

Jin-Soo Kim
(jinsoo.kim@snu.ac.kr)
Systems Software &
Architecture Lab.
Seoul National University

Fall 2019

Machine-level Representation of Programs



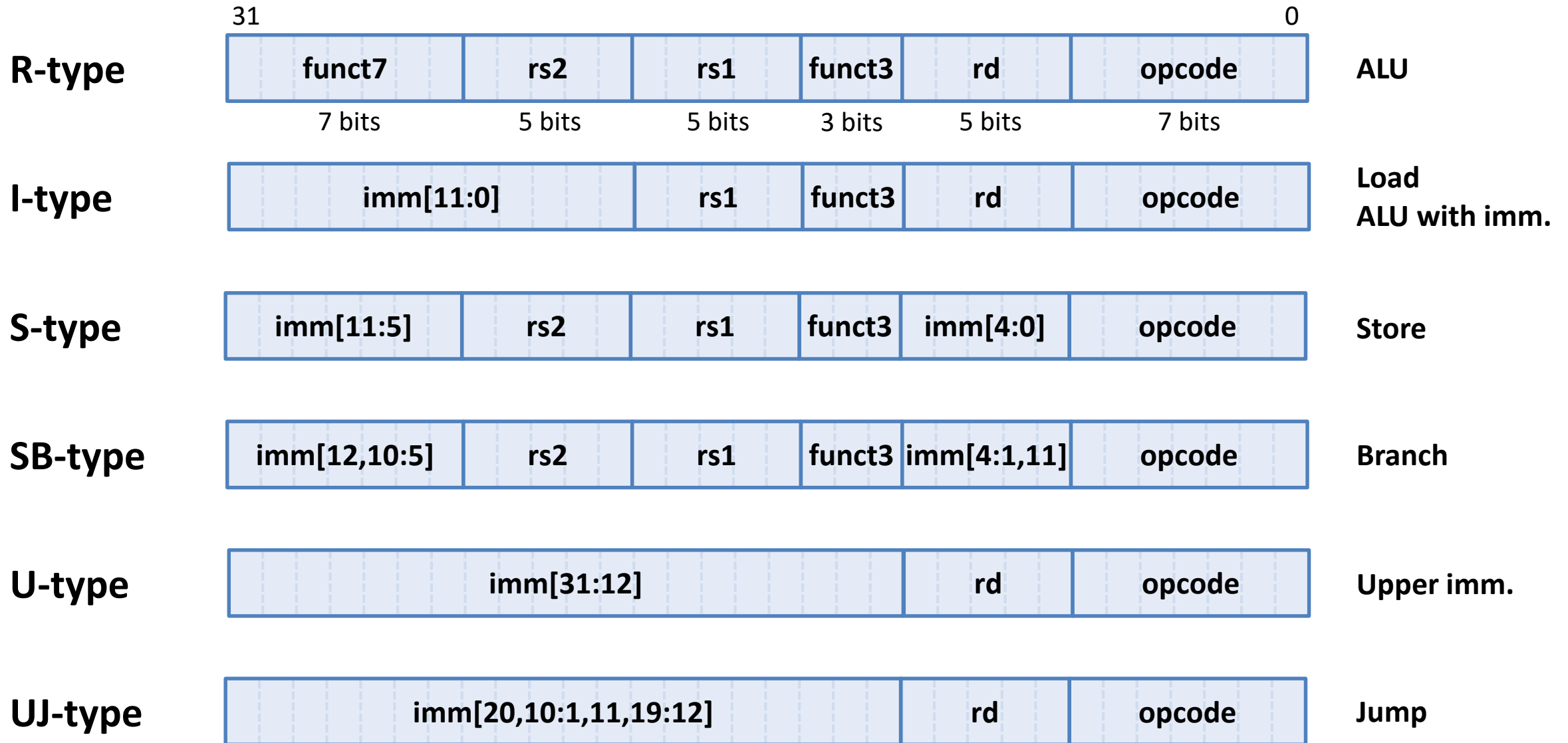
RISC-V: Representing Instructions

Chap. 2.5

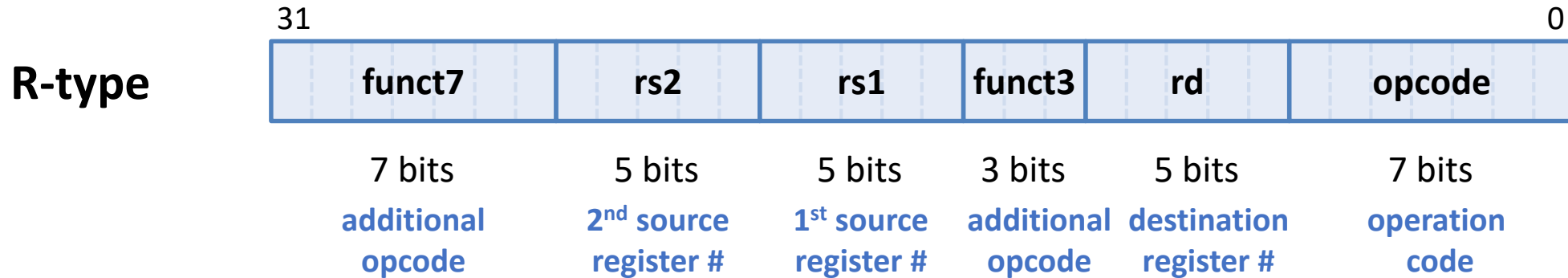
Representing Instructions

- 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

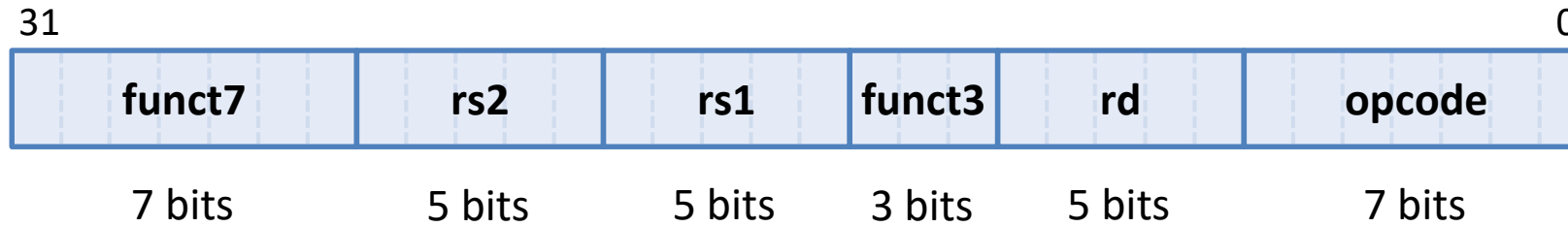


RISC-V R-type Instructions

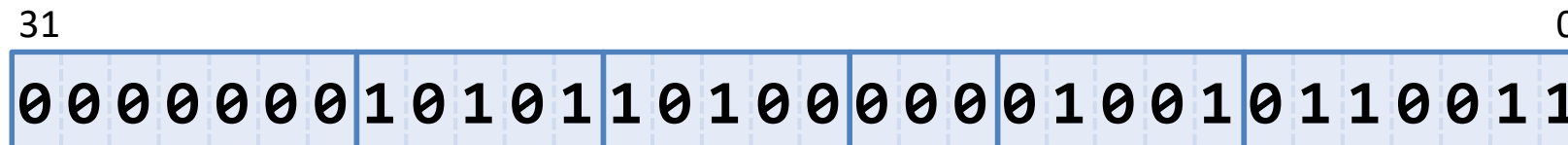


Instruction	Type	Example	funct7	funct3	opcode
add	R	add rd, rs1, rs2	0000000	000	0110011
sub	R	sub rd, rs1, rs2	0100000	000	0110011
sll	R	sll rd, rs1, rs2	0000000	001	0110011
slt	R	slt rd, rs1, rs2	0000000	010	0110011
sltu	R	sltu rd, rs1, rs2	0000000	011	0110011
xor	R	xor rd, rs1, rs2	0000000	100	0110011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

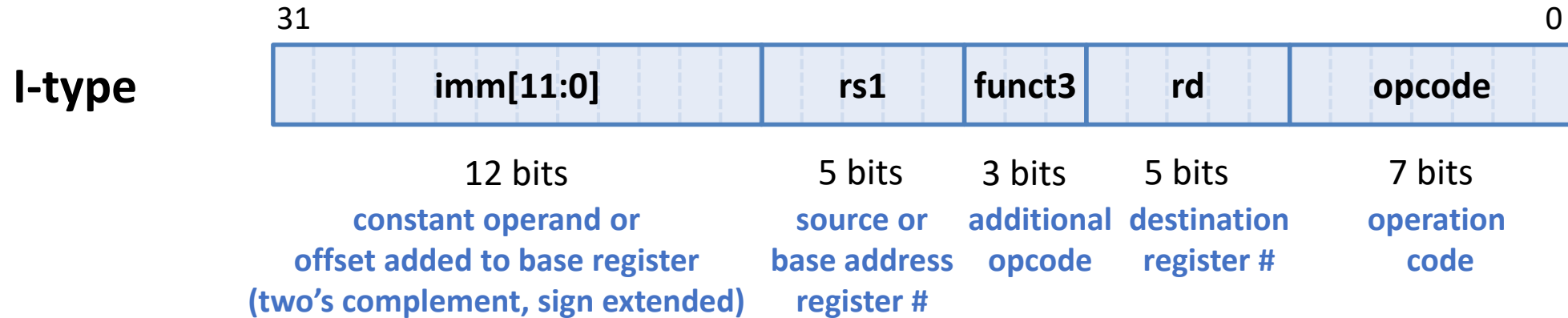
R-type Example



add x9, x20, x21 == 015A04B3₁₆



RISC-V I-type Instructions

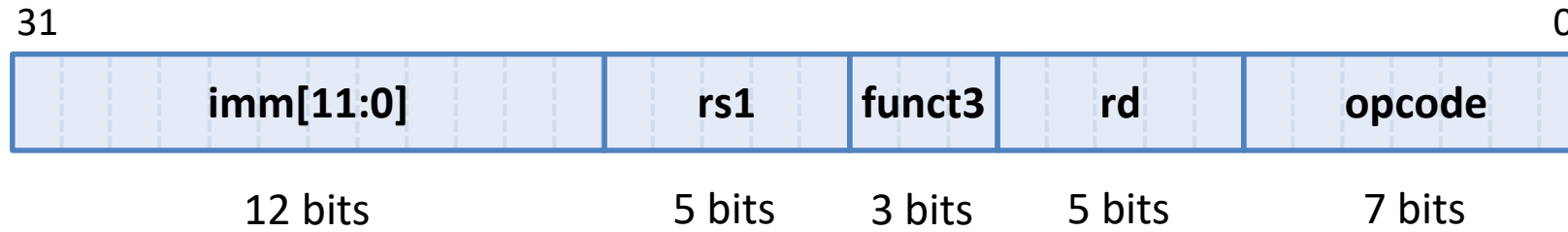


