

CS152: Computer Systems Architecture

RISC-V Assembly, x86 Assembly (And Encoding)



Sang-Woo Jun

Fall 2022



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: 0x000000002,
LW: 0x000000003,
SW: 0x000000004, ...?
- ❑ 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?

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)	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

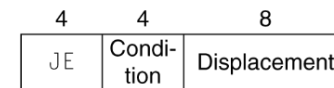
We're not going to look at everything...

x86 encoding

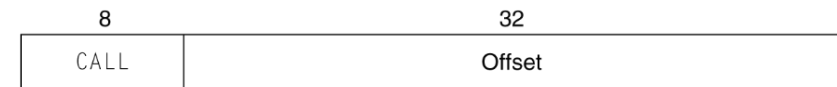
- ❑ Many many complex instructions
 - Fixed-size encoding will waste too much space
 - Variable-length encoding!
 - 1 byte – 15 bytes encoding
- ❑ Complex decoding logic in hardware
 - Hardware translates instructions to simpler micro operations
 - Simple instructions: 1–1
 - Complex instructions: 1–many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

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

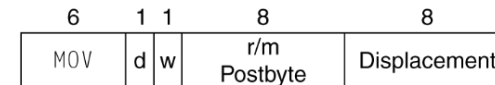
a. JE EIP + displacement



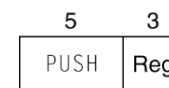
b. CALL



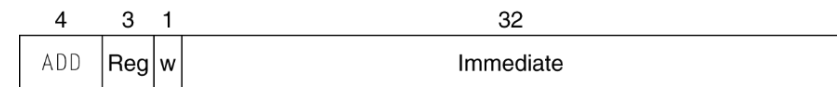
c. MOV EBX, [EDI + 45]



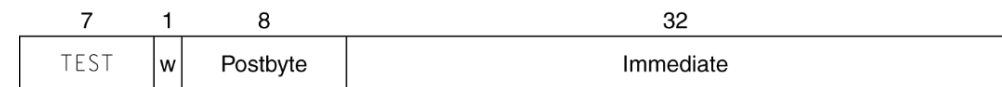
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



Let's look at examples!

Three types of instructions

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

RISC-V 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

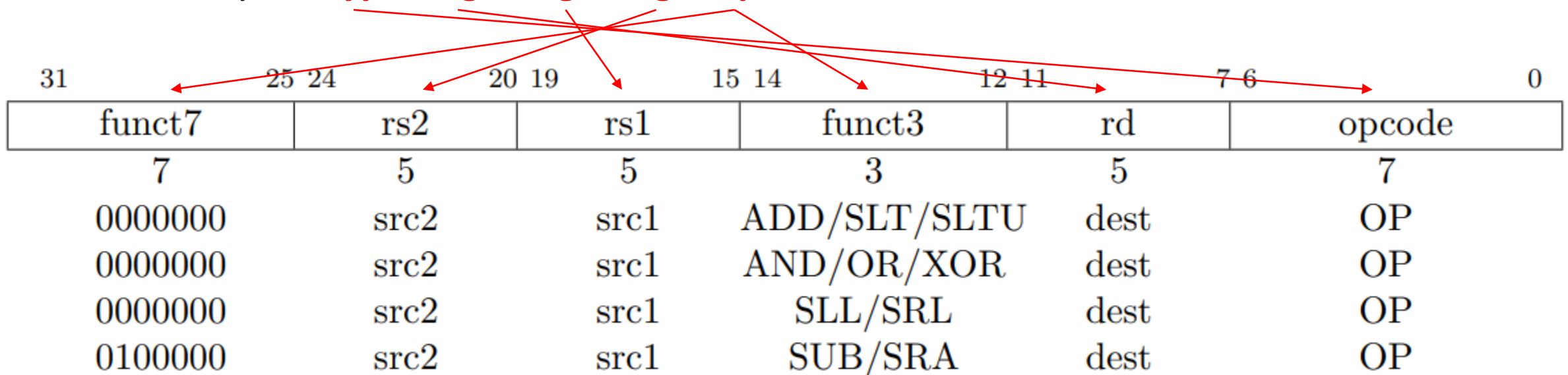
Arithmetic/logical?

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



RISC-V 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

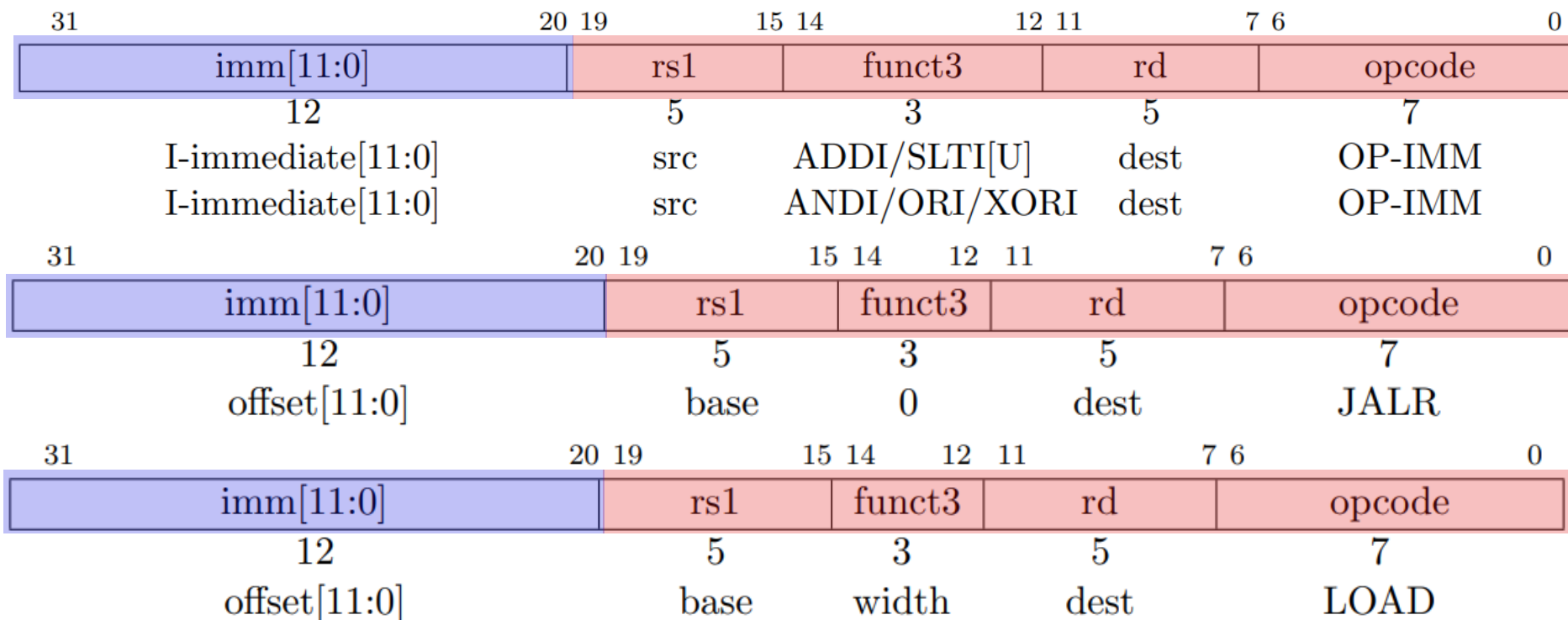
No “subi” instead use negative with “addi”

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'

Aside: Signed and unsigned operations

- ❑ Registers store 32-bits of data, no type
- ❑ Some operations interpret data as signed, some as unsigned values

operation	Meaning
add d, a, b	$d = sx(a) + sx(b)$
slt d, a, b	$d = sx(a) > sx(b) ? 1 : 0$
sltu d, a, b	$d = ux(a) > ux(b) ? 1 : 0$
sll d, a, b	$d = ux(a) \ll b$
srl d, a, b	$d = ux(a) \gg b$
sra d, a, b	$d = sx(a) \gg b$

