Sequential Y86-64 Implementations

CSCI 237: Computer Organization
16th Lecture, Mar 17, 2025

Jeannie Albrecht

Administrative Details

- Lab 3
  - Due this week (submit using submit237 3 files)
- Midterm details
  - This week during lab; closed book/notes; you can have a calculator
  - Focuses on Chapters 1-4 and Labs 1, 2, **and 3**
    - Emphasis on Ch 2-3, Labs 1-2, **a little bit on Y86-64 (Lab 3) and Ch 4**
  - Sample exams posted on webpage
- No class on Wed
  - I'll be in my office from 9:30-12 if you have questions
- Office hours: today 2:00-3:00, tomorrow 1:30-3:00
- Review session Tue 8pm in TPL 205
- We **have** class on Friday (but no office hours in the afternoon)
  - I will post the slides (as always) if you will miss

Review: CMPX and JXX

- cmpq computes the difference between two integer operands and updates the OF, SF, ZF, and CF flags according to the result
  - cmpq ra, rb computes rb-ra and sets flags (condition codes)
- Conditional jump or move happens based on condition codes
- Example: jle jumps if ZF or (OF xor SF)
  - ZF handles the equals case
  - OF xor SF being set indicates that rb was less than ra in previous operation
  - So it jumps if rb <= ra
- In general,
  - cmpq ra, rb
  - jOP jumps if rb OP ra
- In Y86, subX is the same as cmpX, andX is the same as testX

Review: CMPX and JXX

- cmpq ra, rb
- jOP jumps if rb OP ra

```
int test(long x, long y) {
  if (x < y)
    return 0;
  else
    return 1;
}
```

```
test:
    cmpq    %rsi, %rdi  # y:x
    jge     .L3          # x>=y
    movl    $0, %eax
    ret
.L3:                     # else
    movl    $1, %eax
    ret
```

Review: CMPX and JXX

- cmpq ra, rb
- jOP jumps if rb OP ra

```
int test(long x, long y) {
  if (x >= y)
    return 0;
  else
    return 1;
}
```

```
test:
    cmpq    %rsi, %rdi  # y:x
    jl      .L3         # x<y
    movl    $0, %eax
    ret
.L3:                    # else
    movl    $1, %eax
    ret
```

Review: CMPX and JXX

- cmpq ra, rb
- jOP jumps if rb OP ra

```
int test(long x) {
  if (x > 5)
    return 0;
  else
    return 1;
}
```

```
test:
    cmpq    $5, %rdi     # 5:x
    jle     .L3          # x<=5
    movl    $0, %eax
    ret
.L3:                      # else
    movl    $1, %eax
    ret
```

Last Time

- Discussed combinational circuits
- Learned about HCL
- Overview of memory and clocking

Recap: HCL Summary

- Book introduces a very simple hardware description language
- Can only express limited aspects of hardware operation
- Data Types
  - `bool`: Boolean
    - `a`, `b`, `c`, …
  - `int`: words
    - `A`, `B`, `C`, …
    - Does not specify word size—64-bit words, …
- Statements
  - `bool a =` *`bool-expr`* `;`
  - `int A =` *`int-expr`* `;`

