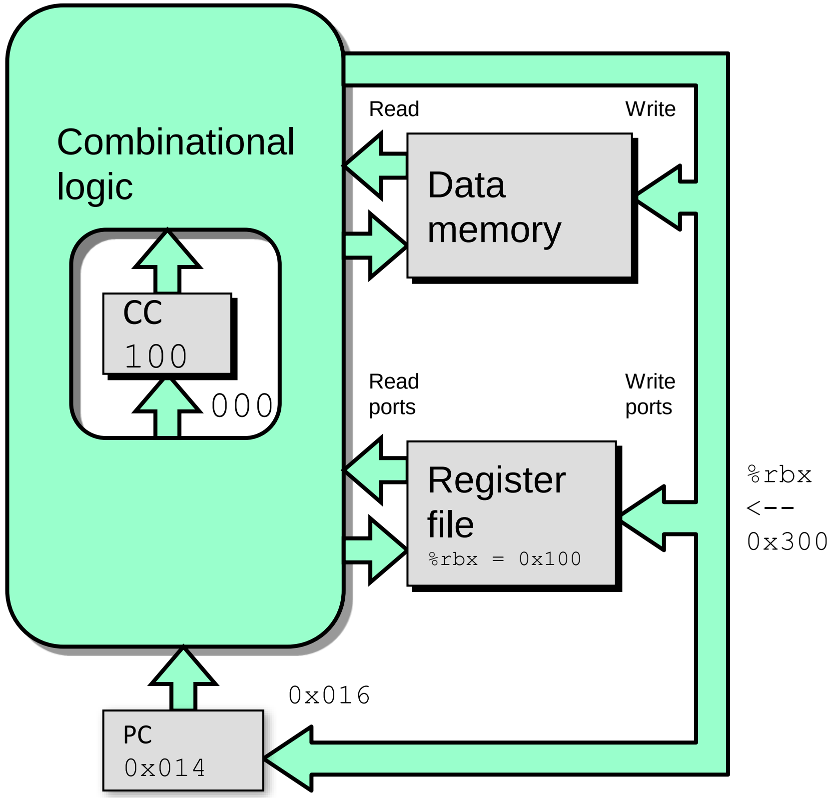


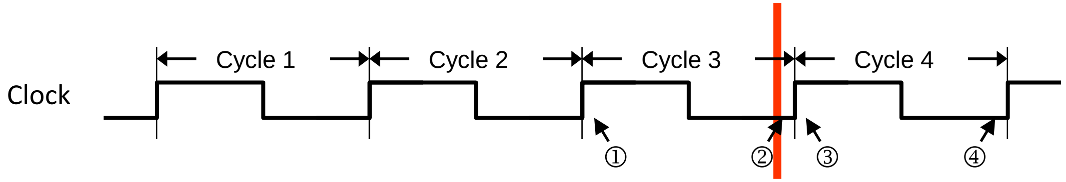


Cycle 1:	`0x000:    irmovq $0x100,%rbx   # %rbx <-- 0x100`
Cycle 2:	`0x00a:    irmovq $0x200,%rdx   # %rdx <-- 0x200`
Cycle 3:	`0x014:    addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000`
Cycle 4:	`0x016:    je dest              # Not taken`
Cycle 5:	`0x01f:    rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300`

- state set according to second `irmovq` instruction
- combinational logic starting to react to state changes