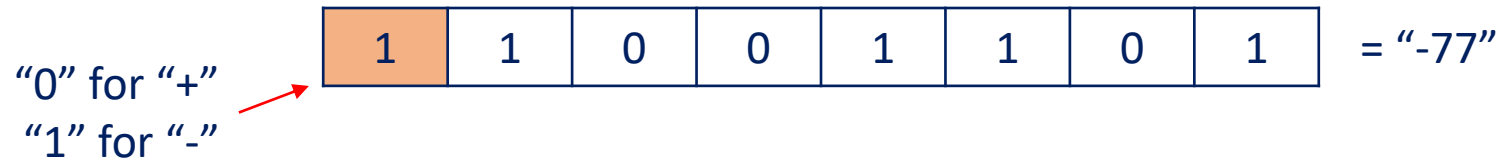
sx: interpret as signed, ux, interpret as unsigned

No sla operation. Why? Two's complement ensures sla == sll

(Same for x86, SHL and SAL have same opcode)

Aside: Two's complement encoding

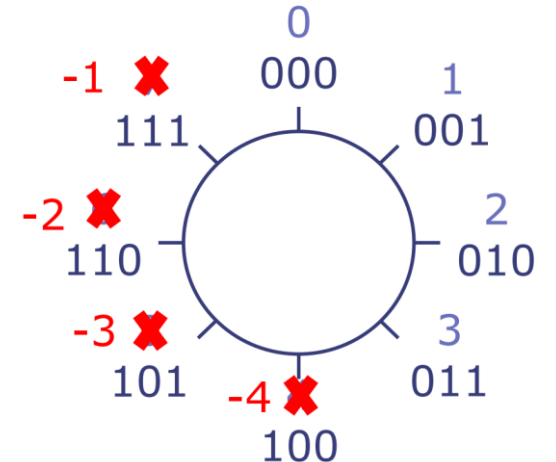
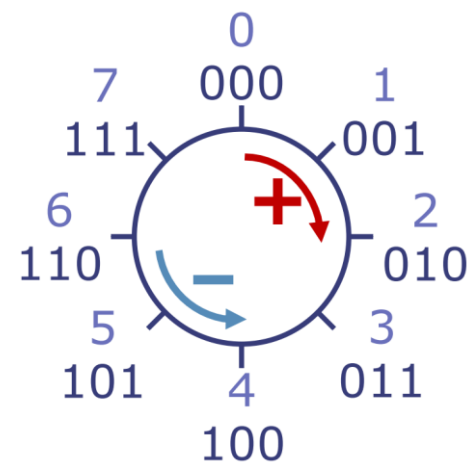
- ❑ How should we encode negative numbers?
- ❑ Simplest idea: Use one bit to store the sign



- ❑ Is this a good encoding? No!
 - Two representations for "0" ("+0", "-0")
 - Add and subtract require different algorithms

Aside: Two's complement encoding

- ❑ The larger half of the numbers are simply interpreted as negative
- ❑ Background: Overflow on fixed-width unsigned numbers wrap around
 - Assuming 3 bits, $100 + 101 = 1001$ (overflow!) = stores 001
 - “Modular arithmetic”, equivalent to following modN to all operations
- ❑ Relabeling allows natural negative operations via modular arithmetic
 - e.g., $111 + 010 = 1001$ (overflow!) = stores 001
equivalent to $-1 + 2 = 1$
 - Subtraction uses same algorithm as add
e.g., $a - b = a + (-b)$



Aside: Two's complement encoding

□ Some characteristics of two's encoded numbers

- Negative numbers have "1" at most significant bit (sign bit)
- Most negative number = $10\dots000 = -2^{N-1}$
- Most positive number = $01\dots111 = 2^{N-1} - 1$
- If all bits are 1 = $11\dots111 = -1$
- Negation works by flipping all bits and adding 1

$$-A + A = 0$$

$$-A + A = -1 + 1$$

$$-A = (-1 - A) + 1$$

$$-A = \sim A + 1$$

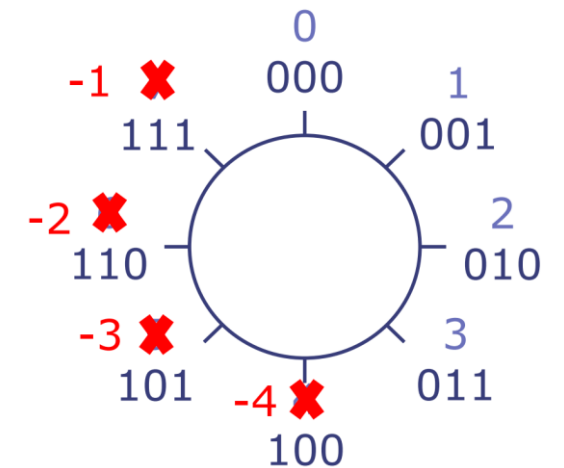
Because -1 is all 1s,
there is no borrowing,
therefore subtracting A from -1 is flipping all bits

e.g.,

111111

-100010

=011101



Aside: Shifting with two's complement

□ Right shift requires both logical and arithmetic modes

- Assuming 4 bits
- $(4_{10}) \gg 1 = (0100_2) \gg 1 = 0010_2 = 2_{10}$ Correct!
- $(-4_{10}) \gg_{\text{logical}} 1 = (1100_2) \gg_{\text{logical}} 1 = 0110_2 = 6_{10}$ For signed values, Wrong!
- $(-4_{10}) \gg_{\text{arithmetic}} 1 = (1100_2) \gg_{\text{arithmetic}} 1 = 1110_2 = -2_{10}$ Correct!
- Arithmetic shift replicates sign bits at MSB

□ Left shift is the same for logical and arithmetic

- Assuming 4 bits
- $(2_{10}) \ll 1 = (0010_2) \ll 1 = 0100_2 = 4_{10}$ Correct!
- $(-2_{10}) \ll_{\text{logical}} 1 = (1110_2) \ll_{\text{logical}} 1 = 1100_2 = -4_{10}$ Correct!

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”

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

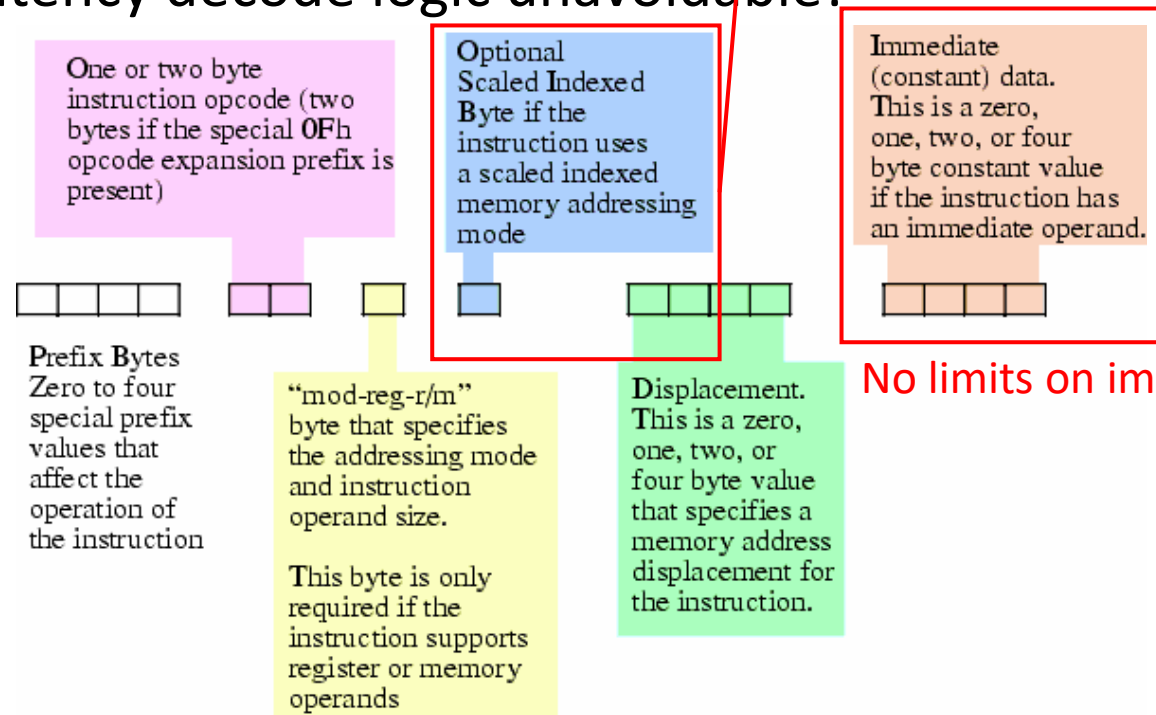
x86: CISC requires complex encoding!

