Introduction to Computer Architecture
Chapter 2

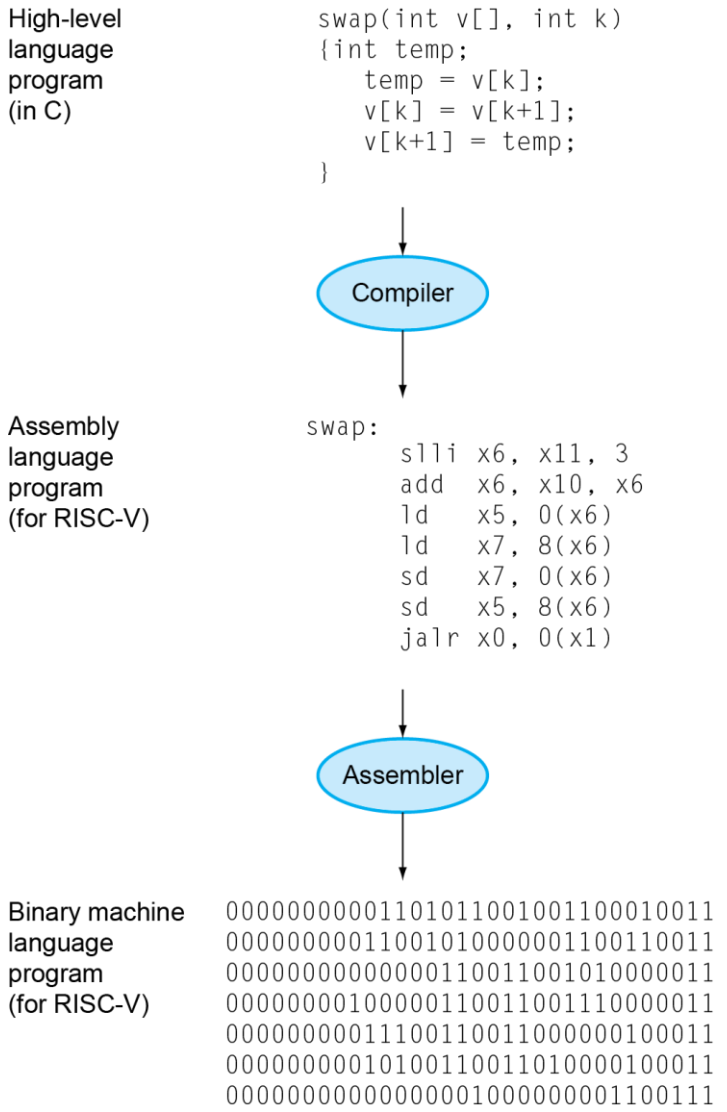
Instructions: Language of the Computer - 1

Hyungmin Cho

Department of Computer Science and Engineering
Sungkyunkwan University

Levels of Program Code

- High-level code
  - Closer to the problem domain
  - Better productivity & portability
- Assembly code
  - Human-readable representation of the machine code
  - Can use symbolic names (labels)
- Machine code
  - Instructions encoded in binary format
  - No symbols



Instruction Set

- The collection of instructions of a computer
  - Arithmetic operations (**`add`**, **`sub`**, …)
  - Logic operations (**`and`**, **`or`**, **`xor`**, …)
  - Multiplication / Division
  - Load from memory / Store to memory
  - Jump / branch

  - Exception handling
  - Floating point

Chapter 2

- We will learn the RISC-V ISA
  - Understand basic instructions in RISC-V ISA
  - Writing simple programs in RISC-V assembly language

  - Binary representations of the instructions (machine code)

The RISC-V Instruction Set

- Used as the textbook's primary example for explaining the CPU architecture

- Developed by UC Berkeley as an **open ISA**
- Currently managed by RISC-V International (riscv.org)

- RISC-V simulators
  - Spike
    - https://github.com/riscv-software-src/riscv-isa-sim
  - RARS Simulator
    - https://github.com/TheThirdOne/rars
  - Complete list of simulators supporting RISC-V ISA:
    - https://riscv.org/exchange/software/

Arithmetic Operations

- Add and subtract operations have three operands
  - One destination and two sources

```
add a, b, c    # a ← b + c
```

- All arithmetic operations follow this form

Arithmetic Operation Example

- Suppose you have "**add**" and "**sub**" instructions with two operands.

- Example C code:

```
f = (g + h) - (i + j); // 4 operands
```

- Corresponding RISC-V assembly code:

```
add t0, g, h   # t0 = g + h   // temporary
add t1, i, j   # t1 = i + j   // temporary
sub f, t0, t1  # f = t0 - t1
```

Registers

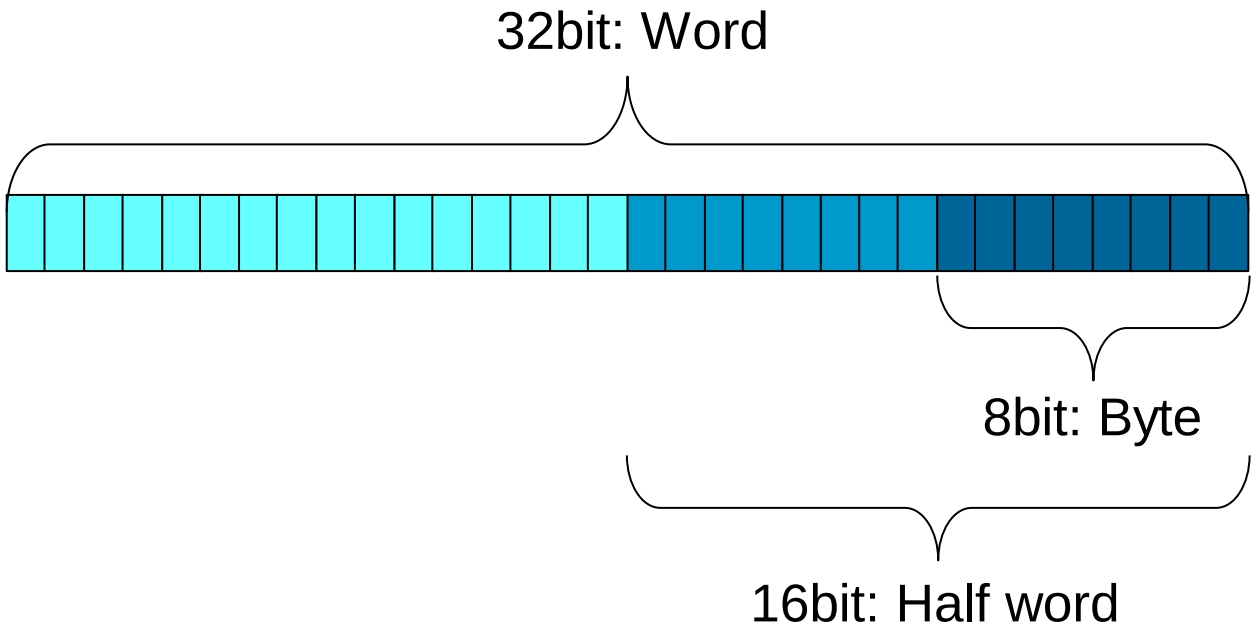
- Small, high-speed storage inside the CPU
- Registers hold data during the execution
  - Arithmetic instructions use register operands