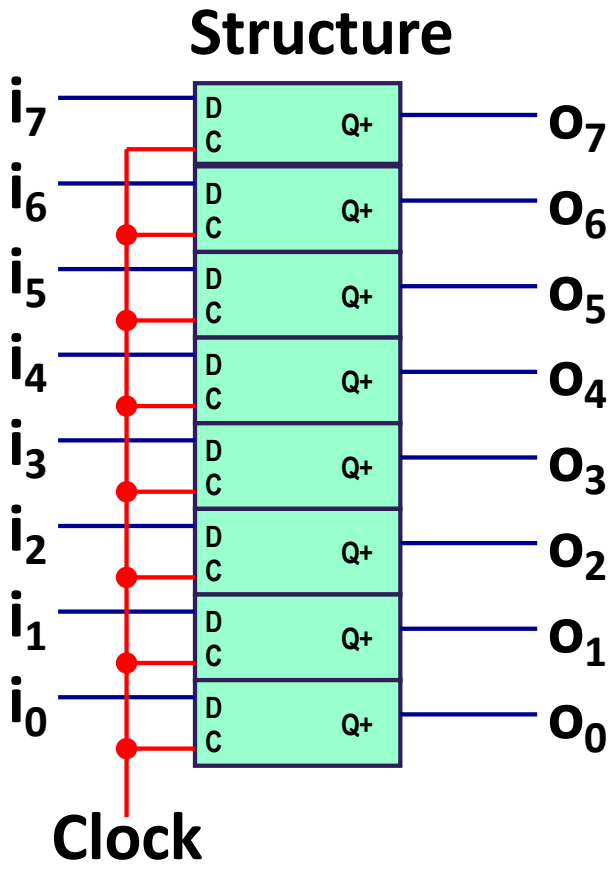
Recap: HCL Operations

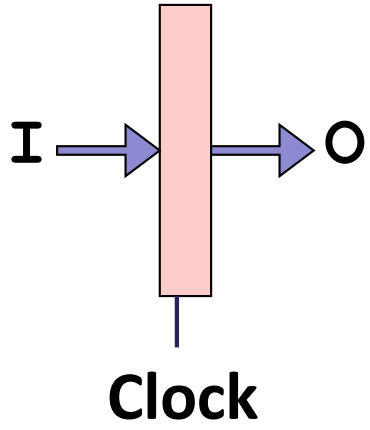
- Classify by type of value returned
- Boolean Expressions
  - Logic Operations
    - `a && b, a || b, !a`
  - Word Comparisons
    - `A == B, A != B, A < B, A <= B, A >= B, A > B`
  - Set Membership
    - `A in { B, C, D }`
      - Same as `A == B || A == C || A == D`
- Word Expressions
  - Case expressions
    - `[ a : A; b : B; c : C ]`
    - Evaluate test expressions `a`, `b`, `c`, … in sequence
    - Return word expression `A`, `B`, `C`, … for first successful test

Today

- Finish up Ch 4.2
- Move on to sequential Y86-64 implementations (Ch 4.3)
  - Organizing Processes into stages
  - SEQ Hardware Structure
  - SEQ Stage Implementations
  - SEQ Timing

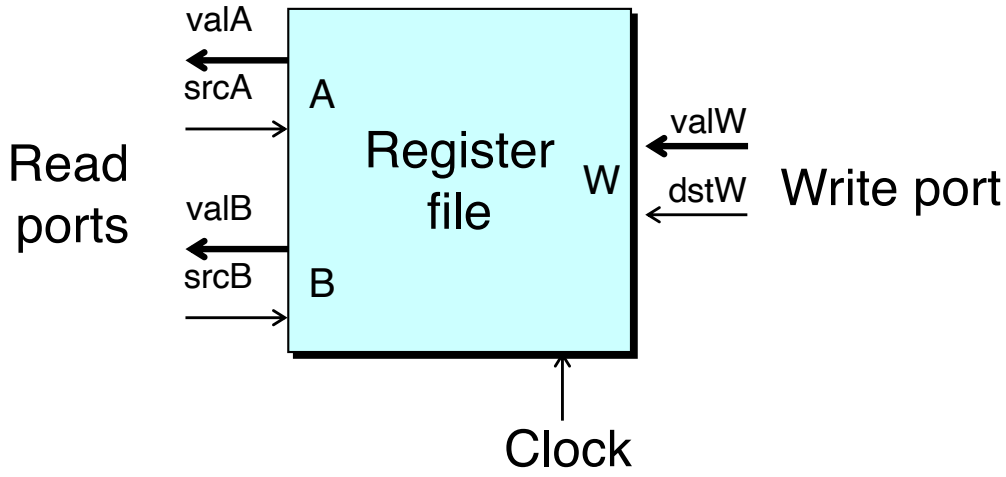
Recap: Hardware Registers





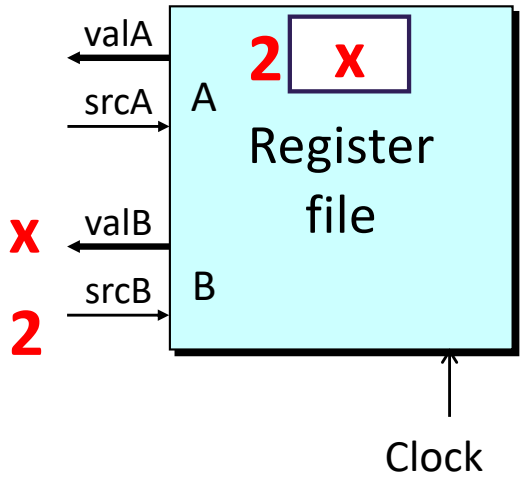
- Stores word of data
- Different from *program registers* seen in assembly code (e.g., %rdi)
- Collection of edge-triggered *latches*
- Loads input on *rising edge* of clock

