

Jin-Soo Kim  
([jinsoo.kim@snu.ac.kr](mailto:jinsoo.kim@snu.ac.kr))

Systems Software &  
Architecture Lab.

Seoul National University

Fall 2022

# Machine-level Representation of Programs



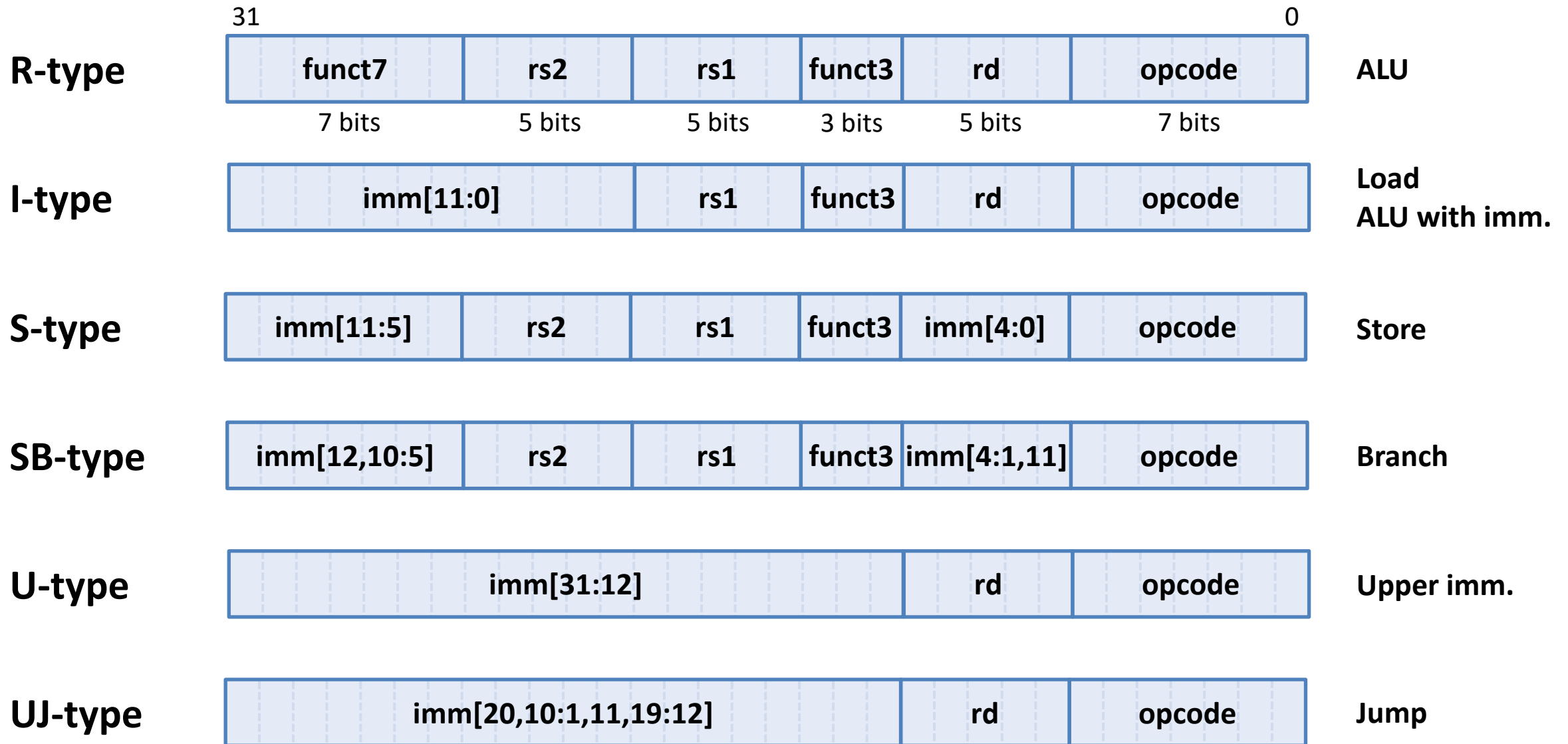
# RISC-V: Representing Instructions

Chap. 2.5

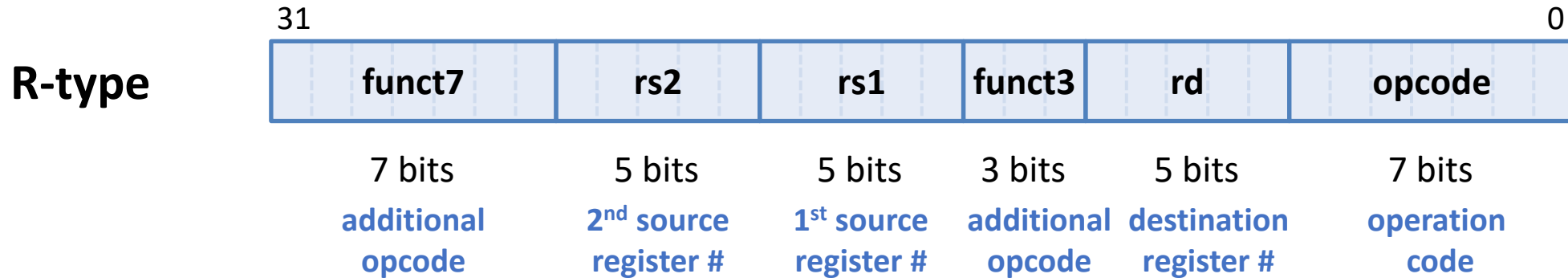
# Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, ...
  - Regularity!