- Immediate arithmetic or load instructions
- **Design Principle 3: Good design demands good compromises**
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

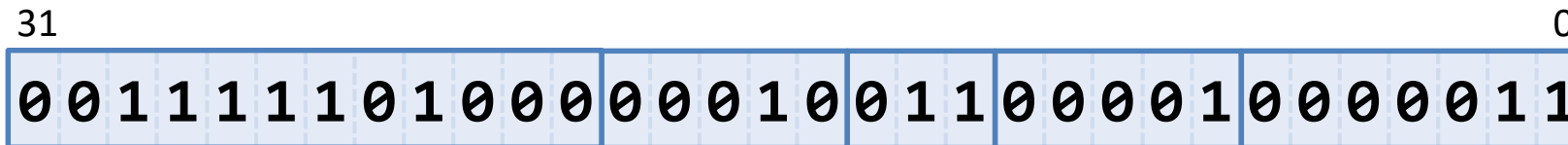
RISC-V I-type Instructions (cont'd)

Instruction	Type	Example	funct7		funct3	opcode
addi	I	addi rd, rs1, imm12	-		000	0010011
slti	I	slti rd, rs1, imm12	-		010	0010011
sltiu	I	sltiu rd, rs1, imm12	-		011	0010011
xori	I	xori rd, rs1, imm12	-		100	0010011
ori	I	ori rd, rs1, imm12	-		110	0010011
andi	I	andi rd, rs1, imm12	-		111	0010011
slli	I	slli rd, rs1, shamt	000000	shamt	001	0010011
srli	I	srli rd, rs1, shamt	000000	shamt	101	0010011
srai	I	srai rd, rs1, shamt	010000	shamt	101	0010011
lb	I	lb rd, imm12(rs1)	-		000	0000011
lh	I	lh rd, imm12(rs1)	-		001	0000011
lw	I	lw rd, imm12(rs1)	-		010	0000011
ld	I	ld rd, imm12(rs1)	-		011	0000011
lbu	I	lbu rd, imm12(rs1)	-		100	0000011
lhu	I	lhu rd, imm12(rs1)	-		101	0000011
lwu	I	lwu rd, imm12(rs1)	-		110	0000011
jalr	I	jalr rd, imm12(rs1)	-		000	1100111