Random-Access Memory

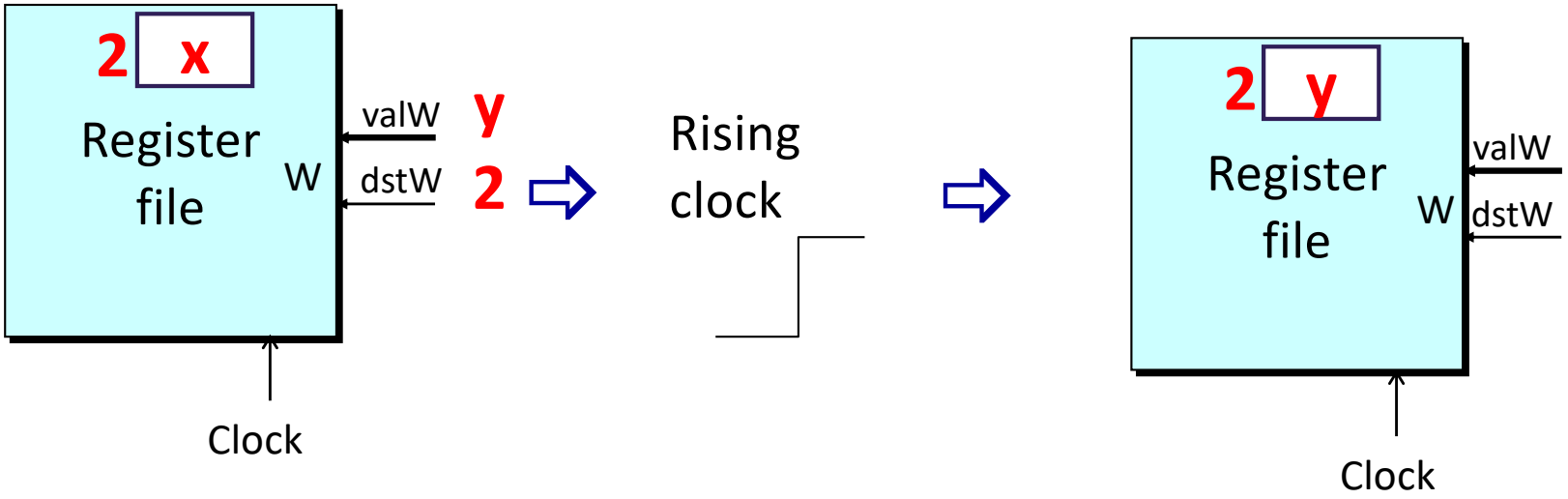


- Stores multiple words of memory (has internal storage)
  - Address input specifies which word to read or write
- Register file
  - Holds values of program registers (`%rax`, `%rsp`, etc.)
  - Register identifier serves as address
    - ID 15 (0xF) implies no read or write performed
- Multiple Ports
  - Can read and/or write multiple words in one cycle
    - Each has separate address and data input/output

Register File Timing



- Reading
  - Like combinational logic
  - Output data generated based on input addr
    - After some small delay
- Writing
  - Like hardware register wrt timing
  - Update only as clock rises

