CSE 502: Computer Architecture

Instruction Decode

RISC ISA Format

- This should be review…
  - Fixed-length
    - MIPS all insts are 32-bits/4 bytes
  - Few formats
    - MIPS has 3 formats: 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)

6	21	5
opcode	other	func

R-format only

000xxx = Br/Jump
(except for 000000)

001xxx = Immediate

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

opcode[2,0]

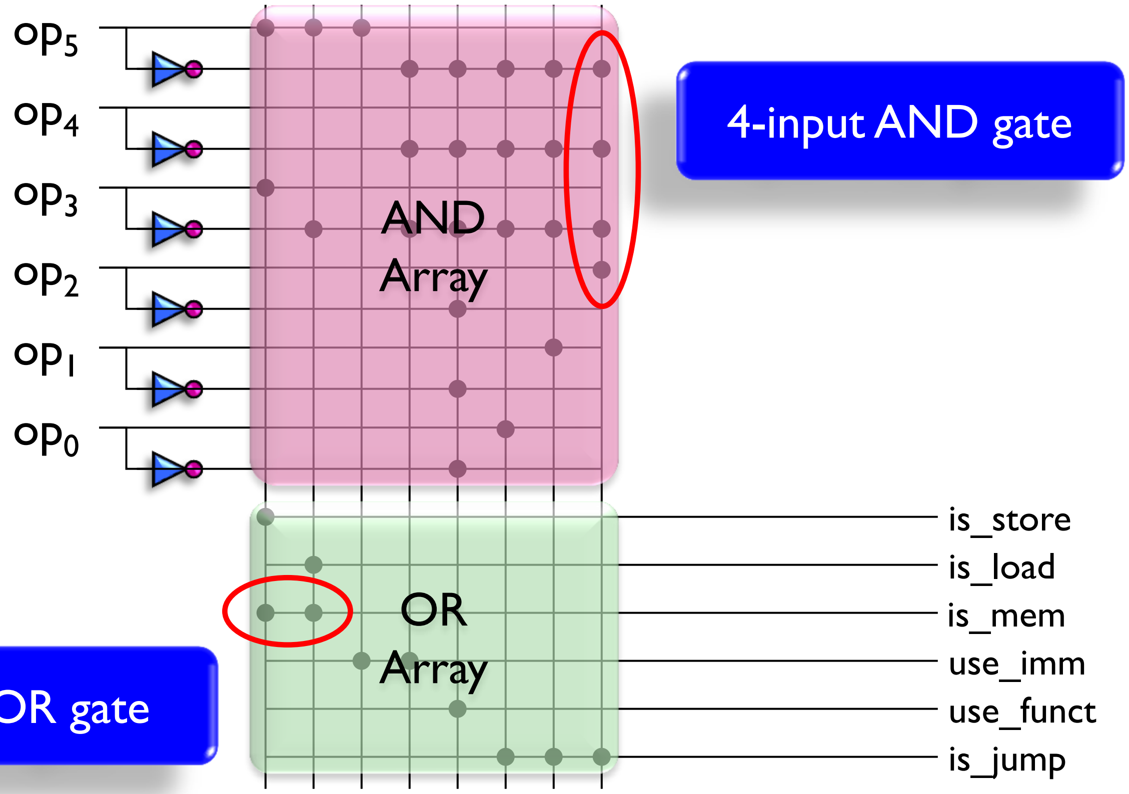
e[5,3]

	000	001	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			s	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				

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)

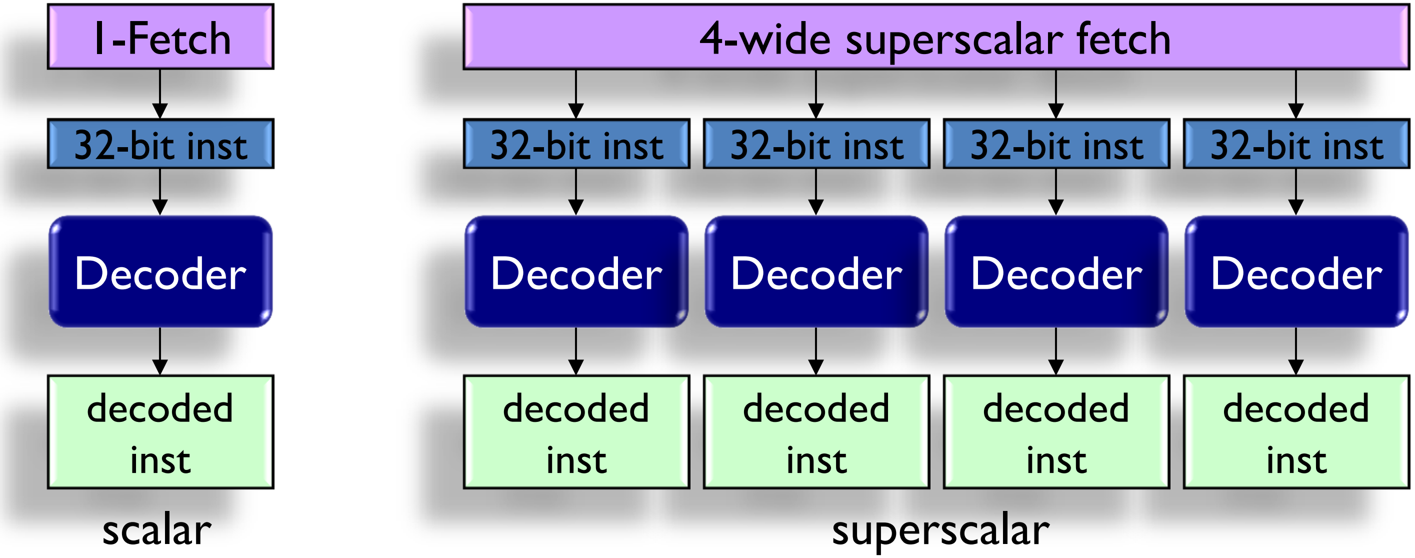


4-input AND gate

2-input OR gate

Superscalar Decode for RISC ISAs

- Decode X insns. per cycle (e.g., 4-wide)
  - Just duplicate the hardware
  - Instructions aligned at 32-bit boundaries