❑ So many possibilities within a single instruction

- Complex, variable-width data to encode

- Complex, high-latency decode logic unavoidable!

Address multiplication only support shifts!



No limits on immediate value encoding

Three types of instructions

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

RISC-V Load/Store operations

❑ Format: op dst, offset(base)

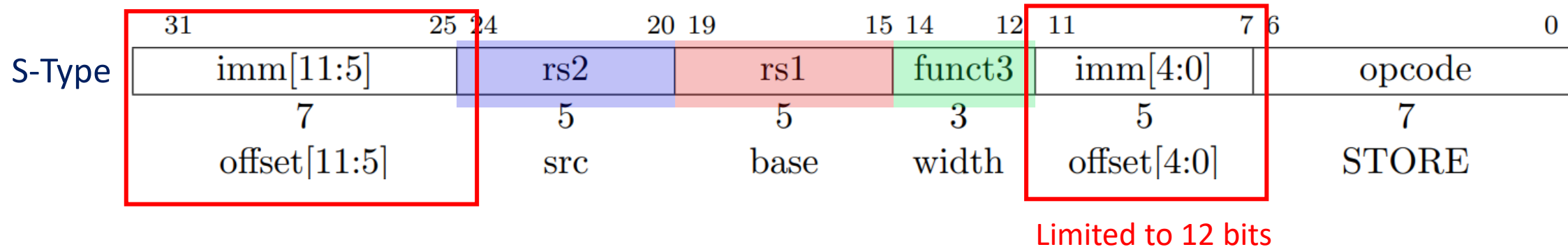
- Address specified by a pair of <base address, offset>
- e.g., lw x1, 4(x2) # Load a word (4 bytes) from [x2]+4 to x1
- The offset is a small constant

❑ 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

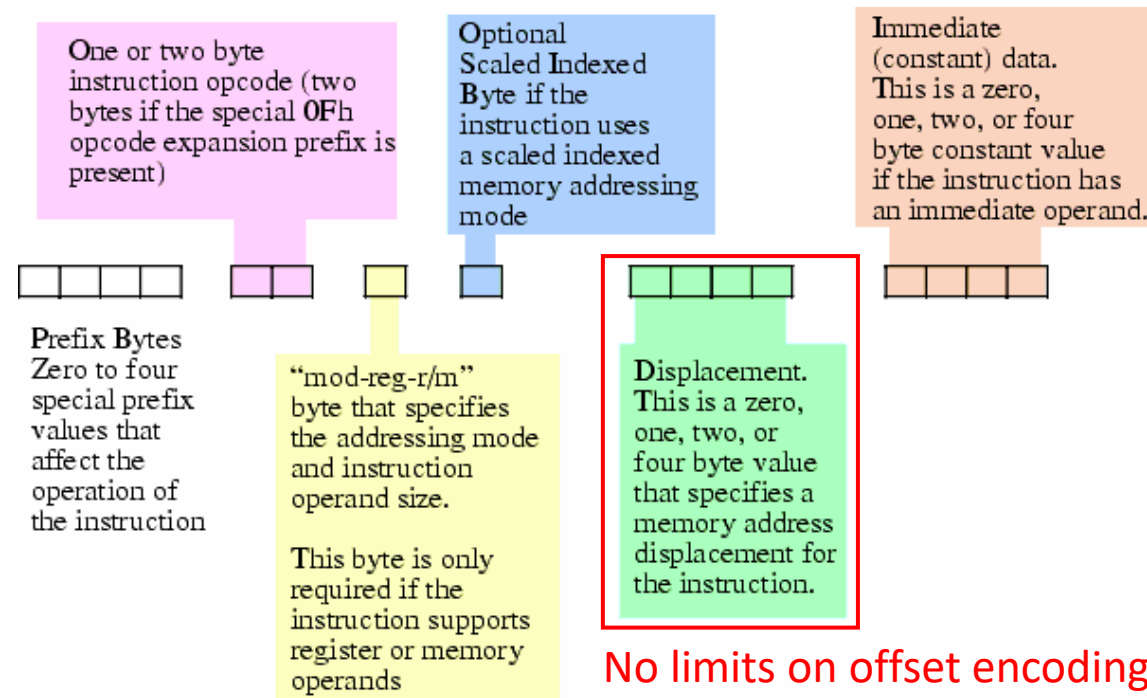
S-Type encoding

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



x86: CISC requires complex encoding!

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



Sign extension

- ❑ Representing a number using more bits
 - Preserve the numeric value
- ❑ Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- ❑ Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
- ❑ In RISC-V instruction set
 - `lb`: sign-extend loaded byte
 - `lbu`: zero-extend loaded byte

Q: Why doesn't stores need sign variants?

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

Three types of instructions

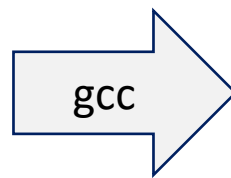
1. Computational operation: from register file to register file
2. Load/Store: between memory and register file
3. Control flow: jump to different part of code