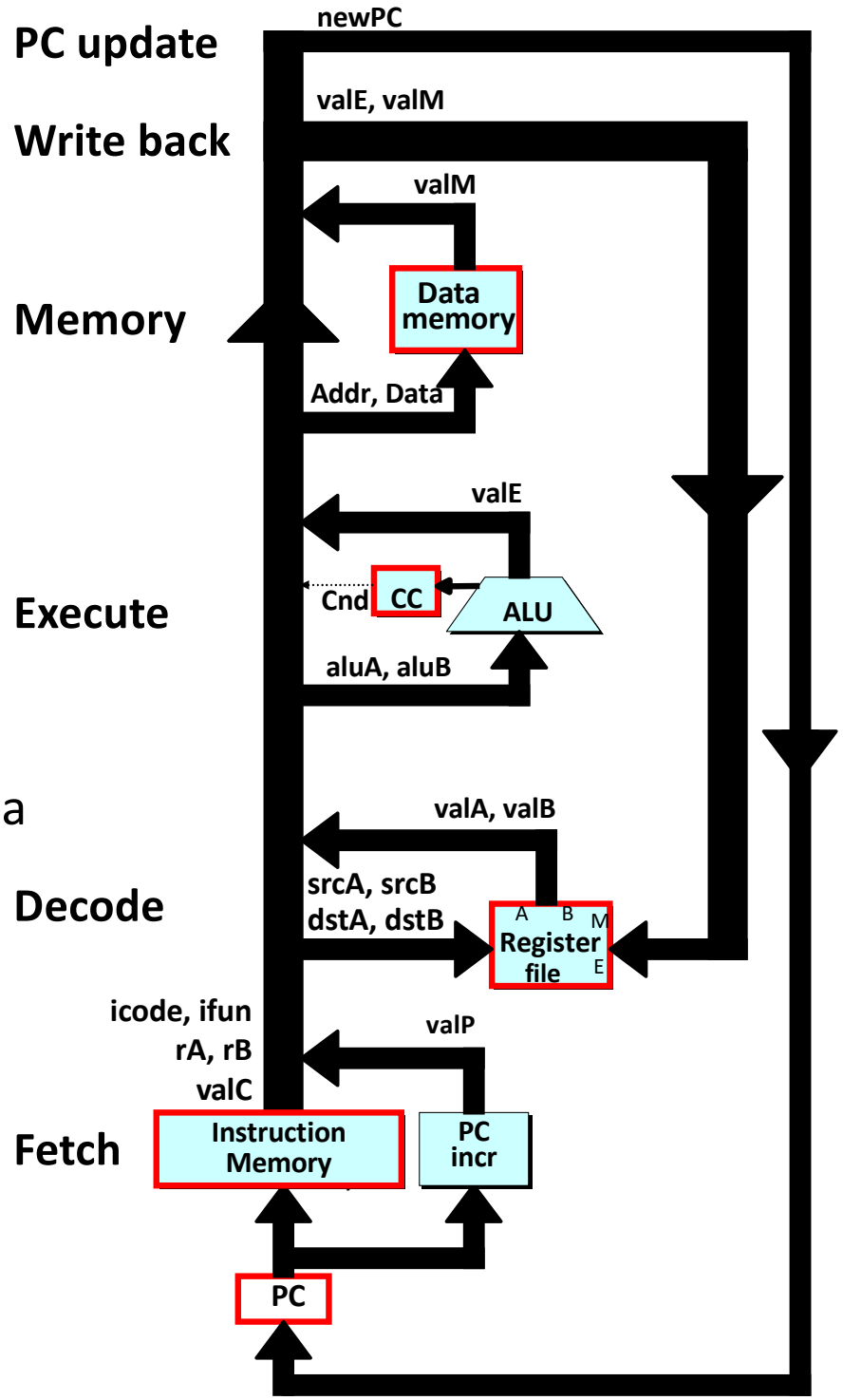


Ch 4.2 Summary

- Computation
  - Performed by combinational logic
  - Computes Boolean functions
  - Continuously reacts to input changes
- Storage
  - Registers (hardware)
    - Hold single words
    - Loaded as clock rises
  - Random-access memories
    - Hold multiple words
    - Possible multiple read or write ports
    - Read word when address input changes
    - Write word as clock rises