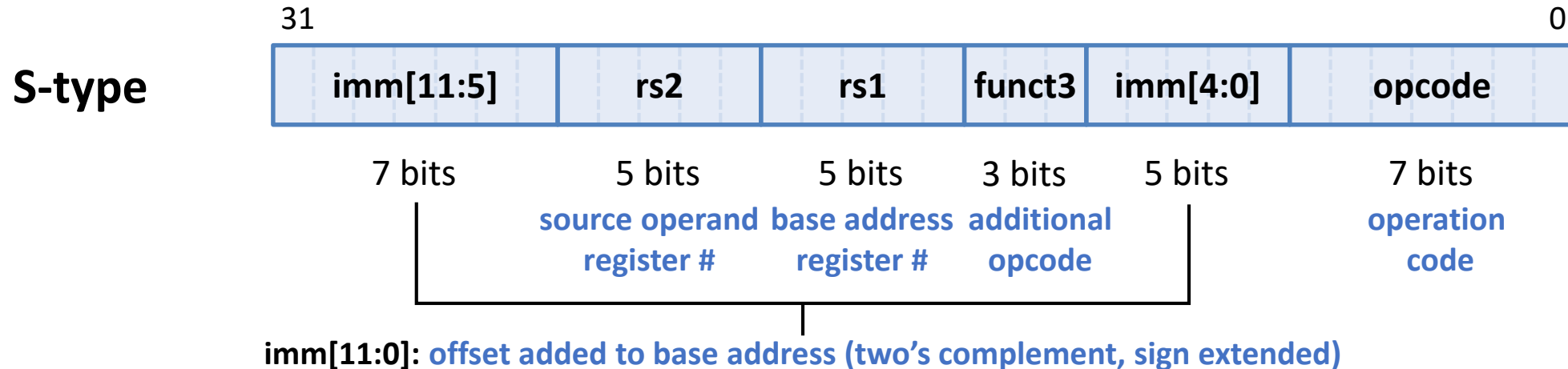
I-type Example



ld x1, 1000(x2) == 3E813083₁₆



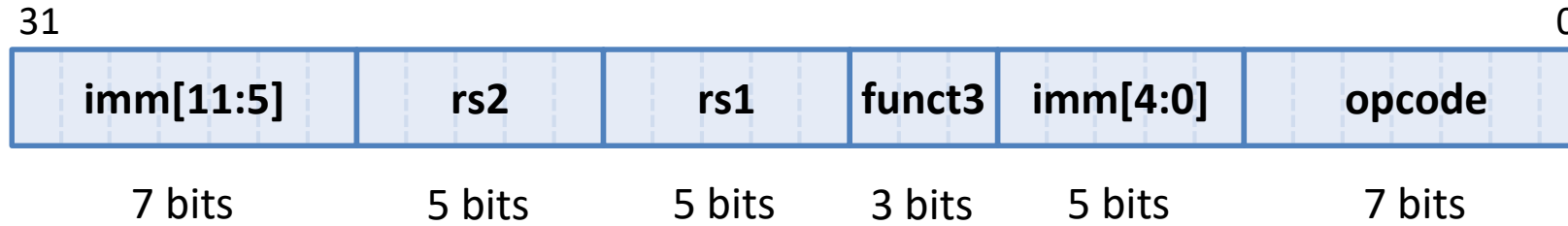
RISC-V S-type Instructions



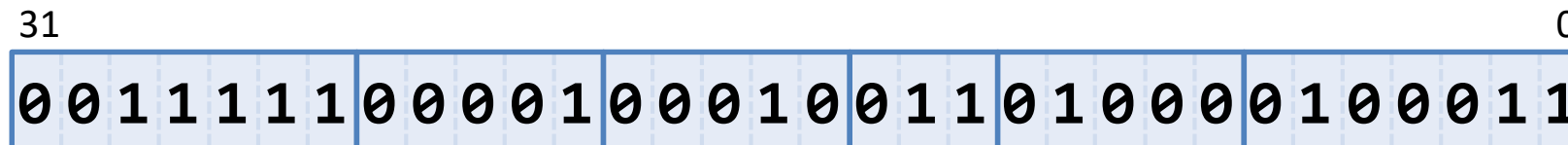
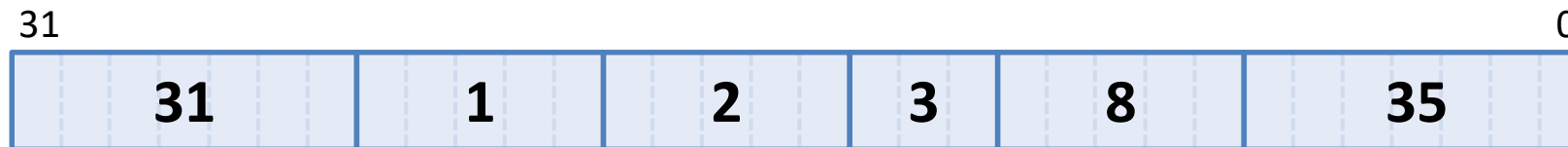
- Different immediate format for store instructions
 - Split so that rs1 and rs2 fields always in the same place

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
sd	S	sd rs2, imm12(rs1)	-	011	0100011

