CS152: Computer Systems Architecture
RISC-V Assembly, x86 Assembly
(And Encoding)

Sang-Woo Jun

Fall 2023

UCI

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson

What does an ISA encoding look like?

- ADD: 0x000000001,
  SUB: 0x00000002,
  LW: 0x000000003,
  SW: 0x00000004, …?
- Haphazard encoding makes processor design complicated!
  - More chip resources, more power consumption, less performance

RISC/CISC decisions

In what way is an ISA “simpler” or “complex”?
And how will it effect hardware design/performance?

The Important Points

- How much work does each instruction do?

- RISC (RISC-V) cleanly divides instructions into three categories
  1. Computational operation: from register file to register file
  2. Load/Store: between memory and register file
  3. Control flow: jump to different part of code

This is every instruction in RISC-V base ISA (RV32I)

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

RV32I Base Instruction Set

imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
fm	pred	succ	rs1	000	rd	0001111	FENCE
1000	0011	0011	00000	000	00000	0001111	FENCE.TSO
0000	0001	0000	00000	000	00000	0001111	PAUSE
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK

RISC-V instruction encoding

- Restrictions
  - 4 bytes per instruction
  - Different instructions have different parameters (registers, immediates, …)
  - Various fields should be encoded to consistent locations
    - Simpler decoding circuitry
- Answer: RISC-V uses 6 “types” of instruction encoding

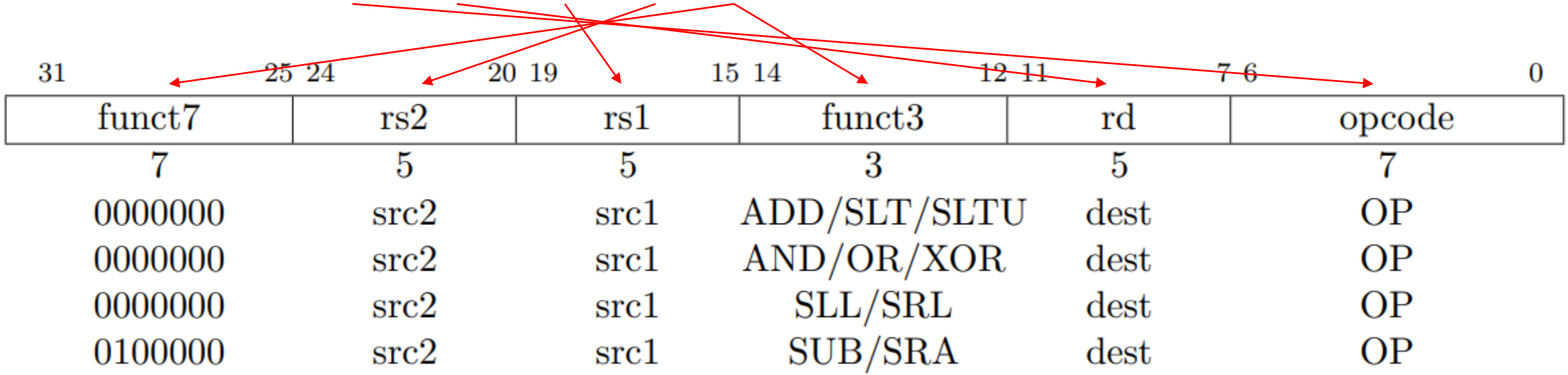
Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Small number of types

We’re not going to look at everything…

1/6: RISC-V R-Type encoding

- Relatively straightforward, register-register operations encoding
- Remember:
  - if (inst.**type** == **ALU**) **rf**[inst.**arg1**] = **alu**(inst.**op**, **rf**[inst.**arg2**], **rf**[inst.**arg3**])
  - In 4 bytes, **type**, **arg1**, **arg2**, **arg3**, **op** needs to be encoded



31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

1/6: R-Type Computational operations

- Arithmetic, comparison, logical, shift operations
- Register-register instructions
  - 2 source operand registers
  - 1 destination register
  - Format: op dst, src1, src2

Arithmetic	Comparison	Logical	Shift
add, sub	slt, sltu	and, or, xor	sll, srl, sra

set less than
set less than unsigned

Signed/unsigned?

Shift left logical
Shift right logical
Shift right arithmetic

Arithmetic/logical?