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:
```

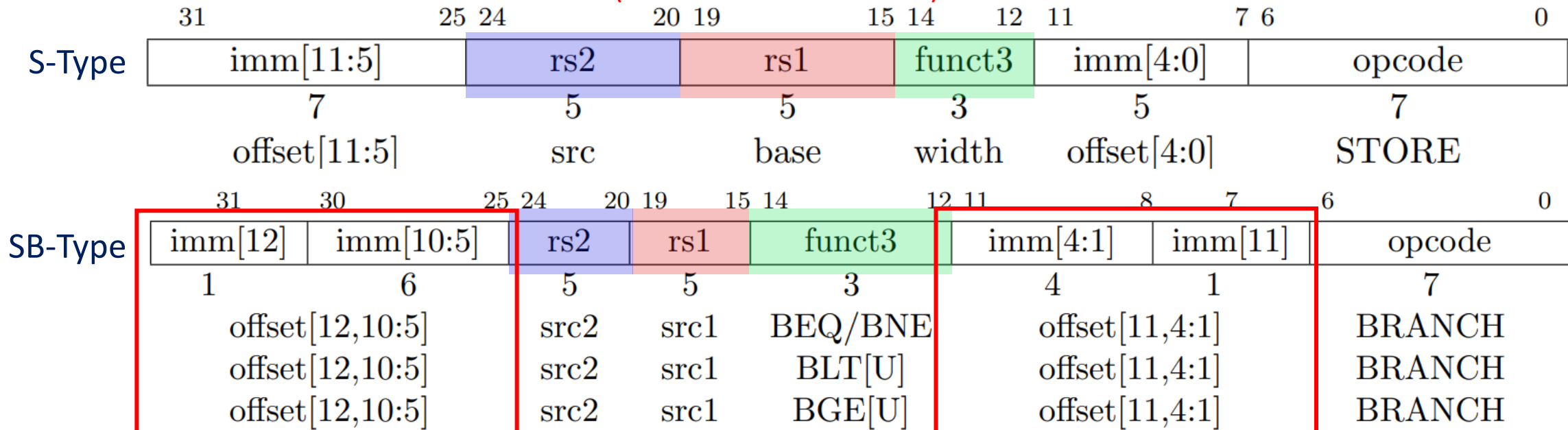
(Assume x1=a; x2=b; x3=c;)

RISC-V S-Type and 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...)

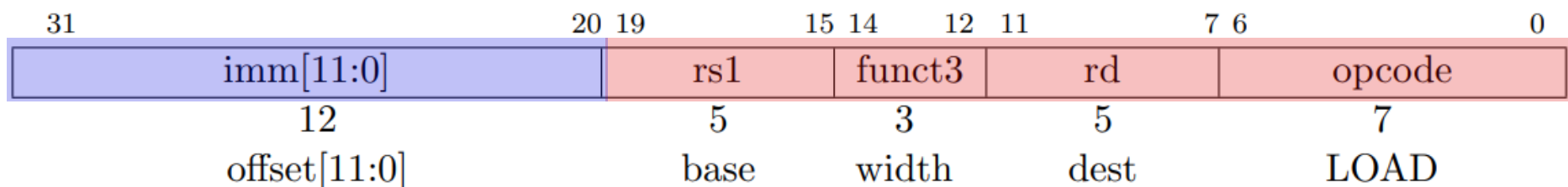
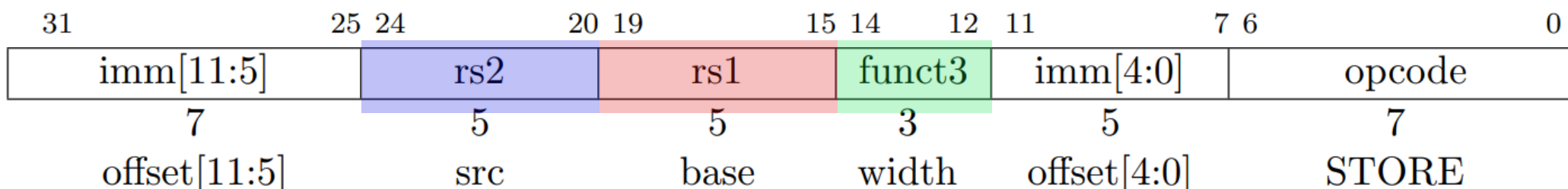
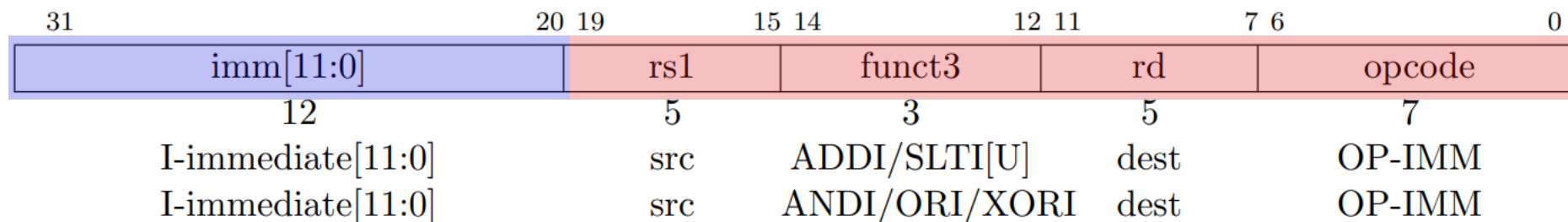


Only 12 bits of offset can fit! -> Jump target can be max 2^{12} bits away

Aside: Why is the immediate field 12 bits?

❑ If most immediate values are larger, this instruction is useless!

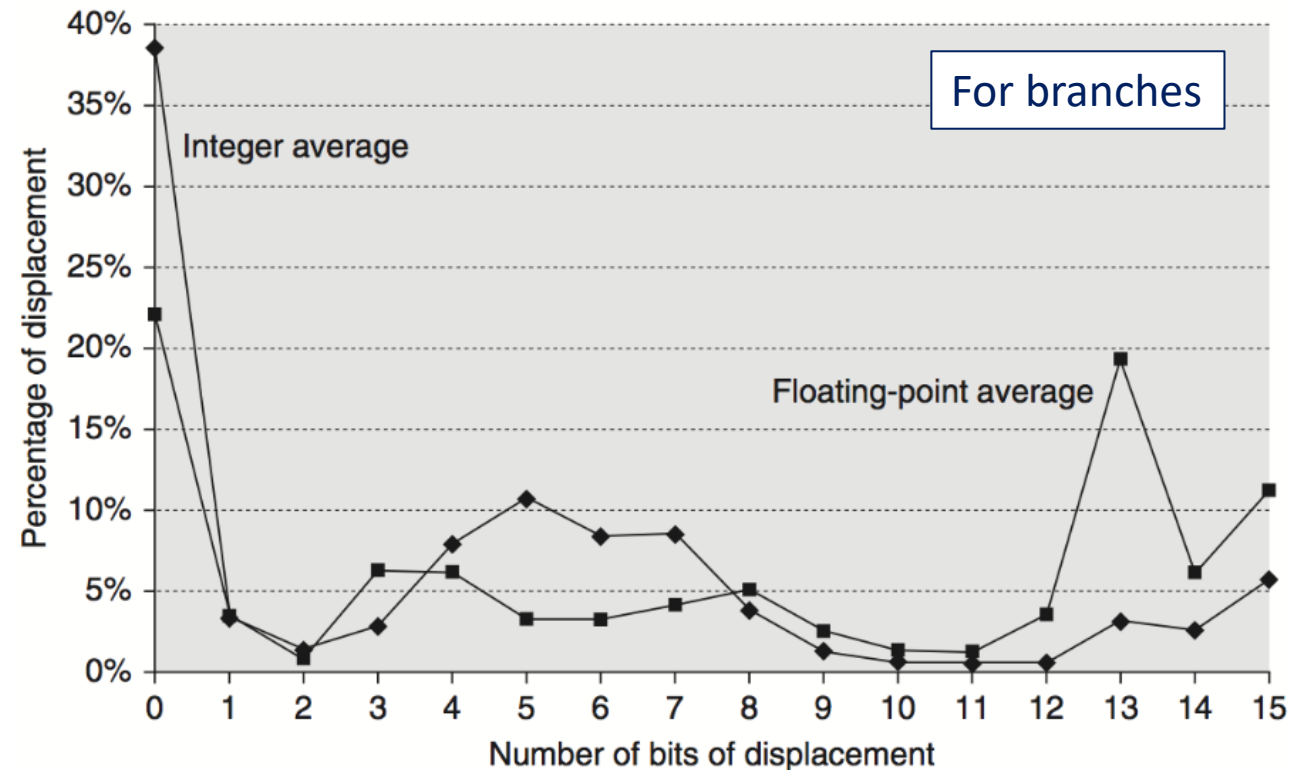
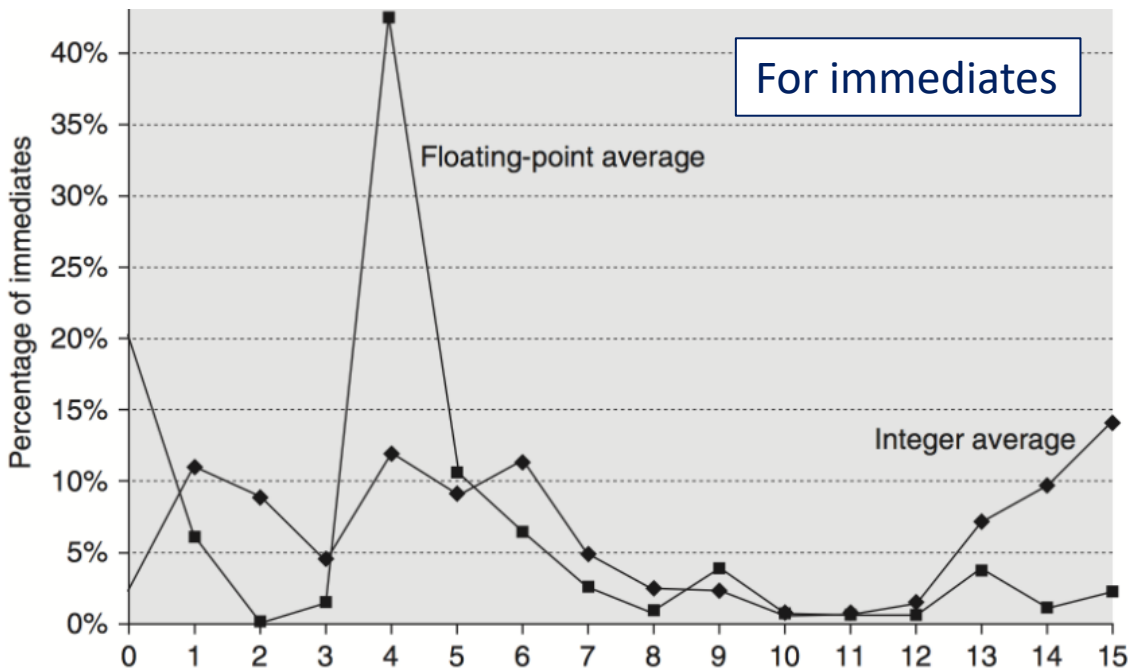
- Why not encode more imm, and reduce register count?



Benchmark-driven ISA design



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



RISC-V 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

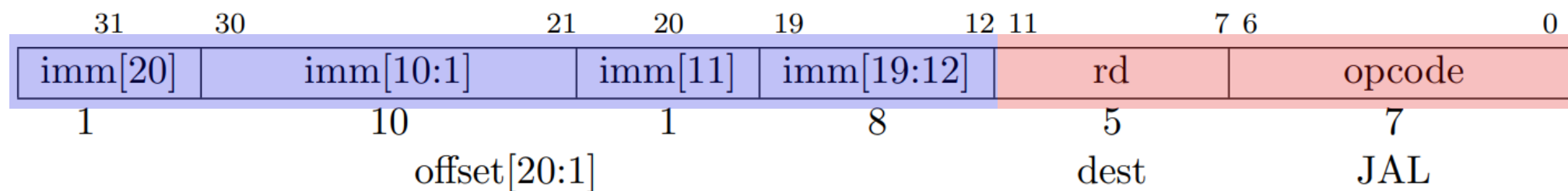
□ Why do we need two variants?

- 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)
```

