Systems I

Datapath Design I

Topics

- Sequential instruction execution cycle
- Instruction mapping to hardware
- Instruction decoding

Overview

How do we build a digital computer?

- Hardware building blocks: digital logic primitives
- Instruction set architecture: what HW must implement

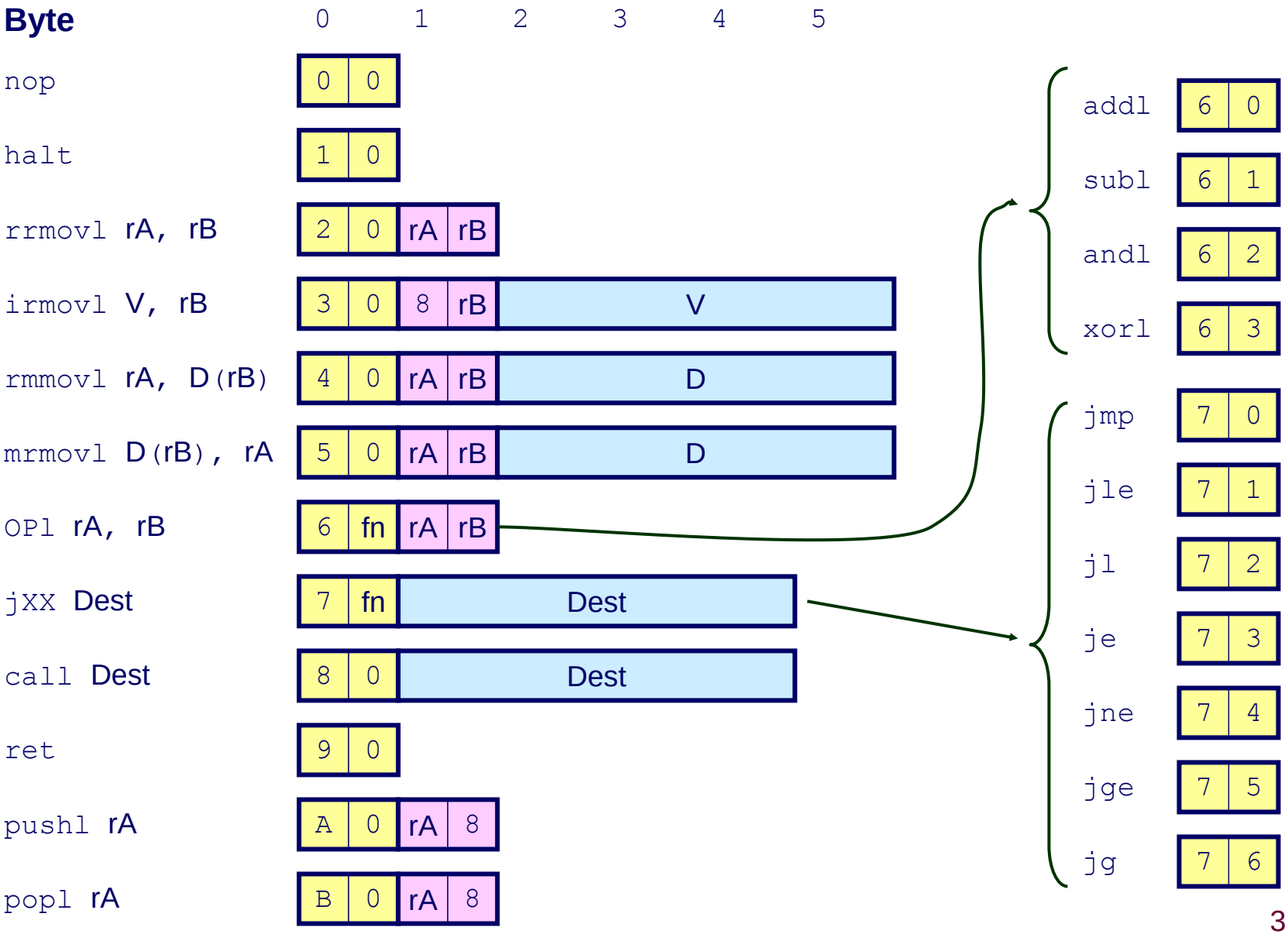
Principled approach

- Hardware designed to implement one instruction at a time
  - Plus connect to next instruction
- Decompose each instruction into a series of steps
  - Expect that most steps will be common to many instructions

Extend design from there

- Overlap execution of multiple instructions (pipelining)
  - Later in this course
- Parallel execution of many instructions
  - In more advanced computer architecture course

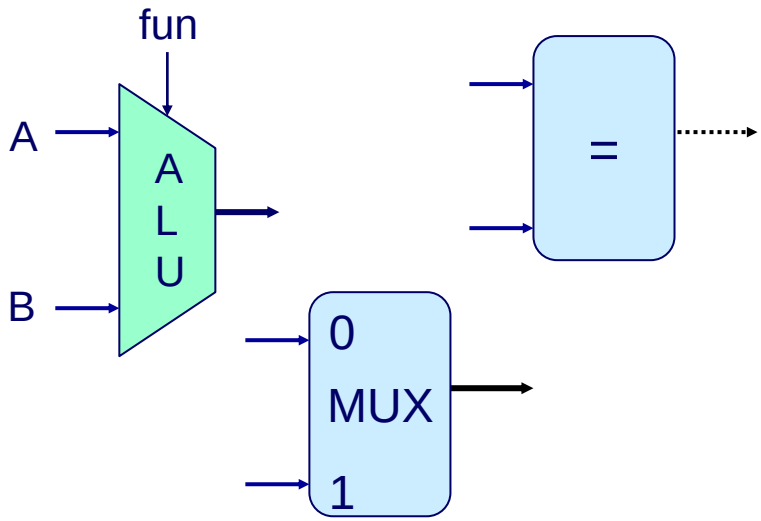
Y86 Instruction Set



Building Blocks

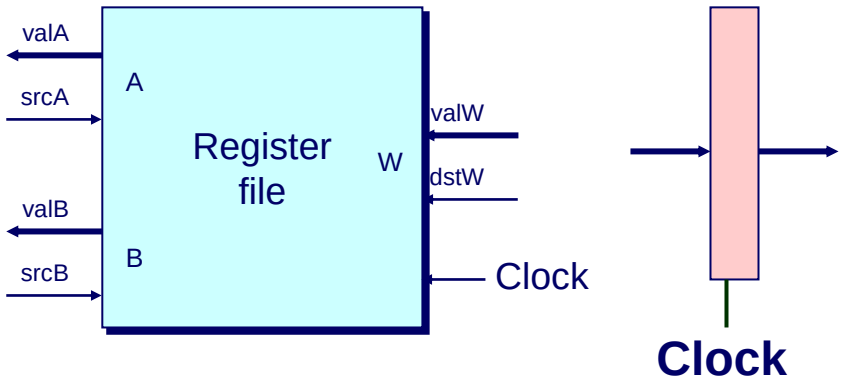
Combinational Logic

- Compute Boolean functions of inputs
- Continuously respond to input changes
- Operate on data and implement control



Storage Elements

- Store bits
- Addressable memories
- Non-addressable registers
- Loaded only as clock rises



Hardware Control Language

- Very simple hardware description language
- Can only express limited aspects of hardware operation
  - Parts we want to explore and modify

Data Types

- `bool`: Boolean
  - `a`, `b`, `c`, …
- `int`: words
  - `A`, `B`, `C`, …
  - Does not specify word size---bytes, 32-bit words, …

Statements

- `bool a =` ***`bool-expr`*** `;`
- `int A =` ***`int-expr`*** `;`

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

An Abstract Processor

What does a processor do?

Consider a processor that only executes nops.

```
void be_a_processor(unsigned int   pc,
                    unsigned char* mem){
    while(1) {
      char opcode = mem[pc];
      assert(opcode == NOP);
      pc = pc + 1;
    }
}
```

Fetch

Decode

Execute

An Abstract Processor

Executes nops and absolute jumps

```
void be_a_processor(unsigned int pc,
                    unsigned char* mem){
   while(1) {
     char opcode = mem[pc];
     switch (opcode) {
        case NOP: pc++;
        case JMP: pc = *(int*)&mem[(pc+1)];
```

Missing execute and memory access

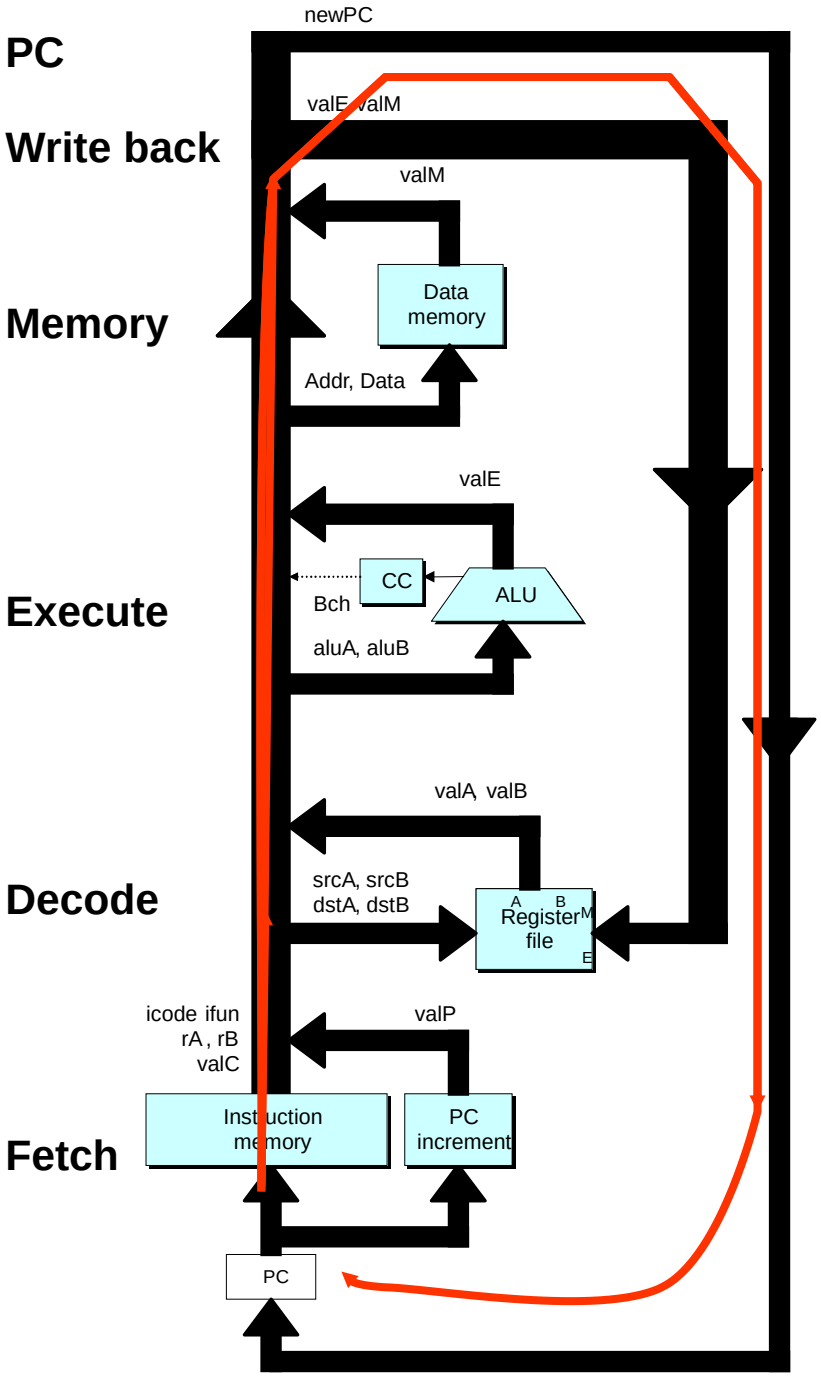
SEQ Hardware Structure

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

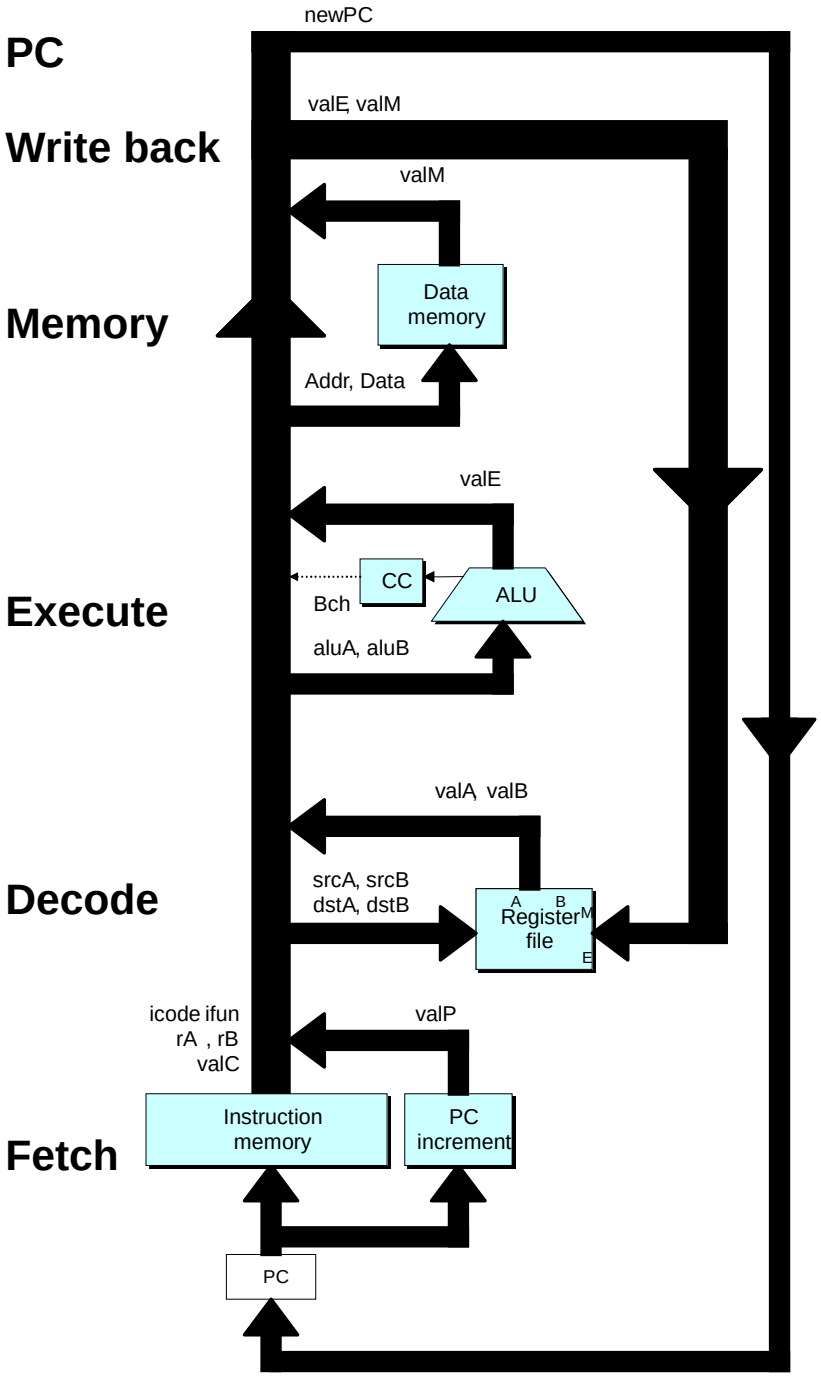
- Read or write data

Write Back

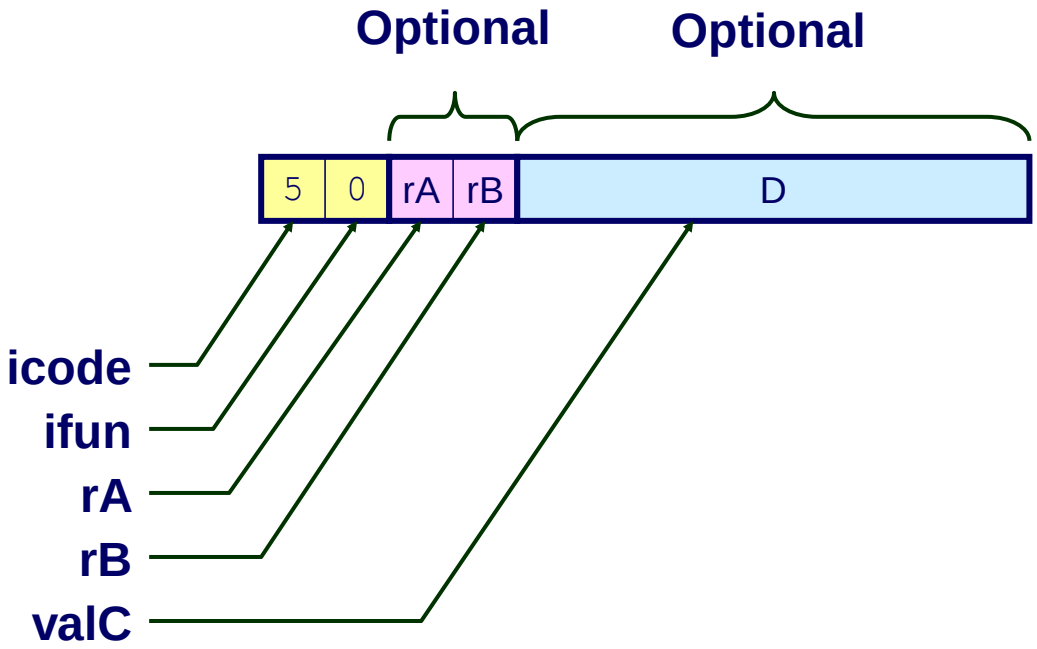
- Write program registers

PC

- Update program counter



Instruction Decoding



Instruction Format

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

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
- Why?

Stage Computation: Arith/Log. Ops

	OPl 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 `rmmovl`

Fetch

- Read 6 bytes

Decode

- Read operand registers

Execute

- Compute effective address

Memory

- Write to memory

Write back

- Do nothing

PC Update

- Increment PC by 6

Stage Computation: `rmmovl`

	`rmmovl` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_4[PC+2]	Read displacement D
	valP ← PC+6	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_4[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- Use ALU for address computation

Executing `popl`

Fetch

- Read 2 bytes

Decode

- Read stack pointer

Execute

- Increment stack pointer by 4

Memory

- Read from old stack pointer

Write back

- Update stack pointer
- Write result to register

PC Update

- Increment PC by 2

Stage Computation: `popl`

	`popl` 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[`%esp`]	Read stack pointer
	valB ← R [`%esp`]	Read stack pointer
Execute	valE ← valB + 4	Increment stack pointer
Memory	valM ← M_4[valA]	Read from stack
Write back	R[`%esp`] ← 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

Summary

Today

- Sequential instruction execution cycle
- Instruction mapping to hardware
- Instruction decoding

Next time

- Control flow instructions
- Hardware for sequential machine (SEQ)