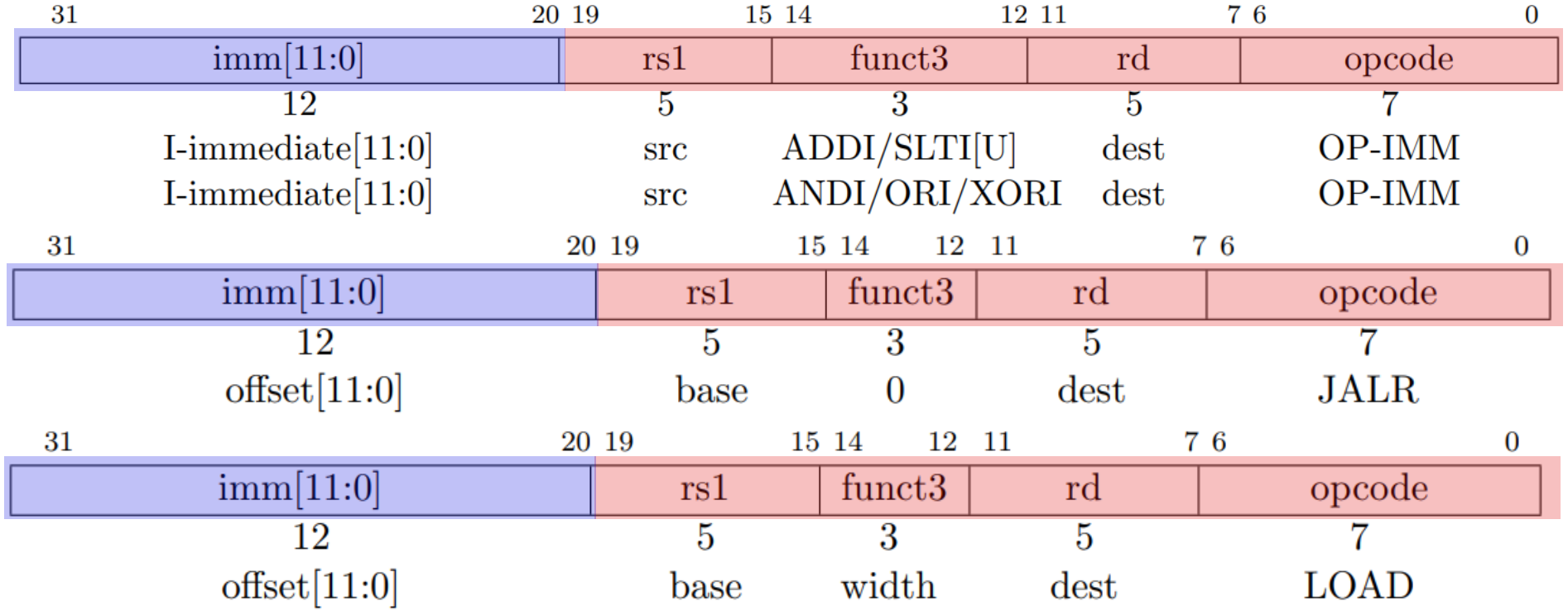
2/6: I-Type Encoding

- Some instructions need “immediate” values
  - e.g., “addi x1, x2, 32” <- 32 is an immediate value encoded in the instruction
  - R-Type does not have slots for this

2/6 RISC-V I-Type encoding

- Register-Immediate operations encoding
  - One register, one immediate as input, one register as output

Operands in same location!



Immediate value limited to 12 bits signed!

addi x5, x6, 2048 # Error: illegal operands `addi x5,x6,2048'

2/6: I-Type Computational operations

- Register-immediate operations
  - 2 source operands
    - One register read
    - One immediate value encoded in the instruction

    Limited to 12 bits! (Why?)
  - 1 destination register
  - Format: op dst, src, imm
    - eg., addi x1, x2, 10

Format	Arithmetic	Comparison	Logical	Shift
register-register	add, sub	slt, sltu	and, or, xor	sll, srl, sra
register-immediate	addi	slti, sltiu	andi, ori, xori	slli, srli, srai

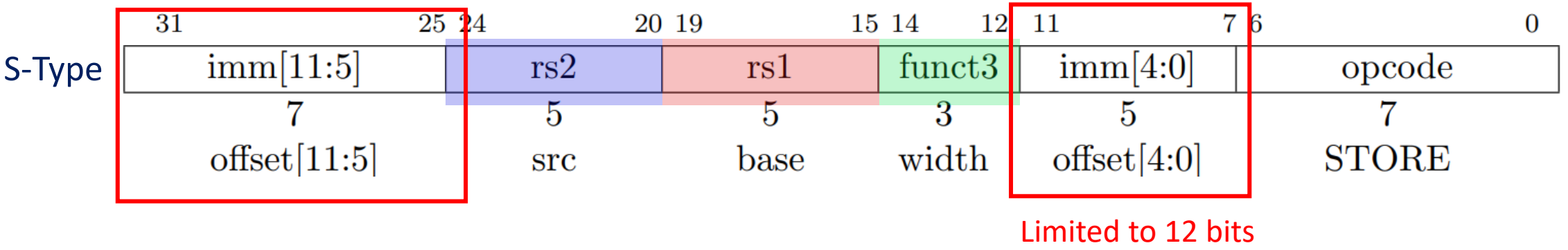
No “subi” instead use negative with “addi”

3/6: RISC-V Load/Store operations

- Format: op dst, offset(base)
  - Address specified by a pair of <base address, offset>
  - e.g., lw x1, 4(x2) # Load a word (4 bytes) from [x2]+4 to x1
  - The offset is a small constant
- Variants for types
  - lw/sw: Word (4 bytes)
  - lh/lhu/sh: Half (2 bytes)
  - lb/lbu/sb: Byte (1 byte)
  - ‘u’ variant is for unsigned loads
    - Half and Byte reads extends read data to 32 bits. Signed loads are sign-bit aware

3/6: S-Type encoding

- Store operation: two register input, no output
  - e.g.,
    sw src, offset(base)

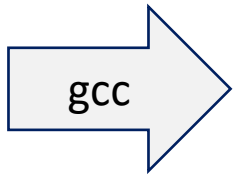


4/6: RISC-V Control flow instructions - Branching

- Format: cond src1, src2, label
- If condition is met, jump to label. Otherwise, continue to next

beq	bne	blt	bge	bltu	bgeu
==	!=	<	>=	<	>=

```
if (a < b):    c = a + 1
      else:    c = b + 2
```