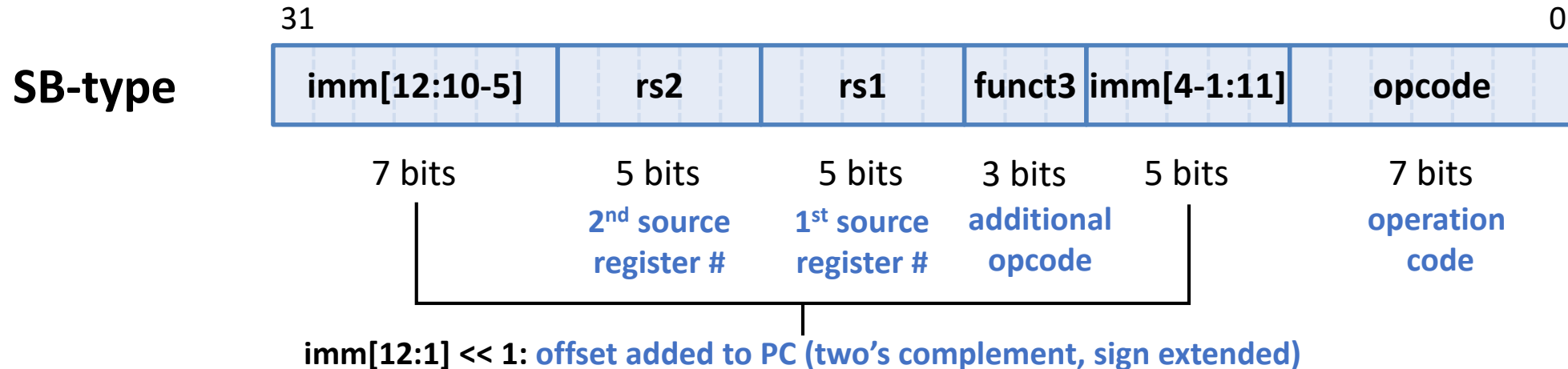
S-type Example



sd x1, 1000(x2) == 3E113423₁₆

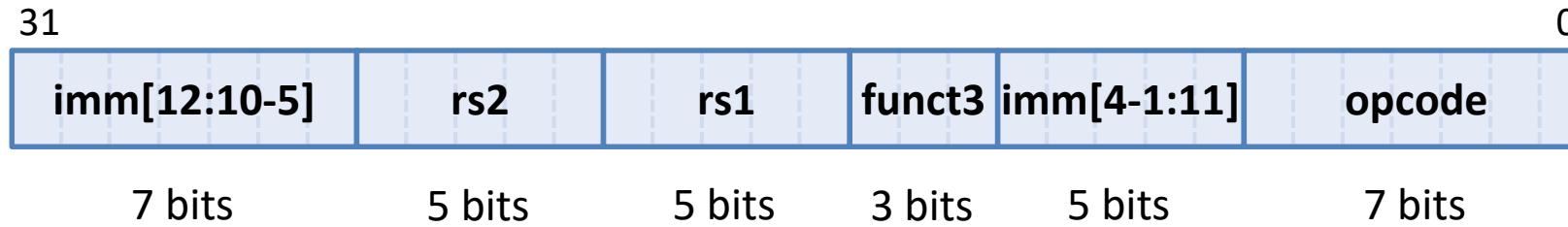


RISC-V SB-type Instructions

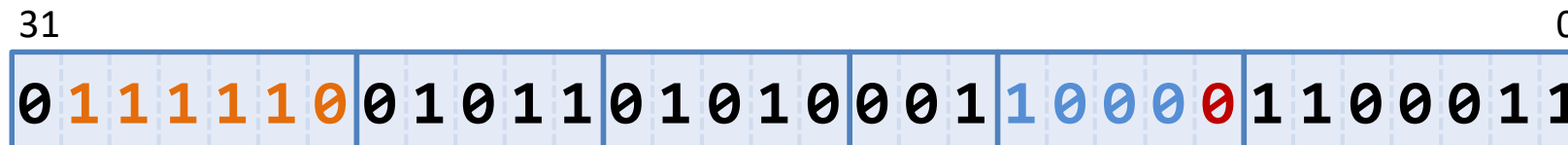


Instruction	Type	Example	funct7	funct3	opcode
beq	SB	beq rs1, rs2, imm12	-	000	1100011
bne	SB	bne rs1, rs2, imm12	-	001	1100011
blt	SB	blt rs1, rs2, imm12	-	100	1100011
bge	SB	bge rs1, rs2, imm12	-	101	1100011
bltu	SB	bltu rs1, rs2, imm12	-	110	1100011
bgeu	SB	bgeu rs1, rs2, imm12	-	111	1100011

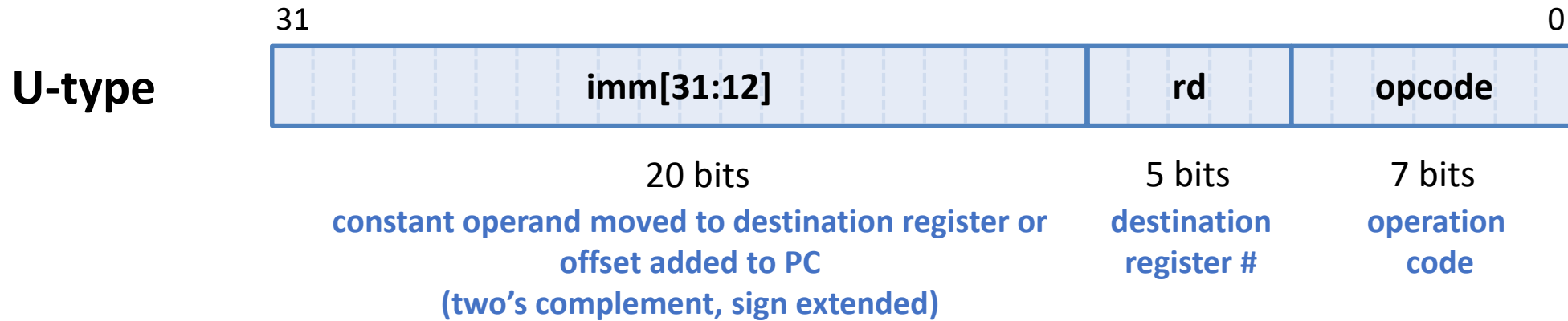
SB-type Example



bne x10, x11, 2000 == 7CB51863₁₆
 (0111 1101 0000₂)