- RISC-V (**RV32I**) ISA has a 32 registers, and the size of each register is 32bits.
  - The collection of those 32 registers is called “register file”
  - Each register has its own number (name): Numbered from `x0` to `x31`
  - A 32-bit value is referred to as a “word”

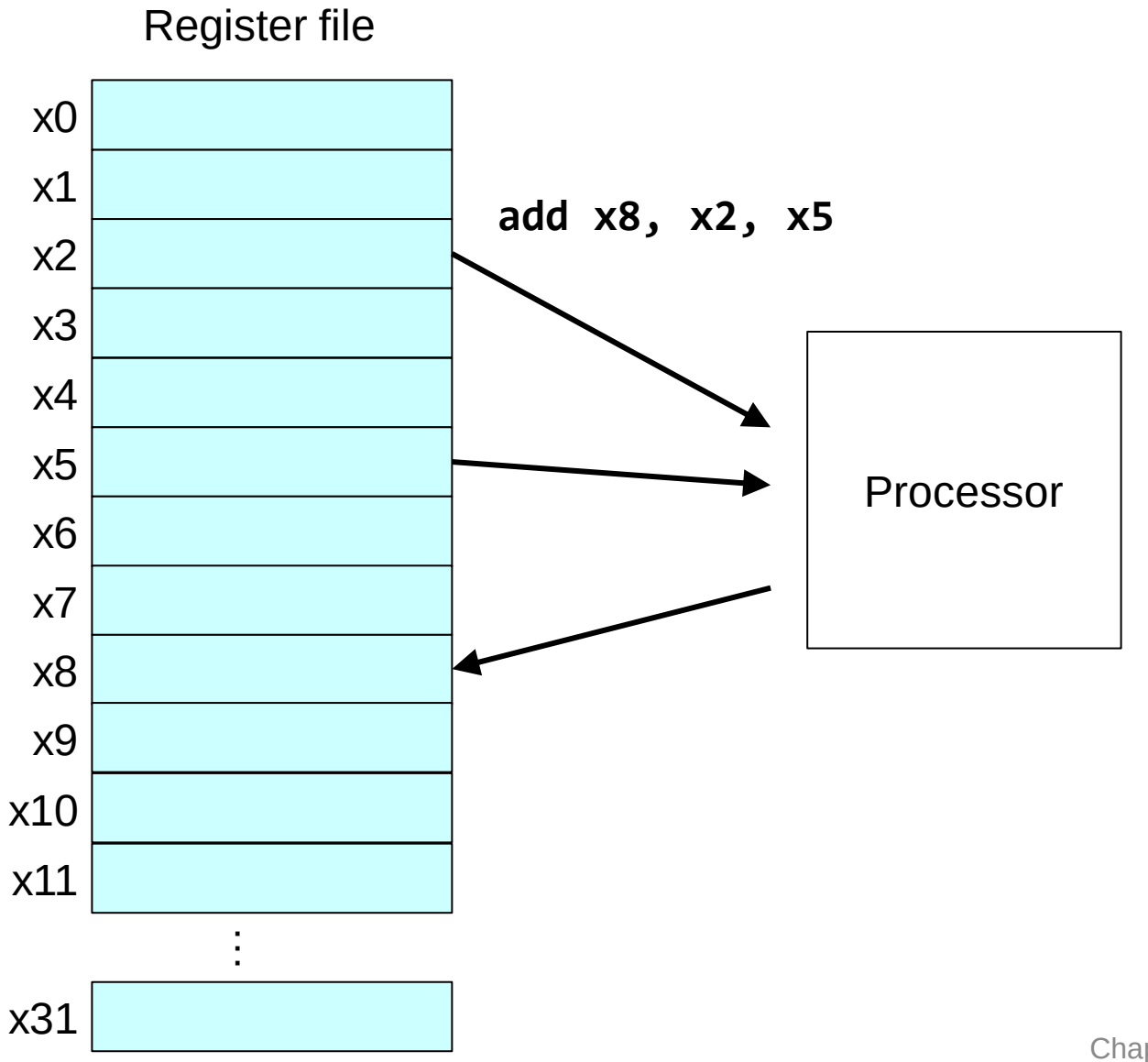
  - Please note that the textbook is based on the 64-bit version (RV64I)
  - This lecture will be using the **32-bit RV32I**, not the 64-bit version

Registers

- RISC-V (**RV32I**) : 32 x 32-bit registers



RISC-V Registers



Memory

- Can’t fit the entire data in registers
- Memory is the main storage for program data
- To apply arithmetic operations on data in memory…
  - Load: memory → registers
  - Compute: (source) register → (destination) register
  - Store: register → memory
- Memory is byte **address**ed
  - Each address: 1 byte (8-bit)