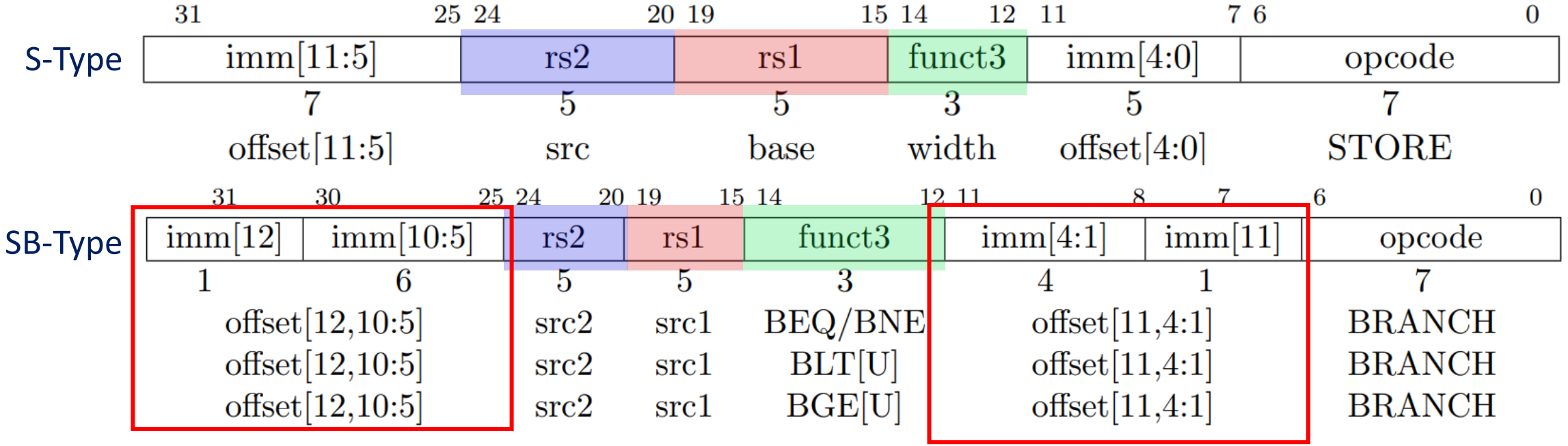

```
        bge x1, x2, else
        addi x3, x1, 1
        beq x0, x0, end
else:   addi x3, x2, 2
end:
```

(Assume x1=a; x2=b; x3=c;)

4.6: SB-Type encoding

- Store operation: two register input, no output
  - e.g.,
    sw src, offset(base)
    beq r1, r2, label

Operands in same location!
(Bit width not to scale…)



Only 12 bits of offset can fit! -> Jump target can be max 2^12 bits away

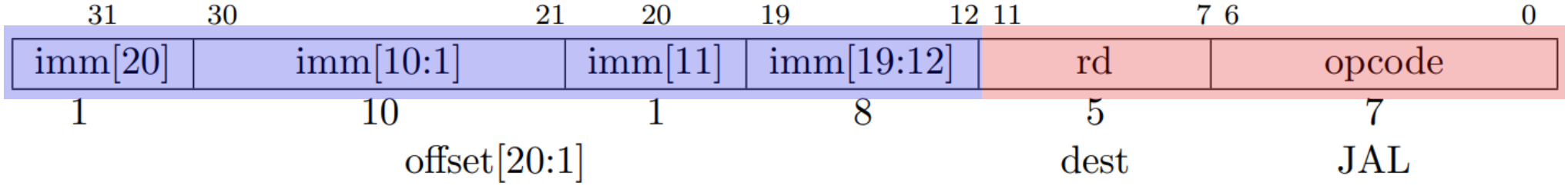
5/6: RISC-V Control flow instructions – Jump and Link

- Format:
  - jal dst, label – Jump to ‘label’, store PC+4 in dst
  - jalr dst, offset(base) – Jump to rf[base]+offset, store PC+4 in dst
    - e.g., jalr x1, 4(x5) Jumps to x5+4, stores PC+4 in x1
- Why do we need two variants?
  - jal has a limit on how far it can jump
    - (Due to immediate value encoding width, shown soon)
  - jalr used to jump to locations defined at runtime
    - Needed for many things including function calls (e.g., Many callers calling one function)

```
    …
    jal x1, function1
    …
function1:
    …
    jalr x0, 0(x1)
```

5/6: UJ-Type encoding

- One destination register, one immediate operand
  - UB-Type: JAL (Jump and link)

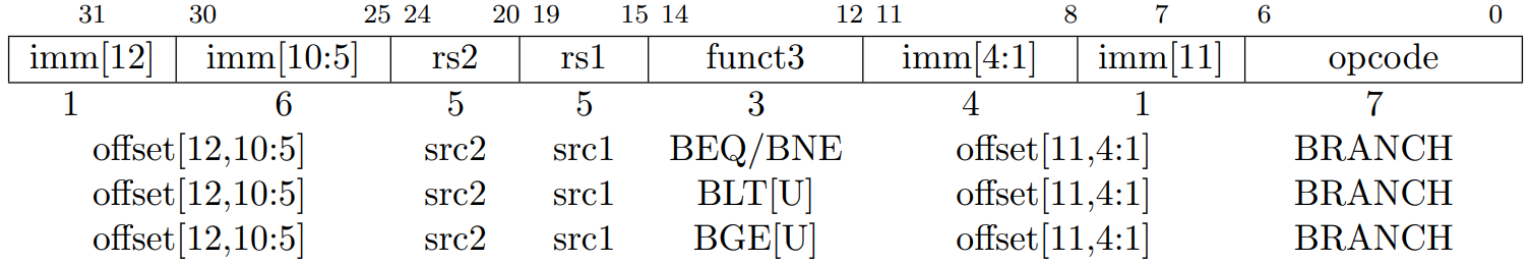


Only 20 bits of offset! What if target is farther?

