CSE 502: Computer Architecture

Basic Instruction Decode

RISC ISA Format

- Fixed-length
  - MIPS all insts are 32-bits/4 bytes
- Few formats
  - MIPS has 3: R (reg, reg, reg), I (reg, reg, imm), J (addr)
  - Alpha has 5: Operate, Op w/ Imm, Mem, Branch, FP
- Regularity across formats (when possible/practical)
  - MIPS & Alpha opcode in same bit-position for all formats
  - MIPS rs & rt fields in same bit-position for R and I formats
  - Alpha ra/fa field in same bit-position for all 5 formats

RISC Decode (MIPS)

opcode[2,0]

opcode[5,3]	[illegible]	[illegible]	010	011	100	101	110	111
000	*func*	*rt*	j	jal	beq	bne	blez	bgtz
001	addi	addiu	slti	sltiu	andi	ori	xori	lui
010			[illegible]	rs				
011								
100	lb	lh	lwl	lw	lbu	lhu	lwr	
101	sb	sh	swl	sw			swr	
110	lwc0	lwc1	lwc2	lwc3				
111	swc0	swc1	swc2	swc3				

000xxx = Br/Jump
(except for 000000)

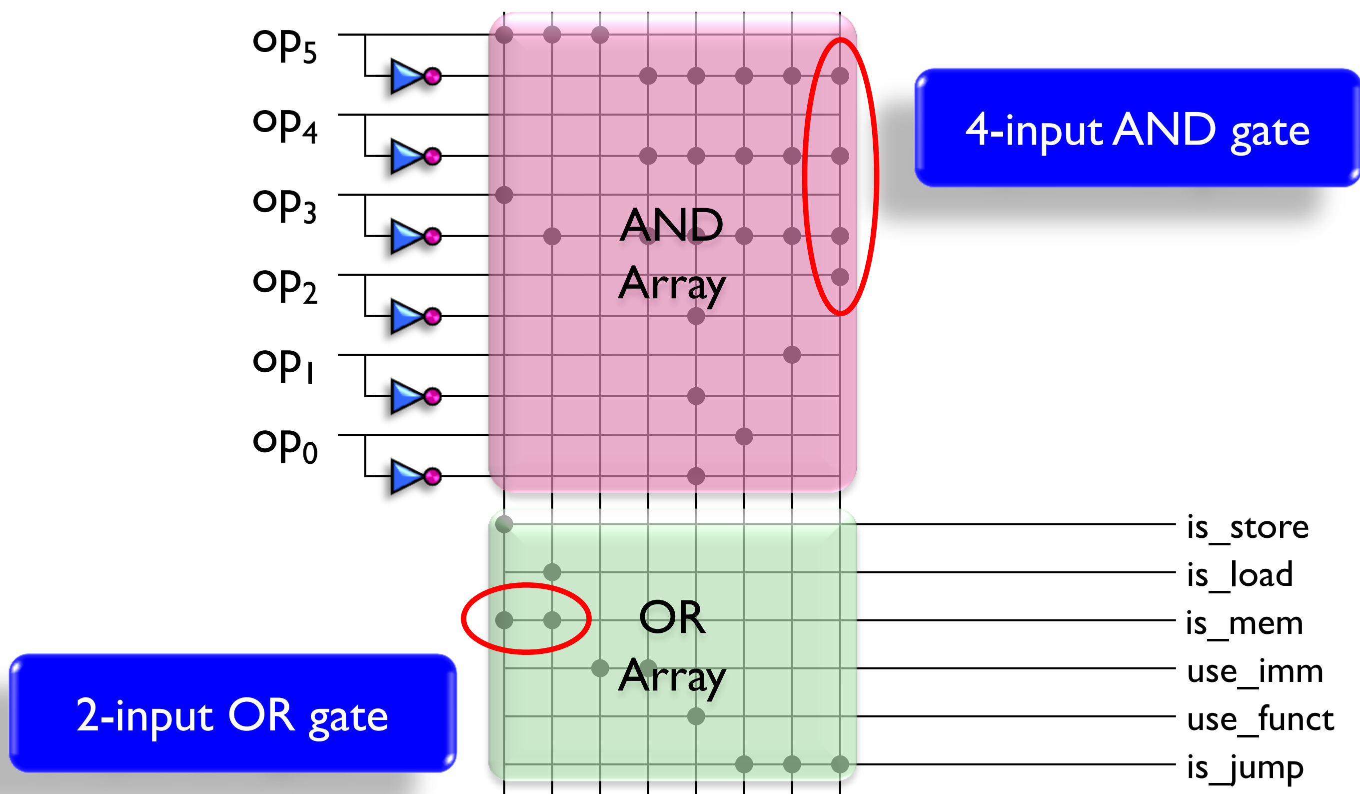
001xxx = Immediate

1xxxxx = Memory
(1x0: LD, 1x1: ST)

PLA Decoders (1/2)

- PLA = Programmable Logic Array
- Simple logic to transform opcode to control signals
  - is_jump = !op5 & !op4 & !op3 & (op2 | op1 | op0)
  - use_funct = !op5 & !op4 & !op3 & !op2 & !op1 & !op0
  - use_imm = op5 | !op5 & !op4 & op3
  - is_load = op5 & !op3
  - is_store = op5 & op3

PLA Decoders (2/2)



CISC ISA

- RISC focus on fast access to information
  - Easy decode, many registers, fast caches
- CISC focus on max expressiveness per min space
  - Designed in era with fewer transistors
  - Each memory access very expensive
    - Pack as much work into as few bytes as possible
    - More “expressive” instructions
      - Better potential code generation in theory
      - More complex code generation in practice

ADD in RISC ISA

Mode	Example	Meaning
Register	ADD R4, R3, R2	R4 <= R3 + R2

ADD in CISC ISA

Mode	Example	Meaning
Register	ADD R4, R3	R4 <= R4 + R3
Immediate	ADD R4, #3	R4 <= R4 + 3
Register Indirect	ADD R4, (R1)	R4 <= R4 + Mem[R1]
Displacement	ADD R4, 100(R1)	R4 <= R4 + Mem[100+R1]
Indexed/Base	ADD R3, (R1+R2)	R3 <= R3 + Mem[R1+R2]
Direct/Absolute	ADD R1, (1234)	R1 <= R1 + Mem[1234]
Memory Indirect	ADD R1, @(R3)	R1 <= R1 + Mem[Mem[R3]]
Auto-Increment	ADD R1,(R2)+	R1 <= R1 + Mem[R2]; R2++
Auto-Decrement	ADD R1, -(R2)	R2--; R1 <= R1 + Mem[R2]

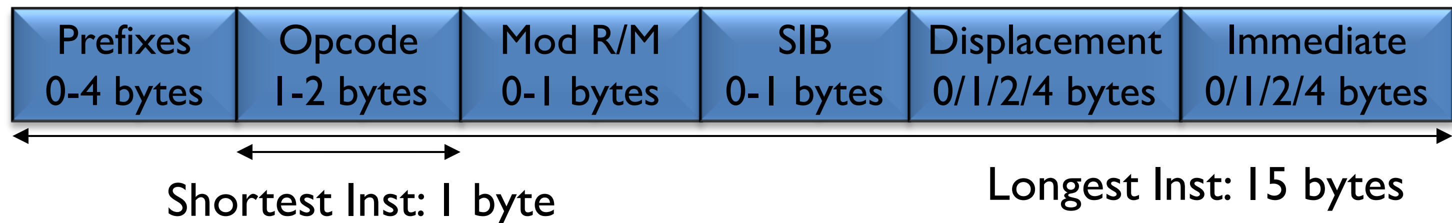
x86

- CISC, stemming from the original 4004 (~1971)
- Example: “Move” instructions
  - General Purpose data movement
    - R→R, M→R, R→M, I→R, I→M
  - Exchanges
    - EAX ↔ ECX, byte order within a register
  - Stack Manipulation
    - push / pop R ↔ Stack, pusha/popa (removed in 64-bit, thankfully)
  - Type Conversion
  - Conditional Moves

Many ways to do the same/similar operation

x86 Encoding

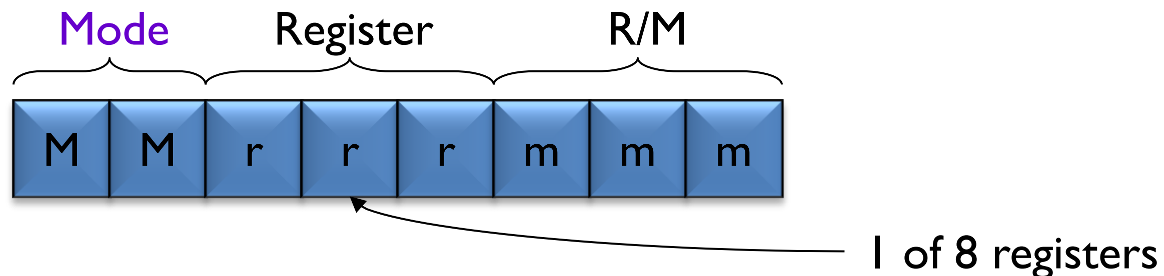
- x86 Instruction Format:



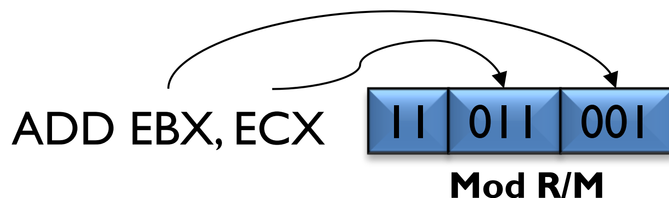
- Opcode indicates if Mod R/M is present
  - Many (not all) instructions use the Mod R/M byte
  - Mod R/M specifies if optional SIB byte is used
  - Mod R/M and SIB may specify additional constants
    - Displacement, Immediate

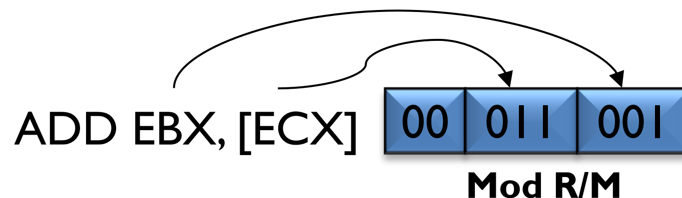
Instruction length not known until *after* decode

x86 Mod R/M Byte



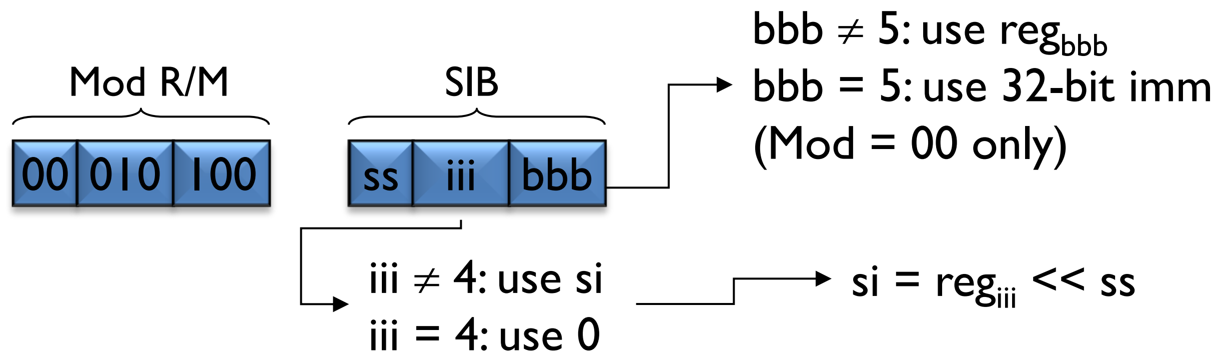
- Mode = 00: No-displacement, use Mem[Rmmm]
- Mode = 01: 8-bit displacement, Mem[Rmmm+disp)]
- Mode = 10: 32-bit displacement (similar to previous)
- Mode = 11: Register-to-Register, use Rmmm





x86 Mod R/M Exceptions

- Mod=00, R/M = 5 → get operand from 32-bit imm
  - Add | 00 | 010 | 101 | 0cff1234 | EDX = EDX+Mem[0cff1234]
- Mod=00, 01, or 10, R/M = 4 → use the "SIB" byte
  - SIB = Scale/Index/Base



x86 Opcode Confusion

- There are different opcodes for A←B and B→A

Opcode	ModR/M	Instruction
10001011	11 000 011	MOV EAX, EBX
10001001	11 000 011	MOV EBX, EAX
10001001	11 011 000	MOV EAX, EBX

- If Opcode = 0F, then use next byte as opcode
- If Opcode = D8-DF, then FP instruction