RISC-V UJ-Type encoding

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

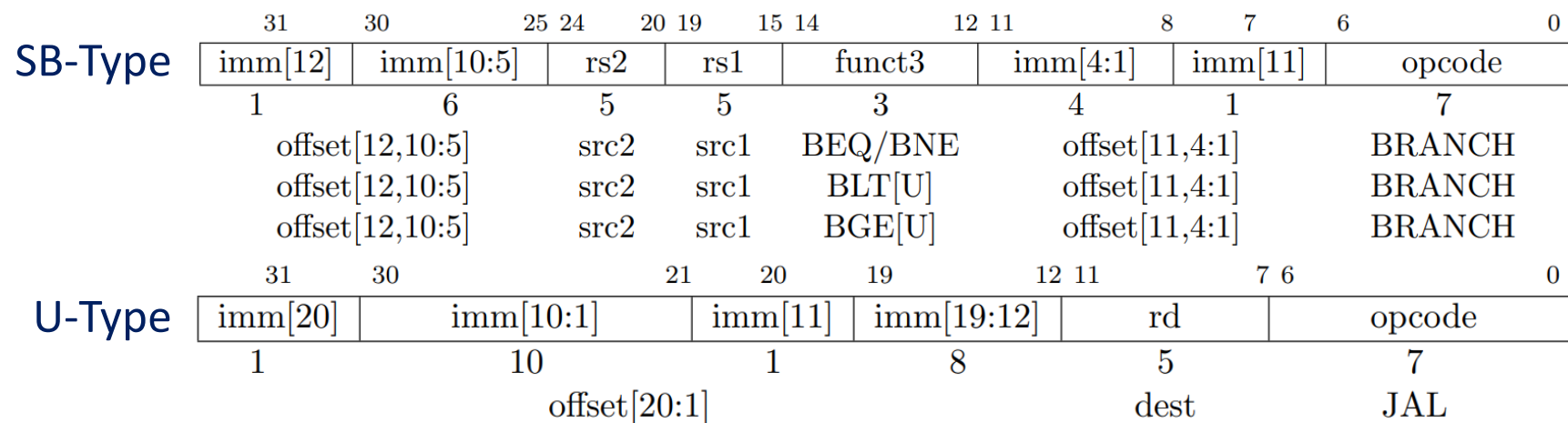


Only 21 bits of offset! What if target is farther?

RISC-V Relative addressing

❑ Problem: jump target offset is small!

- For branches: 13 bits, For JAL: 21 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



Three types of instructions – Part 4

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
4. Load upper immediate: Load (relatively) large immediate value

Problem: Addi/branch can load up to 12 bits! JAL can load up to 21 bits!
How do we encode large values?

RISC-V Load upper immediate instructions

❑ LUI: Load upper immediate

- lui dst, immediate \rightarrow dst = immediate \ll 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

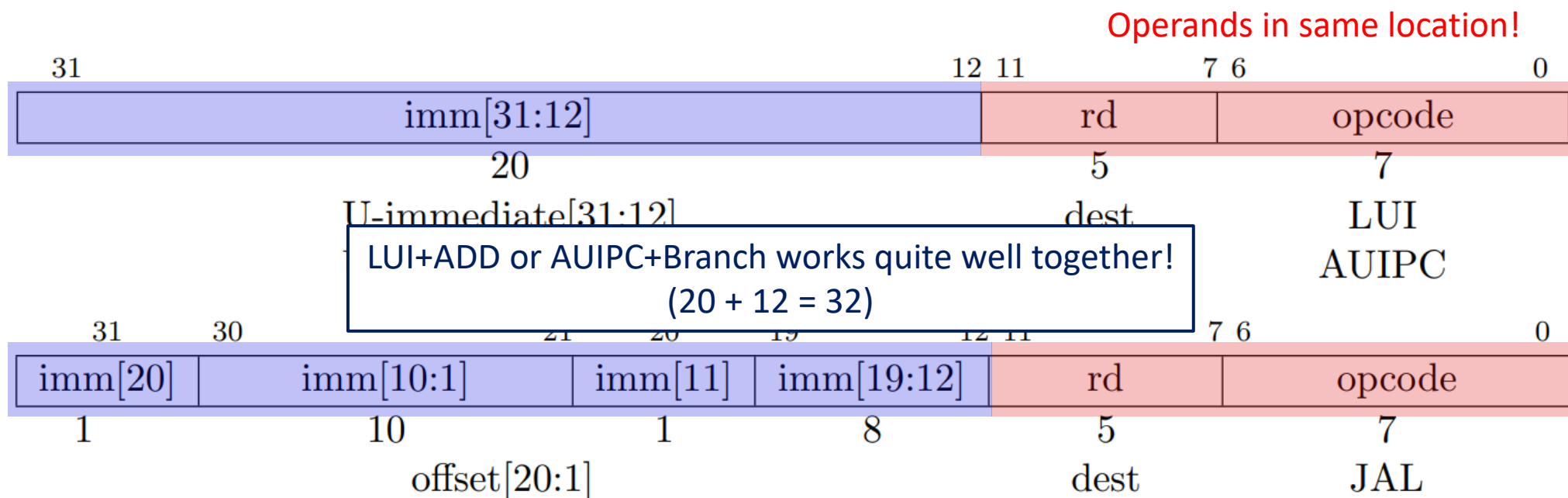
❑ AUIPC: Add upper immediate to PC

- auipc, dst, immediate \rightarrow dst = PC + immediate \ll 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