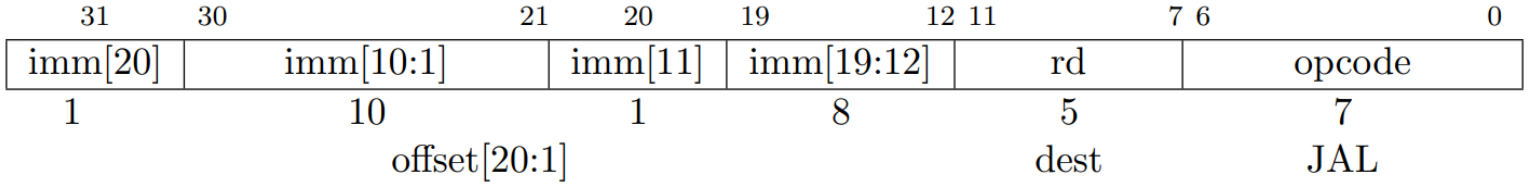
5/6: RISC-V Relative addressing

- Problem: jump target offset is small!
  - For branches: 12 bits, For JAL: 20 bits
  - How does it deal with larger program spaces?
  - Solution: PC-relative addressing (PC = PC + imm)
    - Remember format: beq x5, x6, label
    - Translation from label to offset done by assembler
    - Works fine if branch target is nearby. If not, AUIPC and other tricks by assembler

SB-Type



U-Type



Problem: What if this is not enough!?

6/6: Load upper immediate instructions

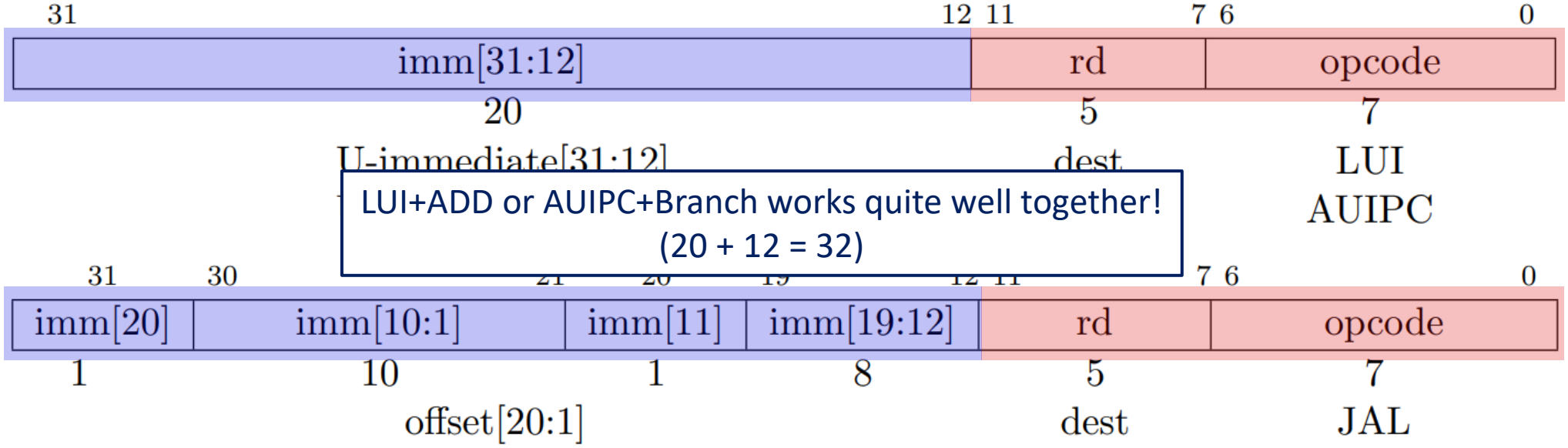
- LUI: Load upper immediate
  - lui dst, immediate → dst = immediate<<12
  - Can load (32-12 = 20) bits
  - Used to load large (~32 bits) immediate values to registers
  - lui followed by addi (load 12 bits) to load 32 bits
- AUIPC: Add upper immediate to PC
  - auipc, dst, immediate → dst = PC + immediate<<12
  - Can load (32-12 = 20) bits
  - auipc followed by addi, then jalr to allow long jumps within any 32 bit address

Typically not used by human programmers!
Assemblers use them to implement complex operations

6/6: RISC-V U-Type and UJ-Type encoding

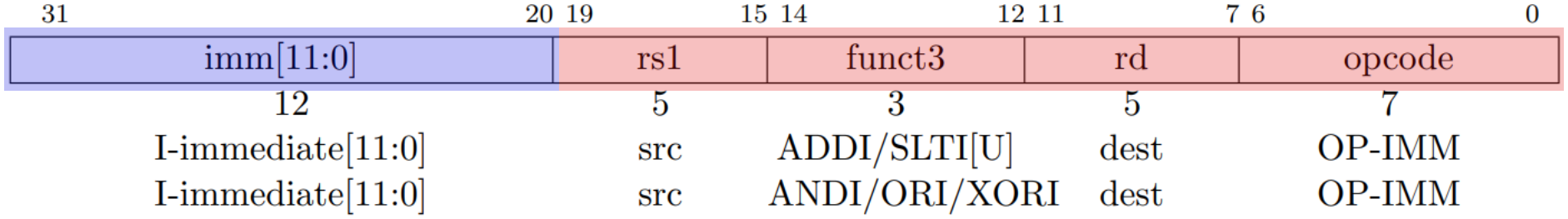
- One destination register, one immediate operand
  - U-Type: LUI (Load upper immediate), AUIPC (Add upper immediate to PC)
    Typically not used by human programmer
  - UB-Type: JAL (Jump and link)

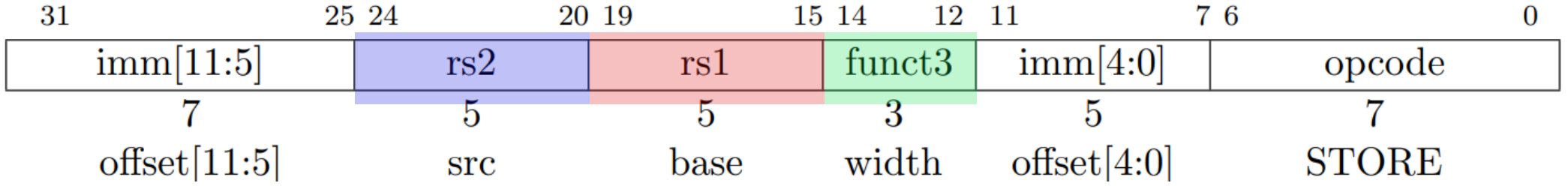
Operands in same location!

