Machine-level Representation of Programs

Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software & Architecture Lab.

Seoul National University

Fall 2021

RISC-V:
Representing Instructions

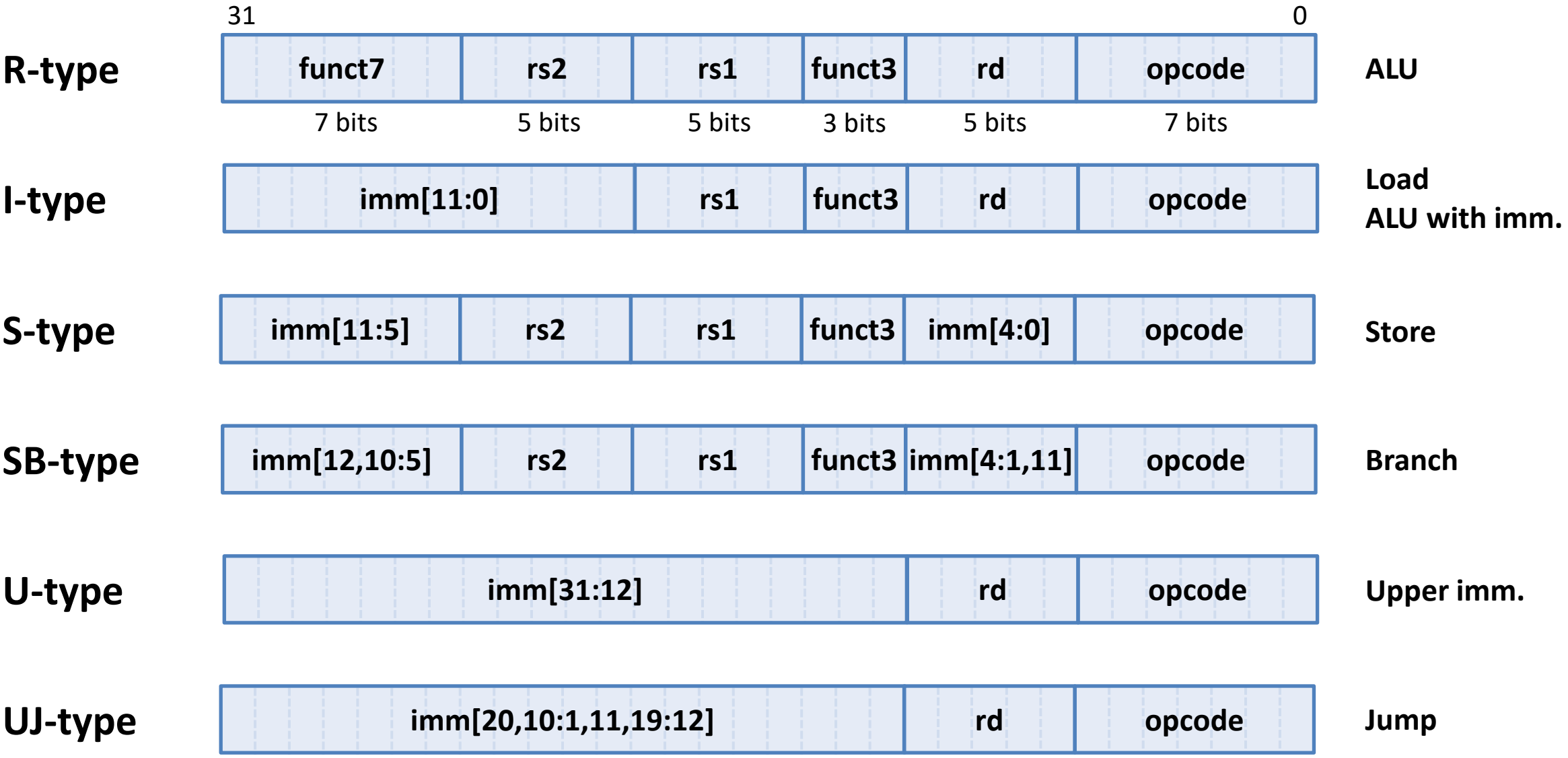
Chap. 2.5

Representing Instructions

- Instructions are encoded in binary
  - Called machine code

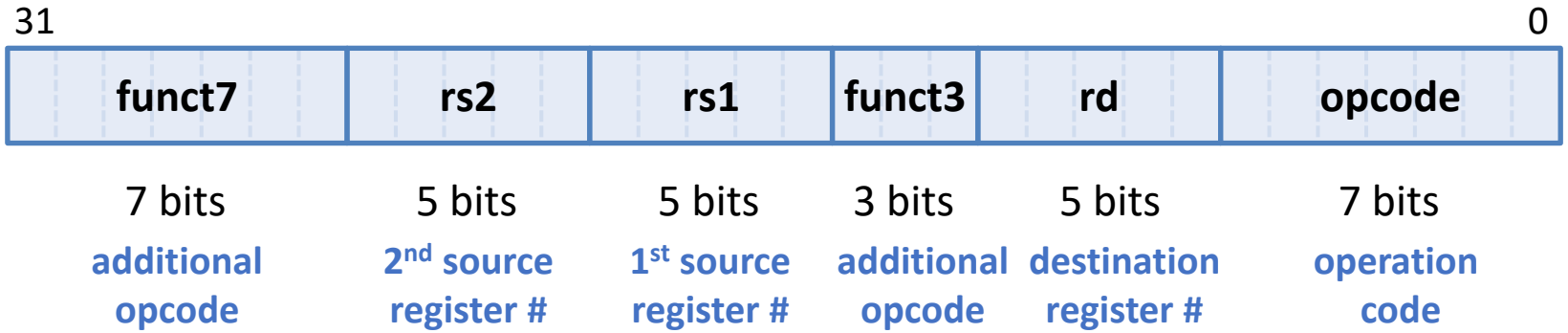
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