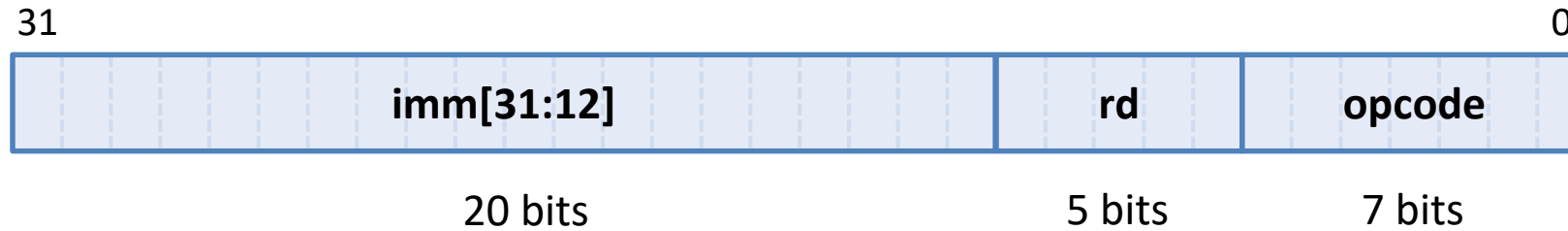
RISC-V U-type Instructions



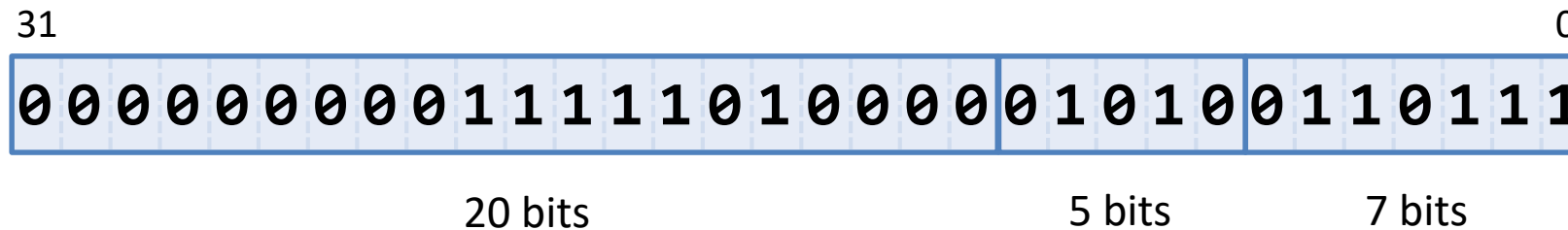
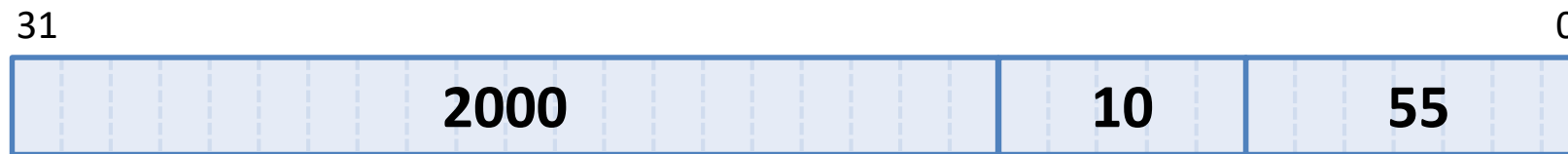
- 20-bit immediate is shifted left by 12 bits

Instruction	Type	Example	funct7	funct3	opcode
lui	U	lui rd, imm20	-	-	0110111
auipc	U	auipc rd, imm20	-	-	0010111

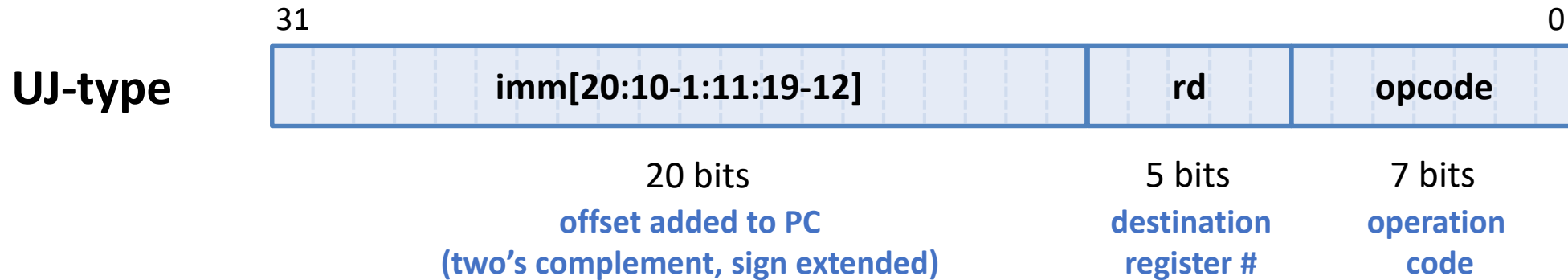
U-type Example



`lui x10, 2000 == 007D053716`
(0000 0000 0111 1101 0000₂)



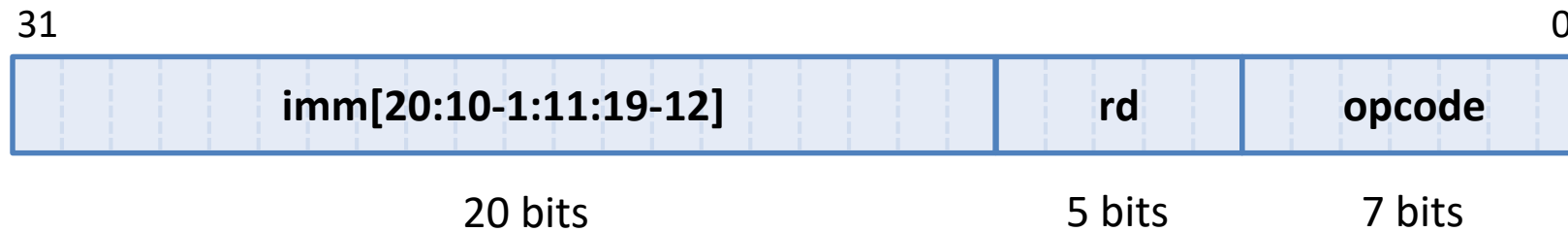
RISC-V UJ-type Instructions



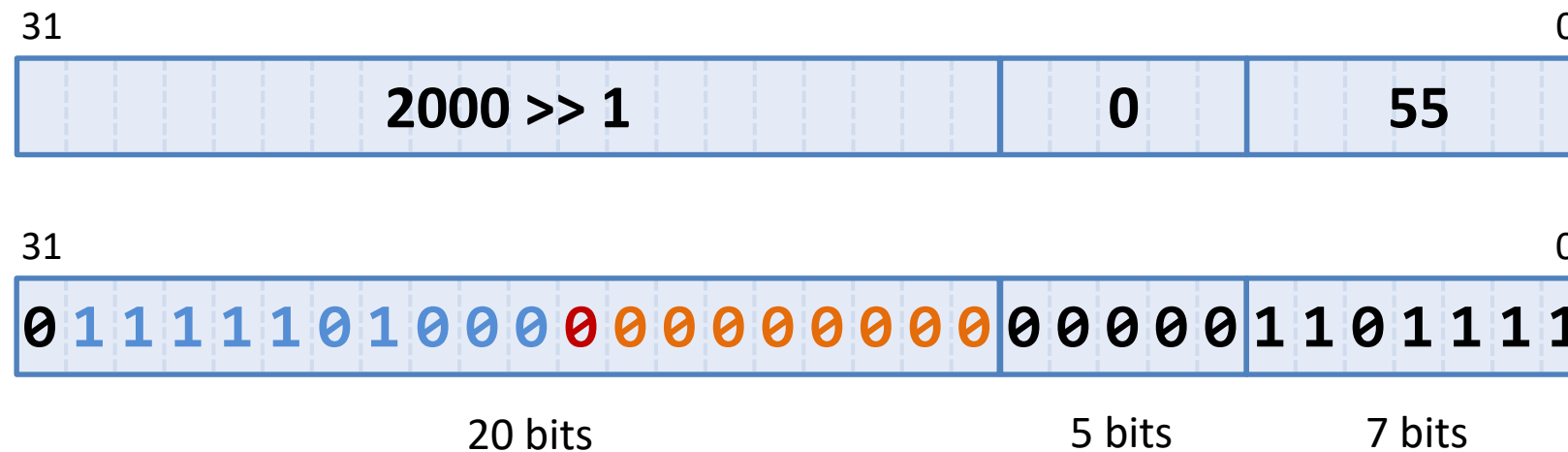
- 20-bit immediate is shifted left by 1 bit and added to PC