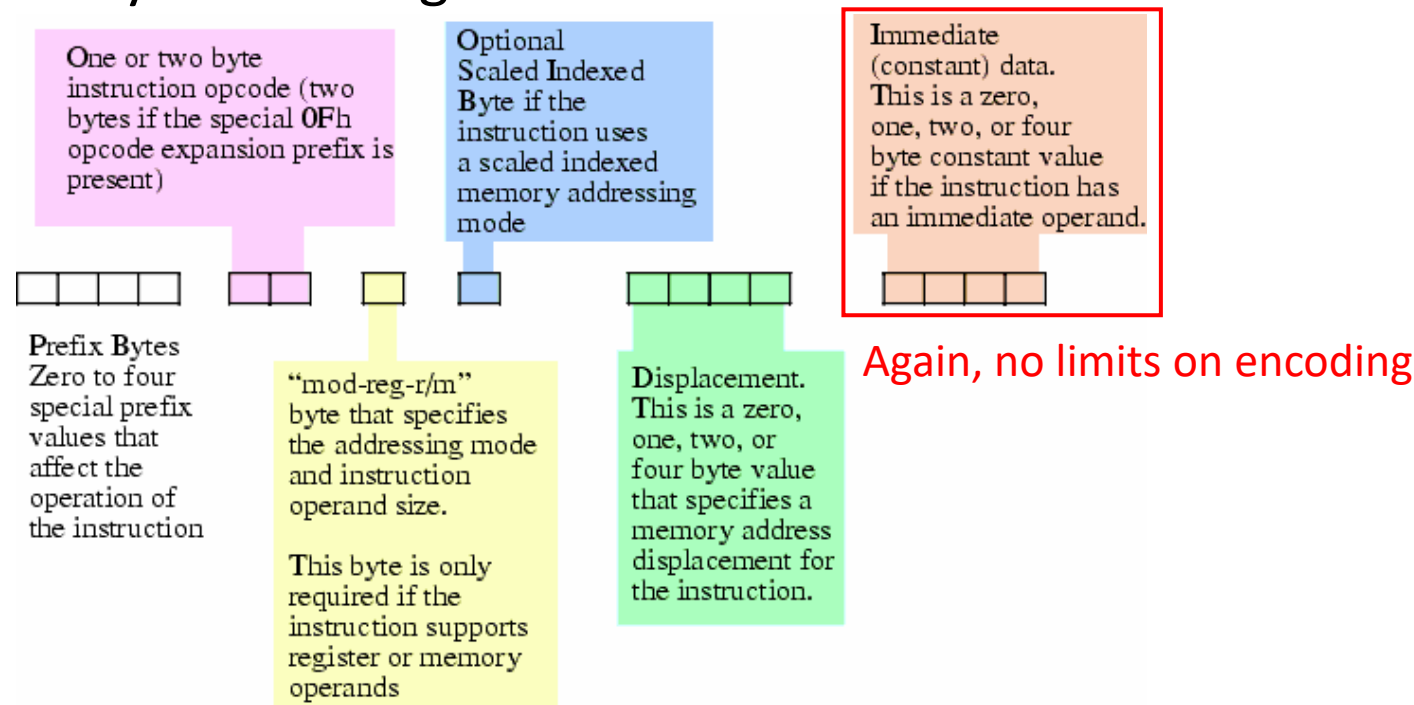
RISC-V U-Type and UJ-Type encoding

- 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)



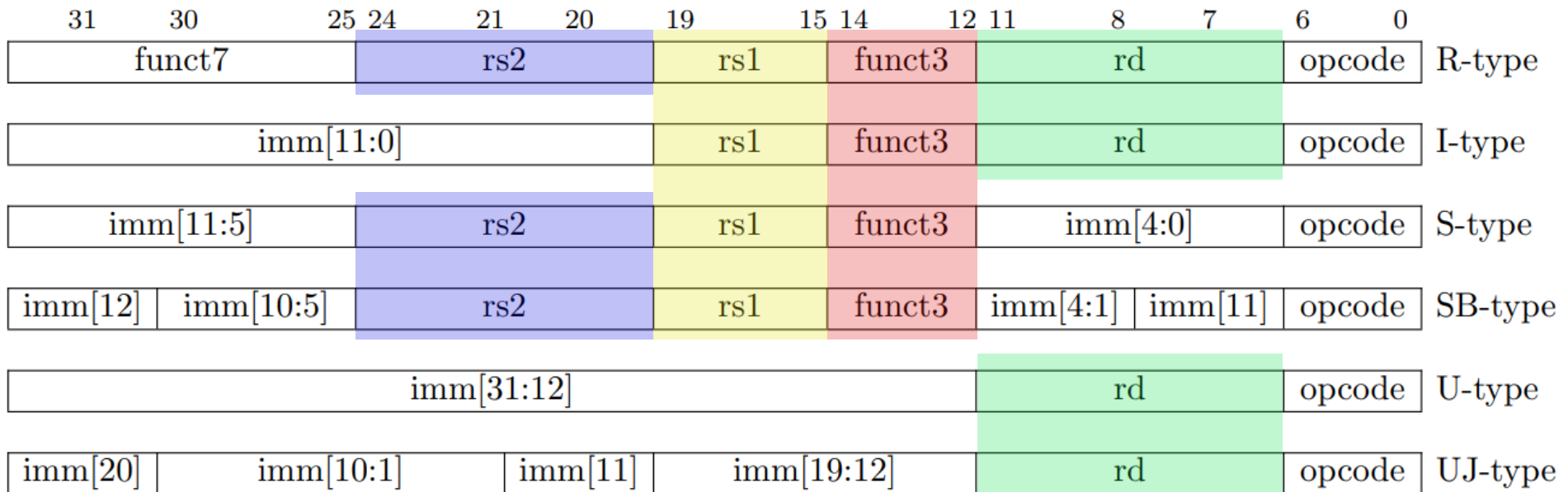
x86: CISC requires complex encoding!

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



RISC-V Design consideration: Consistent operand encoding location

- Simplifies circuits, resulting in less chip resource usage



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

```
cmp <reg>,<reg>
```

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

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

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

❑ e.g., x86 “cmp” compares two values and sets condition code flags

- Usual addressing modes
- Jump instruction variants read condition code flags

```
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)
```

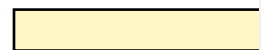
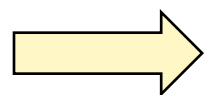
Conditional execution in CISC: Condition codes

- ❑ 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

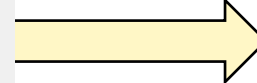
```
if (a > 10) {  
    a = 10;  
} else {  
    a = a + 1;  
}
```

C Code



```
    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.
```

Without predicated instructions



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

With predicated instructions

RISC-V Condition codes

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

CS152: Computer Systems Architecture

Programming With Assembly



Sang-Woo Jun

Fall 2022



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

RISC-V Pseudoinstructions

- ❑ Using raw RISC-V instructions is complicated for programmer
 - e.g., How can I load a 32-bit immediate into a register? **Trivial on x86**
- ❑ Solved by “Pseudoinstructions” that are not implemented in hardware
 - Assembler expands it to one or more instructions

Pseudo-Instruction	Description
li dst, imm	Load immediate
la dst, label	Load label address
bgt, ble, bgtu, bleu, ...	Branch conditions translated to hardware-implemented ones
jal label	jal x1, 0(label)
ret	Return from function (jalr x0, x1, 0)


...and more! Look at provided ISA reference

Why x0, why x1?



RISC-V register conventions

- ❑ Convention: **Not enforced** by hardware, but agreed by programmers
 - Except x0 (zero). Value of x0 is always zero regardless of what you write to it
 - Used to discard operations results. e.g., jalr x0, x1, 0 ignores return address

Registers	Symbolic names	Description	Saver  ?
x0	zero	Hardwired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporary registers	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x11	a0-a1	Function arguments and return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

 Symbolic names also used in assembler syntax

RISC-V Calling conventions and stack

□ Some register conventions during function call

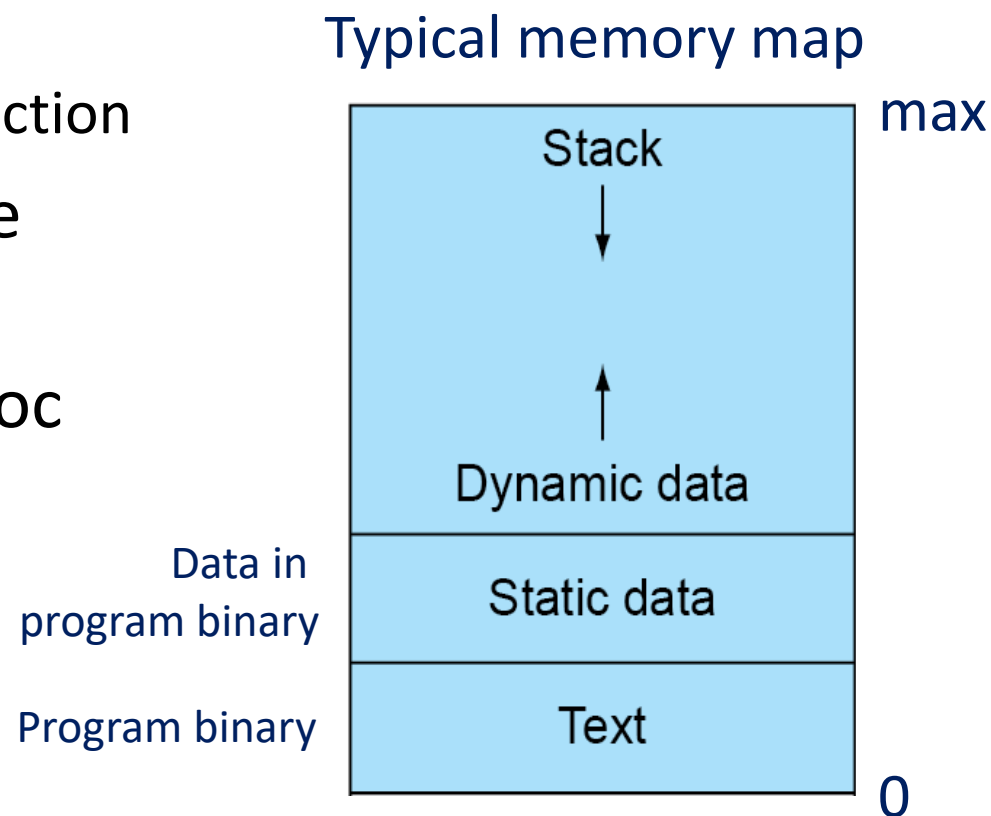
- ra (x1): typically holding return address
 - Saver is “caller”, meaning a function caller must save its ra somewhere before calling
- sp (x2): typically used as stack pointer
- t0-t6: temporary registers “Save” where? Registers are limited
 - Saver is “caller”, meaning a function caller must save its values somewhere before calling, if its values are important (Callee can use it without worrying about losing value)
- a0-a7: arguments to function calls and return value
 - Saver is “caller”
- s0-s11: saved register
 - Saver is “callee”, meaning if a function wants to use one, it must first save it somewhere, and restore it before returning

Merely conventions, not enforced.

Does not matter if your code never calls other people’s code, or is called by other people’s code

RISC-V Calling conventions and stack

- ❑ Registers saved in off-chip memory across function calls
- ❑ Stack pointer x2 (sp) used to point to top of stack
 - sp is callee-save
 - No need to save if callee won't call another function
- ❑ Stack space is allocated by decreasing value
 - Referencing done in sp-relative way
- ❑ Aside: Dynamic data used by heap for malloc



RISC-V Example: Using callee-saved registers

❑ Will use s0 and s1 to implement f

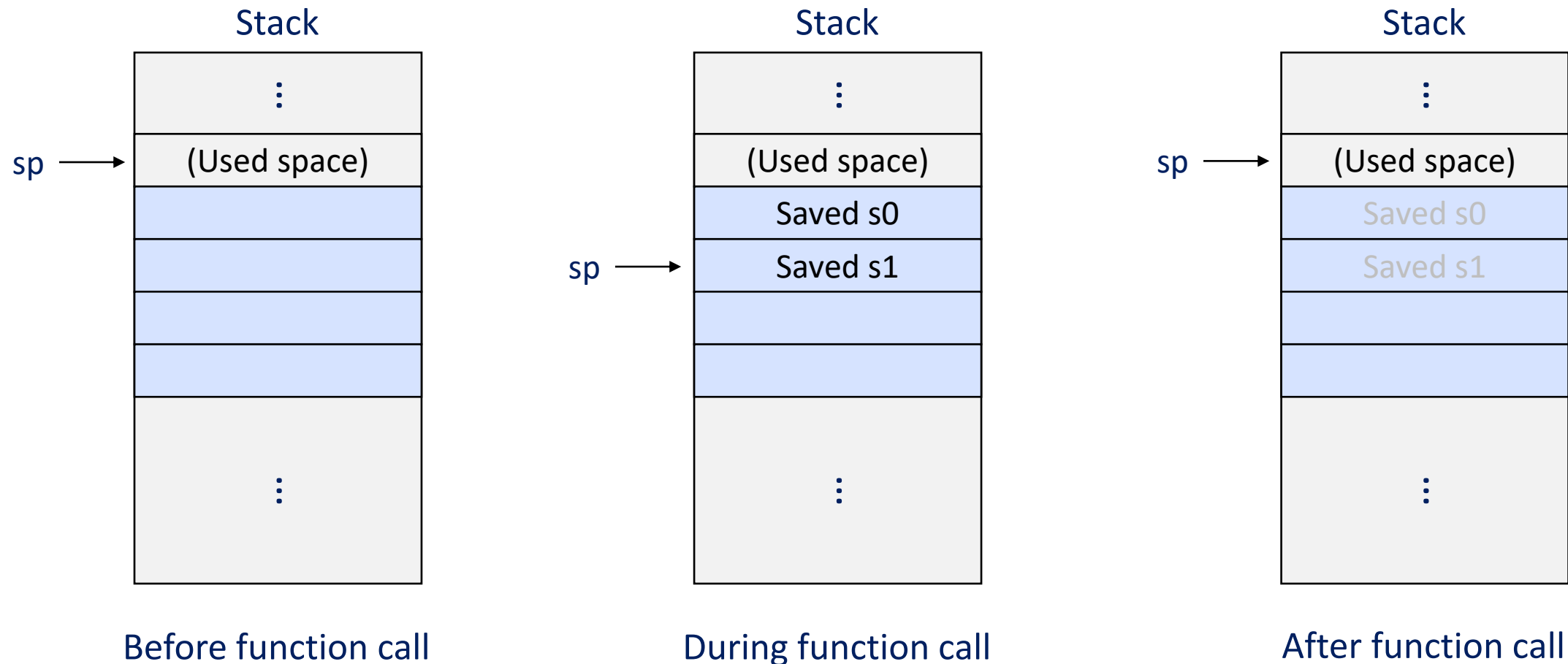
```
int f(int x, int y) {  
    return (x + 3) | (y + 123456);  
}
```

f:

```
addi sp, sp, -8 // allocate 2 words (8 bytes) on stack  
sw s0, 4(sp) // save s0  
sw s1, 0(sp) // save s1  
addi s0, a0, 3  
li s1, 123456  
add s1, a1, s1  
or a0, s0, s1  
lw s1, 0(sp) // restore s1  
lw s0, 4(sp) // restore s0  
addi sp, sp, 8 // deallocate 2 words from stack  
 // (restore sp)
```

ret

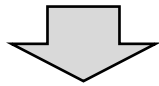
RISC-V Example: Using callee-saved registers



RISC-V Example: Using caller-saved registers

Caller

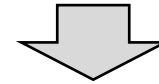
```
int x = 1;
int y = 2;
int z = sum(x, y);
int w = sum(z, y);
```



```
li a0, 1
li a1, 2
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp) // save y
jal ra, sum
// a0 = sum(x, y)
lw a1, 4(sp) // restore y
jal ra, sum
// a0 = sum(z, y)
lw ra, 0(sp)
addi sp, sp, 8
```

Callee

```
int sum(int a, int b) {
    return a + b;
}
```



```
sum:
    add a0, a0, a1
    ret
```

Source: MIT 6.004 2019 L03

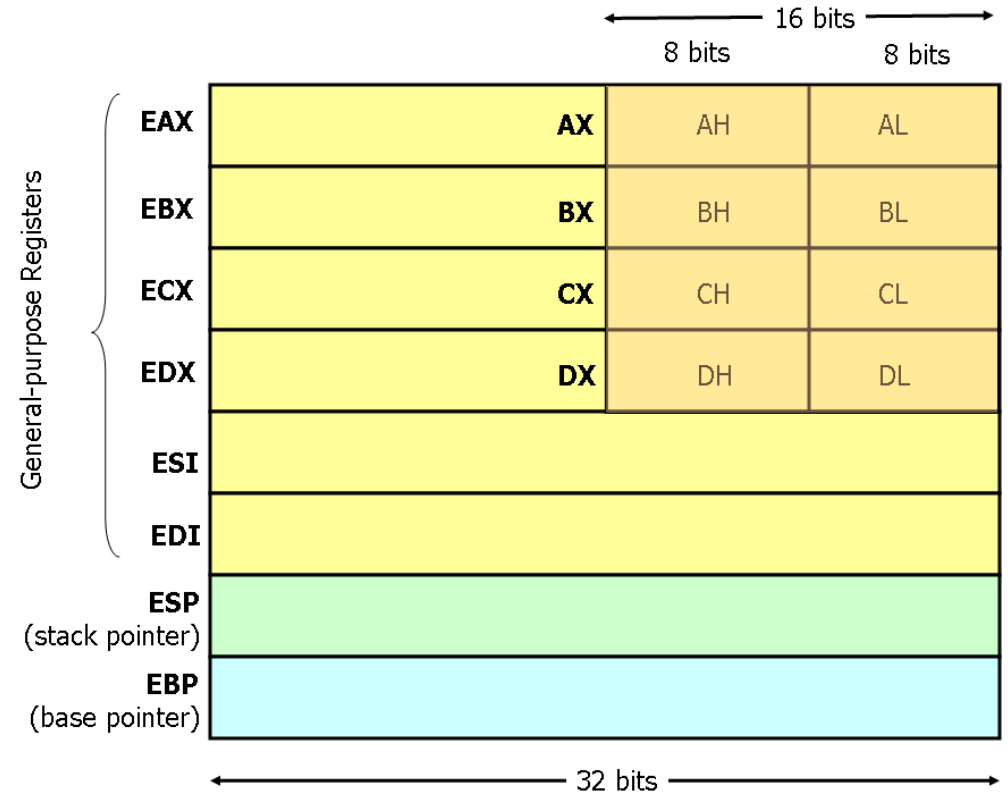
ra is saved, meaning even if callee calls another function, caller can still retrieve its ra

Why did the caller save a1?
We don't know which registers callee will use
Caller must save all caller-save registers it cares about

x86 register conventions

- ❑ Four 'general purpose' registers
 - A,B,C,D (Too few!)
- ❑ Two 'index' registers (for string ops)
 - ESI (Source index), EDI (Destination index)
- ❑ Special registers for stack management
 - ESP (stack pointer), EBP (base pointer)
- ❑ Caller-saved, callee-saved
 - Caller: EAX,ECX,EDX, ESI, EDI
 - Callee: EBX, ESP, EBP

Why only EBX...



Register convention comparisons

❑ RISC-V

- Instructions have almost no implicit effects
 - add x1 x2 x3 updates only x1. No other state changes
- Registers are almost entirely symmetrical
 - Only x0 is special (zero). All others are same. Conventions not enforced.

❑ x86

- Instructions can have a large number of implicit effects
 - “there are a number of instructions which (unexpectedly?) use one of them—but which one?—implicitly.”*
- Many instructions can only work with certain registers
 - Counter instructions update ECX, etc.
 - Remnant of backwards compatibility to 8080, a single-register (Accumulator) architecture with additional special registers

*<https://stackoverflow.com/questions/1856320/purpose-of-esi-edi-registers>

More details can be found here: <https://stackoverflow.com/questions/45538021/how-to-know-if-a-register-is-a-general-purpose-register>

Aside: Strange x86 optimizations: ESP

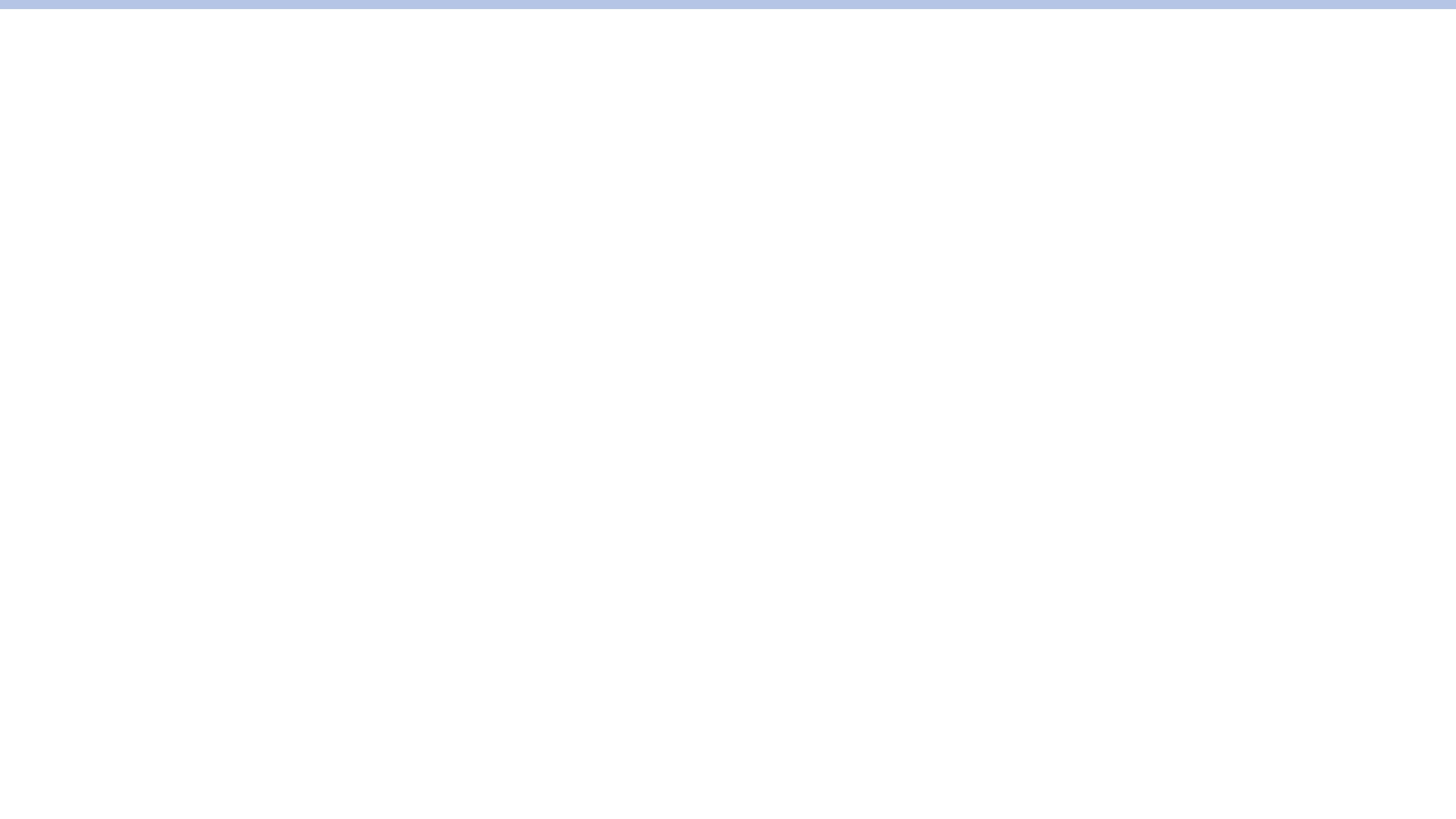
- ❑ ESP (Stack pointer) typically stores top of stack address
- ❑ It can also be used for (almost) general-purpose operations
 - Additional register can prevent a costly memory access!
 - Some restrictions. e.g., Cannot be used as address index
 - ESP must be saved somewhere and restored later
- ❑ No compiler will do this automatically (I think)
 - Really in-depth performance engineering requires assembly code

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!

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



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 \nRightarrow 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