RISC-V Instruction Formats



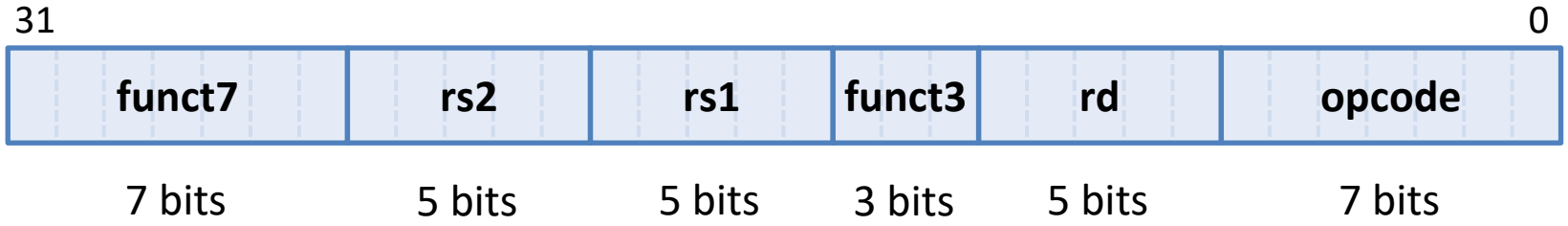
RISC-V R-type Instructions

R-type

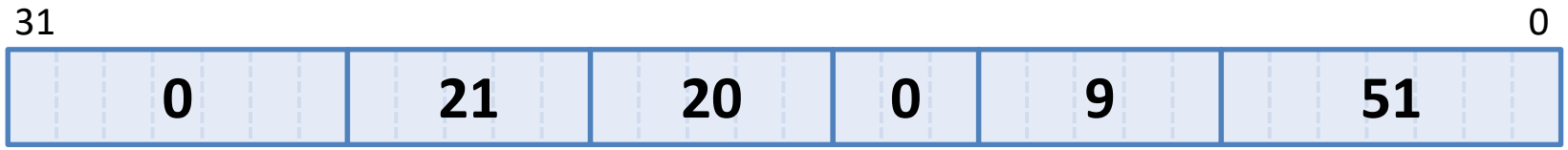


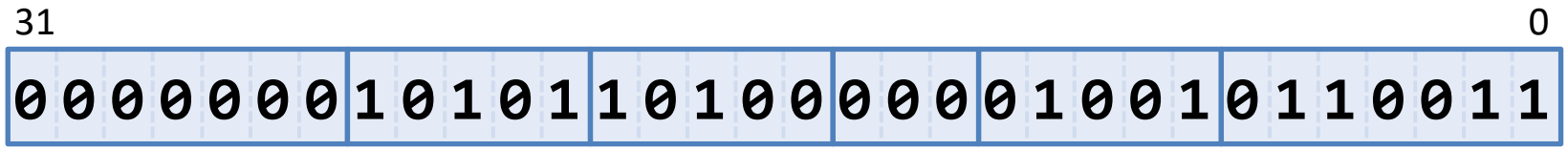
Instruction	Type	Example	funct7	funct3	opcode
add	**R**	`add rd, rs1, rs2`	0000000	000	0110011
sub	**R**	`sub rd, rs1, rs2`	0100000	000	0110011
sll	**R**	`sll rd, rs1, rs2`	0000000	001	0110011
slt	**R**	`slt rd, rs1, rs2`	0000000	010	0110011
sltu	**R**	`sltu rd, rs1, rs2`	0000000	011	0110011
xor	**R**	`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

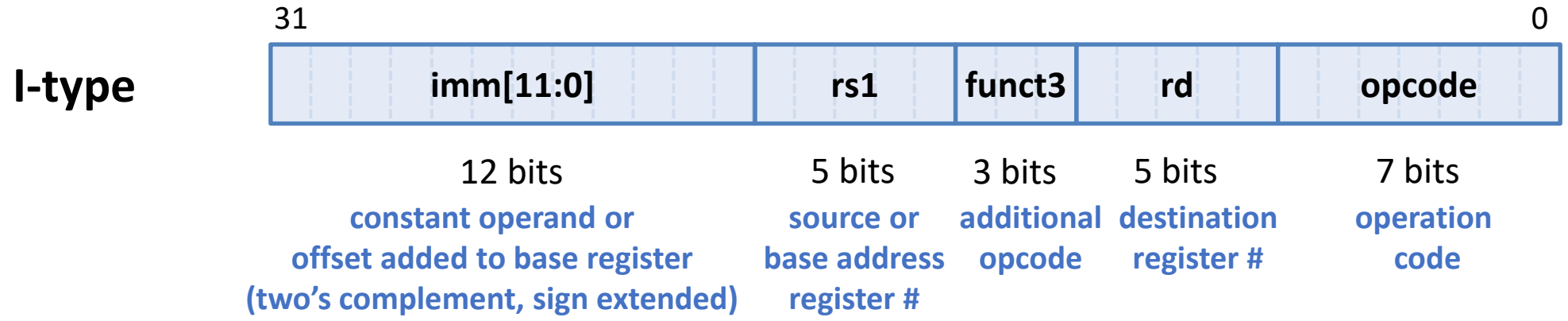


`add x9, x20, x21` == $015A04B3_{16}$





RISC-V I-type Instructions

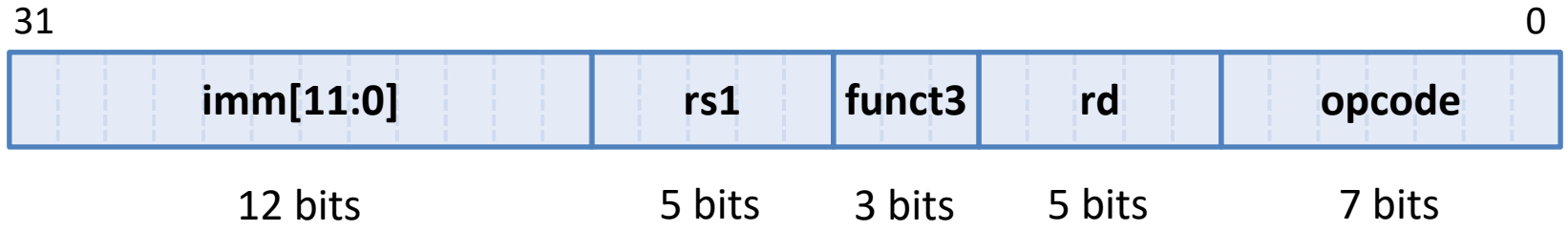


- Immediate arithmetic or load instructions
- Design Principle 3: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