SEQ Operation #3

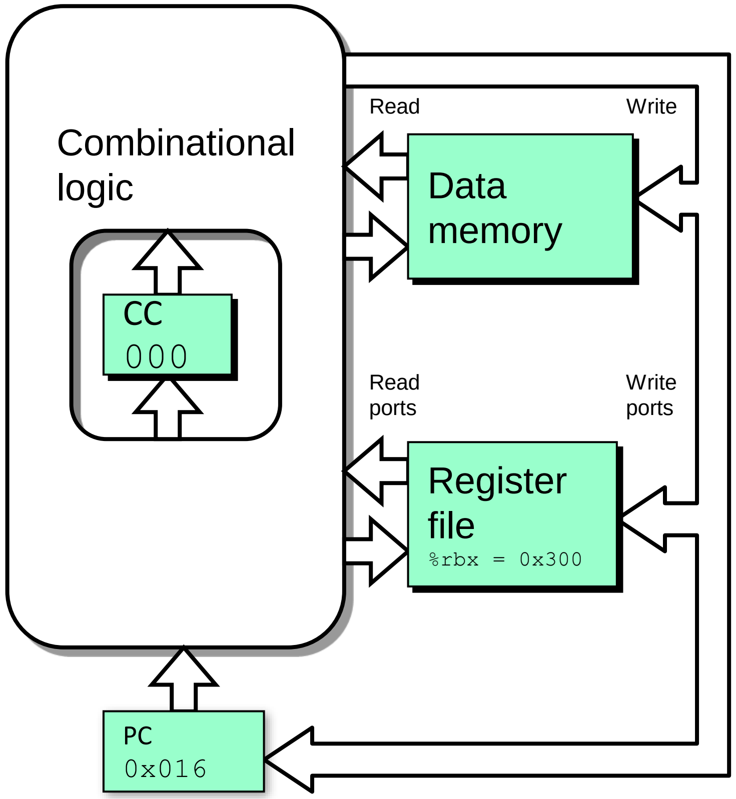


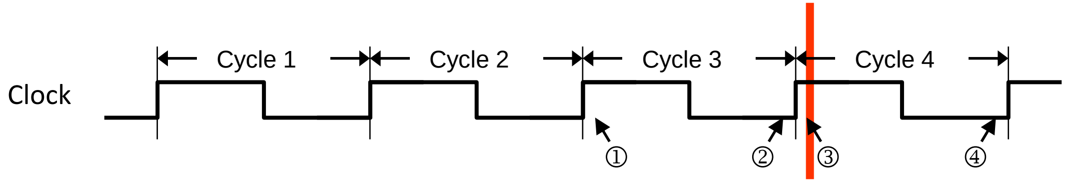


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

- state set according to second `irmovq` instruction
- combinational logic generates results for `addq` instruction

SEQ Operation #4

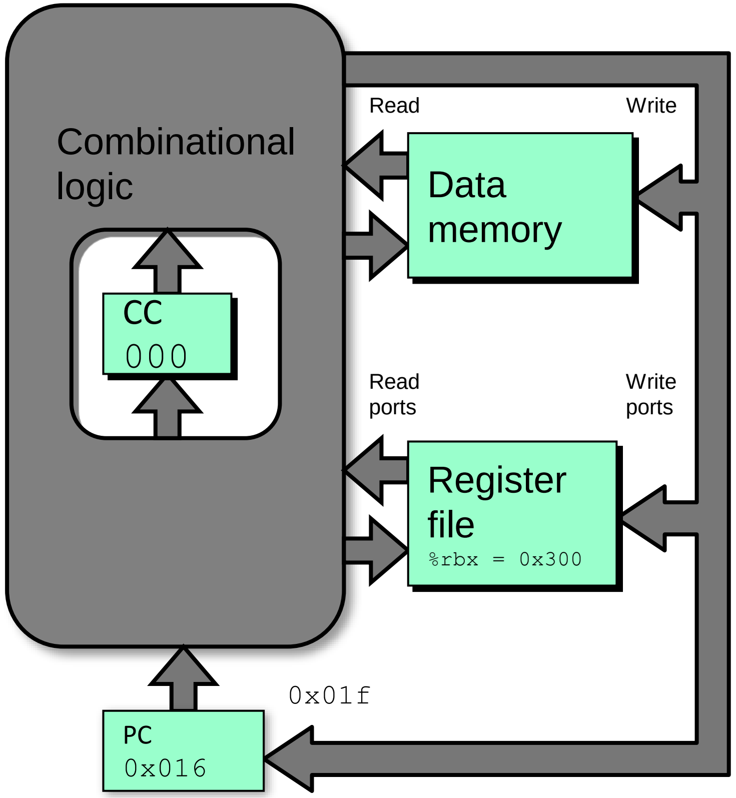


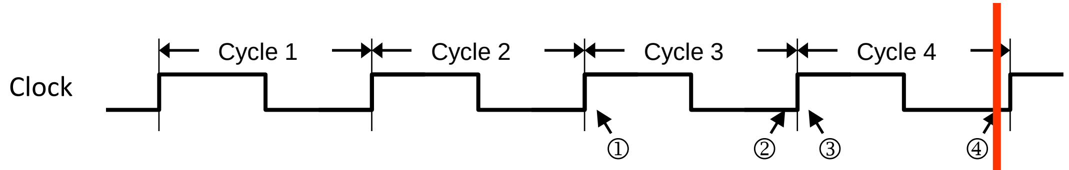


Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
Cycle 4:	0x016:	je dest	# Not taken
Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

- state set according to `addq` instruction
- combinational logic starting to react to state changes

SEQ Operation #5





Cycle 1:	0x000:	`irmovq $0x100,%rbx   # %rbx <-- 0x100`
Cycle 2:	0x00a:	`irmovq $0x200,%rdx   # %rdx <-- 0x200`
Cycle 3:	0x014:	`addq %rdx,%rbx       # %rbx <-- 0x300 CC <-- 000`
Cycle 4:	0x016:	`je dest              # Not taken`
Cycle 5:	0x01f:	`rmmovq %rbx,0(%rdx)  # M[0x200] <-- 0x300`

- state set according to `addq` instruction
- combinational logic generates results for `je` instruction

SEQ Summary

Implementation

- Express every instruction as series of simple steps
- Follow same general flow for each instruction type
- Assemble registers, memories, predesigned combinational blocks
- Connect with control logic

Limitations

- Too slow to be practical
- In one cycle, must propagate through instruction memory, register file, ALU, and data memory
- Would need to run clock very slowly
- Hardware units only active for fraction of clock cycle

Thank You!