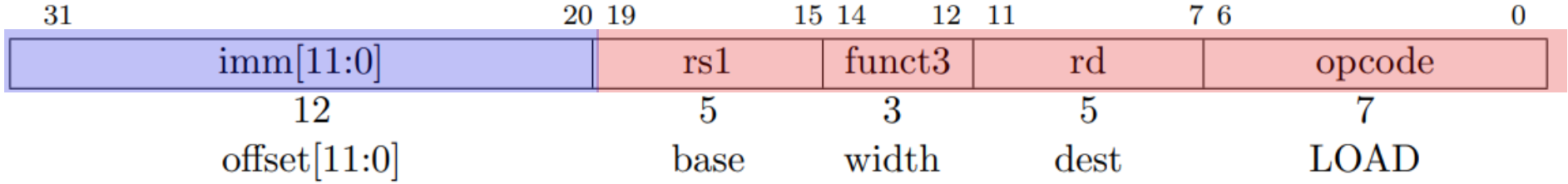


Aside: Why is the immediate field 12 bits?

- If most immediate values are larger, this instruction is useless!
  - Why not encode more imm, and reduce register count?

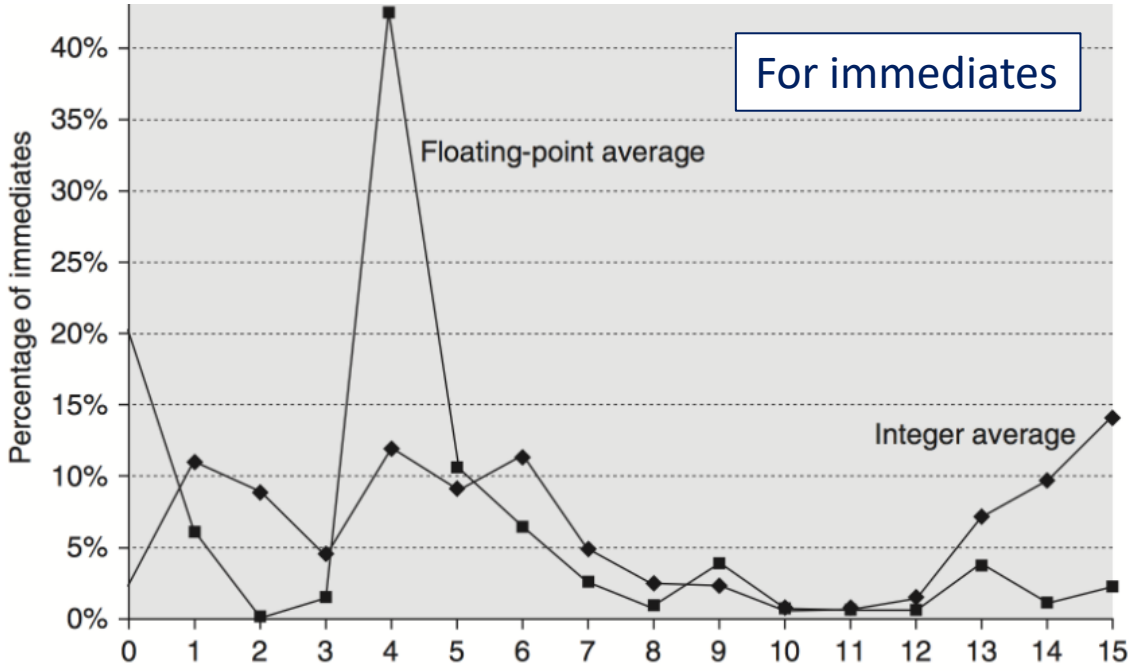


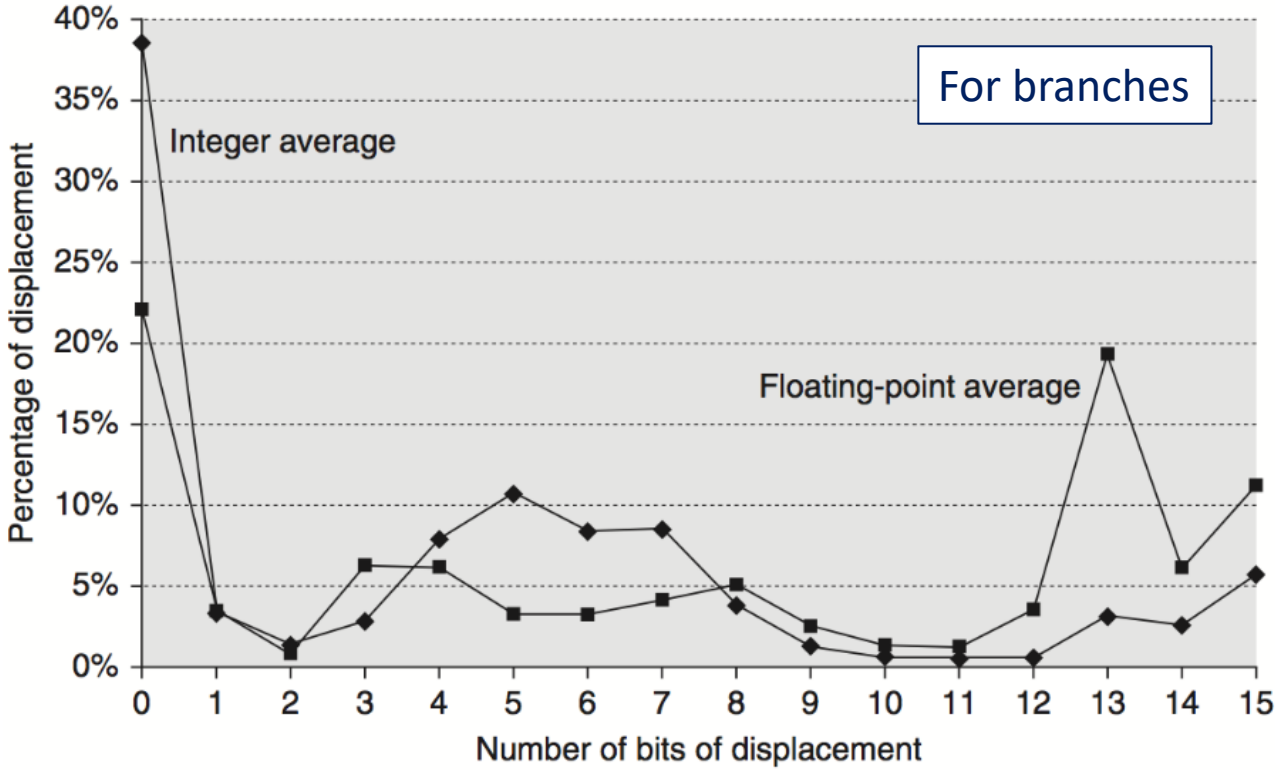




Benchmark-driven ISA design