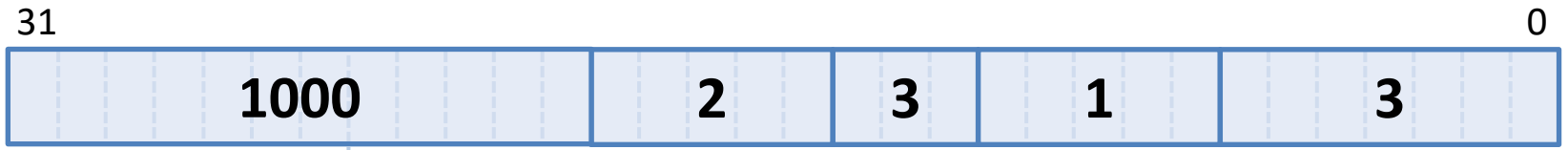
RISC-V I-type Instructions (cont'd)

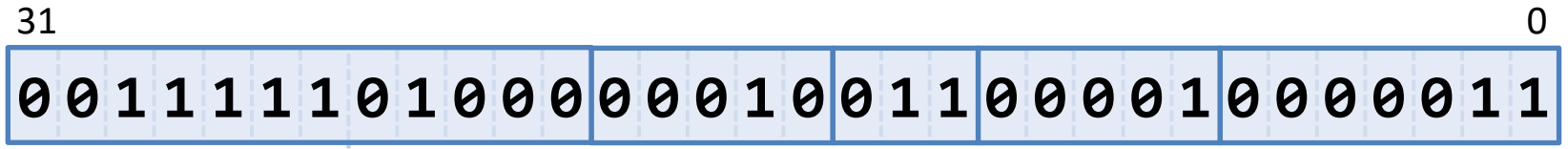
Instruction	Type	Example	funct7	funct3	opcode
addi	I	addi rd, rs1, imm12	-	000	0010011
slti	I	slti rd, rs1, imm12	-	010	0010011
sltiu	I	sltiu rd, rs1, imm12	-	011	0010011
xori	I	xori rd, rs1, imm12	-	100	0010011
ori	I	ori rd, rs1, imm12	-	110	0010011
andi	I	andi rd, rs1, imm12	-	111	0010011
slli	I	slli 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

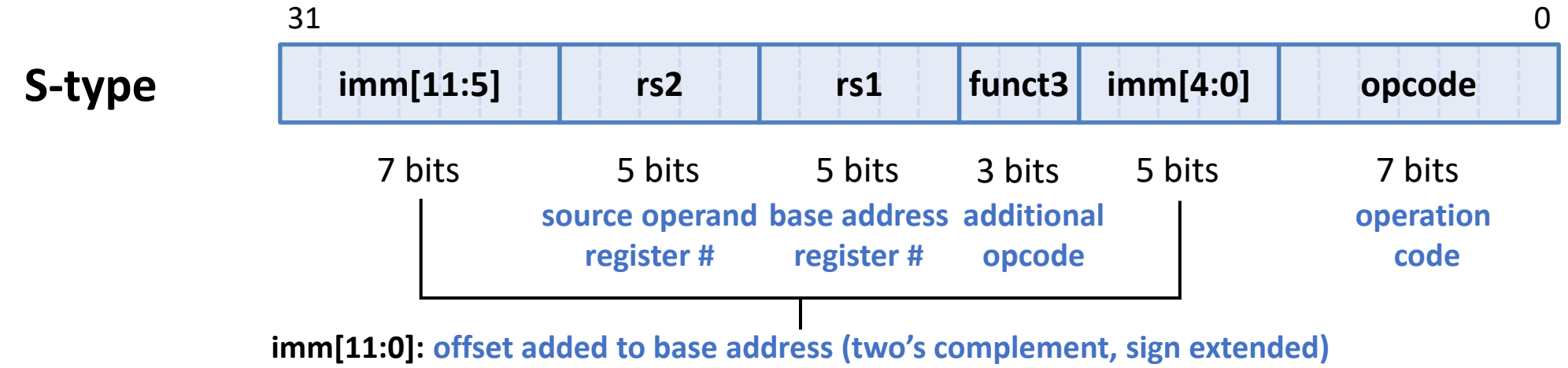


`ld x1, 1000(x2)` == $3E813083_{16}$





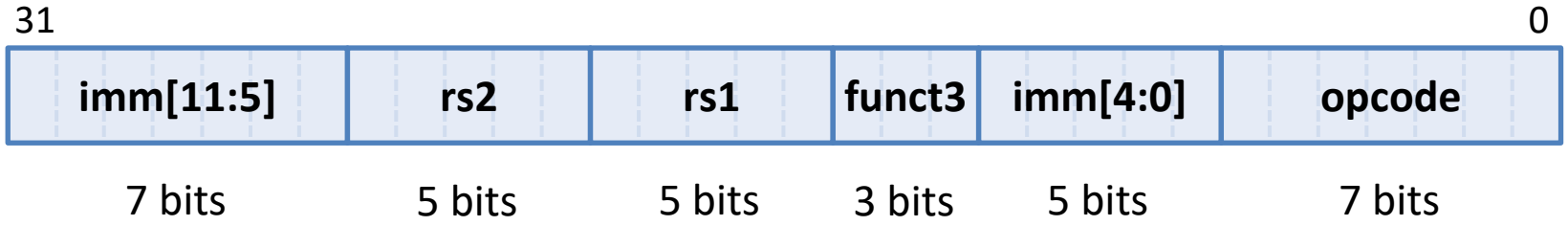
RISC-V S-type Instructions



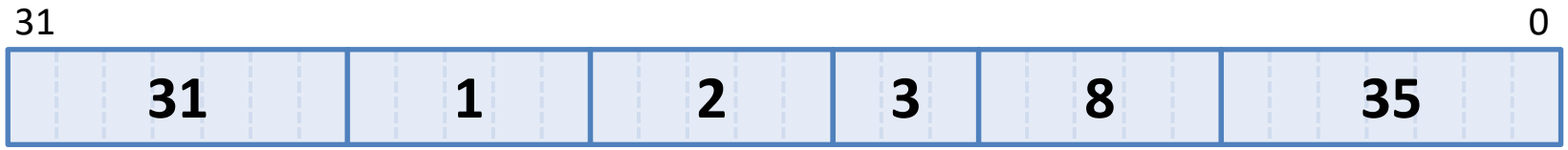
- Different immediate format for store instructions
  - Split so that rs1 and rs2 fields always in the same place

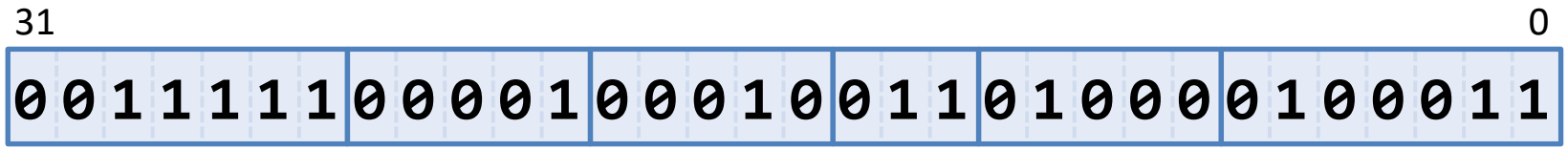
Instruction	Type	Example	funct7	funct3	opcode
sb	**S**	`sb rs2, imm12(rs1)`	-	000	0100011
sh	**S**	`sh rs2, imm12(rs1)`	-	001	0100011
sw	**S**	`sw rs2, imm12(rs1)`	-	010	0100011
sd	**S**	`sd rs2, imm12(rs1)`	-	011	0100011

S-type Example

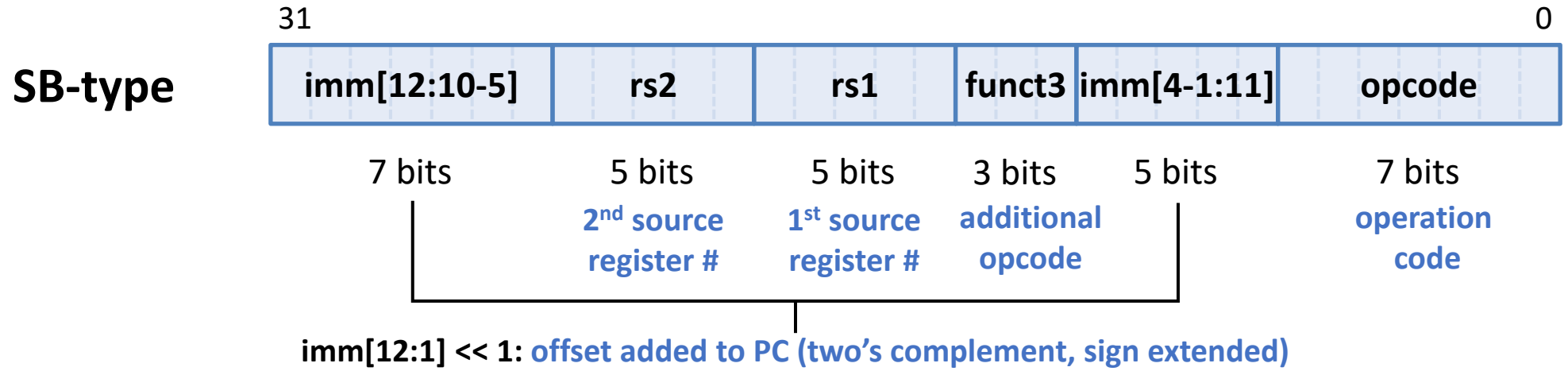


`sd x1, 1000(x2)` == $3E113423_{16}$



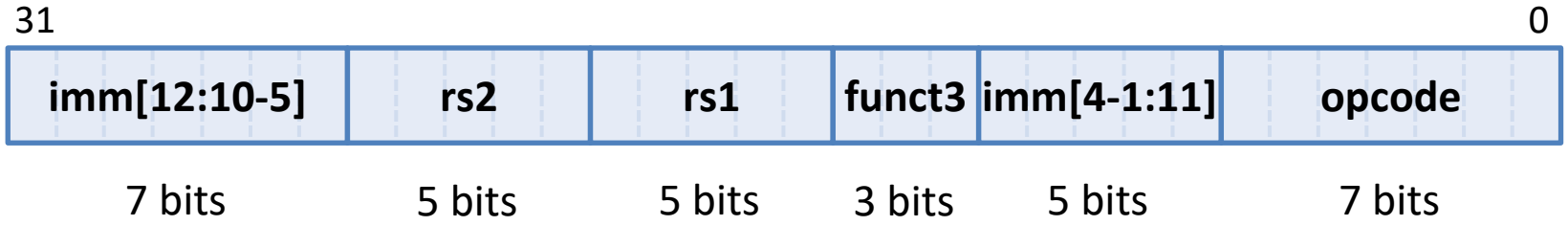


RISC-V SB-type Instructions



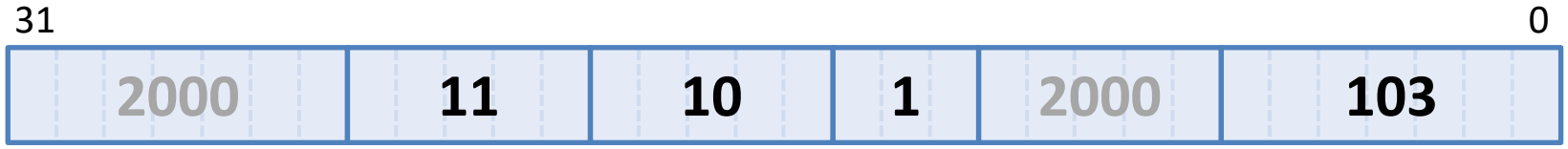
Instruction	Type	Example	funct7	funct3	opcode
beq	**SB**	`beq rs1, rs2, imm12`	-	000	1100011
bne	**SB**	`bne rs1, rs2, imm12`	-	001	1100011
blt	**SB**	`blt rs1, rs2, imm12`	-	100	1100011
bge	**SB**	`bge rs1, rs2, imm12`	-	101	1100011
bltu	**SB**	`bltu rs1, rs2, imm12`	-	110	1100011
bgeu	**SB**	`bgeu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

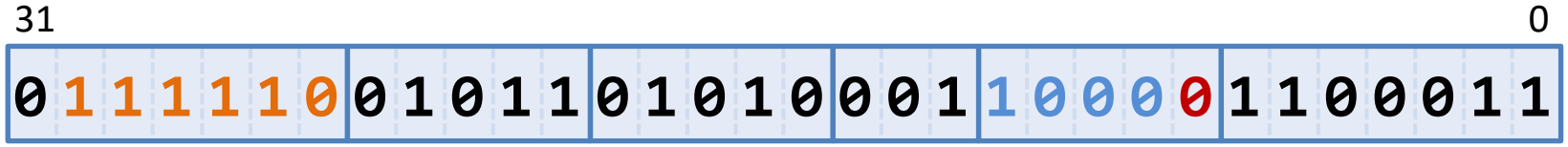
SB-type Example



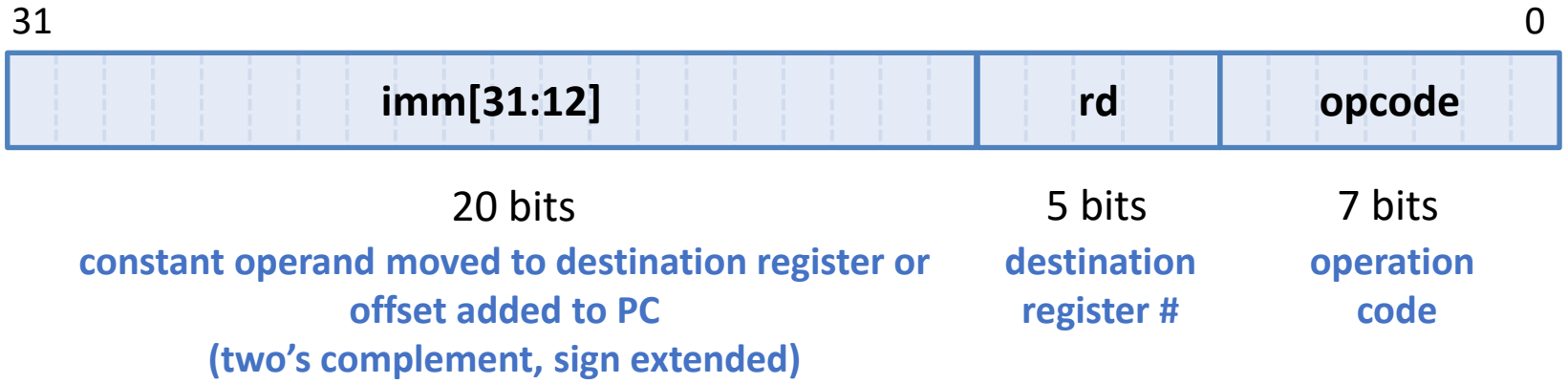
bne x10, x11, 2000 == $7CB51863_{16}$

(0111 1101 0000$_2$)





RISC-V U-type Instructions



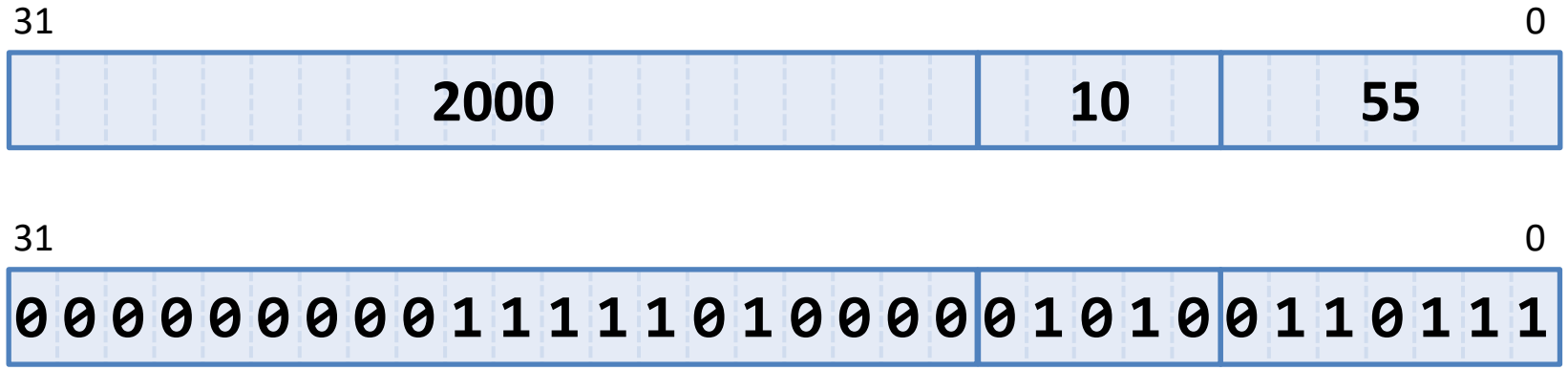
- 20-bit immediate is shifted left by 12 bits

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

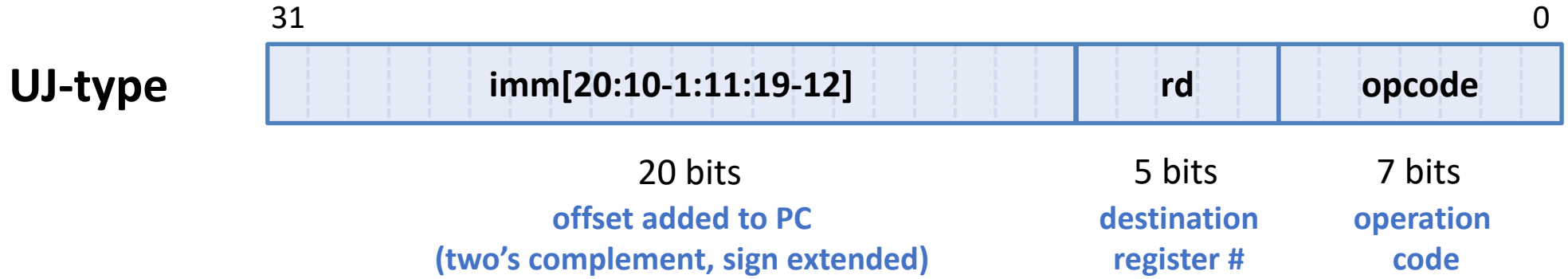
U-type Example

`lui x10, 2000` == $007D0537_{16}$

($0000\ 0000\ 0111\ 1101\ 0000_2$)



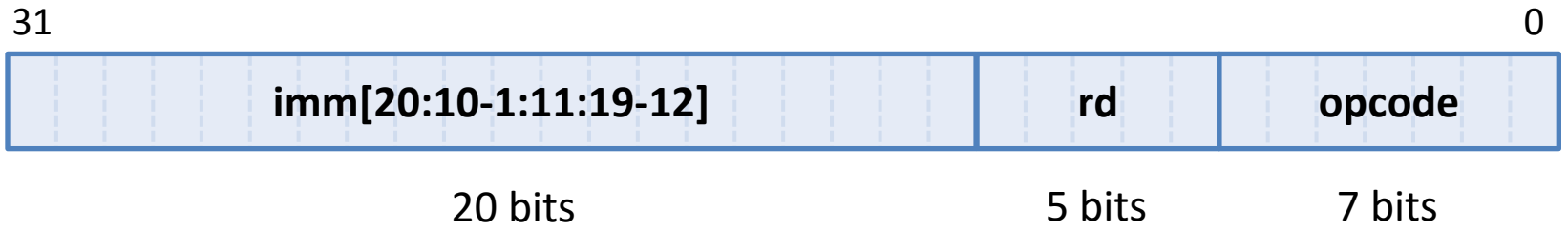
RISC-V UJ-type Instructions



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

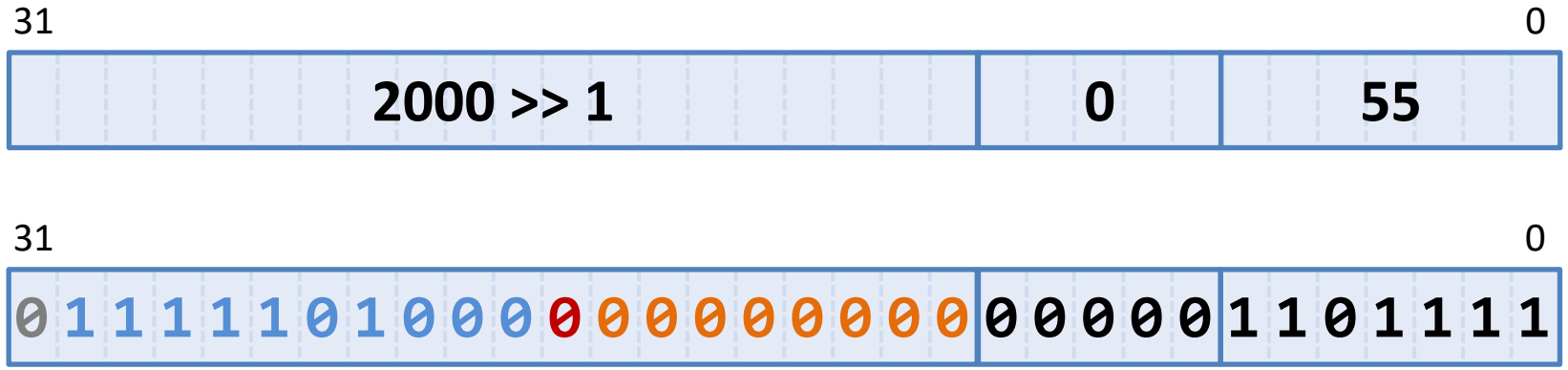
Instruction	Type	Example	funct7	funct3	opcode
jal	**UJ**	`jal  rd, imm20`	-	-	1101111

UJ-type Example

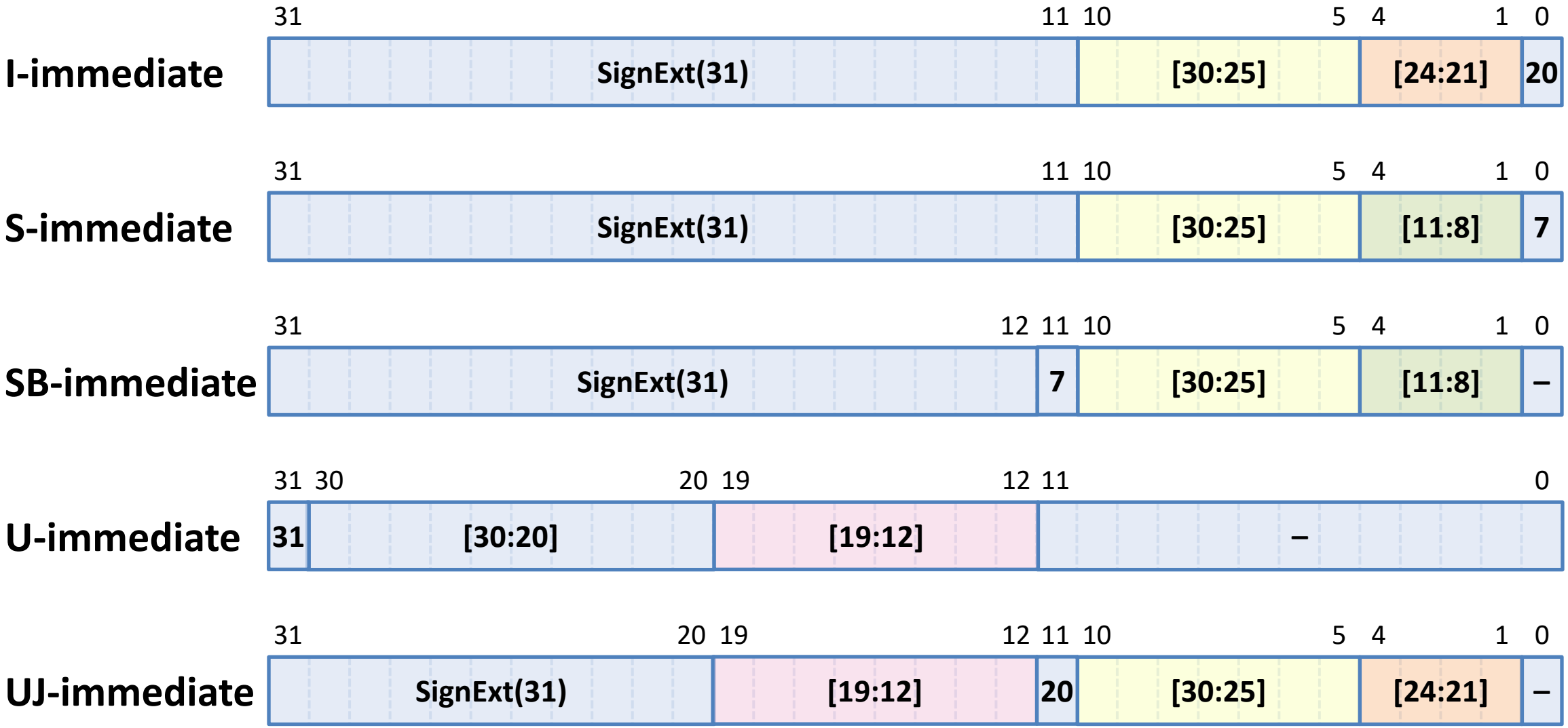


`jal x0, 2000` == $7D00006F_{16}$

(0000 0000 0111 1101 0000$_2$)