Register vs. Memory

- Registers are much faster than memory

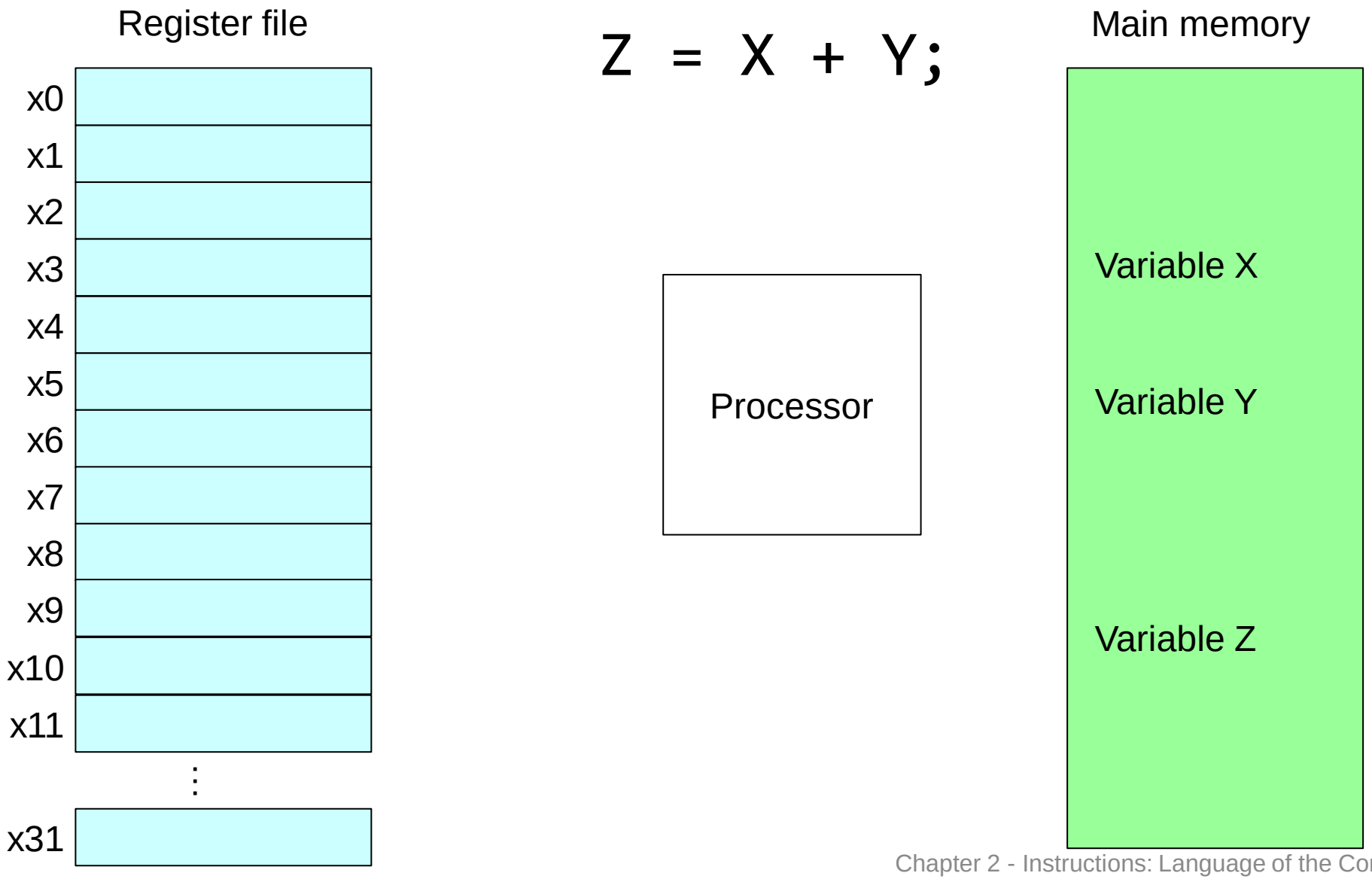
- Can’t perform computations directly on the data within memory
  - Need Load → Compute → Store steps
  - If need multiple operations, Load → Compute 1 → Compute 2 → … → Store

  - Note: while some CISC-style CPUs support memory operands, such instructions will be broken down into multiple sub-operations internally.

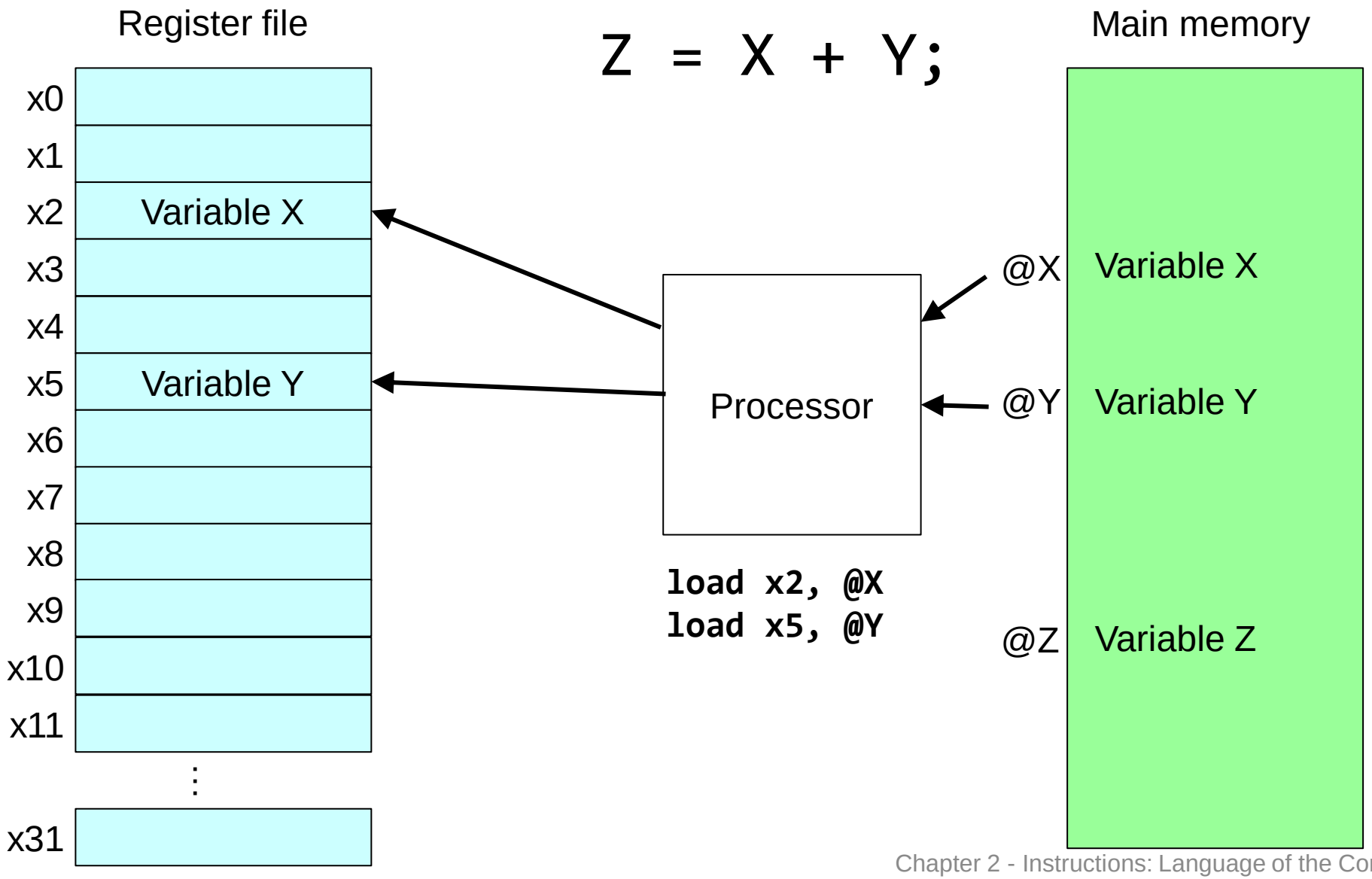
Register Optimization: Crucial for Performance.