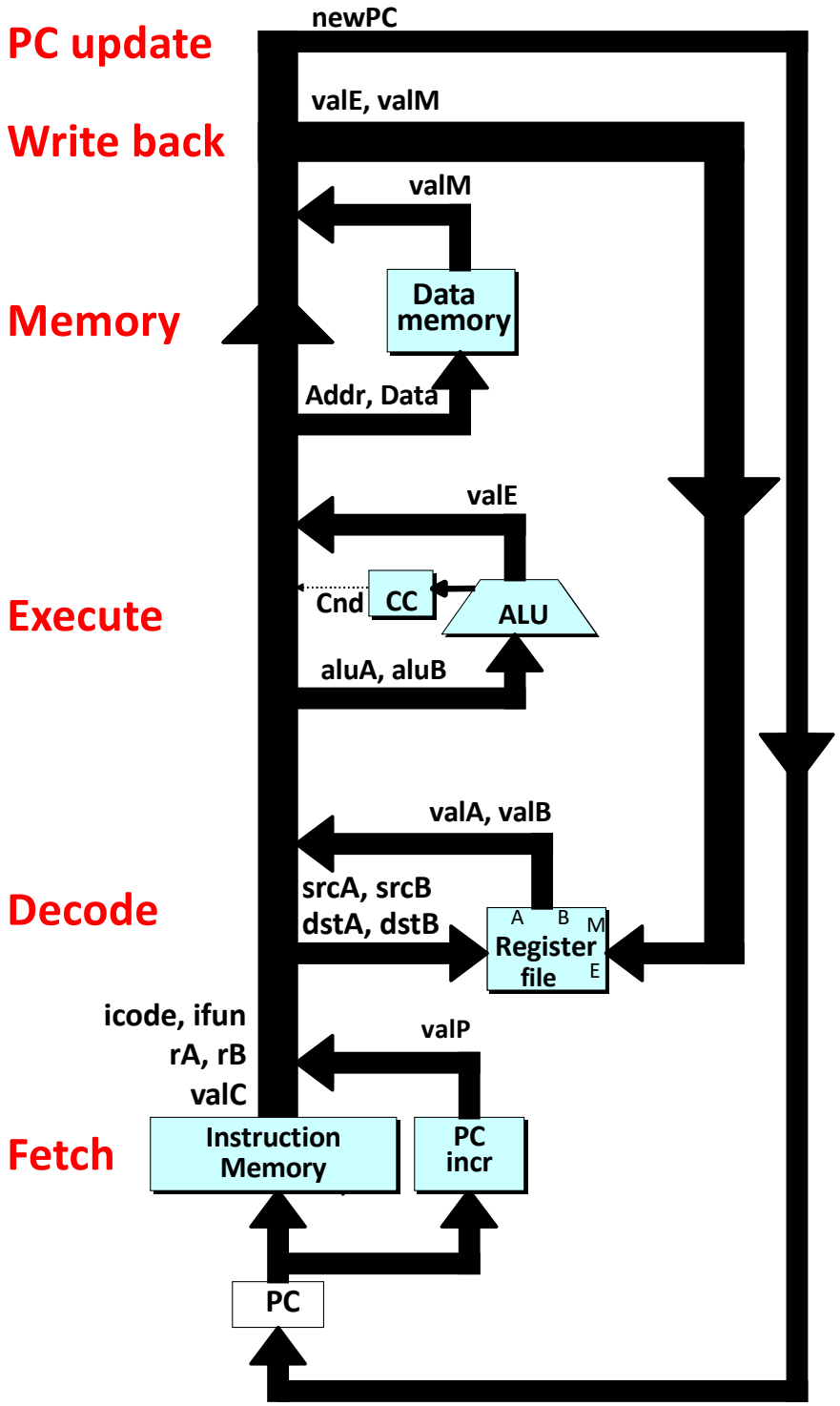
Moving on to Ch 4.3: SEQ Hardware Structure

- State (red boxes)
  - Program counter register (PC)
  - Condition code register (CC)
  - Register File
  - Memories
    - Access same memory space
    - Data: for reading/writing program data
    - Instruction: for reading instructions
- Instruction Flow (arrows)
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