# RISC-V Instruction Formats

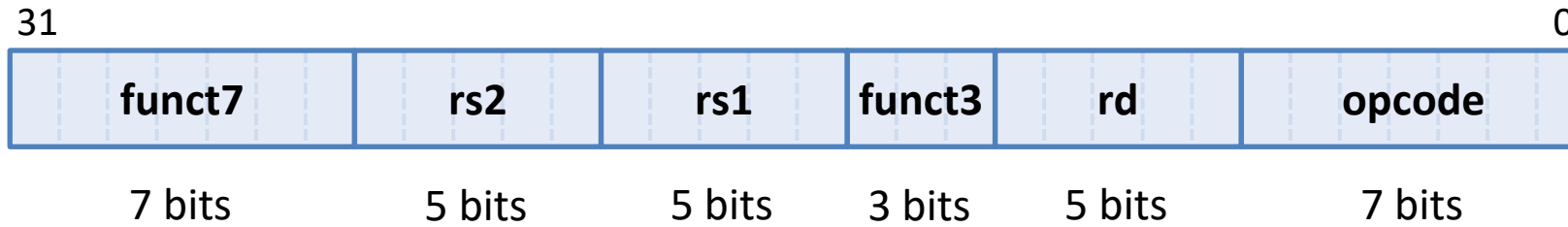


# RISC-V R-type Instructions

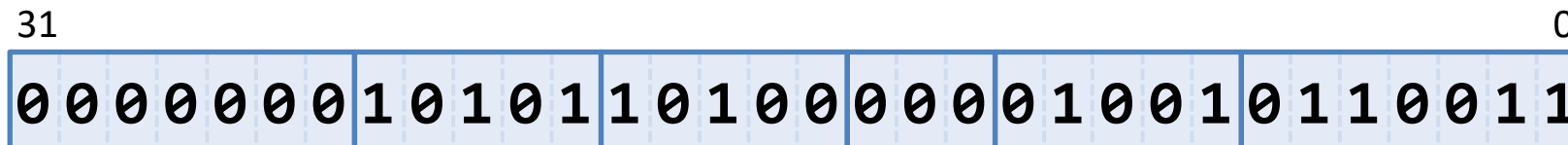


Instruction	Type	Example	funct7	funct3	opcode
<b>add</b>	R	add rd, rs1, rs2	0000000	000	0110011
<b>sub</b>	R	sub rd, rs1, rs2	0100000	000	0110011
<b>sll</b>	R	sll rd, rs1, rs2	0000000	001	0110011
<b>slt</b>	R	slt rd, rs1, rs2	0000000	010	0110011
<b>sltu</b>	R	sltu rd, rs1, rs2	0000000	011	0110011
<b>xor</b>	R	xor rd, rs1, rs2	0000000	100	0110011
<b>srl</b>	R	srl rd, rs1, rs2	0000000	101	0110011
<b>sra</b>	R	sra rd, rs1, rs2	0100000	101	0110011
<b>or</b>	R	or rd, rs1, rs2	0000000	110	0110011
<b>and</b>	R	and rd, rs1, rs2	0000000	111	0110011

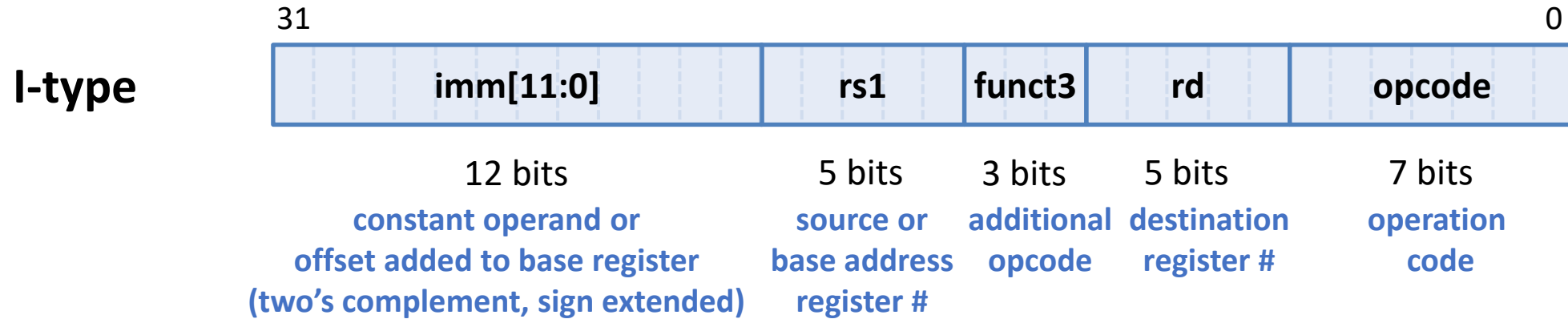
# R-type Example



add x9, x20, x21 == 015A04B3<sub>16</sub>



# RISC-V I-type Instructions



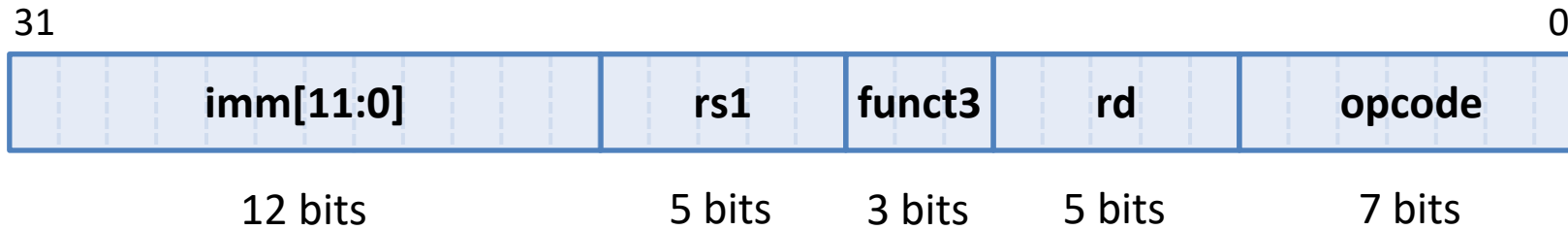
- Immediate arithmetic or load instructions
- **Design Principle 3: Good design demands good compromises**
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

# RISC-V I-type Instructions (cont'd)

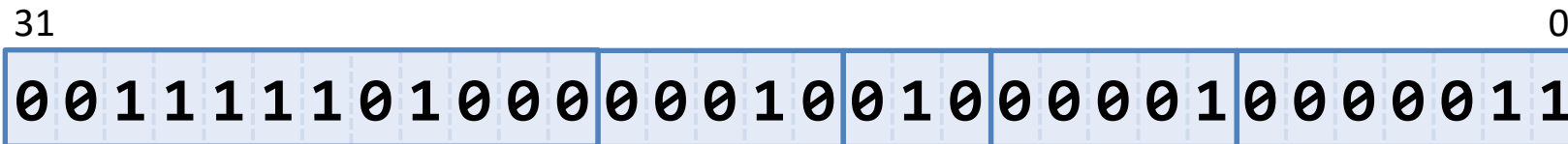
Instruction	Type	Example	funct7	rs2	funct3	opcode
<b>addi</b>	I	addi rd, rs1, imm12	-		000	0010011
<b>slti</b>	I	slti rd, rs1, imm12	-		010	0010011
<b>sltiu</b>	I	sltiu rd, rs1, imm12	-		011	0010011
<b>xori</b>	I	xori rd, rs1, imm12	-		100	0010011
<b>ori</b>	I	ori rd, rs1, imm12	-		110	0010011
<b>andi</b>	I	andi rd, rs1, imm12	-		111	0010011
<b>slli</b>	I	slli rd, rs1, shamt	0000000	shamt	001	0010011
<b>srli</b>	I	srli rd, rs1, shamt	0000000	shamt	101	0010011
<b>srai</b>	I	srai rd, rs1, shamt	0100000	shamt	101	0010011
<b>lb</b>	I	lb rd, imm12(rs1)	-		000	0000011
<b>lh</b>	I	lh rd, imm12(rs1)	-		001	0000011
<b>lw</b>	I	lw rd, imm12(rs1)	-		010	0000011
<b>lbu</b>	I	lbu rd, imm12(rs1)	-		100	0000011
<b>lhu</b>	I	lhu rd, imm12(rs1)	-		101	0000011
<b>jalr</b>	I	jalr rd, imm12(rs1)	-		000	1100111