- A good compiler should minimize memory access.
  - If an operation can be performed entirely using the registers only, the compiler can avoid memory accesses.
  - If the data size is larger than the registers, the compiler should prioritize frequently accessed data for registers and store less frequently accessed data in memory.

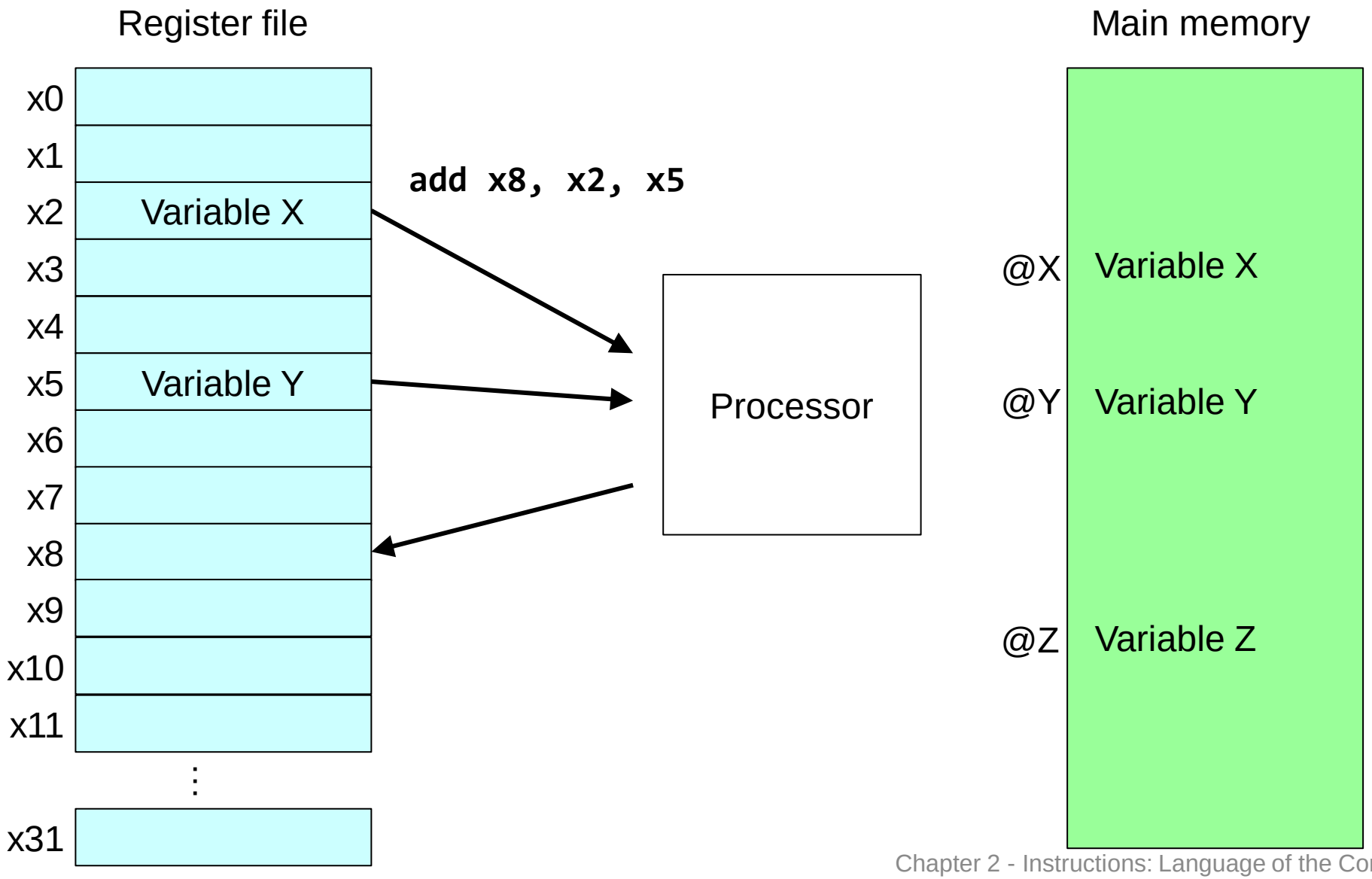
Using Memory Values



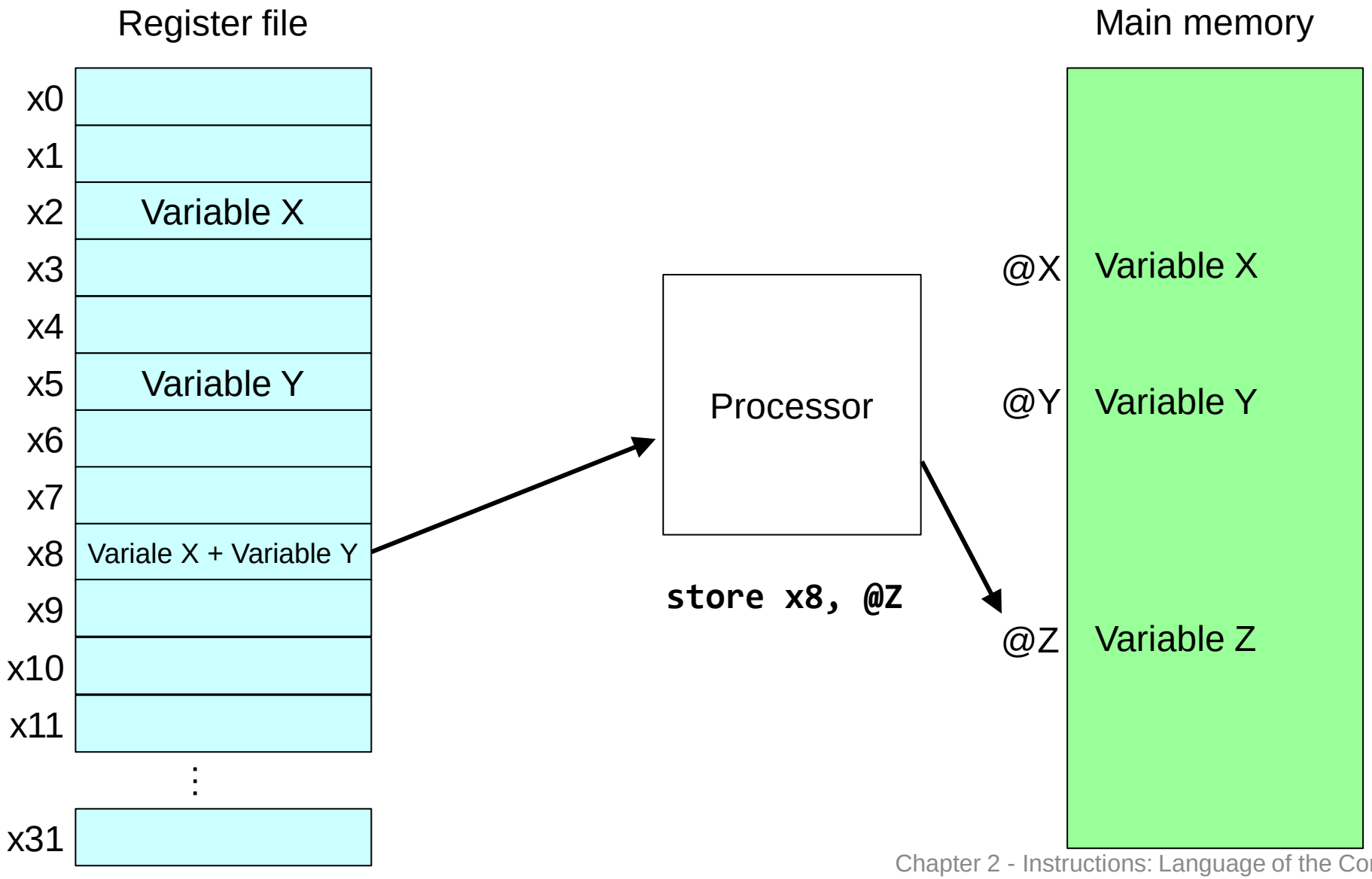
Using Memory Values



Using Memory Values



Using Memory Values



RISC-V Register Usage

Register Number	Name	Usage
x0	zero	Constant 0 (hardwired)
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7, x28-x31	t0 - t6	Temporaries
x8	s0 / fp	Frame pointer
x9, x18-x27	s1 - s11	Saved registers
x10-x11	a0 - a1	Function arguments / results
x12-x17	a2 - a7	Function arguments

Register Operand Example

- Example C code:

```
f = (g + h) - (i + j);
```

  - Suppose `f, g, h, i, j` in `x19, x20, x21, x22, x23`

- Corresponding RISC-V assembly code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

RISC-V Arithmetic Instructions

- RISC-V arithmetic instructions: `add, sub, addi`

Sample Code

- Suppose the registers have the following values

```
x1: 100, x2: 300,  x3: 500, x4: 700
```

- What would be the value of `a4(x14)` after executing the following code?

```
add x11, x1, x2          //x11: 400
sub x12, x0, x3          //x12: -500
add x13, x11, x12        //x13: -100
add x14, x13, x4         //x14: 600
```

The Number of Registers is Finite

- What happens if you have no more registers?
  - You will eventually run out of registers if you use a new register for each instr.

- Reuse registers that are no longer used

```
add x1, x2, x3
sub x5, x0, x4
add x6, x1, x5
add x8, x6, x7
```

```
add x1, x2, x3
sub x5, x0, x4
add x1, x1, x5
add x8, x1, x7
```

- If all registers will be used later? Save some register values in memory
  - “Register spill”
  - Need to avoid as much as possible

Immediate Operands