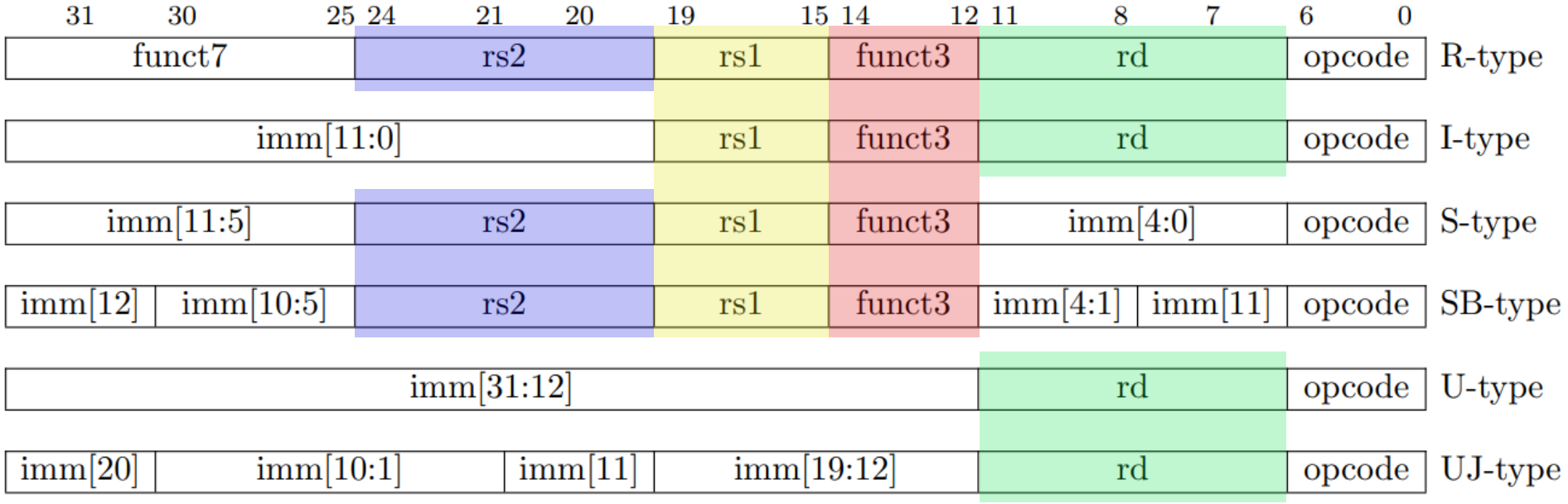
- Make the common case fast!
  - 12~16 bits capture most cases





RISC-V Design consideration: Consistent operand encoding location

- Simplifies circuits, resulting in less chip resource usage



CS152: Computer Systems Architecture
x86 Assembly
(And Encoding)