CISC ISA

- RISC focus on fast access to information
  - Easy decode, I$, large RF’s, D$
- CISC focus on max expressiveness per min space
  - Designed in era with fewer transistors, chips
  - 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
Displacement	ADD R4, 100(R1)	R4 = R4 + Mem[100+R1]
Register Indirect	ADD R4, (R1)	R4 = R4 + Mem[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]

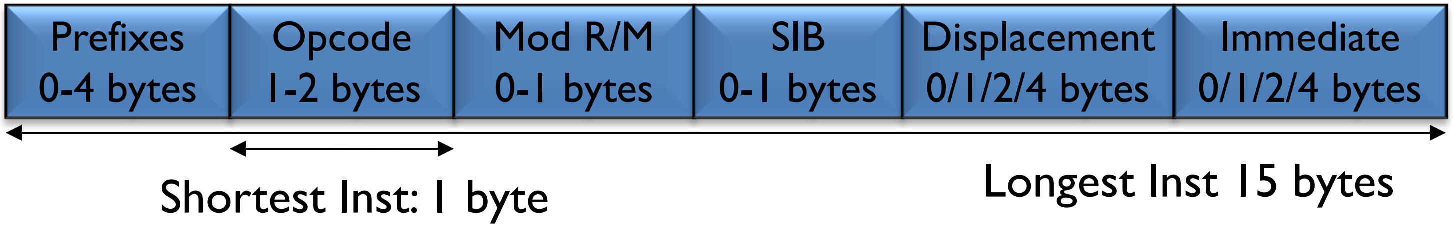
x86

- 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
  - Type Conversion
  - Conditional Moves

Many ways to do the same/similar operation

x86 Encoding

- Basic x86 Instruction:



Prefixes	Opcode	Mod R/M	SIB	Displacement	Immediate
0-4 bytes	1-2 bytes	0-1 bytes	0-1 bytes	0/1/2/4 bytes	0/1/2/4 bytes

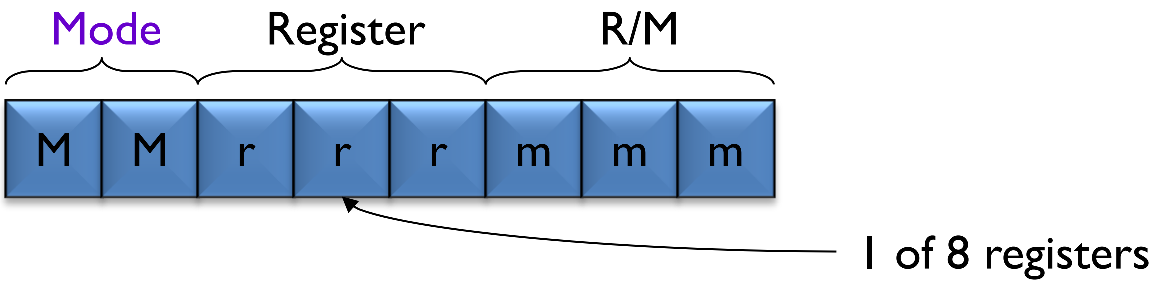
Shortest Inst: 1 byte

Longest Inst 15 bytes

- Opcode has flag indicating Mod R/M is present
  - Most 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

Instruction length not known until *after* decode

x86 Mod R/M Byte

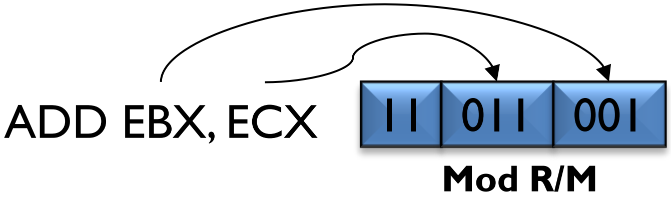


Mode Register R/M

M M r r r m m m

1 of 8 registers

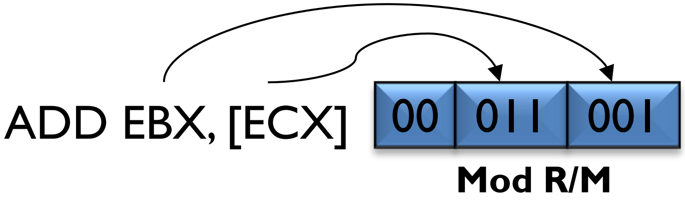
- Mode = 00: No-displacement, use Mem[Rmmm]
- Mode = 01: 8-bit displacement, Mem[Rmmm+disp)]
- Mode = 10: 32-bit displacement (similar to previous)
- Mode = 11: Register-to-Register, use Rmmm



ADD EBX, ECX

11 | 011 | 001

Mod R/M



ADD EBX, [ECX]

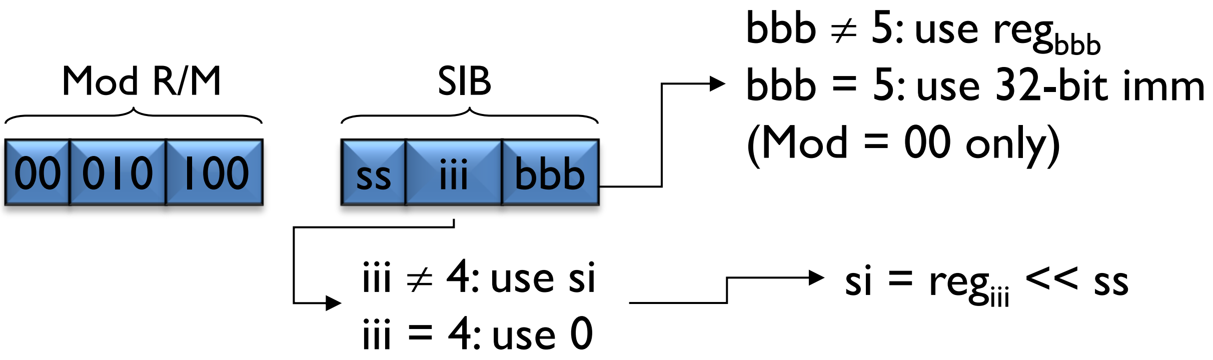
00 | 011 | 001

Mod R/M

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



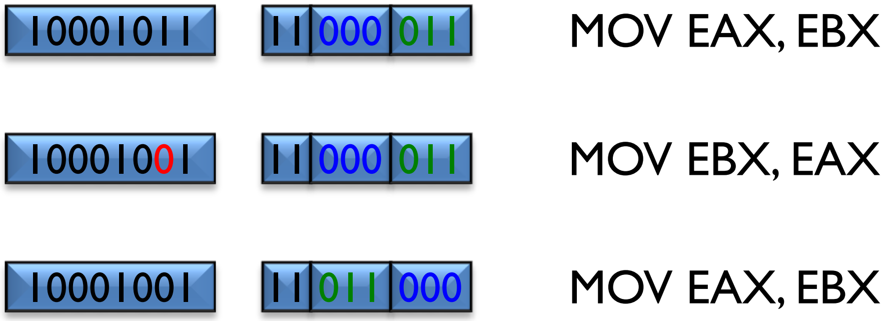
bbb ≠ 5: use reg_{bbb}
bbb = 5: use 32-bit imm
(Mod = 00 only)

iii ≠ 4: use si
iii = 4: use 0

si = reg_{iii} << ss

x86 Opcode Confusion

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

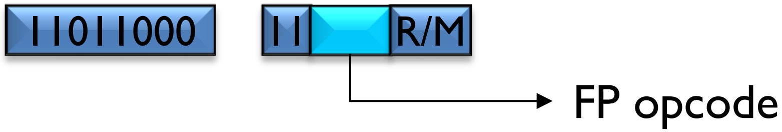


10001011 | 11 | 000 | 011 — MOV EAX, EBX

10001001 | 11 | 000 | 011 — MOV EBX, EAX

10001001 | 11 | 011 | 000 — MOV EAX, EBX

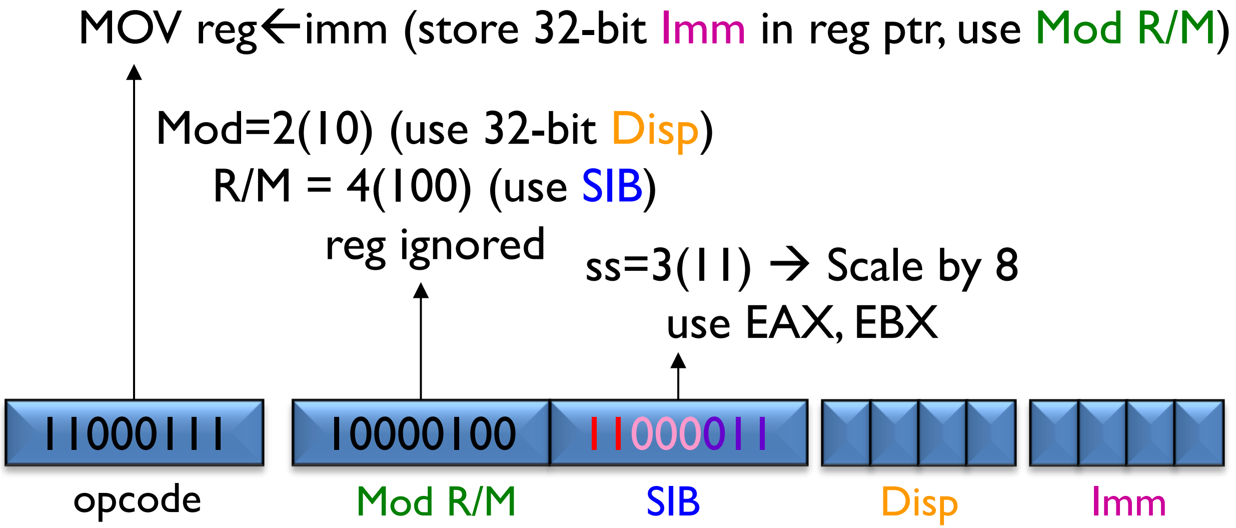
- If Opcode = 0F, then use next byte as opcode
- If Opcode = D8-DF, then FP instruction



11011000 | 11 | | R/M

FP opcode

x86 Decode Example



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

Total: 11 byte instruction

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

RISC (MIPS) vs CISC (x86)

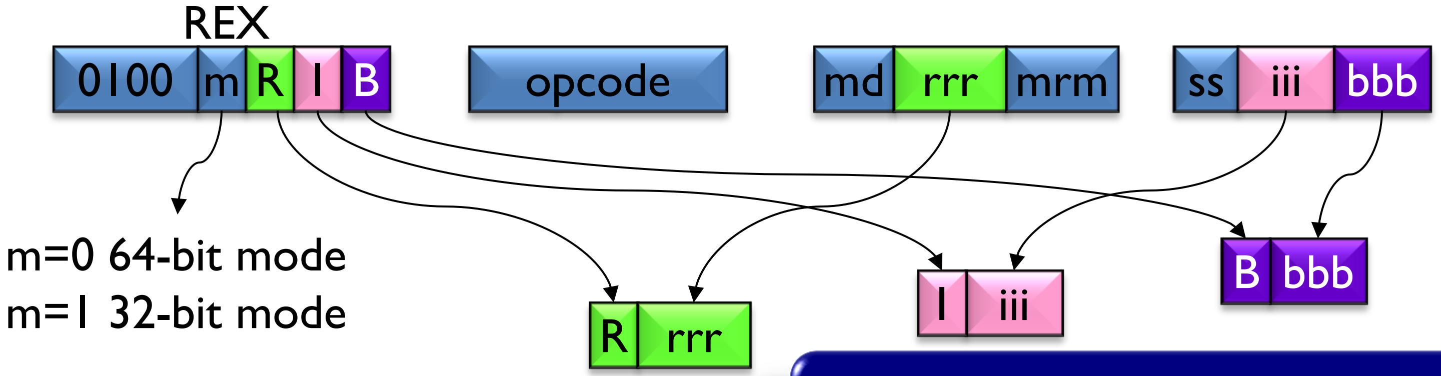
```
lui R1, Disp[31:16]
ori R1, R1, Disp[15:0]
add R1, R1, R2
shli R3, R3, 3
add R3, R3, R1
lui R1, Imm[31:16]
ori R1, R1, Imm[15:0]
st [R3], R1
```

```
MOV [EBX+EAX*8+Disp], Imm
```

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



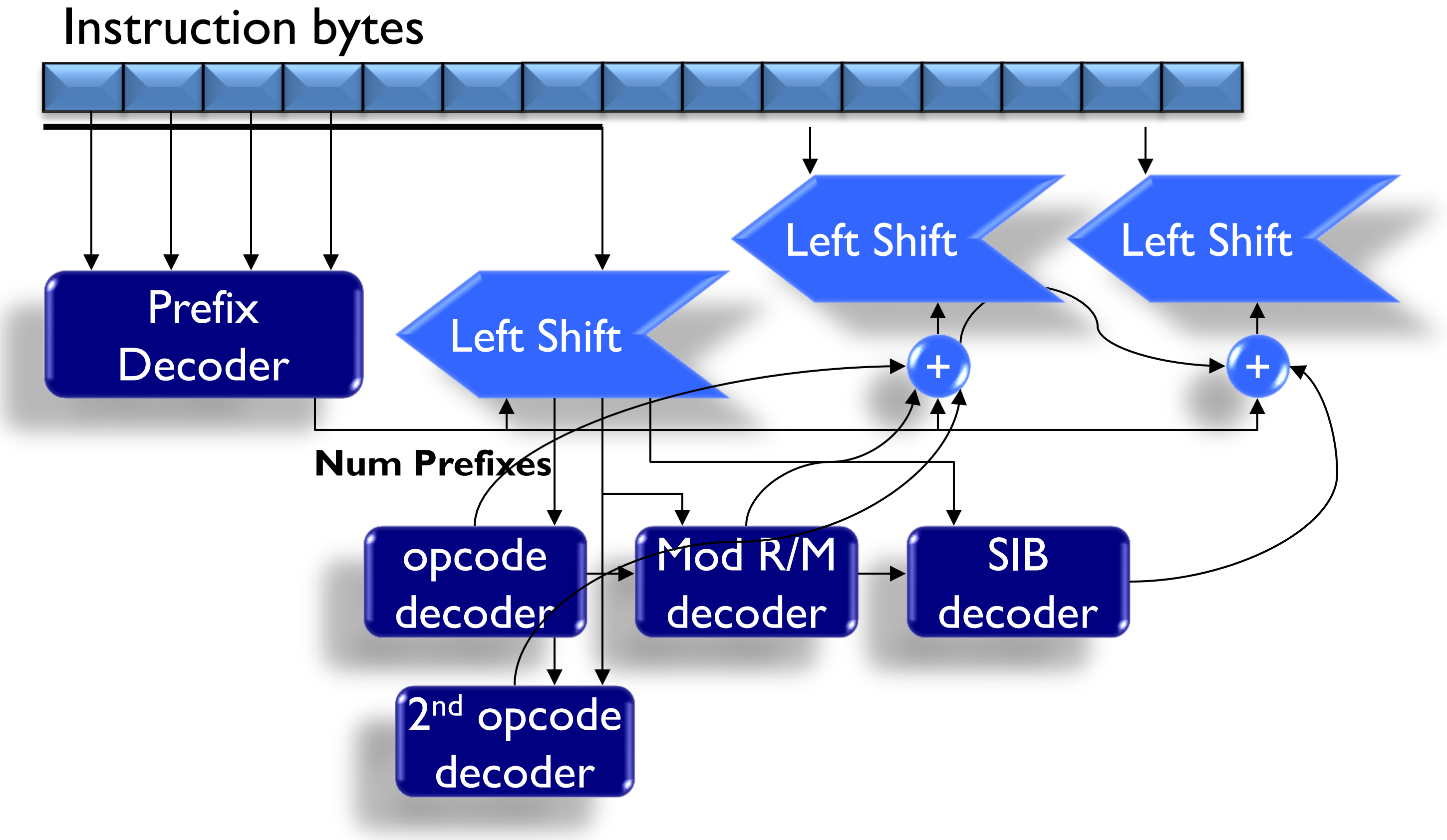
m=0 64-bit mode
m=1 32-bit mode

Register specifiers are now 4 bits each: can choose 1 of 16 registers

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

Ugly? Scary? But it works…

x86 Decode Hardware



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
  - Segment information
  - “rep” specifiers
    - Would lead to 100’s of bits
    - Common case only needs a fraction → a lot of waste