- **Constant** data included in an instruction

  `addi x22, x23, 4`

- No subtract immediate instruction (~~**`subi`**~~)
  - Instead, just use a negative integer value.

    `addi x22, x23, -1`

The Constant Zero

- RISC-V register 0 (`x0`, or '`zero`') is the constant 0
  - Cannot be overwritten

- Useful for common operations
  - e.g., move (copy) between registers

    `add x1, x2, zero`

  - Loading a constant (immediate) value to a register

    `addi x1, x0, 10`

Load Immediate Value

- **`li`** instruction

  `li` *`xd`*`, immediate_value (32-bit)`

- Example:

```
li x1, 100
li x2, 0x10
add x3, x1, x2
```

- Not a real instruction (*pseudo-instruction*)

Sample Code (2)

- x1: 100, x2: 300, x3: 500, x4: 700

```
li x1, 100
li x2, 300
li x3, 500
li x4, 0x2BC          //0x2bc = 700
add x10, x1, x2
sub x11, x0, x3
add x12, x10, x11
add x13, x12, x4
```

Multiply (1)

- Example: $11_2 \times 11_2 = 110_2 + 11_2 = 1001_2$

- 32-bit × 32-bit → Result can be up to 64-bit

- Need 2 destination registers to completely hold the result!

Multiply (2)

- "**M**" Extension for Multiply / Division
  - The base "RV32I" RISC-V ISA does not contain multiply / division instructions
  - "RV32I**M**" version supports multiply / division instructions

- `mul x3, x1, x2`
  - Multiply x1 and x2, and put the lower 32 bits to x3

- `mulh x3, x1, x2`
  - Multiply `x1` and `x2`, and put the upper 32 bits to `x3`

Multiply (3)

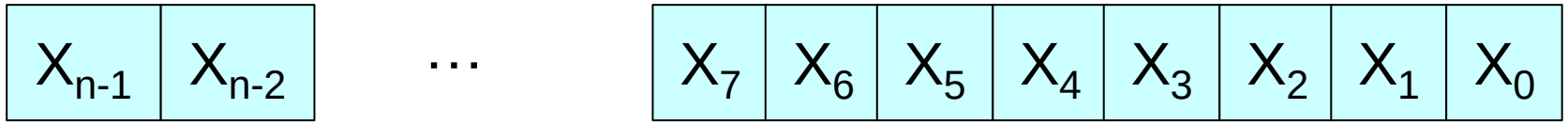
- `mulh x3, x1, x2`
  - Multiply `x1` and `x2`, and put the upper 32 bits to `x3`
  - `x1` and `x2` contains signed numbers

- `mulhu x3, x1, x2`
  - Multiply `x1` and `x2`, and put the upper 32 bits to `x3`
  - `x1` and `x2` contains unsigned numbers

- `mulhsu x3, x1, x2`
  - Multiply `x1` and `x2`, and put the upper 32 bits to `x3`
  - `x1`: signed
  - `x2`: unsigned

Unsigned Binary Integers

- Given an ***n***-bit number



$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example:

  $0000\ 0000\ 0000\ \ldots.\ 0000\ 1011_2$

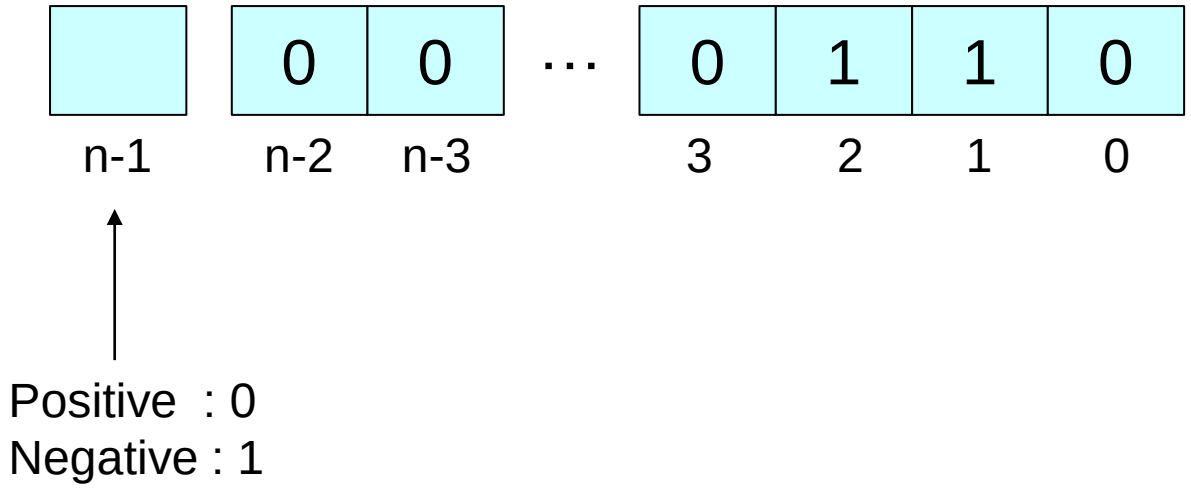
  $= 0\times2^{31} + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$

  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 32 bits: 0 to +4,294,967,295

Signed Binary Integers

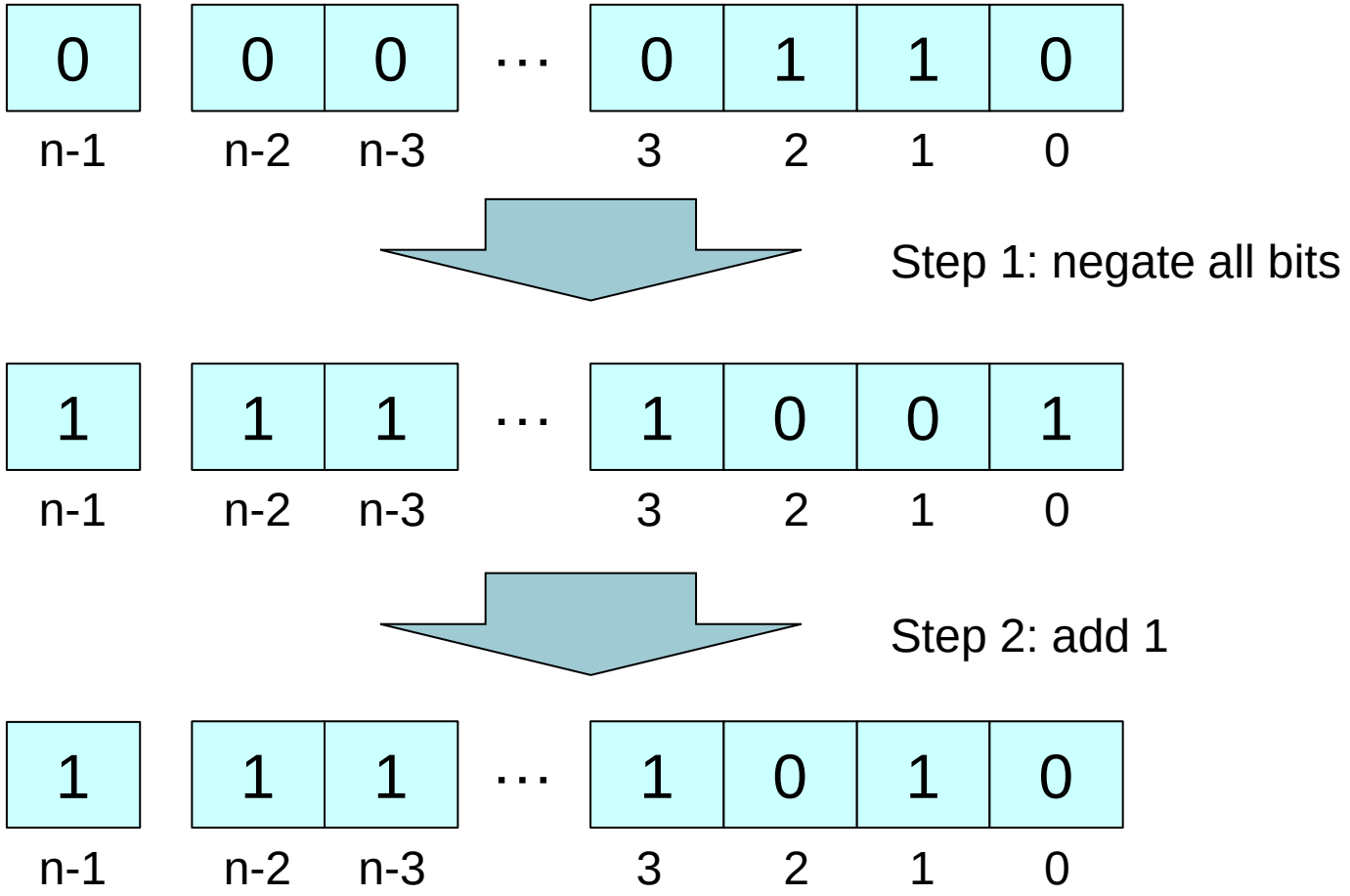
- How to represent a negative integer?



- Sign Magnitude: Any downside?