Instruction	Type	Example	funct7	funct3	opcode
jal	UJ	jal rd, imm20	-	-	1101111

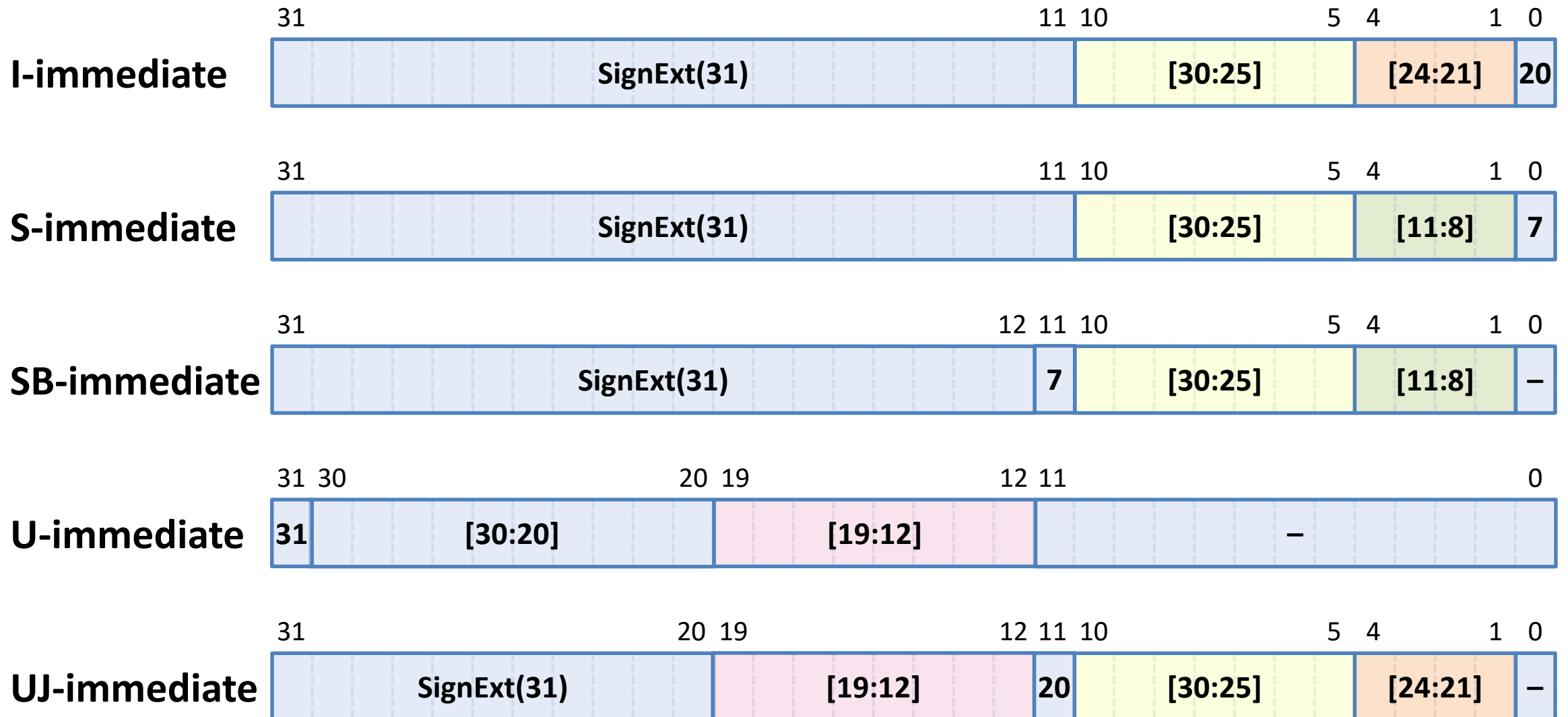
UJ-type Example



jal x0, 2000 == 7D00006F₁₆
 (0000 0000 0111 1101 0000₂)



RISC-V Immediate Values

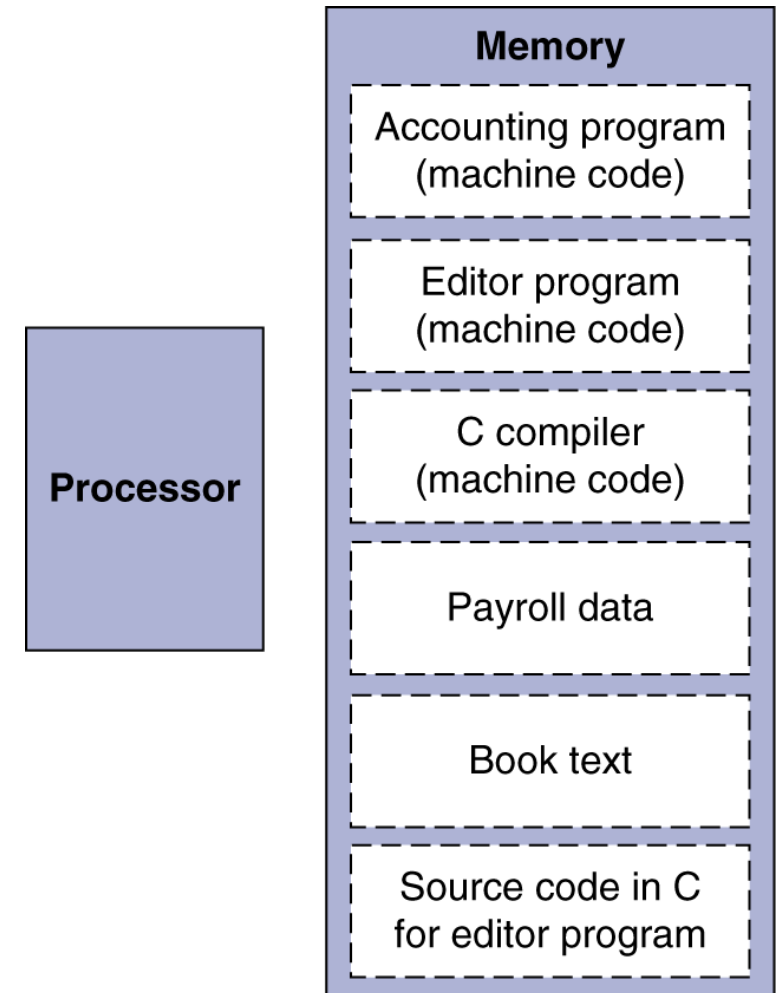


Translating and Starting a Program

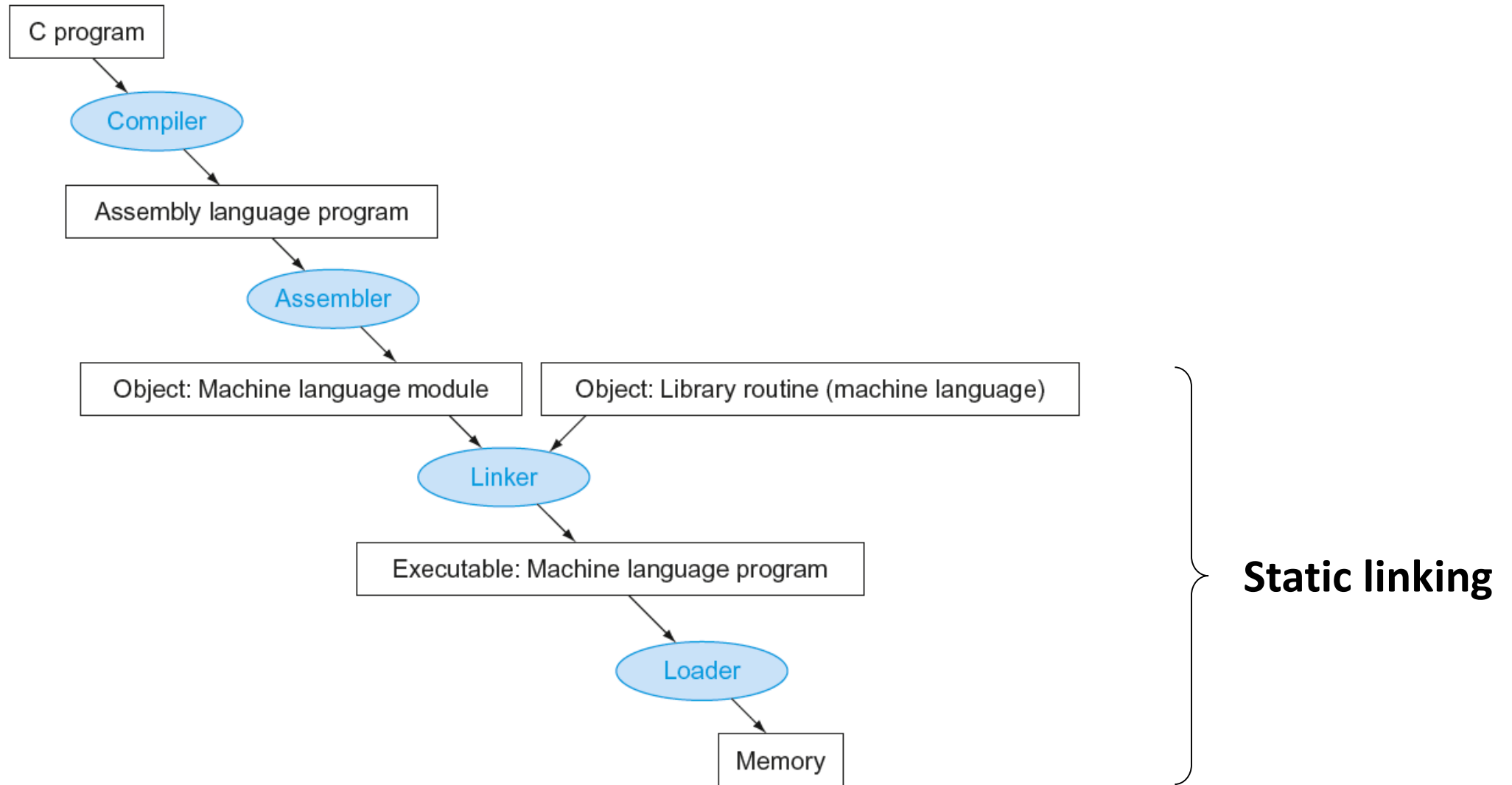
Chap. 2.12

Stored Program Computers

- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
 - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
 - Standardized ISAs

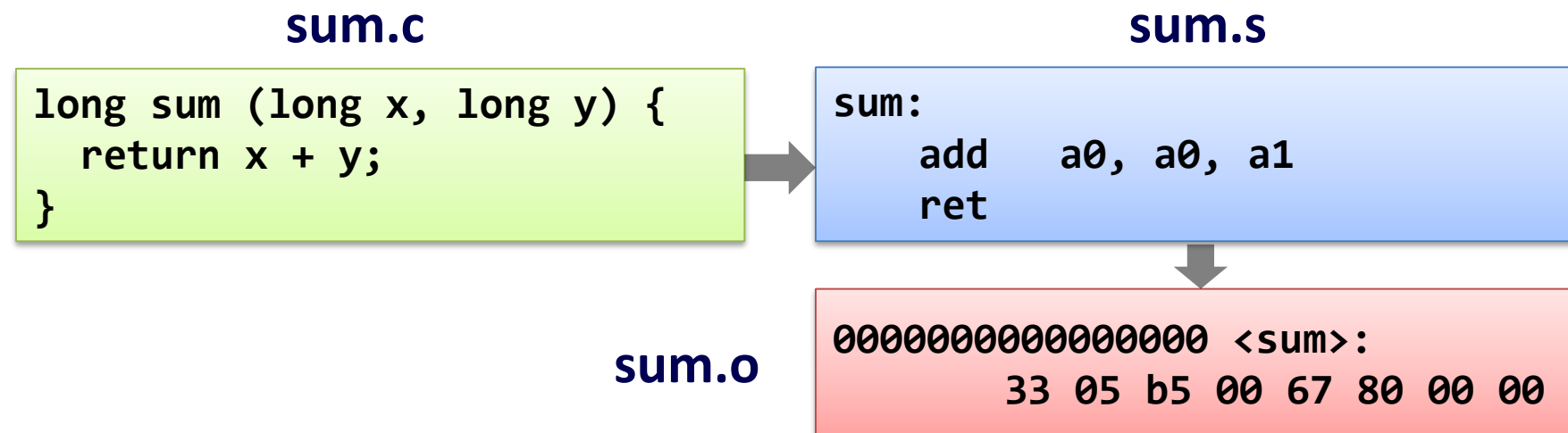


Translation and Startup



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
- `riscv64-unknown-elf-gcc -Og -S sum.c`



Producing an Object Module

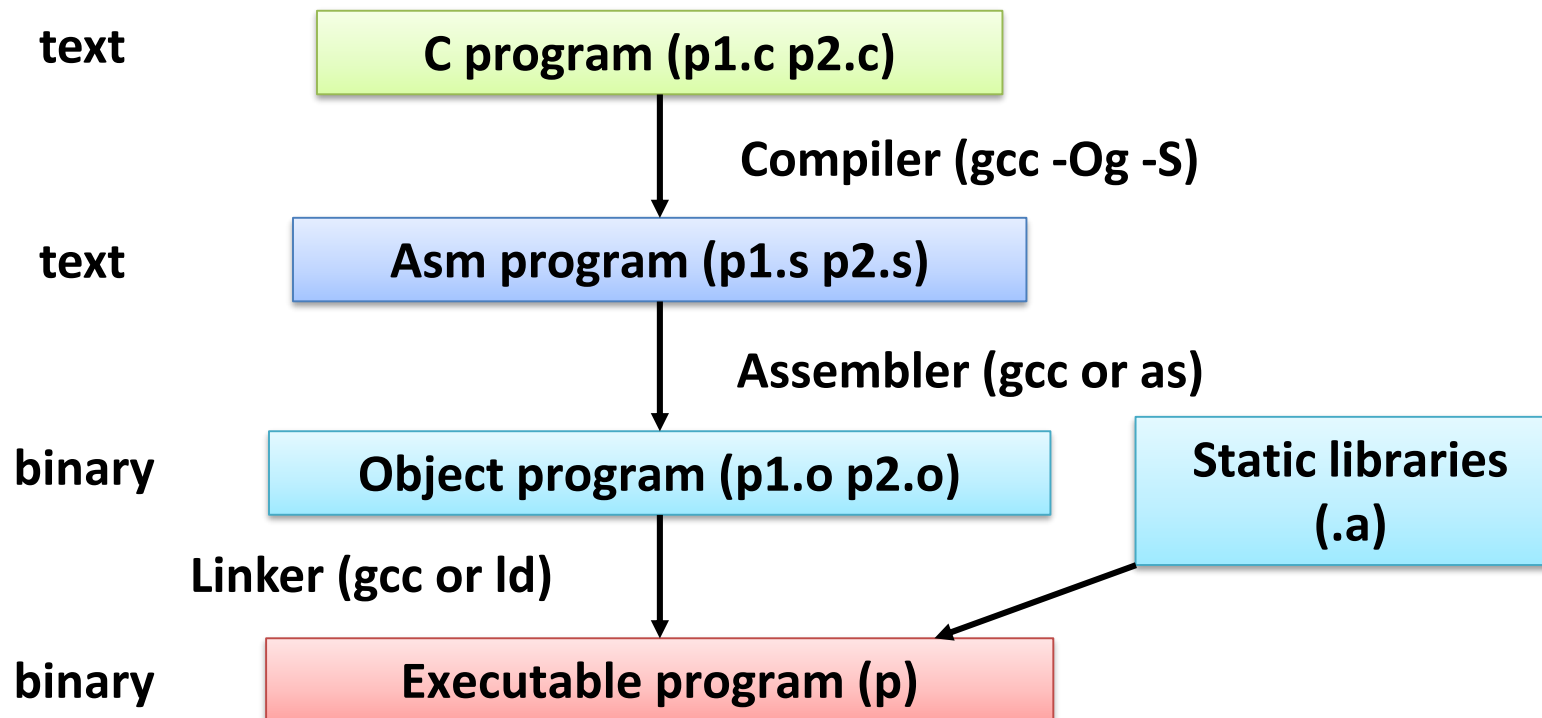
- Assembler (or compiler) translates program into machine instructions
- Nearly-complete image of executable code
- Missing linkages between code in different files
- Provides information for building a complete program from the pieces
 - Header: described contents of object module
 - Text segment: translated instructions
 - Static data segment: data allocated for the life of the program
 - Relocation info: for contents that depend on absolute location of loaded program
 - Symbol table: global definitions and external references
 - Debug info: for associating with source code

Linking Object Modules

- Produces an executable image
 - Merges segments
 - Resolve labels (determine their addresses)
 - Patch location-dependent and external references
 - Combines with static run-time libraries
 - Some libraries are dynamically linked: linking occurs when program begins execution
- Could leave location dependencies for fixing by a relocating loader
 - But with virtual memory, no need to do this
 - Program can be loaded into absolute location in virtual memory space

Turning C into Executable Program

- **riscv64-unknown-elf-gcc -Og p1.c p2.c -o p**
 - Use basic optimizations (-Og)
 - Put resulting binary in file p



Loading a Program

- Create a process
- Load from image file on disk into memory
 - Read header to determine segment sizes
 - Copy text and initialized data into memory
 - Set up stack
 - Initialize registers (including sp, fp, gp)
- Jump to startup routine
 - Copies arguments to x10, ... and calls main
 - When main returns, do exit syscall

Memory Layout

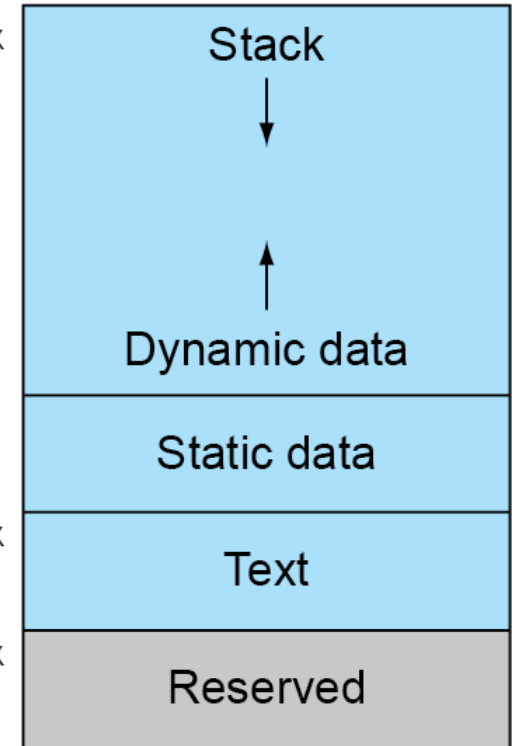
- **Text:** program code
- **(Static) Data:** global variables
 - e.g., static variables in C, constant arrays and strings
 - x3 (global pointer or gp) initialized to address allowing $\pm$ offsets into this segment
- **Heap:** dynamic data
 - e.g., malloc in C, new in Java
- **Stack:** automatic storage

SP $\rightarrow$ 0000 003f ffff fff0_{hex}

0000 0000 1000 0000_{hex}

PC $\rightarrow$ 0000 0000 0040 0000_{hex}

0



Disassembling

- Disassembler: **riscv64-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 elf64-littleriscv

Disassembly of section .text:

0000000000000000 <sum>:
   0:    00b50533          add     a0,a0,a1
   4:    00008067          ret
```

Disassembling with gdb

- Disassemble procedure `sum`

```
$ riscv64-unknown-elf-gcc -Og -g sum.c main.c
$ riscv64-unknown-elf-gdb a.out
(gdb) disassemble sum
Dump of assembler code for function sum:
   0x00000000000010164 <+0>:      add      a0,a0,a1
   0x00000000000010168 <+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
```