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

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

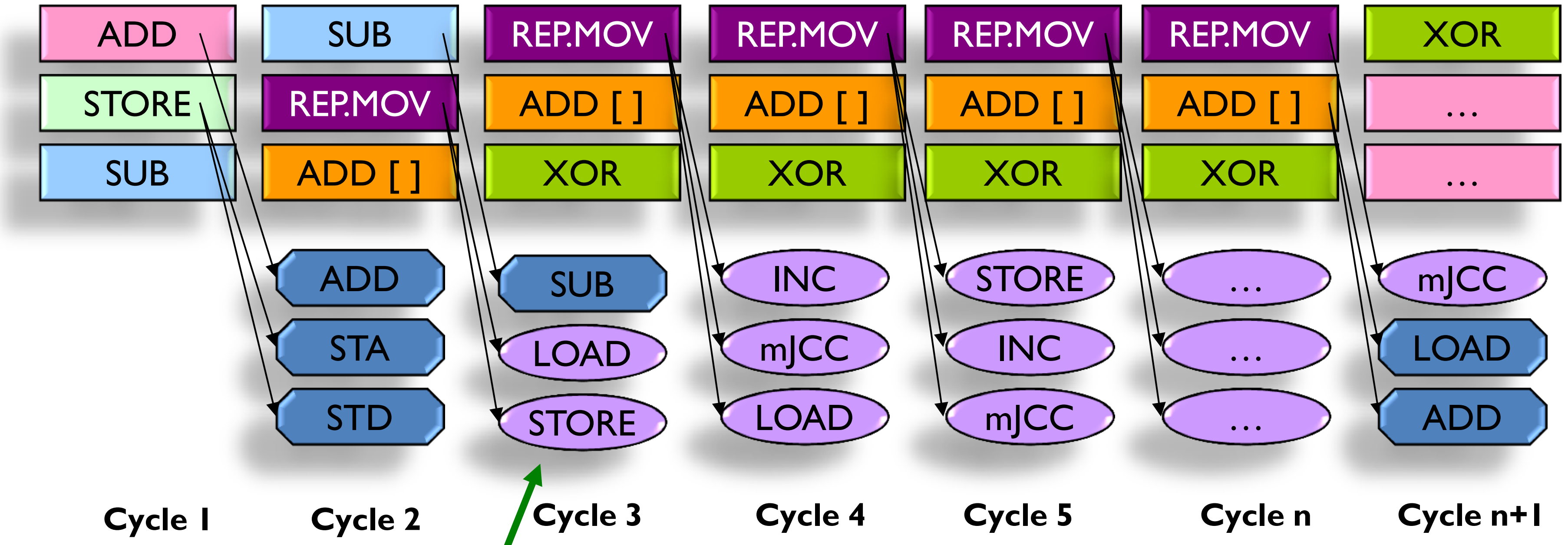
uop Limits

- How many uops can a decoder generate?
  - For complex x86 insts, many are needed (10’s, 100’s?)
  - Makes decoder horribly complex
  - Typically there’s a limit to keep complexity under control
    - One x86 instruction → 1-4 uops
    - Most instructions translate to 1.5-2.0 uops
- What if a complex insn. needs more than 4 uops?

UROM/MS for Complex x86 Insts

- UROM (microcode-ROM) stores the uop equivalents
  - Used for nasty x86 instructions
    - Complex like > 4 uops (PUSHA or STRREP.MOV)
    - Obsolete (like AAA)
- Microsequencer (MS) handles the UROM interaction

UROM/MS Example (3 uop-wide)



Complex instructions, get uops from mcode sequencer

Fetch- x86 insts

Decode - uops

UROM - uops

Superscalar CISC Decode

- Instruction Length Decode (ILD)
  - – Where are the instructions?
    - Limited decode – just enough to parse prefixes, modes
- Shift/Alignment
  - – Get the right bytes to the decoders
- Decode
  - – Crack into uops

Do this for N instructions per cycle!

ILD Recurrence/Loop

PCi = X

PCi+1 = PCi + sizeof(Mem[PCi])

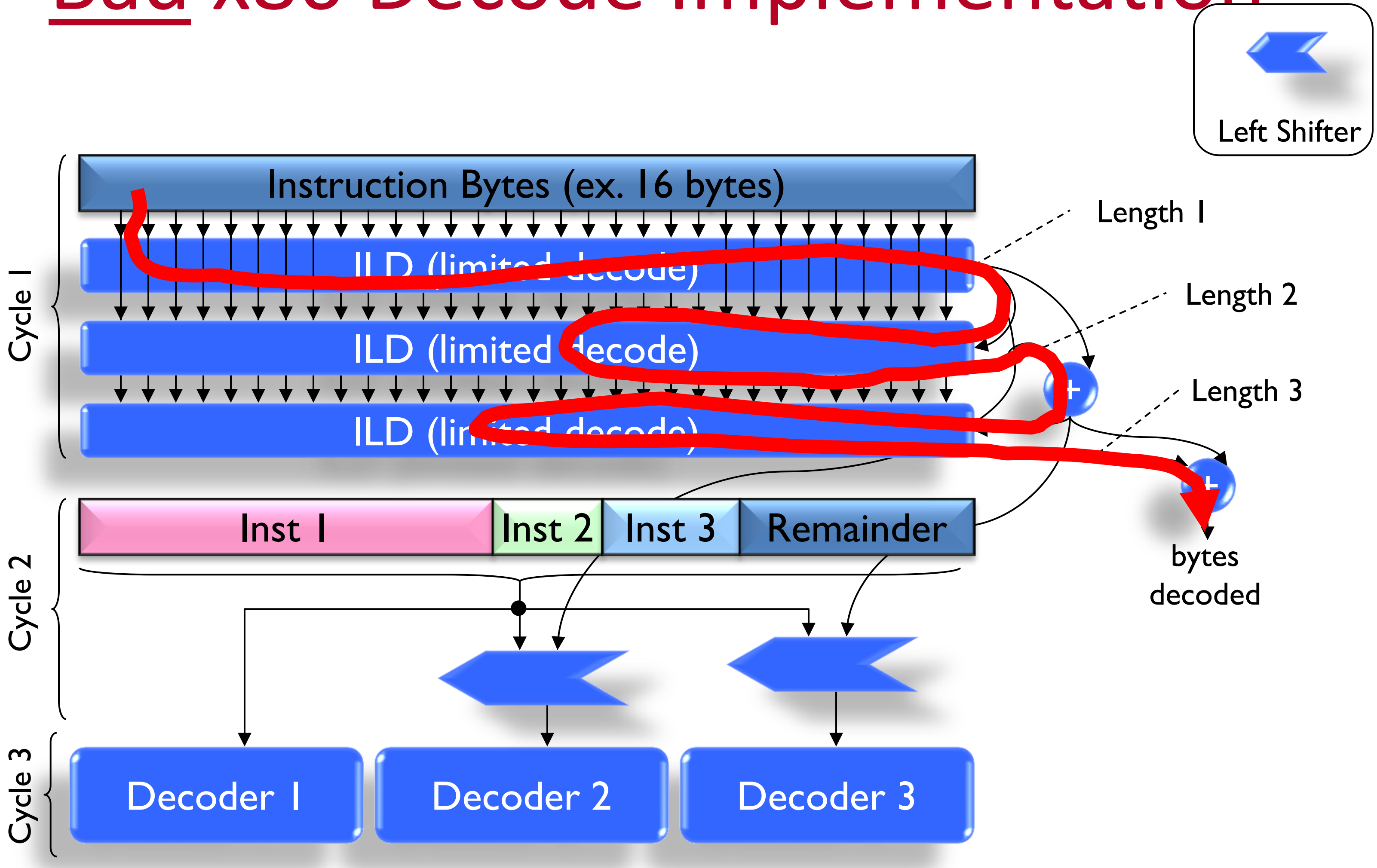
PCi+2 = PCi+1 + sizeof(Mem[PCi+1])

= PCi + sizeof(Mem[PCi]) + sizeof(Mem[PCi+1])

- Can’t find start of next insn. before decoding the first
- Must do ILD serially
  - ILD of 4 insns/cycle implies cycle time will be 4x

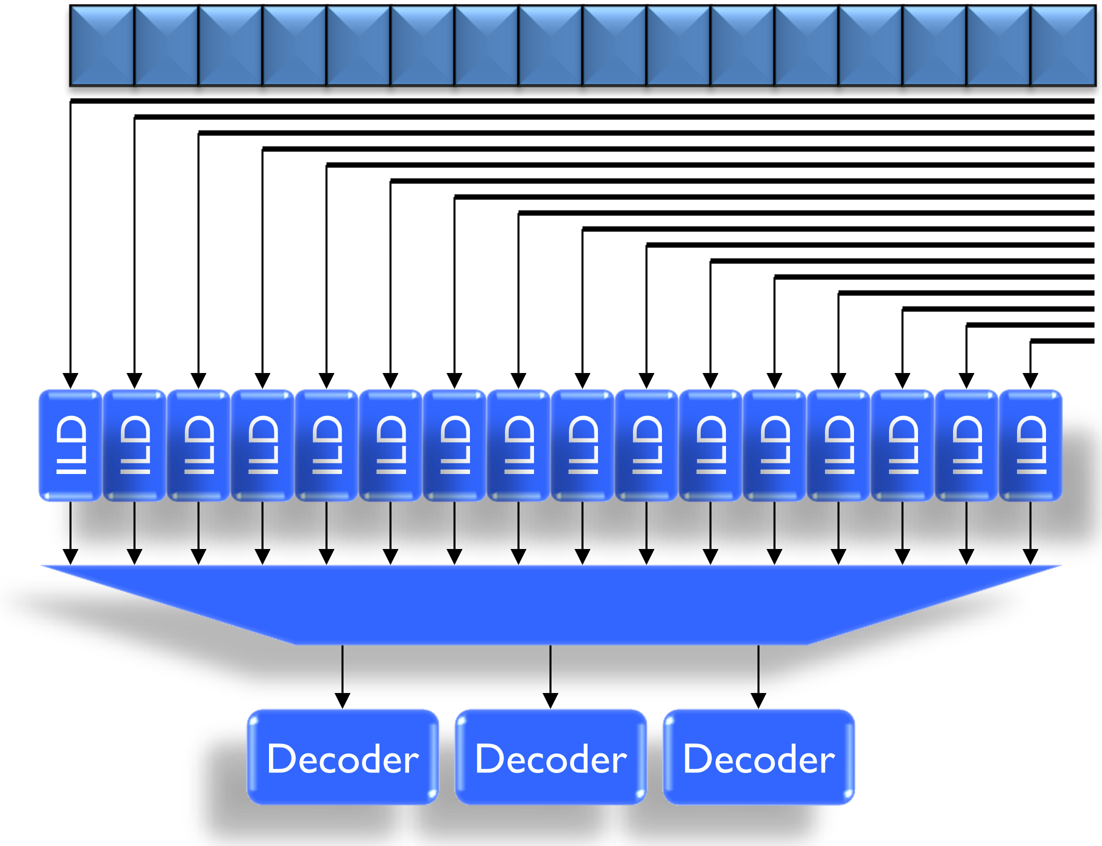
Critical component not pipeline-able

Bad x86 Decode Implementation



ILD dominates cycle time; not scalable

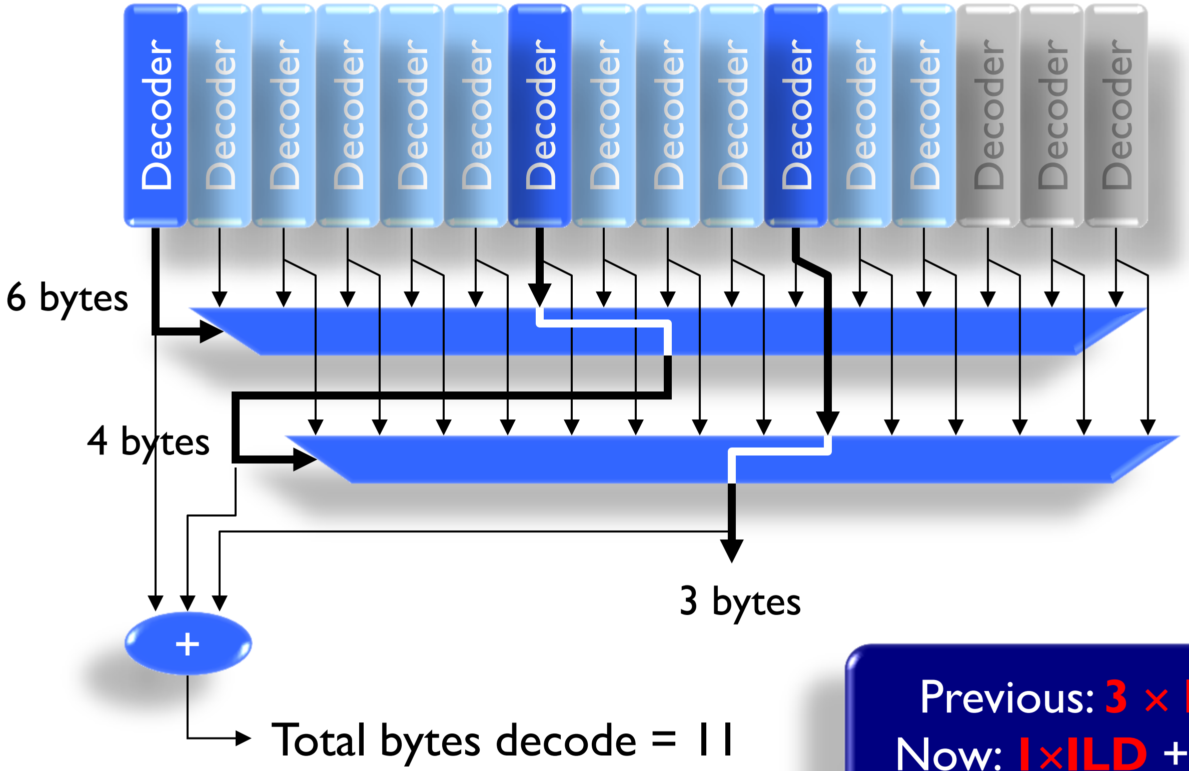
Hardware-Intensive Decode



Decode from every possible instruction starting point!

Giant MUXes to select instruction bytes

ILD in Hardware-Intensive Approach



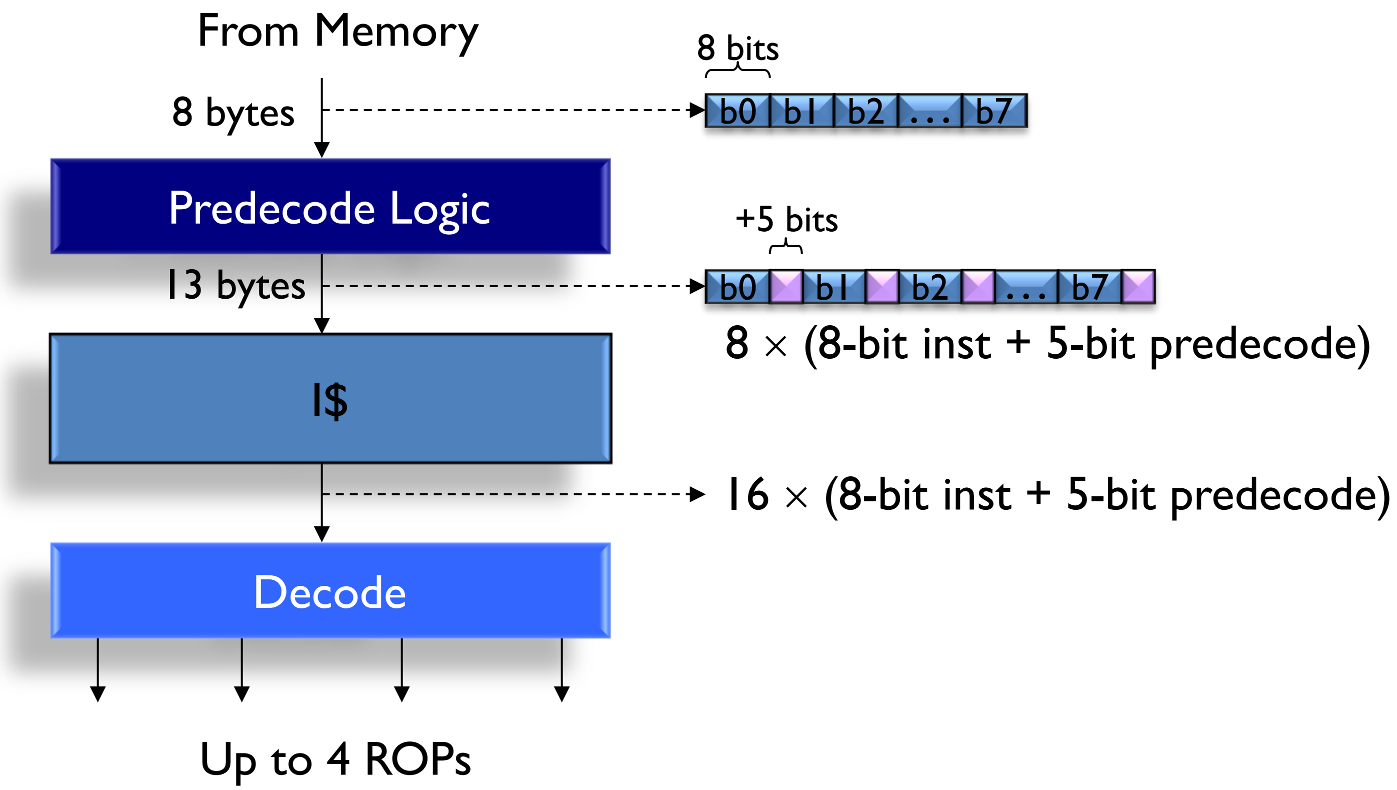
Previous: 3 × ILD + 2×add
Now: 1×ILD + 2×(mux+add)

Predecoding

- ILD loop is hardware intensive
  - Impacts latency
  - Consumes substantial power
- If instructions A, B and C are decoded
  - ... lengths for A, B, and C will still be the same next time
  - No need to repeat ILD

Possible to cache the ILD work

Decoder Example: AMD K5 (1/2)

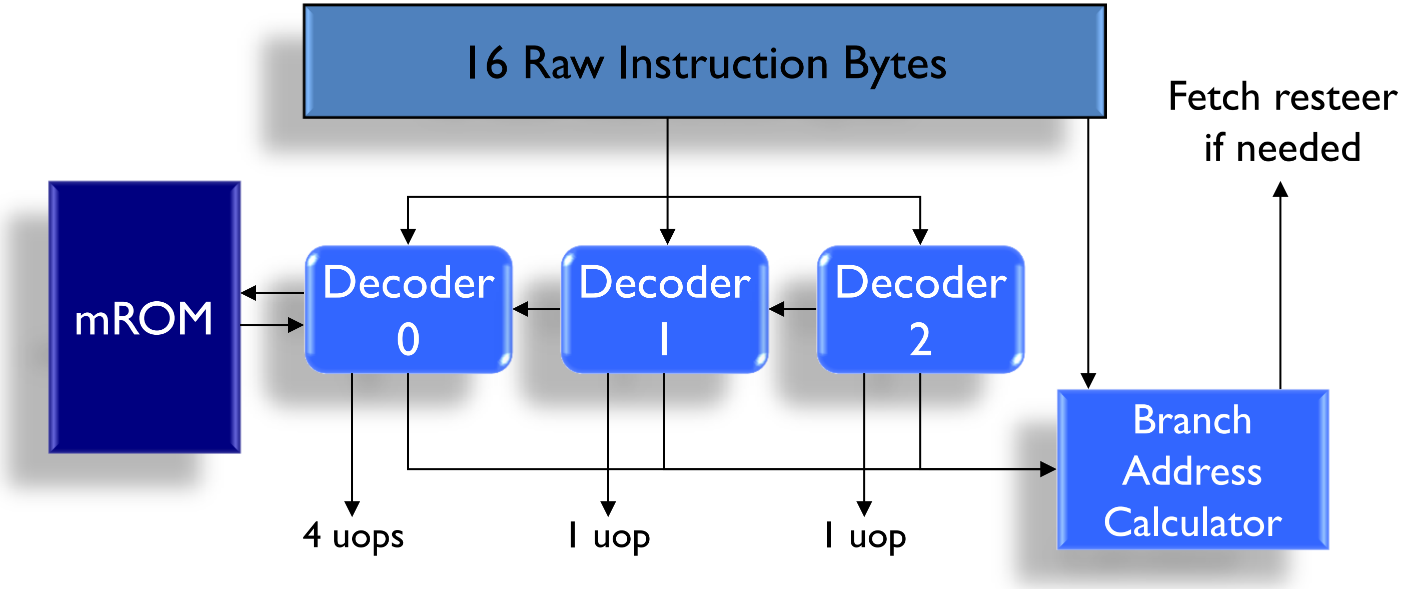


Compute ILD on fetch, store ILD in the I$

Decoder Example: AMD K5 (2/2)

- Predecode makes decode easier by providing:
  - Instruction start/end location (ILD)
  - Number of ROPs needed per inst
  - Opcode and prefix locations
- Power/performance tradeoffs
  - Larger I$ (increase data by 62.5%)
    - Longer I$ latency, more I$ power consumption
  - Remove logic from decode
    - Shorter pipeline, simpler logic
    - Cache and reused decode work → less decode power
  - Longer effective I-L1 miss latency (ILD on fill)

Decoder Example: Intel P-Pro



- Only Decoder 0 interfaces with the uROM and MS
- If insn. in Decoder 1 or Decoder 2 requires > 1 uop
  1) do not generate output
  2) shift to Decoder 0 on the next cycle