2’s-Complement Signed Integers

- Given an ***n-1*** bit positive integer… to make it a negative number with the same absolute value,



2's-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$

- Example (32-bit):

  $1111\ 1111\ \ldots\ 1111\ 1100_2$

  $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$

  $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

- Using 32 bits: –2,147,483,648 to +2,147,483,647

2’s-Complement Advantages

- Representing 1 more number (-2n-1)
- Only 1 representation of zero (0)
  - 0000 0000 …. 0000
- Use the same adder logic with the unsigned integer values

Singed / Unsigned Multiplication

- `0xFFFF_FFFF × 0x0000_0002`
  - If signed numbers: -1 × 2 = -2
    - -2 in 64 bit: `FFFF FFFF FFFF FFFE`
  - If unsigned numbers: 4,294,967,295 × 2 = 8,589,934,590
    - 8,589,934,590 in 64 bit: `0000 0001 FFFF FFFE`

Division

- **`div x3, x1, x2`**
  - Divide `x1` by `x2`, and put the quotient to `x3`
- **`rem x3, x1, x2`**
  - Divide `x1` by `x2`, and put the remainder to `x3`
- **`divu / remu`**
  - Unsigned versions

Logical Operations

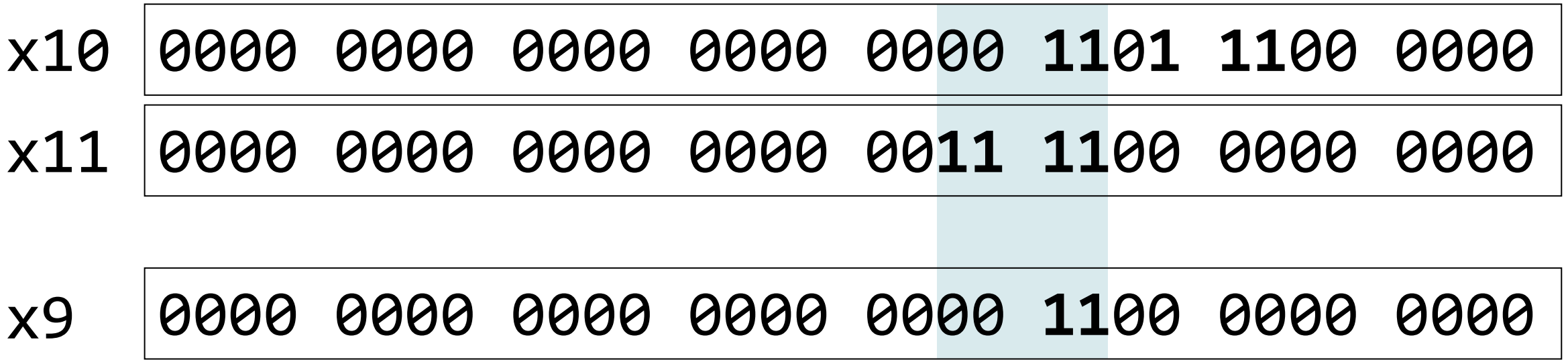
- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Bitwise AND	`&`	`&`	`and, andi`
Bitwise OR	`|`	`|`	`or, ori`
Bitwise XOR	`^`	`^`	`xor, xori`
Bitwise NOT	`~`	`~`	
Shift left	`<<`	`<<`	`slli`
Shift right	`>>`	`>>>`	`srli`

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

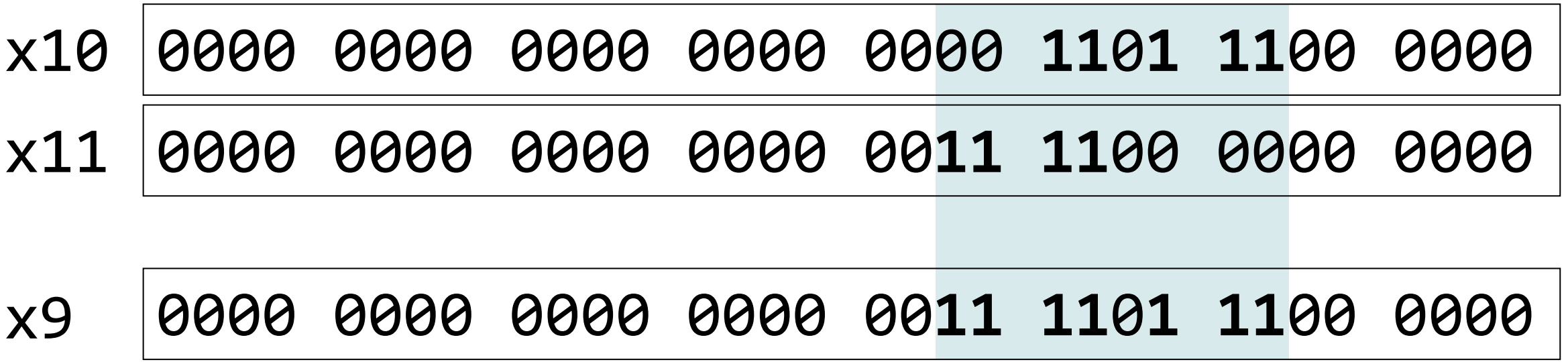
```
and x9, x10, x11
```



OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or x9, x10, x11
```



XOR Operations

- Differencing operation
  - Set some bits to 1, leave others unchanged

```
xor x9, x10, x11  // if x11 = -1 (0xFFFFFFFF), x9 = ~x10
                  // i.e., xor x9, x10, -1 = not x9, x10
```

x10	0000	0000	0000	0000	0000	**1101**	**11**00	0000
x11	1111	1111	1111	1111	1111	1111	1111	1111
x9	1111	1111	1111	1111	1111	0010	0011	1111

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- RISC-V has **`xori`**
  - **`a`** XORI **`-1`** == NOT **`a`**

- Pseudo-instruction: **not**

```
not x10, x11
      ↓
xori x10, x11, -1
```

`xori rd, rs, -1`

`-1`	`1111 1111 1111 1111 1111 1111 1111 1111`
`rs`	`0000 0000 0000 0000 0011 1100 0000 0000`
XOR	
`rd`	`1111 1111 1111 1111 1100 0011 1111 1111`

Basic Shifting

- Shift types
  - Logical (or unsigned) shift
  - Arithmetic (or signed) shift

- Shift directions
  - Left (multiply by powers of 2)

  - Right (divide by powers of 2)
    - Take floor value if the result is not an integer
    - Floor value of X (or $\lfloor X \rfloor$) is the largest integer less than or equal to X
      - 5>>1 = $\lfloor 5/2 \rfloor$ = 2
      - -3>>1 = $\lfloor -3/2 \rfloor$ = -2

Logical Shift

- Logical shift left
  - MSB: shifted out
  - LSB: shifted in with a 0
  - `11001011 << 1 = 10010110`
  - `11001011 << 3 = 01011000`
- Logical shift right
  - MSB: shifted in with a 0
  - LSB: shifted out
  - `11001011 >> 1 = 01100101`
  - `11001011 >> 3 = 00011001`
- Logical shifts are useful to perform multiplication or division of unsigned integer by powers of two

Arithmetic Shift

- Arithmetic shift left
  - MSB: shifted out.
    - Be aware of overflow/underflow
  - LSB: shifted in with a 0
  - `1100 << 1 = 1000  //-4 * 2 = -8`
  - `1100 << 3 = 0000  //-4 * 8 = 0 ??`
- Arithmetic shift right
  - MSB: retain the sign bit
  - LSB: shifted out
  - `0100 >> 1 = 0010 //4 / 2 = 2`
  - `1100 >> 1 = 1110 //-4 / 2 = -2`
- Logical shifts are useful to perform multiplication or division of signed integer by powers of two
- The result of arithmetic shift left is the same as logical shift left

RISC-V Instructions – Shift operations (1)

- **`slli, srli`**
  - `e.g., slli rd, rs1, 4` `( rd ← rs1 << 4)`

- **`srai`**
  - Shift right arithmetic – extend the sign bit
  - Useful for dividing 2's complement numbers

RISC-V Instructions – Shift operations (2)

- **sll, srl, sra**
  - e.g., **sll rd, rs1, rs2** (rd ← rs1 << rs2)

- Shift by the amount held in the lower 5 bits of register **rs2**