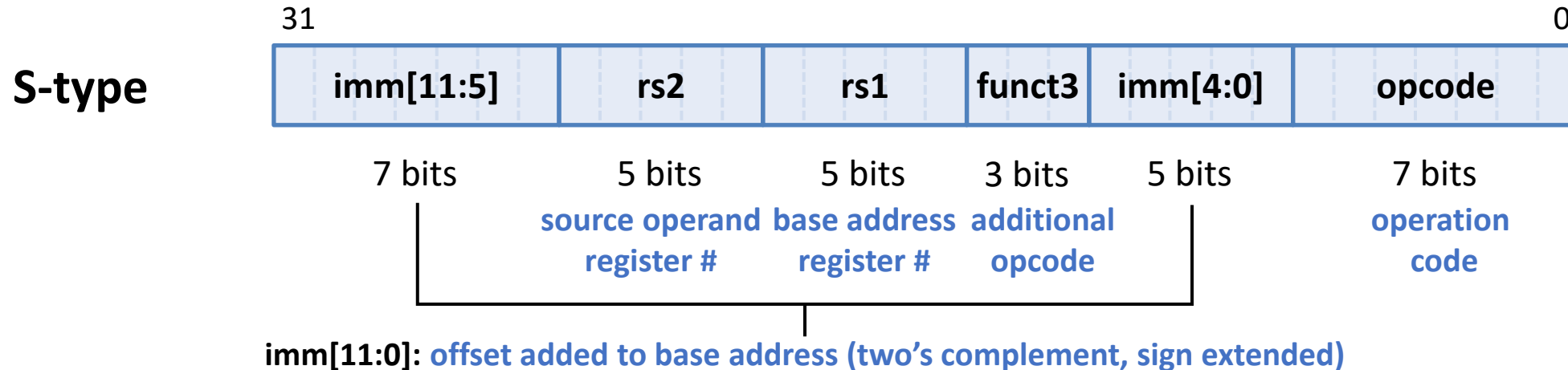
# I-type Example



`lw x1, 1000(x2) == 3E81208316`



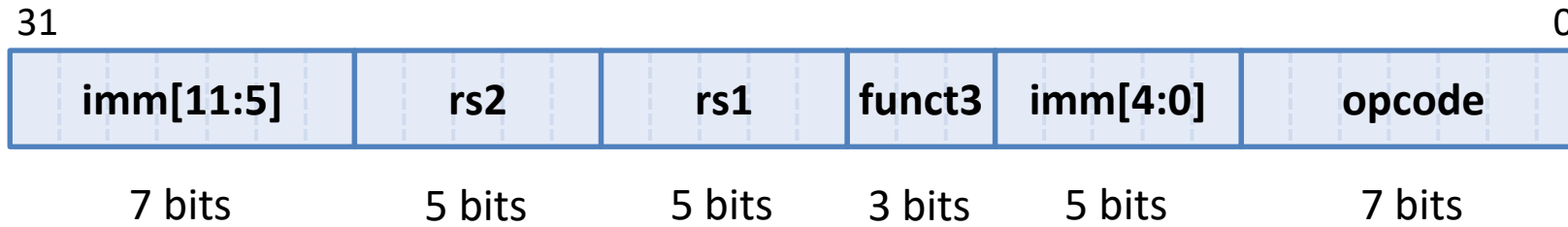
# RISC-V S-type Instructions



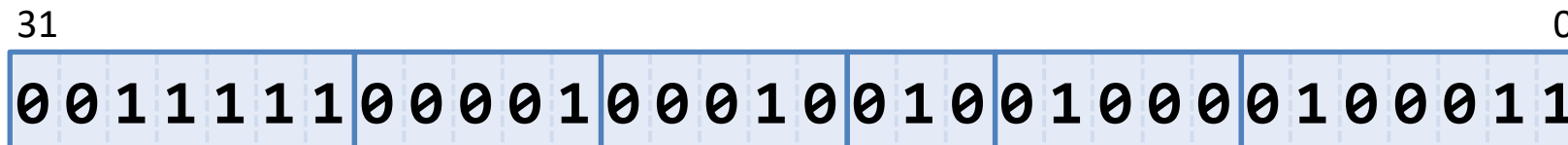
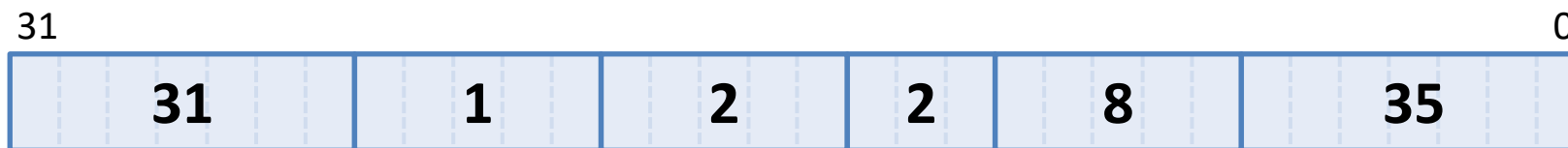
- Different immediate format for store instructions
  - Split so that rs1 and rs2 fields always in the same place

Instruction	Type	Example	funct7	funct3	opcode
sb	S	sb rs2, imm12(rs1)	-	000	0100011
sh	S	sh rs2, imm12(rs1)	-	001	0100011
sw	S	sw rs2, imm12(rs1)	-	010	0100011