11011000 | 11 | [FP opcode] | R/M

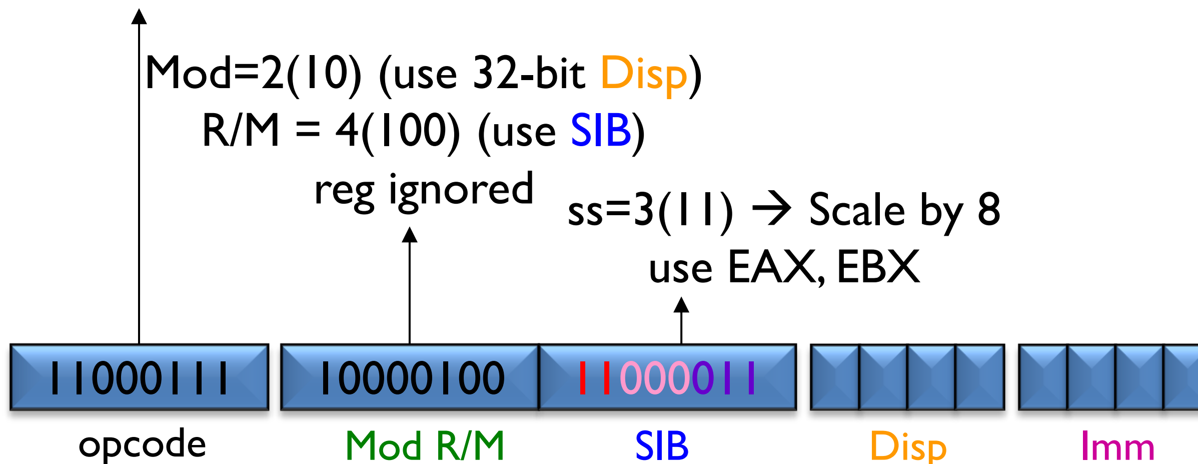
FP opcode

x86 Decode Example

```
struct { long a; long b; long c;} myobj[5];
myobj[2]->b = 0xF001234;
```

`MOV 0xF001234,0x8(%ebx,%eax,8)`

MOV reg←imm (store 32-bit Imm in reg ptr, use Mod R/M)



*((EAX<<3) + EBX + Disp) = Imm

Total: 11 byte instruction

Note: Add 4 prefixes, and you reach the max size

RISC (MIPS) vs CISC (x86)

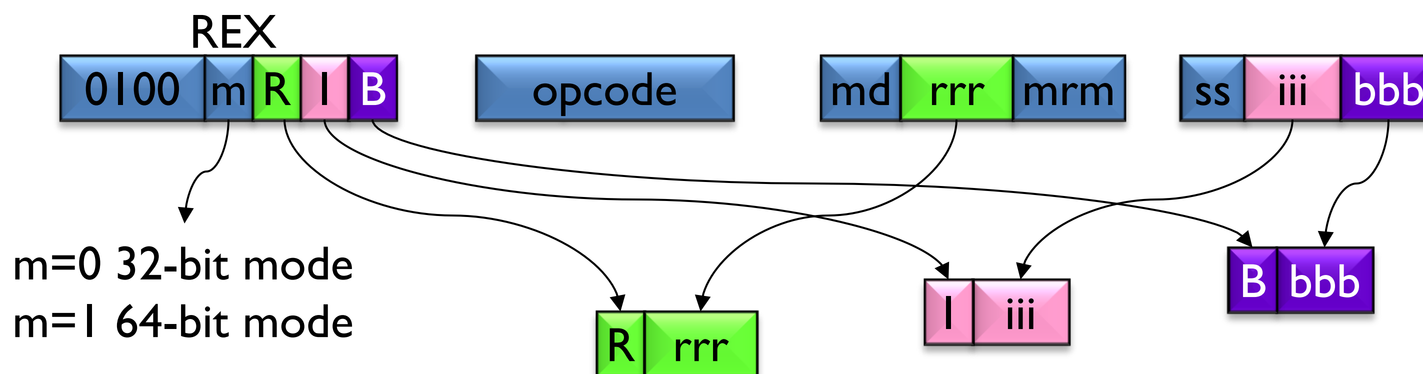
```
lui R1, Disp[31:16]
ori R1, R1, Disp[15:0]
add R1, R1, R2
shli R3, R3, 3
add R3, R3, R1
lui R1, Imm[31:16]
ori R1, R1, Imm[15:0]
st [R3], R1
```