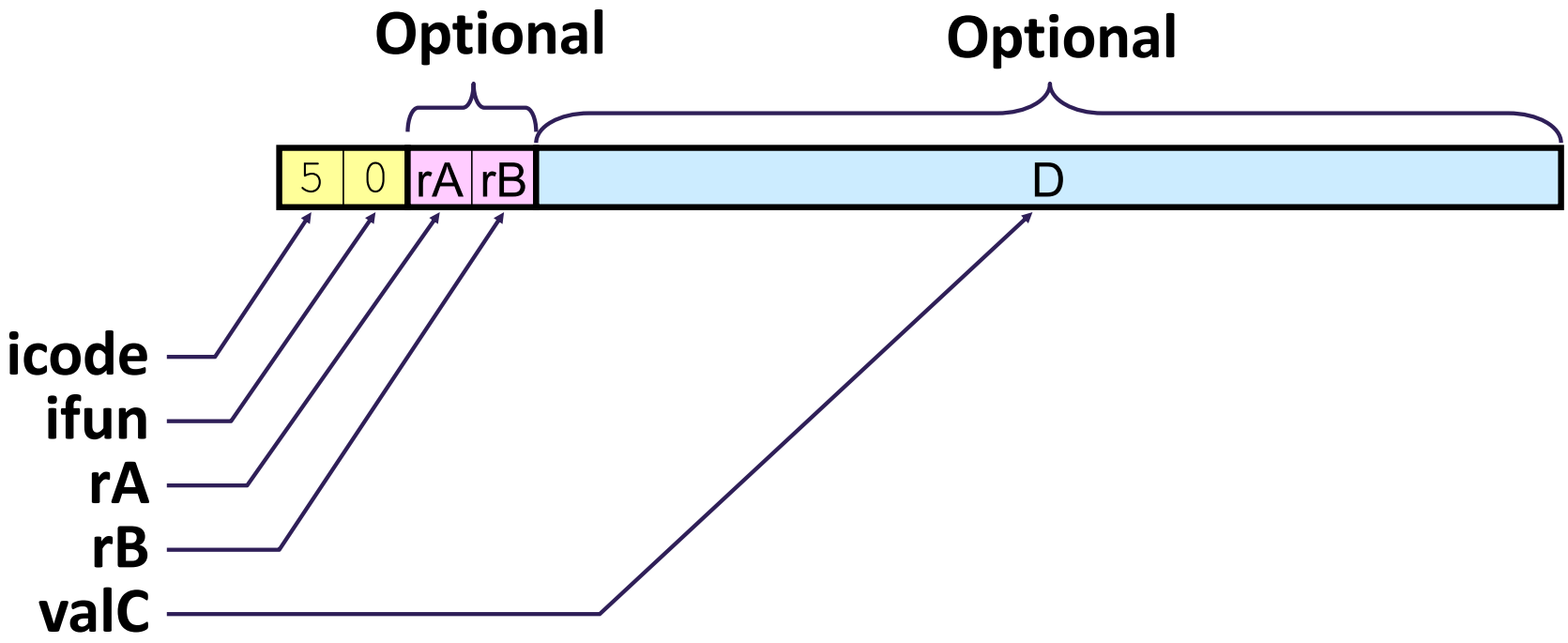


SEQ Stages

- Fetch
  - Read instruction from instr memory
- Decode
  - Read program registers from instr
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write Back
  - Write program registers to register file
- PC update
  - Update program counter

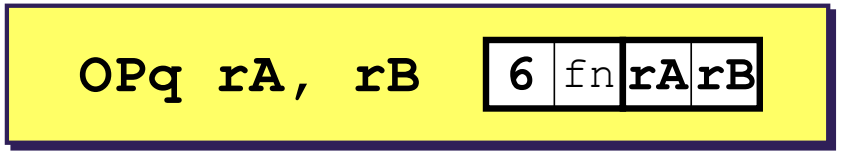


Instruction Decoding



- Instruction Format
  - Instruction byte — icode:ifun
  - Optional register byte — rA:rB
  - Optional constant word — valC

Executing Arithmetic/Logical Operation



- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - Perform operation
  - Set condition codes
- Memory
  - Do nothing
- Write back
  - Update register rB
- PC Update
  - Increment PC by 2 bytes

Stage Computation: Arith/Log Ops

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

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions
- Note: M_1 indicates we're accessing 1 byte of memory (usually instruction memory), while M_8 indicates we're accessing 8 bytes (usually data memory)

Executing `rmmovq`

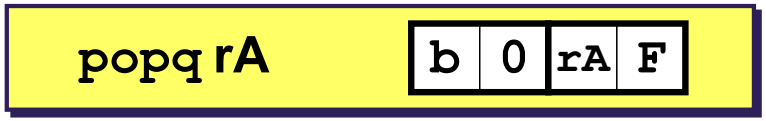
- Fetch
  - Read 10 bytes
- Decode
  - Read operand registers
- Execute
  - Compute effective address
- Memory
  - Write to memory (rB+D)
- Write back
  - Do nothing
- PC Update
  - Increment PC by 10