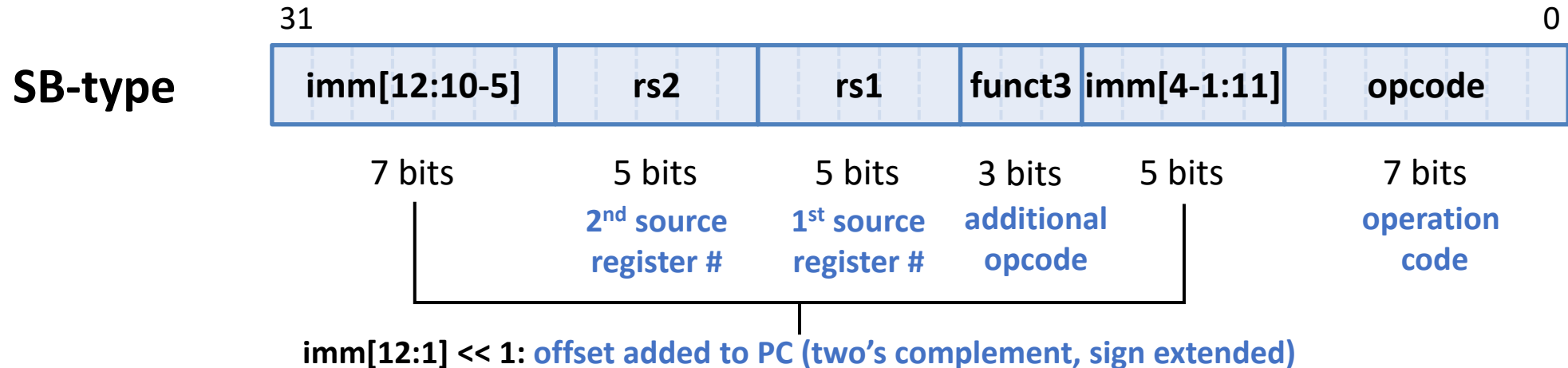
# S-type Example



`sw x1, 1000(x2) == 3E11242316`

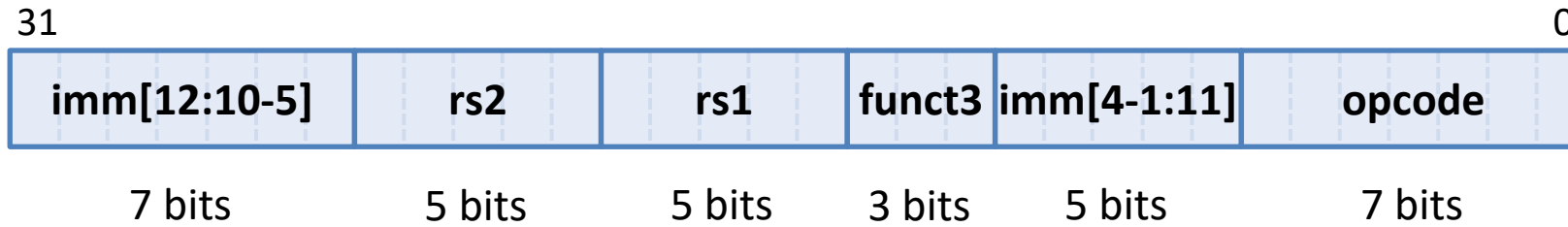


# RISC-V SB-type Instructions

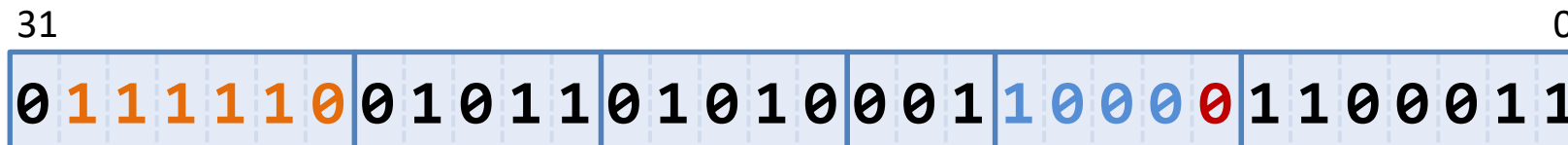


Instruction	Type	Example	funct7	funct3	opcode
beq	SB	beq rs1, rs2, imm12	-	000	1100011
bne	SB	bne rs1, rs2, imm12	-	001	1100011
blt	SB	blt rs1, rs2, imm12	-	100	1100011
bge	SB	bge rs1, rs2, imm12	-	101	1100011
bltu	SB	bltu rs1, rs2, imm12	-	110	1100011
bgeu	SB	bgeu rs1, rs2, imm12	-	111	1100011

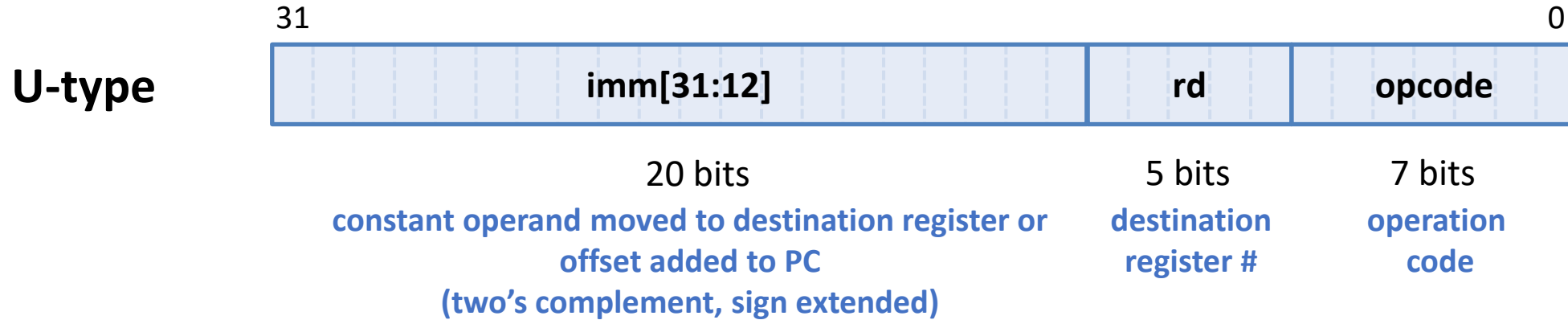
# SB-type Example



bne x10, x11, 2000 == 7CB51863<sub>16</sub>  
(0111 1101 0000<sub>2</sub>)



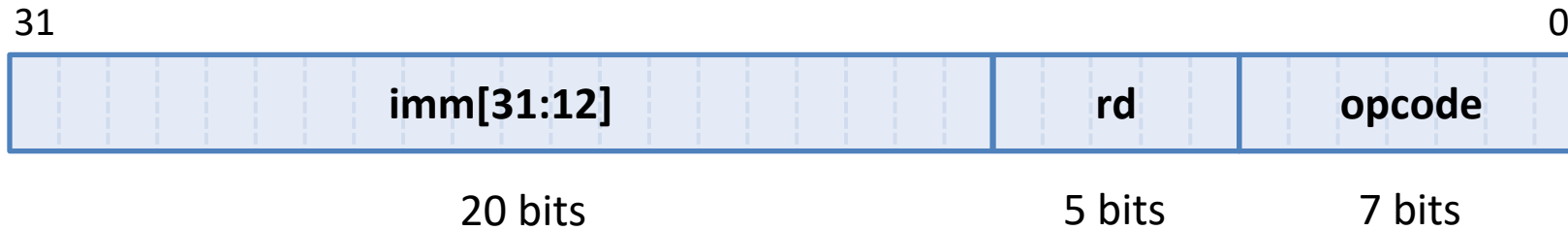
# RISC-V U-type Instructions



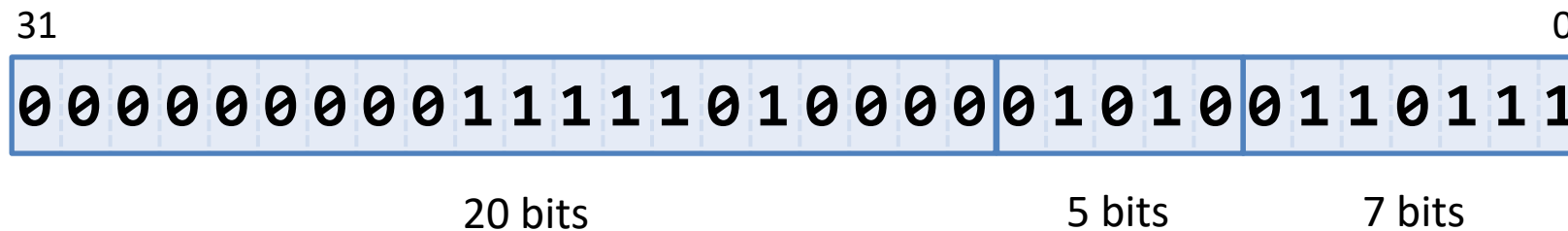
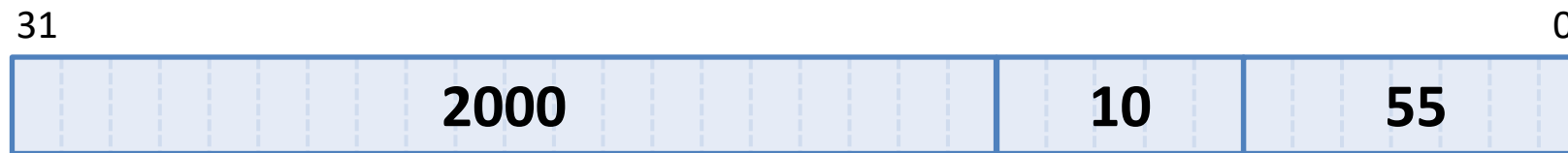
- 20-bit immediate is shifted left by 12 bits

Instruction	Type	Example	funct7	funct3	opcode
<b>lui</b>	U	lui rd, imm20	-	-	0110111
<b>auipc</b>	U	auipc rd, imm20	-	-	0010111

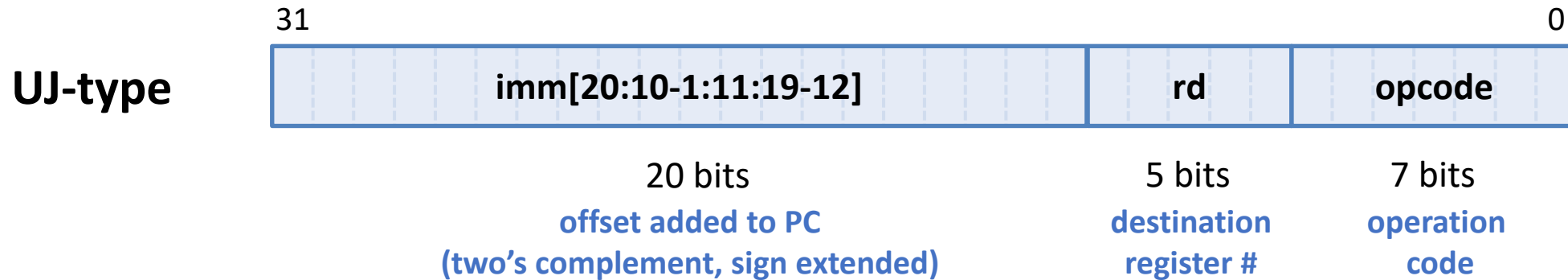
# U-type Example



`lui x10, 2000 == 007D053716`  
(0000 0000 0111 1101 0000<sub>2</sub>)



# RISC-V UJ-type Instructions

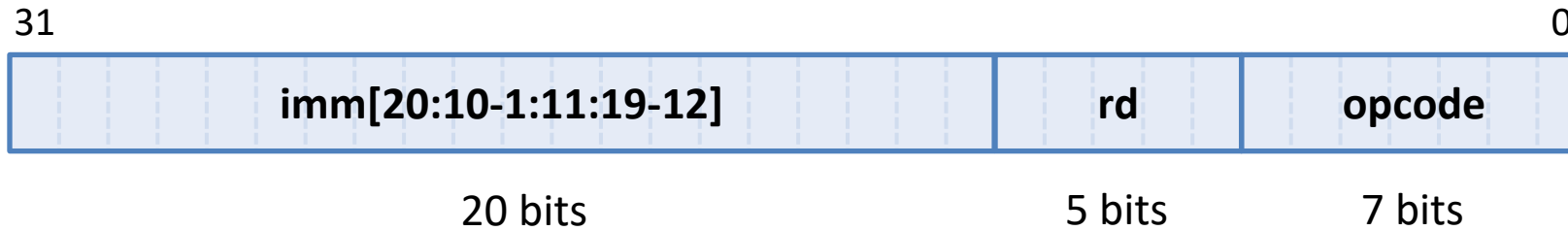


- 20-bit immediate is shifted left by 1 bit and added to PC

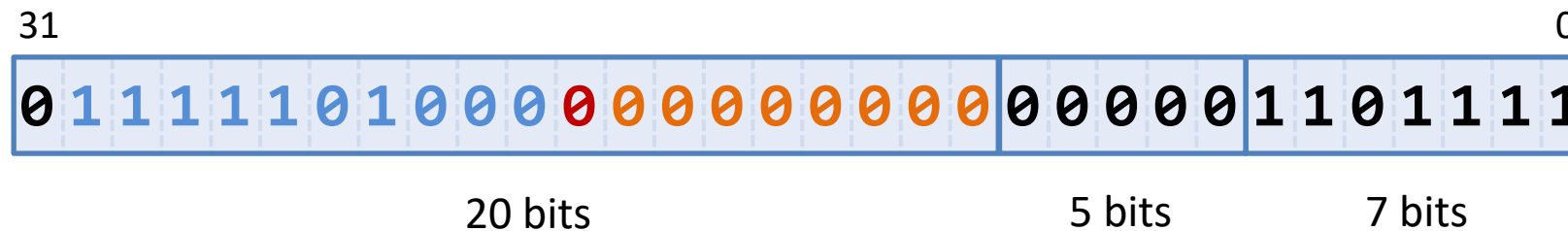
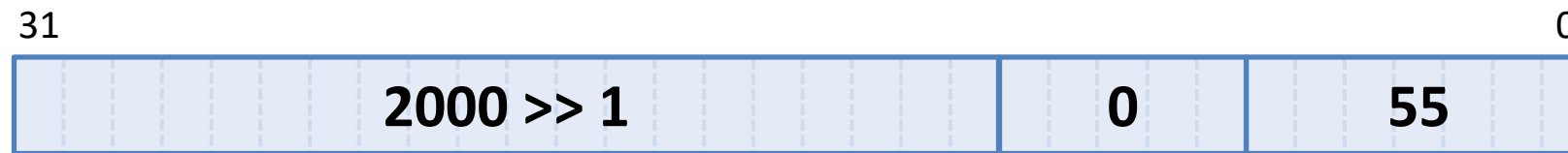
Instruction	Type	Example	funct7	funct3	opcode
<b>jal</b>	UJ	jal rd, imm20	-	-	1101111



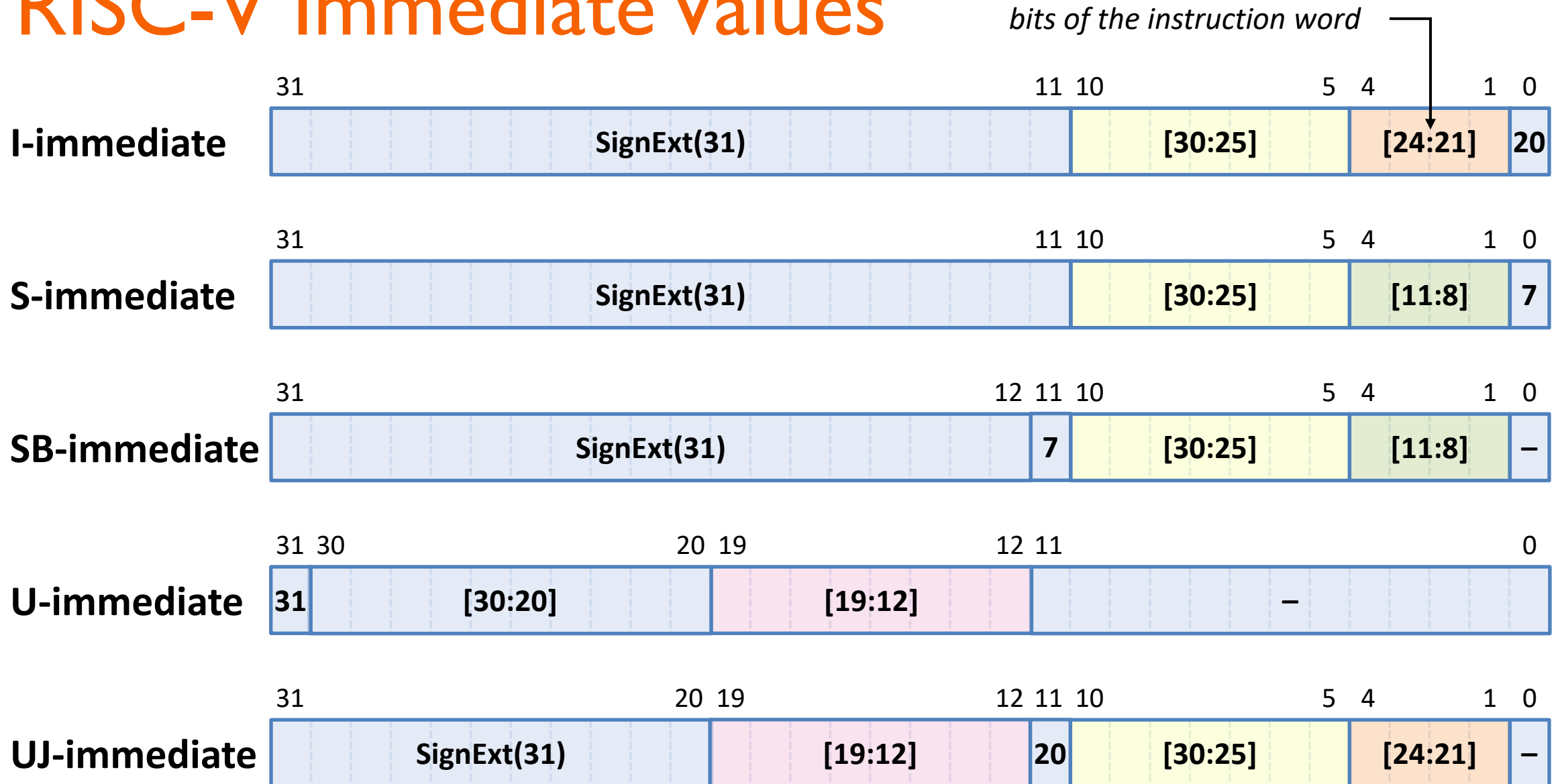
# UJ-type Example



jal x0, 2000 == 7D00006F<sub>16</sub>  
(0000 0000 0111 1101 0000<sub>2</sub>)

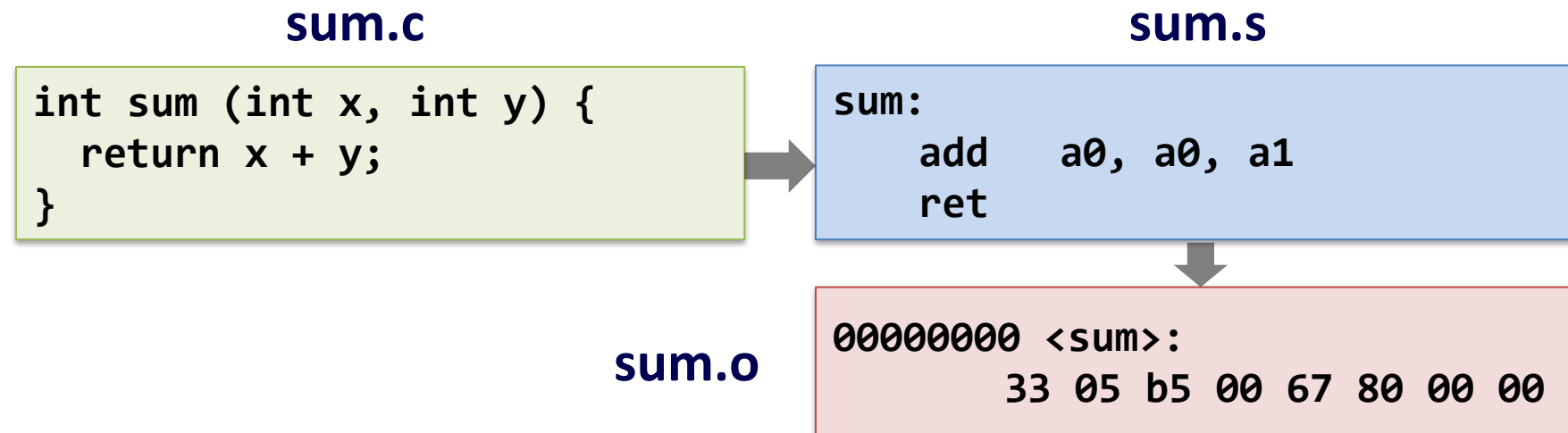


# RISC-V Immediate Values



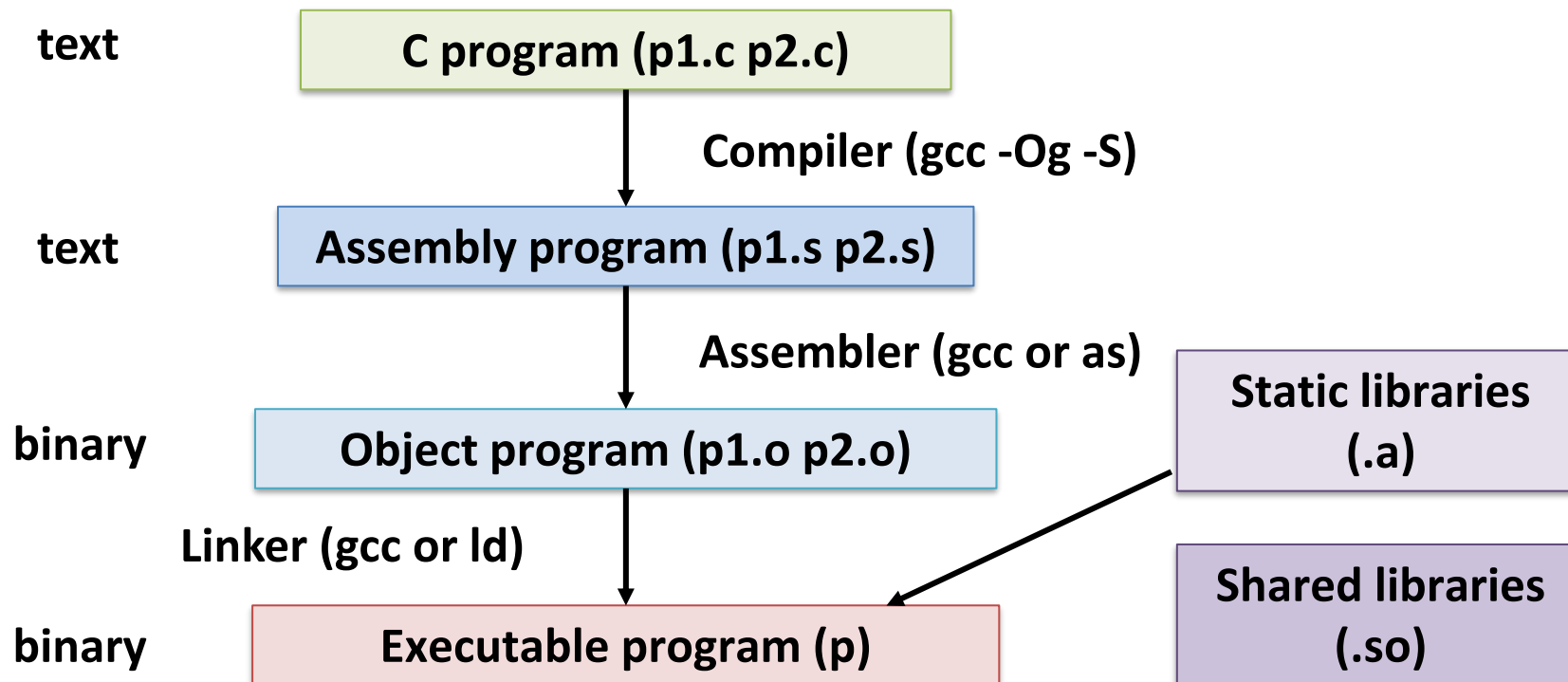
# Compiling into Machine Code

- Machine code (or binary code)
  - The byte-level programs that a processor executes
- Assembly code
  - A text representation of machine code
- `riscv32-unknown-elf-gcc -Og -S sum.c`



# Turning C into Executable Program

- `riscv32-unknown-elf-gcc -Og p1.c p2.c -o p`
  - Use basic optimizations (-Og)
  - Put resulting binary in file p



# Disassembling

- Disassembler: `riscv32-unknown-elf-objdump -d sum.o`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` (object code) file

```
sum.o:      file format elf32-littleriscv

Disassembly of section .text:

00000000 <sum>:
   0:      00b50533      add     a0,a0,a1
   4:      00008067      ret
```

# Disassembling with gdb

- Disassemble procedure `sum`

```
$ riscv32-unknown-elf-gcc -Og -g sum.c main.c
$ riscv32-unknown-elf-gdb a.out
(gdb) disassemble sum
Dump of assembler code for function sum:
   0x00010164 <+0>:      add     a0,a0,a1
   0x00010168 <+4>:      ret
End of assembler dump.
```

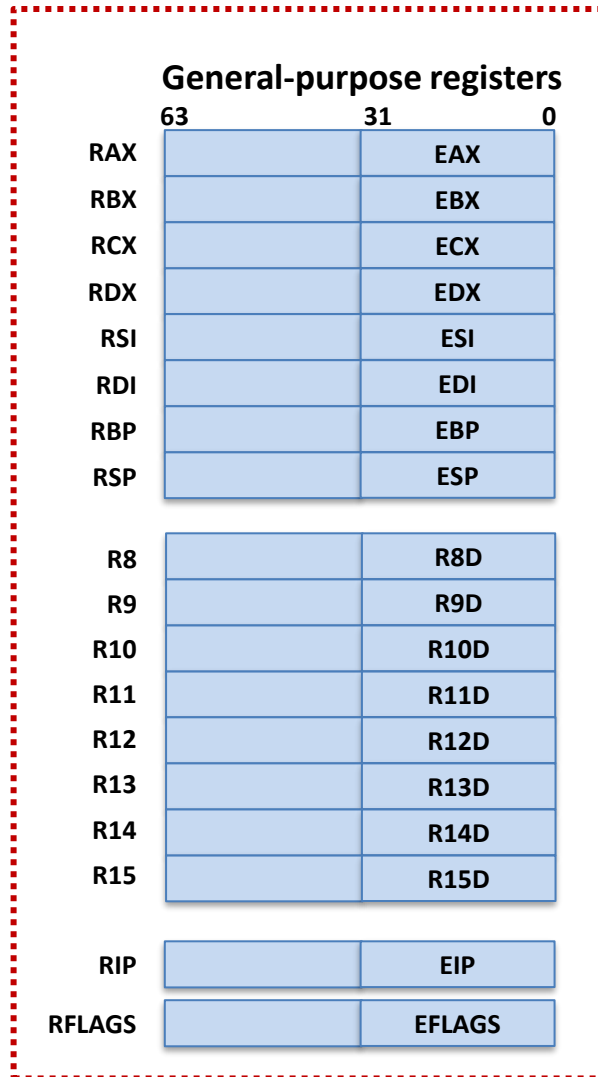
- Examine 8 bytes starting at `sum`

```
(gdb) x/8xb sum
0x10164 <sum>:  0x33    0x05    0xb5    0x00    0x67    0x80    0x00    0x00
```