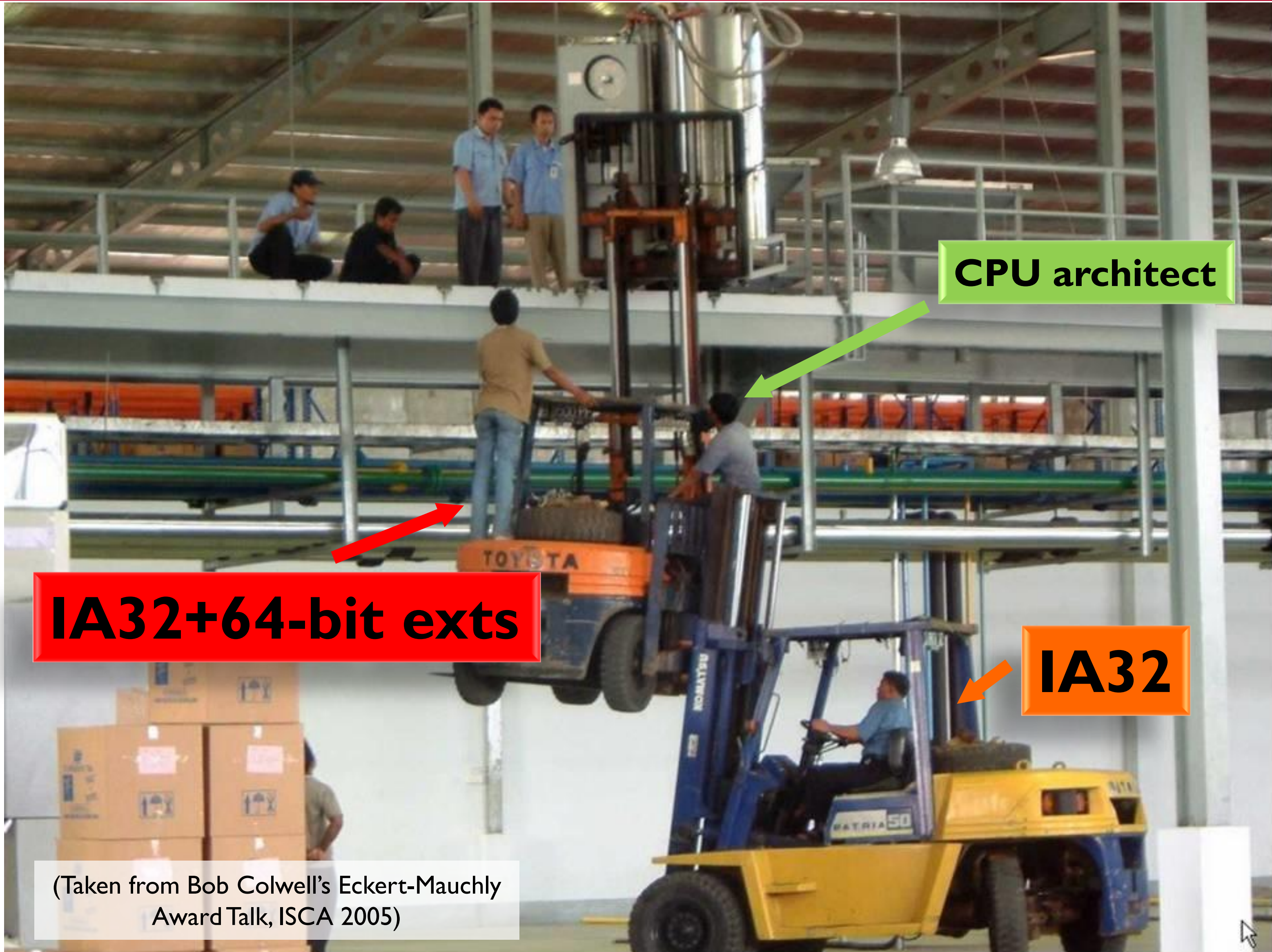
```
MOV Imm, Disp(%EBX,%EAX,8)
```

8 insns. at 32 bits each ***vs*** 1 insn. at 88 bits: ***2.9x!***

x86-64 / EM64T

- 8→16 general purpose registers
  - But we only used to have 3-bit register fields...
- Registers extended from 32→64 bits each
- Default: instructions still 32-bit
  - New “REX” prefix byte to specify additional information





(Taken from Bob Colwell's Eckert-Mauchly Award Talk, ISCA 2005)

Ugly? Scary? But it works…

Decoded x86 Format

- RISC: easy to expand → union of needed info
  - Generalized opcode (not too hard)
  - reg1, reg2, reg3, immediate (possibly extended)
  - Some fields ignored
- CISC: union of all possible info is huge
  - Generalized opcode (too many options)
  - Up to 3 regs, 2 immediates

Too expensive to decode x86 into control bits

x86 → RISC-like mops

- x86 decoded into “uops”(Intel) or “ROPs”(AMD) … (micro-ops or RISC-ops)
  - Each uop is RISC-like
  - uops have limitations to keep union of info practical

x86		uops	count
ADD EAX, EBX	→	ADD EAX, EBX	1 uop
ADD EAX, [EBX]	→	LOAD tmp = [EBX] ADD EAX, tmp	2 uops
ADD [EAX], EBX	→	LOAD tmp = [EAX] ADD tmp, EBX STA EAX STD tmp	4 uops