Stage Computation: `rmmovq`

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

- Use ALU for address computation

Executing popq



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

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

Executing Conditional Moves

rrmovq	2	0
cmovle	2	1
cmovl	2	2
cmove	2	3
cmovne	2	4
cmovge	2	5
cmovg	2	6

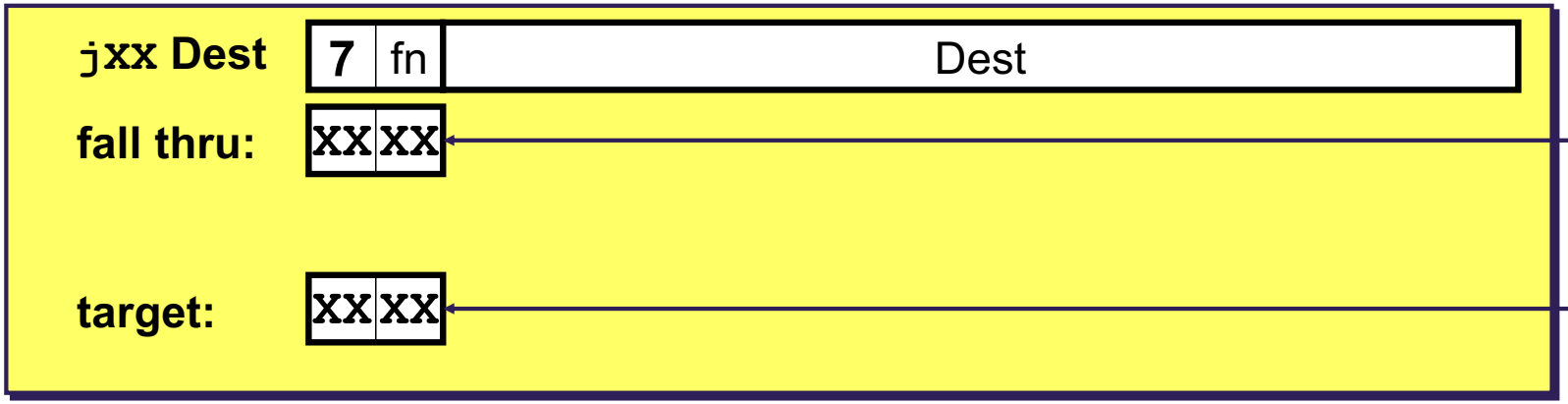
- Fetch
  - Read 2 bytes
- Decode
  - Read operand registers
- Execute
  - If !cond, then set destination register to 0xF
- Memory
  - Do nothing
- Write back
  - Update register (or not)
- PC Update
  - Increment PC by 2

Stage Computation: Cond Move

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



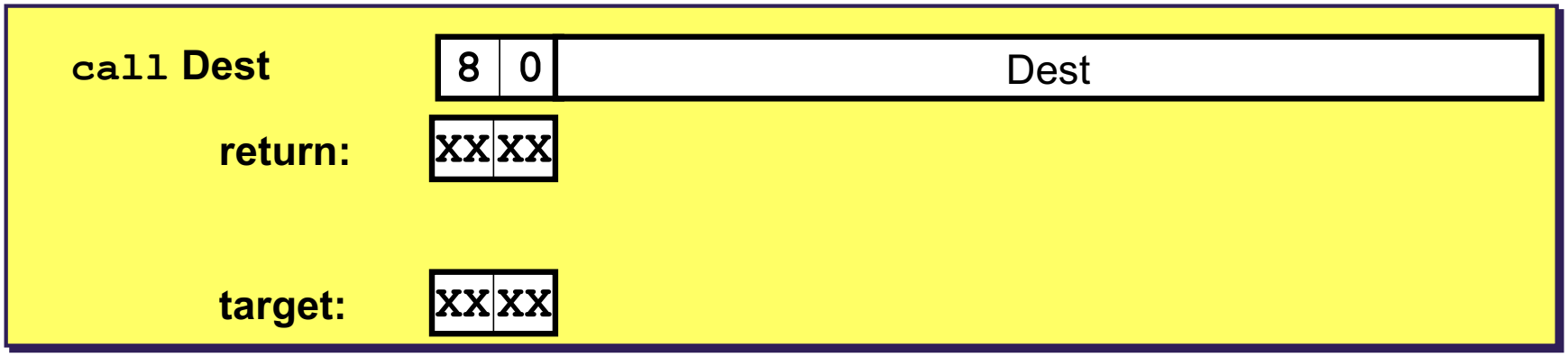
- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Do nothing
- Execute
  - Determine whether to take branch based on jump condition and condition codes
- Memory
  - Do nothing
- Write back
  - Do nothing
- PC Update
  - Set PC to Dest if branch taken or to incremented PC if not branch