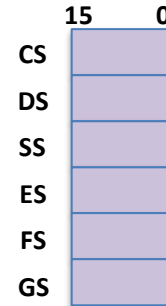
# Intel x86-64

Chap. 2.19

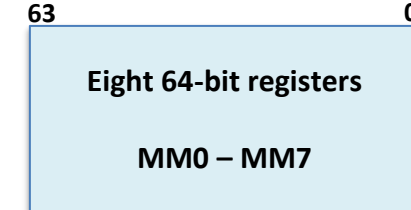
# x86 Basic Execution Environment



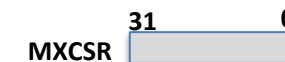
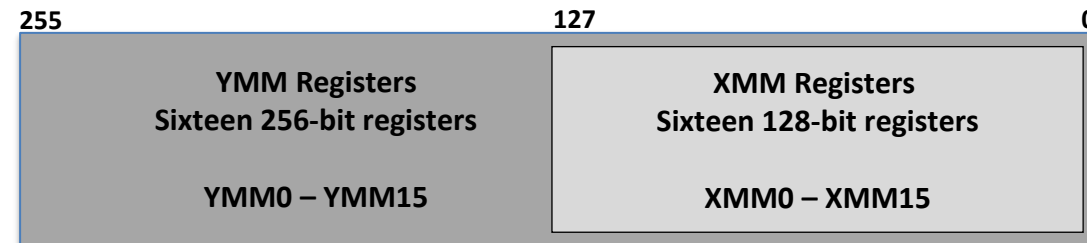
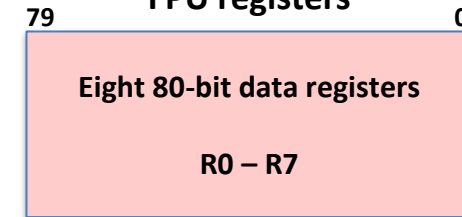
**Segment registers**



**MMX registers**



**FPU registers**





# x86 Operands

- Two operands per instruction
- The source can be an immediate, a register, or a memory location
- The destination can be a register or a memory location

Second source operand	Source/Dest operand	Example
Register	Register	<code>addq %rcx, %rax</code>
Immediate	Register	<code>addq \$4, %eax</code>
Memory	Register	<code>addq 0(%rcx), %rax</code>
Register	Memory	<code>addq %rax, 8(%rax)</code>
Immediate	Memory	<code>addq \$4, 8(%rax)</code>

# x86 Memory Addressing Modes

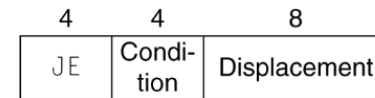
- **D(Rb, Ri, S)**     $\text{Mem}[\text{Reg}[\text{Rb}] + \text{S} * \text{Reg}[\text{Ri}] + \text{D}]$ 
  - **D:**            constant “displacement”: 1, 2, or 4 bytes
  - **Rb:**            Base register: any of 16 integer registers
  - **Ri:**            Index register: any, except for %rsp
  - **S:**            Scale: 1, 2, 4, or 8

Example	Effective address
<code>movq %rax, 8(%rdx)</code>	$\%rdx + 8$
<code>movq %rax, 16(%rdx, %rcx)</code>	$\%rdx + \%rcx + 16$
<code>movq %rax, (,%rcx, 4)</code>	$\%rcx * 4$
<code>movq %rax, 32(%rdx, %rcx, 8)</code>	$\%rdx + \%rcx * 8 + 32$

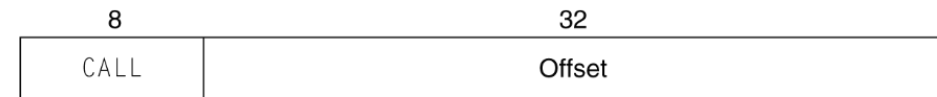
# x86 Instruction Encoding

- Variable length encoding
  - Postfix bytes specify addressing mode
  - Prefix bytes modify operation
    - Operand length, repetition, locking, ...
  - The length of an instruction can be up to 17 bytes
    - Up to 4 prefixes, each of which is 1 byte
    - 1 ~ 3 bytes for opcode
    - Up to 1 byte for simple addressing modes
    - Up to 1 byte for complex addressing modes
    - Up to 4 bytes for displacement
    - Up to 4 bytes for immediate data

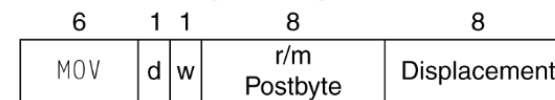
a. JE EIP + displacement



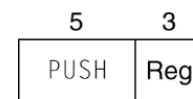
b. CALL



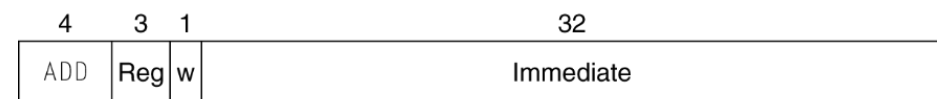
c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



# CISC vs. RISC

Chap. 2.18 – 2.20

# CISC (Complex Instruction Set Computer)

- Add instructions to perform “typical” programming tasks
  - DEC PDP-11 & VAX, IBM System/360, Motorola 68000, IA-32, Intel 64, ...
- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter, etc.
  - Explicit push and pop instructions
- Arithmetic instructions can access memory
  - Requires memory read and write during computation
  - Complex addressing modes
- Instructions have varying lengths
- Condition codes
  - Set as side effect of arithmetic and logical instructions

x86\_64

```
movq    A, %rsi
movq    B, %rdi
movq    n, %rcx
REP MOVS
```

# RISC (Reduced Instruction Set Computer)

- Philosophy: Fewer, simple instructions
  - Might take more to get given task done
  - Can execute them with small and fast hardware
  - MIPS, RISC-V, Sun SPARC, IBM PowerPC, ARM, SuperH, ...
- Register-oriented instruction set
  - Many more (typically 32+) registers
  - Use for arguments, return address, temporaries
- Only load & store instructions can access memory
- Each instruction has fixed size
- No condition codes
  - Test instructions return 0/1 in register

## RISC-V

```
la      a0, A
la      a1, B
li      a2, n
add     a3, a0, a2
L0:
lbu     a4, 0(a0)
sbu     a4, 0(a1)
addi    a0, a0, 1
addi    a1, a1, 1
bne     a0, a3, L0
```

# CISC vs. RISC

## ■ Original debate

- CISC proponents – easy for compiler, fewer code bytes
- RISC proponents – better for optimizing compilers, can make run fast with simple chip design

## ■ Current status

- For desktop/server processors, choice of ISA not a technical issue
  - With enough hardware, can make anything run fast
  - Code compatibility more important
- x86-64 adopted many RISC features
  - More registers, use them for argument passing
  - Hardware translates instructions to simpler  $\mu$ ops
- For embedded processors, RISC makes sense: smaller, cheaper, less power

# Summary

- Design principles
  - Simplicity favors regularity
  - Smaller is faster
  - Good design demands good compromises
- Make the common case fast (and make the rare case correct)
- RISC-V: typical of RISC ISAs