Sang-Woo Jun

Fall 2023

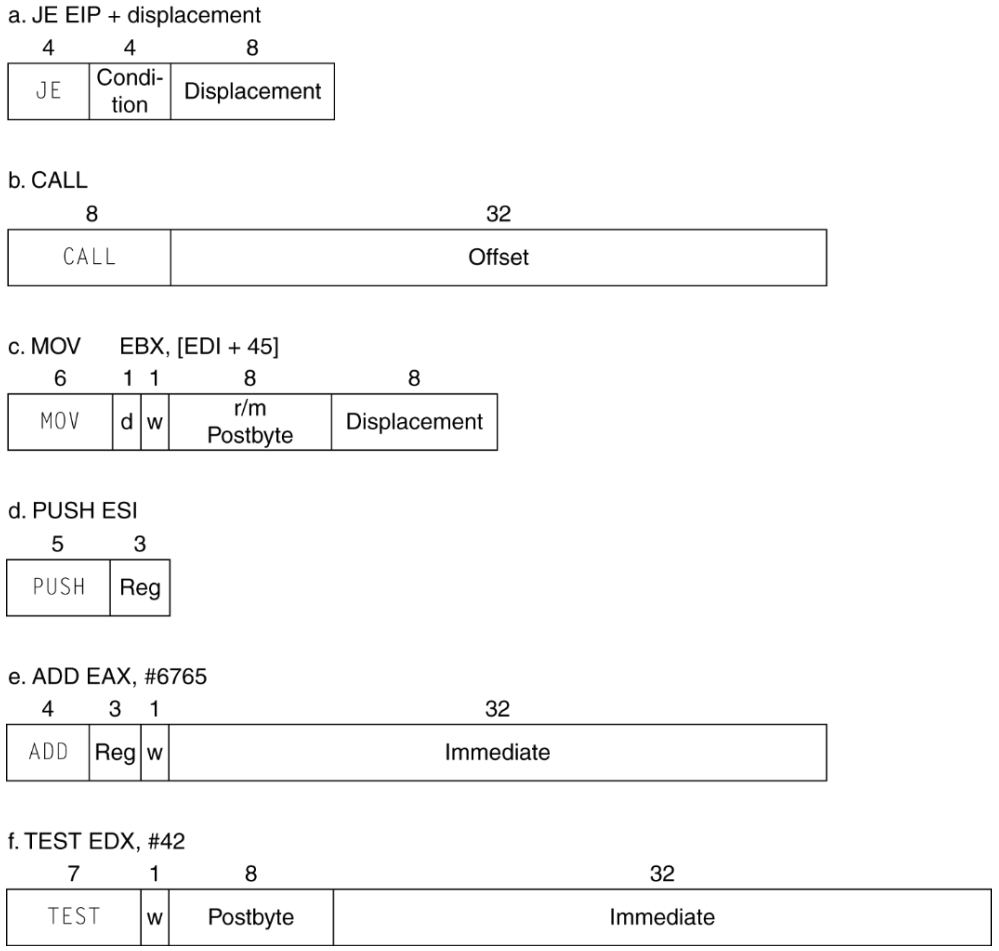
UCI

Large amount of material adapted from MIT 6.004, “Computation Structures”, Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”, and CS 152 Slides by Isaac Scherson

x86 encoding

- Many many complex instructions
  - Fixed-size encoding will waste too much space
  - Variable-length encoding!
  - 1 byte – 15 bytes encoding
- Complex decoding logic in hardware
  - Hardware translates instructions to simpler micro operations
    - Simple instructions: 1–1
    - Complex instructions: 1–many
  - Microengine similar to RISC
  - Market share makes this economically viable

Comparable performance to RISC! But with translation overhead
Compilers avoid complex instructions



Meanwhile: x86 – Addressing modes

- Typical x86 assembly instructions have many addressing mode variants

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- e.g., ‘add’ has two input operands, storing the add in the second

```
add <reg>, <reg>
add <mem>, <reg>
add <reg>, <mem>
add <imm>, <reg>
add <imm>, <mem>
```

Examples
add $10, %eax — EAX is set to EAX + 10
addb $10, (%eax) — add 10 to the single byte stored at memory address stored in EAX

CISC! But no “Memory -> Memory”

Example source: Guide to x86 Assembly - Yale FLINT Group

CISC ISAs typically mix arithmetic + load/store

- Remember x86 “add” example
  - Arithmetic instruction can access memory, store in memory
- Some special Load/Store instructions also do exist
  - e.g., “mov” with same addressing modes
  - e.g., “vmovupd” in AVX extensions…

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

x86 Complex addressing modes: Complex encoding!

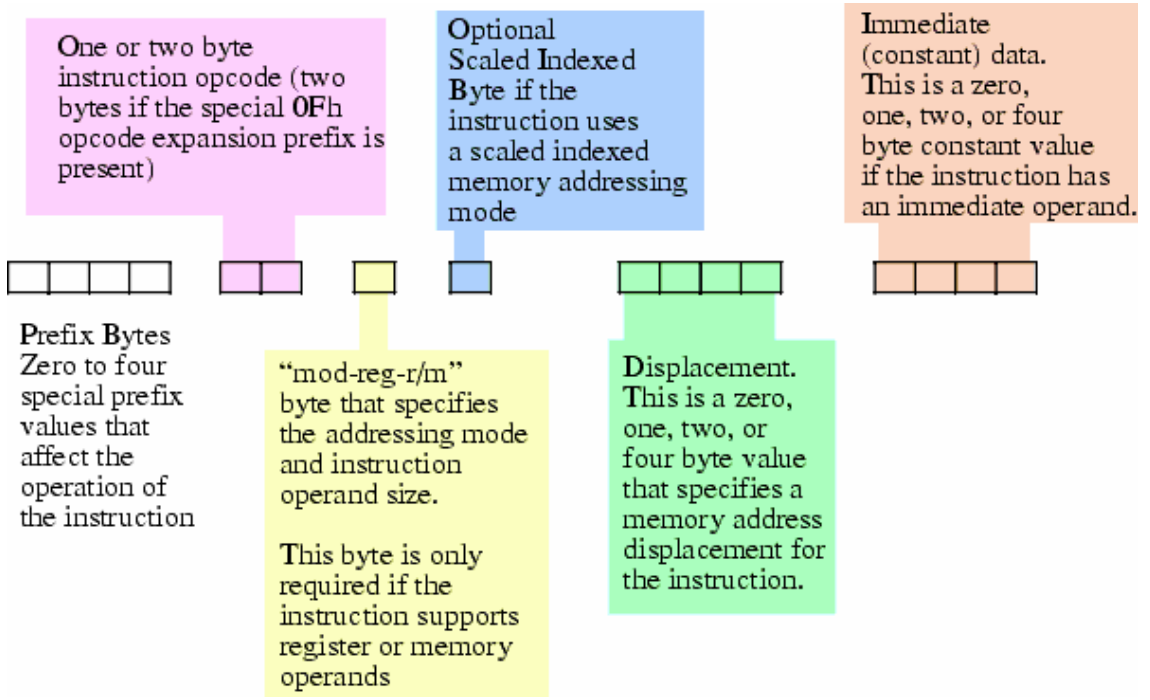
- “`imul eax, [rdx+rcx*4-0x4]`”
  - Encoded to single instruction “`0f af 44 8a fc`”
  - Signed multiplication between eax, and a value from memory
  - Two additions and one multiplication before memory request!
    - (Which architectural component is responsible for this arithmetic?)
  - One multiplication after memory request comes back

- Who performs the memory address arithmetic?
  - Separate ALU? Time-share ALU with actual imul operation?
  - Microarchitectural details not enforced by ISA

x86: CISC requires complex encoding!