RISC-V Immediate Values



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:
   0x0000000000010164 <+0>:     add     a0,a0,a1
   0x0000000000010168 <+4>:     ret
End of assembler dump.
```

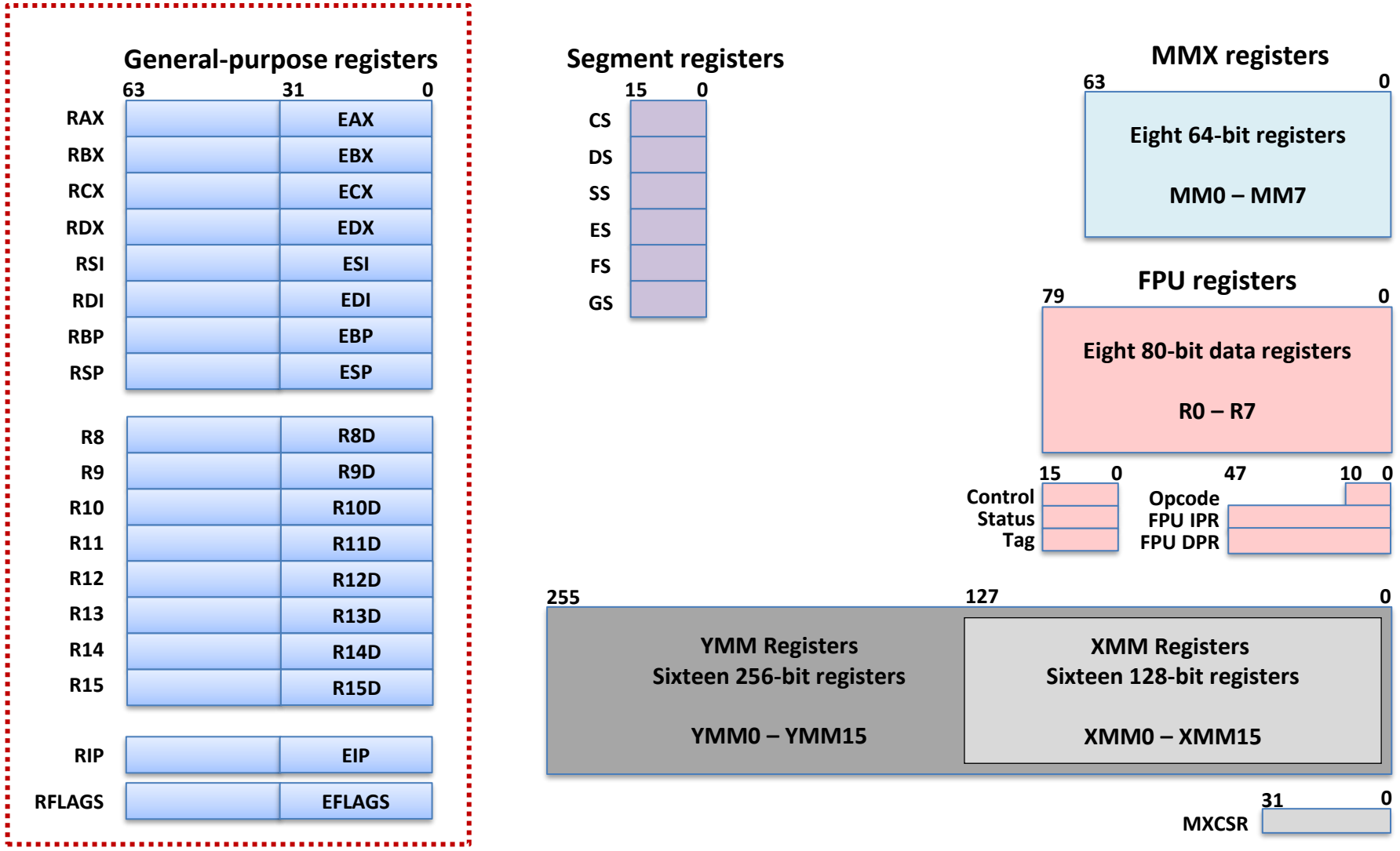
- Examine 8 bytes starting at `sum`

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

Intel x86-64

Chap. 2.19

x86 Basic Execution Environment



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	addq %rcx, %rax
Immediate	Register	addq $4, %eax
Memory	Register	addq 0(%rcx), %rax
Register	Memory	addq %rax, 8(%rax)
Immediate	Memory	addq $4, 8(%rax)

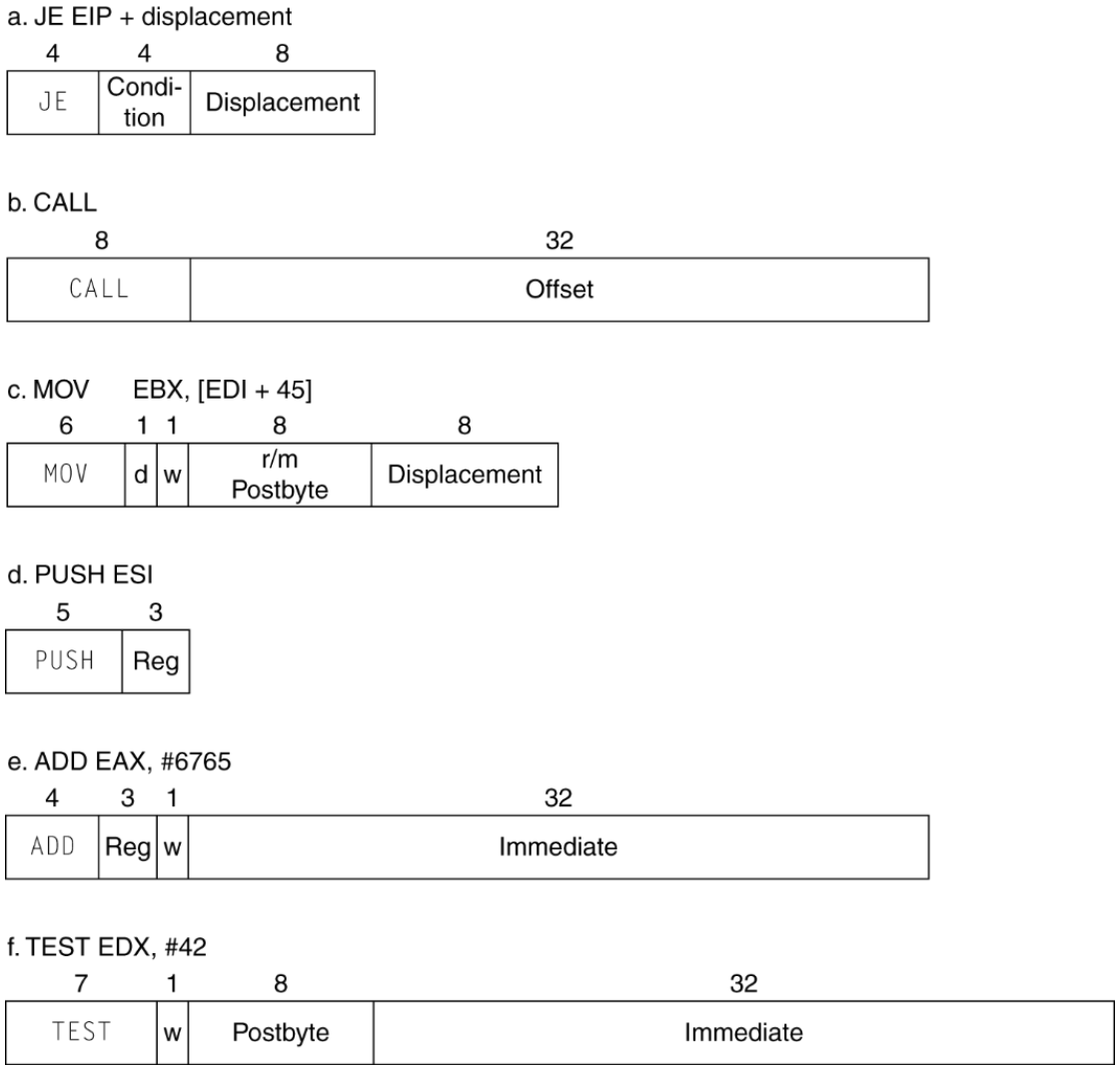
x86 Memory Addressing Modes

- **`D(Rb,Ri,S)`** Mem[Reg[Rb]+S*Reg[Ri]+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
`movq %rax, 8(%rdx)`	`%rdx + 8`
`movq %rax, 16(%rdx, %rcx)`	`%rdx + %rcx + 16`
`movq %rax, (,%rcx, 4)`	`%rcx * 4`
`movq %rax, 32(%rdx, %rcx, 8)`	`%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



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 μops
  - 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