Stage Computation: Jumps

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

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

Executing `call`



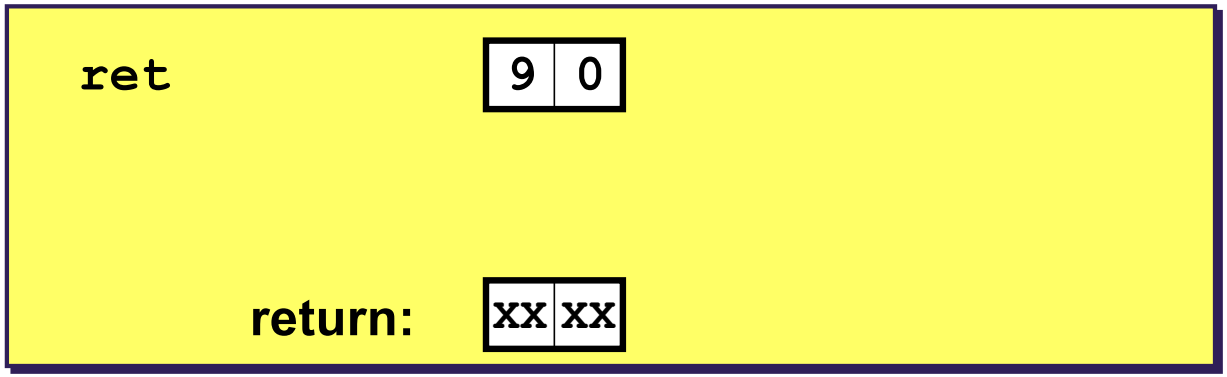
- Fetch
  - Read 9 bytes
  - Increment PC by 9
- Decode
  - Read stack pointer
- Execute
  - Decrement stack pointer by 8
- Memory
  - Write incremented PC to new value of stack pointer
- Write back
  - Update stack pointer
- PC Update
  - Set PC to Dest

Stage Computation: `call`

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

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

Computation Steps

		OPq rA, rB	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB	rA:rB ← M_1[PC+1]	Read register byte
	valC		[Read constant word]
	valP	valP ← PC+2	Compute next PC
Decode	valA, srcA	valA ← R[rA]	Read operand A
	valB, srcB	valB ← R[rB]	Read operand B
Execute	valE	valE ← valB OP valA	Perform ALU operation
	Cond code	Set CC	Set/use cond. code reg
Memory	valM		[Memory read/write]
Write back	dstE	R[rB] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valP	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Computation Steps

		call Dest	
Fetch	icode,ifun	icode:ifun ← M_1[PC]	Read instruction byte
	rA,rB		[Read register byte]
	valC	valC ← M_8[PC+1]	Read constant word
	valP	valP ← PC+9	Compute next PC
Decode	valA, srcA		[Read operand A]
	valB, srcB	valB ← R[%rsp]	Read operand B
Execute	valE	valE ← valB + –8	Perform ALU operation
	Cond code		[Set /use cond. code reg]
Memory	valM	M_8[valE] ← valP	Memory read/write
Write back	dstE	R[%rsp] ← valE	Write back ALU result
	dstM		[Write back memory result]
PC update	PC	PC ← valC	Update PC

- All instructions follow same general pattern
- Differ in what gets computed on each step

Summary of Computed Values

- Fetch
  - icode — Instruction code
  - ifun — Instruction function
  - rA — Instr. Register A
  - rB — Instr. Register B
  - valC — Instruction constant
  - valP — Incremented PC
- Decode
  - srcA — Register ID A
  - srcB — Register ID B
  - dstE — Destination Register E
  - dstM — Destination Register M
  - valA — Register value A
  - valB — Register value B
- Execute
  - valE — ALU result
  - Cnd — Branch/move flag
- Memory
  - valM — Value from memory