- So many possibilities within a single instruction
  - Complex, variable-width data to encode
  - Complex, high-latency decode logic unavoidable!

e.g., Immediate values can use either 0, 1,2,4 bytes to encode



Variable-length: Many fields are optional
→ The location (bit offset) of each field is always changing!

Bristol community college, "CIS-77 Introduction to Computer Systems"

Aside: Conditional execution in CISC and RISC

Conditional execution in CISC: Condition codes

- Implicitly managed bitmap of flags
  - e.g., Carry, Overflow, Negative, Equal to zero, less than, …
  - Flags set by previously executed instruction
- e.g., x86 “cmp” compares two values and sets condition code flags
  - Usual addressing modes
  - Jump instruction variants read condition code flags

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

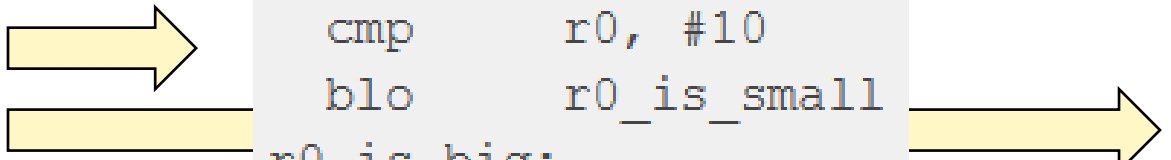
```
je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)
```

Conditional execution in CISC: Condition codes

- Some instructions can execute only if conditions are met
  - “Predicated instructions”
  - ARM MOVHS (Move higher or same) only moves if previous instruction resulted in “higher or same” flag being set. Otherwise NOP
  - Can remove a costly conditional branch instruction if used well
  - Carry bits can be useful for large adds, …

Predicated instructions in ARM

```
if (a > 10) {
  a = 10;
} else {
  a = a + 1;
}
```



```
  cmp     r0, #10
  blo     r0_is_small
r0_is_big:
  mov     r0, #10
  b       continue
r0_is_small:
  add     r0, r0, #1
continue:
  @ Other code.
```



```
cmp     r0, #10
movhs   r0, #10
addlo   r0, r0, #1
```

C Code

Without predicated instructions

With predicated instructions

RISC-V Condition codes

- RISC-V does not have condition codes
  - Designers wanted simpler communications between pipeline stages

Wrapping up

- Two ends of the spectrum: RISC and CISC
  - RISC simplifies processor hardware, but same programs result in more code
  - CISC reduces code volume, but complicates processor hardware
- To reason about this trade-off, we need to know their actual effects
  - How much clock speed degradation do we get with more complex decode?
  - How much transistor overhead is complex decode?
  - How much instruction count increase caused by RISC ISA?

Up next!

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

The Important Points

- RISC-V (RISC) instructions are cleanly divided into categories
  - ALU, Branch, Memory – Specifically the six encoding types
  - Lower encoding density, but simplifies decoding
- x86 (CISC) does NOT cleanly divide work into categories
  - Each instruction can do a combination of ALU, Branch, Memory
  - Higher density, but complicates decoding

The Important Points

- RISC-V (RISC) instructions are fixed-width
  - immediate values cannot be encoded in full (32 bits) into one instruction
  - e.g., addi encodes 12 bits, AUIPC encodes 20 bits
  - ISA carefully designed to require only two instructions per 32-bit word
  - (register file being 32, opcode being 7 bits, all balance into this)
- x86 (CISC) instructions are variable-width
  - One immediate value can be encoded into one long (4bytes+) instructions
  - Complicates encoding, but fewer instructions

Aside: Handling I/O

- How can a processor communicate with the outside world?
- Special instructions? Sometimes!
  - RISC-V defines CSR (Control and Status Registers) instructions
  - Check processor capability (I/M/E/A/..?), performance counters, system calls, …
  - “Port-mapped I/O”
- E.g., x86 has “IN”, “OUT” instructions
  - Goes back to how 8080 did I/O
  - “IN $0x60, %al” reads a keyboard input from the PS/2 controller

Source: Wikipedia

Aside: Handling I/O

- For efficient communication, memory-mapped I/O
  - Happens outside the processor
  - I/O device directed to monitor CPU address bus, intercepting I/O requests
    - Each device assigned one or more memory regions to monitor
    - Some memory commands handles by memory, some by peripherals!

Example:
In the original Nintendo GameBoy, reading from address 0xFF00
returned a bit mask of currently pressed buttons

Both approaches require one CPU instruction per word I/O…

Aside: Handling I/O

- Even faster option: DMA (Direct Memory Access)
  - Off-chip DMA Controller can be directed to read/write data from memory without CPU intervention
  - Once DMA transfer is initiated, CPU can continue doing other work
  - Used by high-performance peripherals like PCIe-attached GPUs, NICs, and SSDs
    - Hopefully we will have time to talk about PCIe!
  - Contrast: Memory-mapped I/O requires one CPU instruction for one word of I/O
    - CPU busy, blocking I/O hurts performance for long latency I/O

Wrapping up…

- Design principles
  1. Simplicity favors regularity
  2. Smaller is faster
  3. Good design demands good compromises
- Make the common case fast

- Powerful instruction ⇏ higher performance
  - Fewer instructions required, but complex instructions are hard to implement
    - May slow down all instructions, including simple ones
  - Compilers are good at making fast code from simple instructions