```
li x10, 200
li x11, 100   // 100_10 = 0000 0000 0000 0000 0000 0000 0110 0100_2
sll x12, x10, x11 //x12 = 200 << 00100_2 = 200 * 2^4 = 3200
```

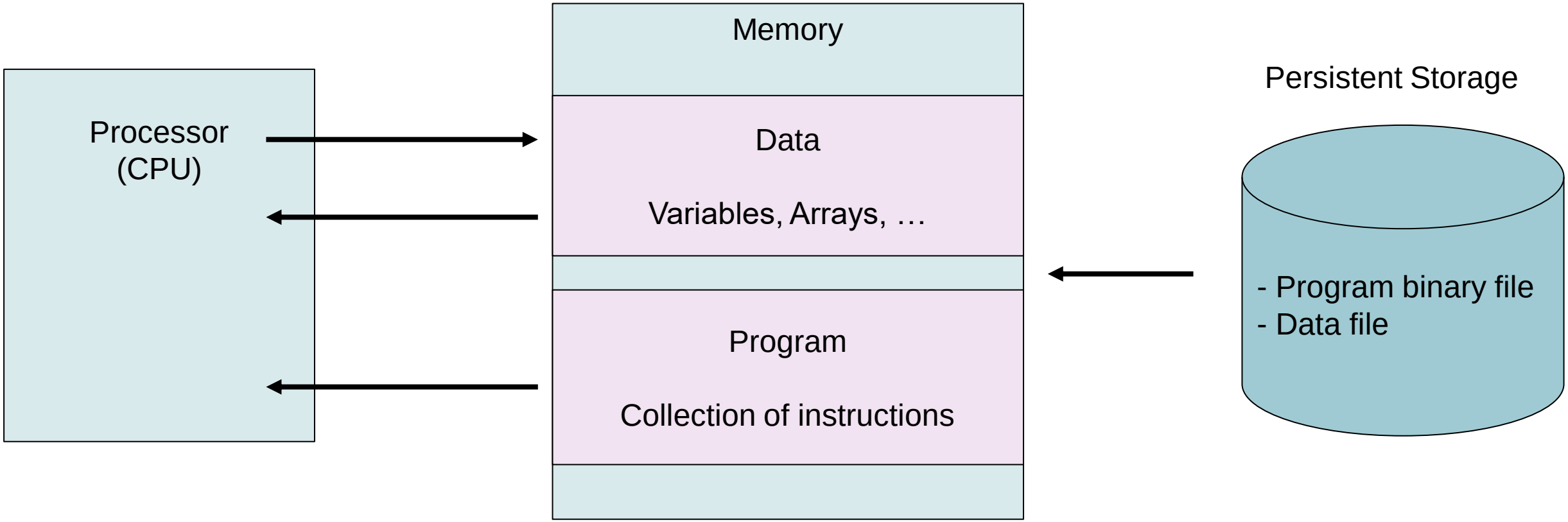
RISC-V Instructions – Shift operations (3)

- $-123 / 2^4 = -123 / 16 = -7.6875 = -7$? -8?
  - Shift right arithmetic: “round down”

```
li x10, -123  // -123_10 = 1111 1111 1111 1111 1111 1111 1000 0101_2

li x11, 4

sra x12, x10, x11 //x12 = 1111 1111 1111 1111 1111 1111 1111 1000_2
                          = -8_10
```

MACHINE CODE

Stored Program Concept



- Instructions and data are represented in binary
- Instructions and data are stored in memory
- CPU fetches instructions and data to execute

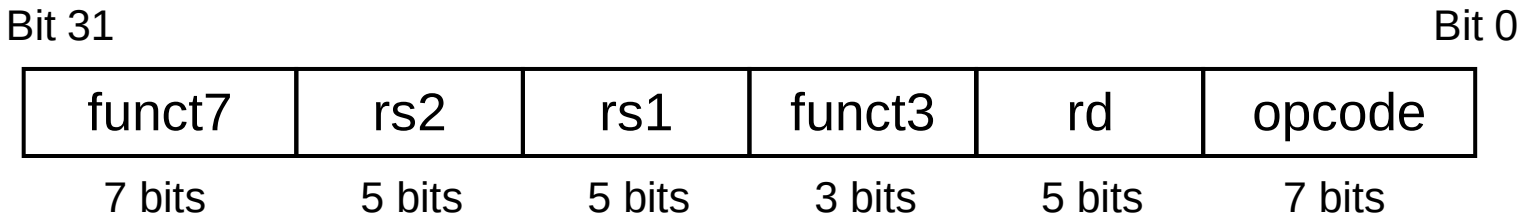
Representing Instructions (1)

- Instructions are encoded in binary
  - Called machine code

- RISC-V instructions
  - Encoded as 32-bit (4 bytes) instruction words

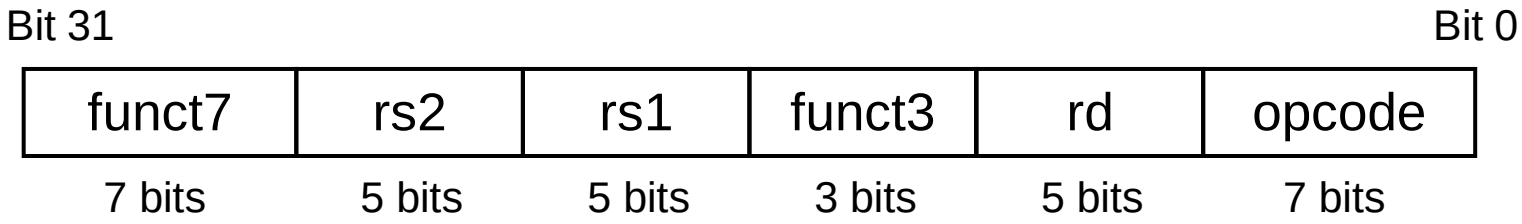
  - CISC processors have variable instruction length
    - x86: 1byte – 15 bytes

RISC-V R-format Instructions



- