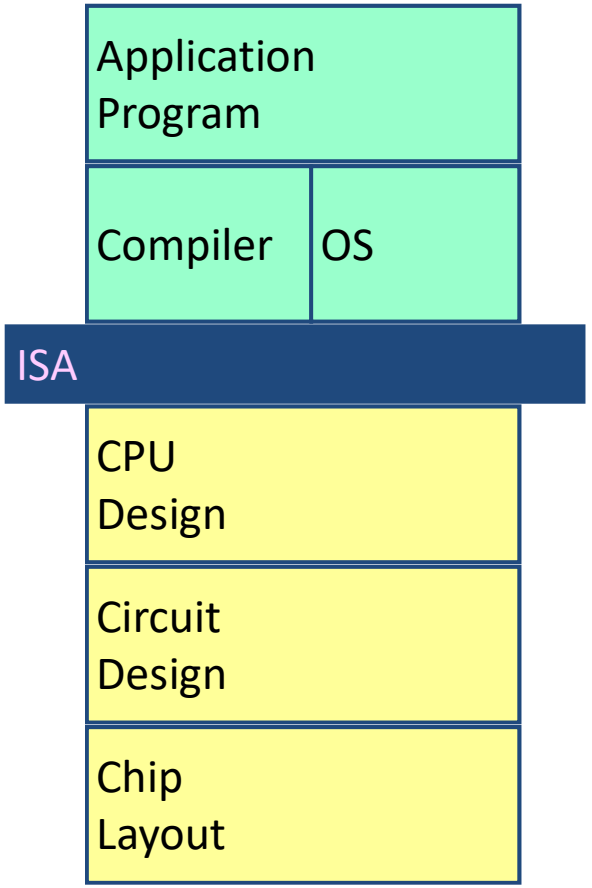
Computer Systems Organization
CS2.201

Topic 4

Based on chapter 4 from Computer Systems by Randal E. Bryant and David R. O'Hallaron

Instruction Set Architecture

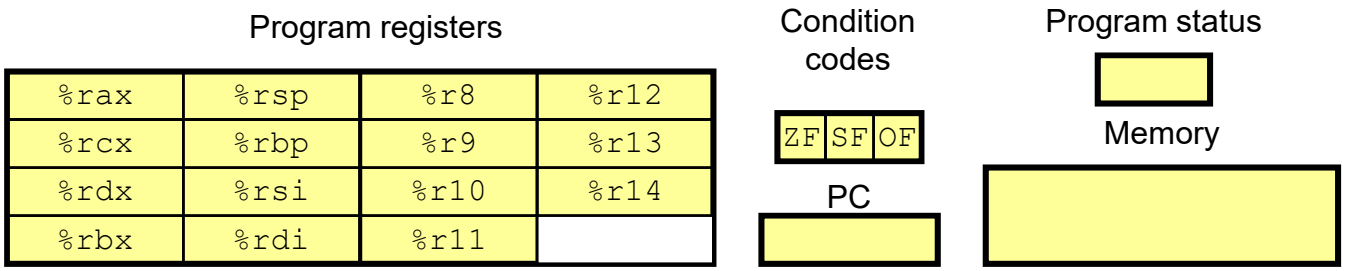
- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addq`, `pushq`, `ret`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



What is Y86-64 ?

- Y86 is a “toy” machine that is similar to the x86 but much simpler and acts as a gentler introduction to assembly level programming.
  - Just a few instructions as opposed to hundreds for the x86;
  - Everything you learn about the Y86 will apply to the x86 with very little modification
  - Chapter introduces pipelining in that context of Y86

Y86-64 Processor State



- Program Registers
  - 15 registers (omit `%r15`). Each 64 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - ZF: Zero SF:Negative OF: Overflow
- Program Counter
  - Indicates address of next instruction
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

Encoding Registers

- Each register has 4-bit ID

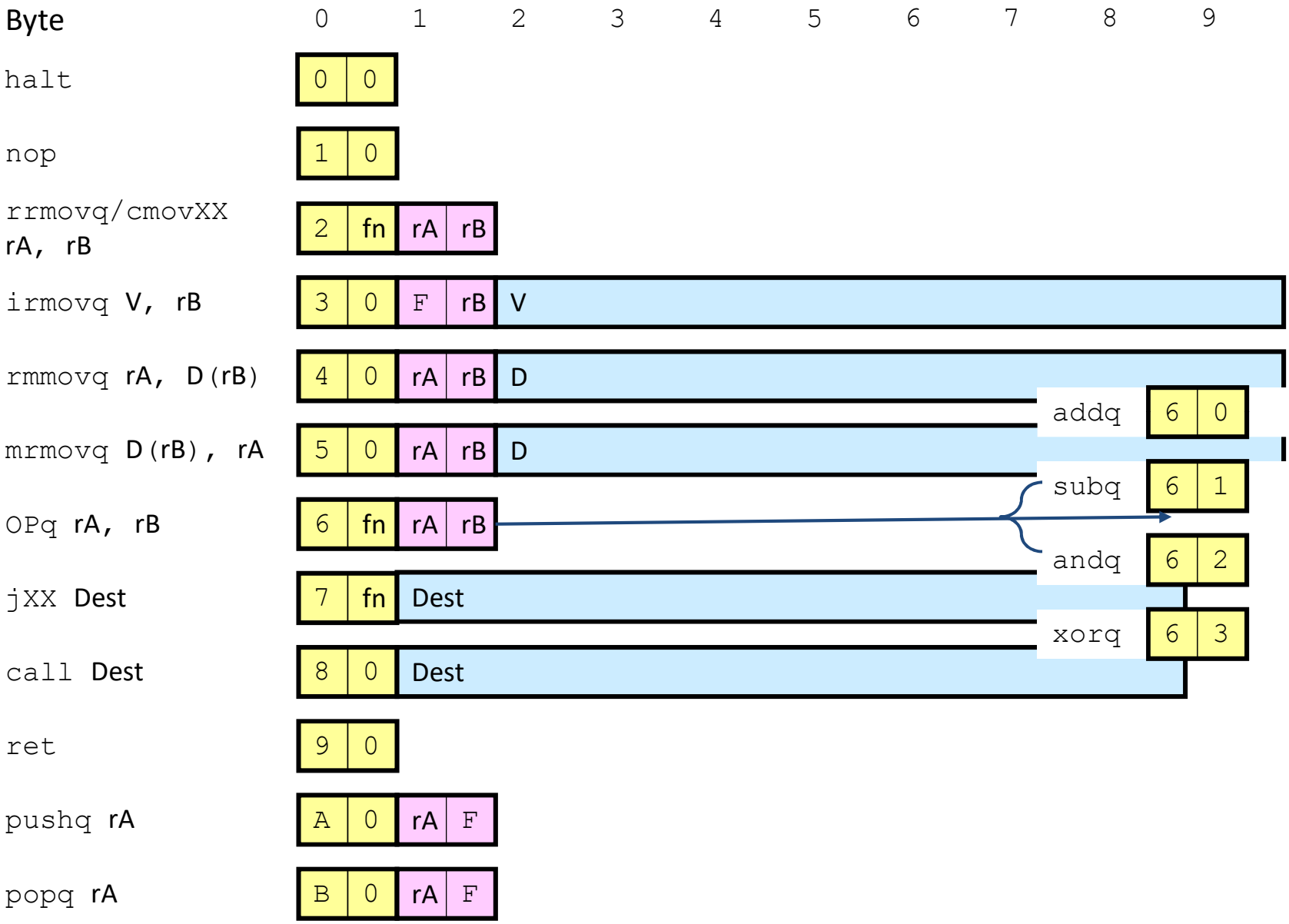
%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

  - Same encoding as in x86-64
- Register ID 15 (`0xF`) indicates "no register"
  - Useful in hardware design in multiple places

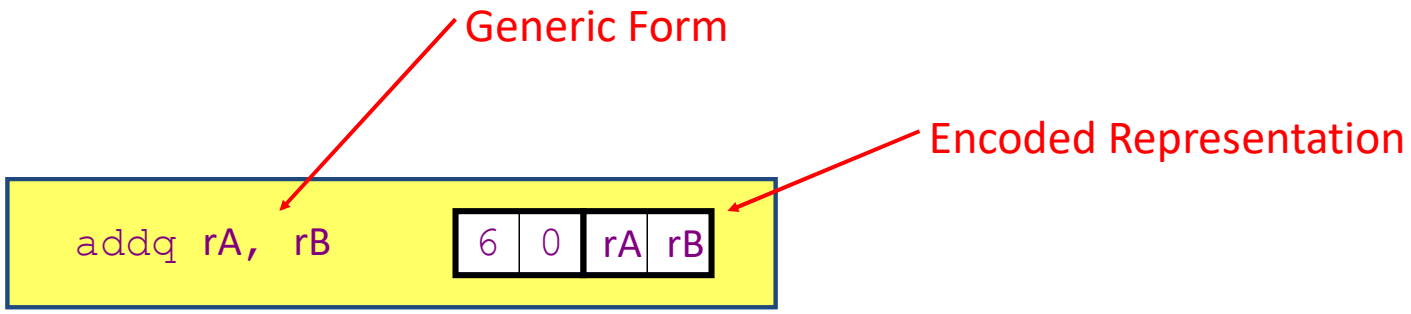
Y86-64 Instructions

- Format
  - 1–10 bytes of information read from memory
    - Can determine instruction length from first byte
    - Not as many instruction types, and simpler encoding than with x86-64
  - Each accesses and modifies some part(s) of the program state

Y86-64 Instruction Set



Instruction Example



- Addition Instruction
  - Add value in register rA to that in register rB
    - Store result in register rB
    - Y86-64 only allows addition to be applied to register data
  - Set condition codes based on result
  - e.g., **`addq %rax,%rsi`** Encoding: **`60 06`**
  - Two-byte encoding
    - First indicates instruction type
    - Second gives source and destination registers

Move Operations

Register ➔ Register

Immediate ➔ Register

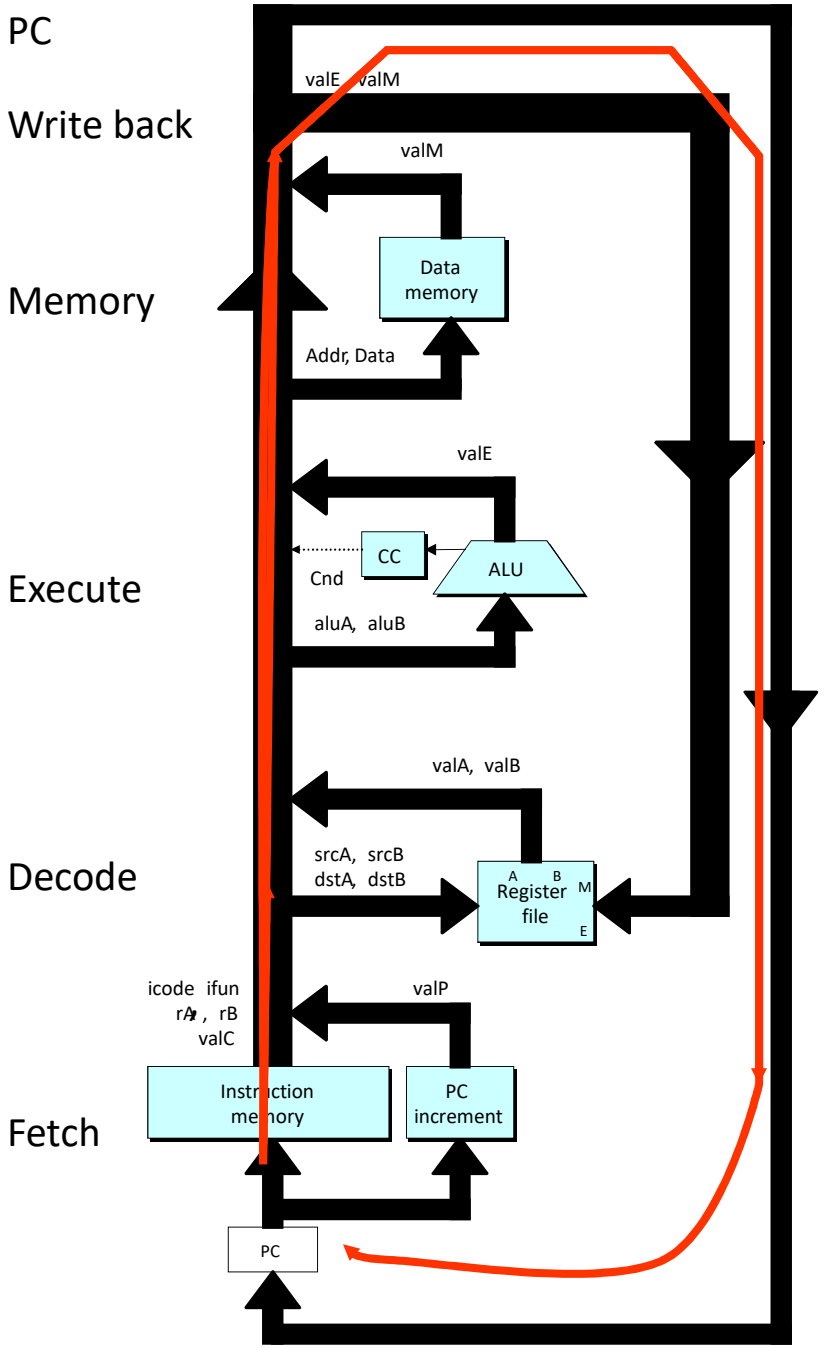
Register ➔ Memory

Memory ➔ Register

- Like the x86-64 `movq` instruction
- Simpler format for memory addresses
- Different names to keep them distinct

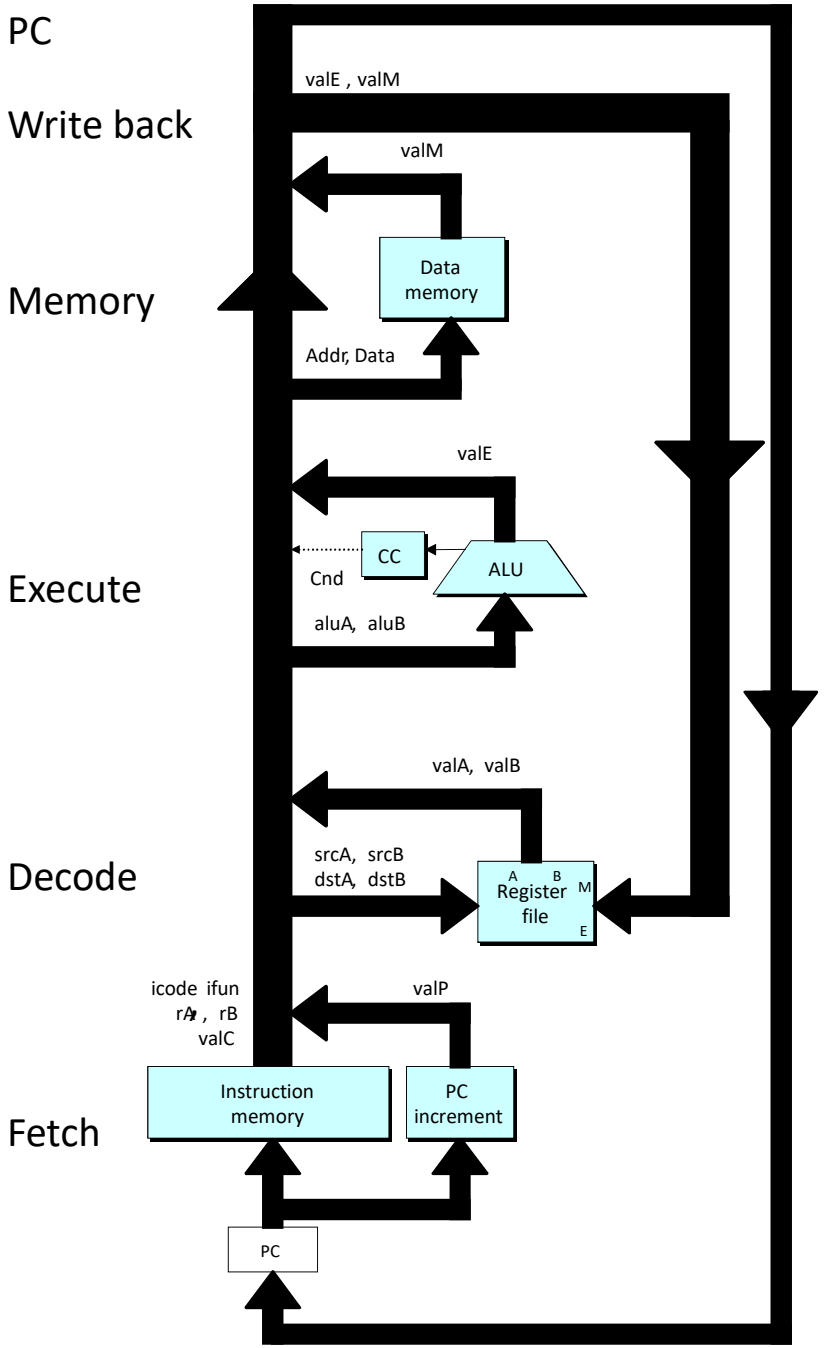
Sequential (SEQ) Hardware Structure

- State
  - 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
  - Read instruction at address specified by PC
  - Process through stages
  - Update program counter

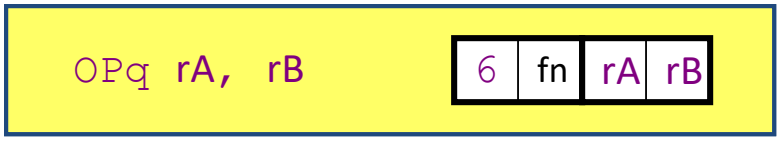


SEQ Stages

- 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



Executing Arith./Logical Operation



- 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: Arith/Log. Ops

	OPq 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 ← 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

Executing `rmmovq`

- 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 ← 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

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

Real-World Pipelines: Car Washes

Sequential

Parallel

Pipelined

- Idea
  - Divide process into independent stages
  - Move objects through stages in sequence
  - At any given times, multiple objects being processed
  - Spraying water and soap, scrubbing, applying wax, drying etc.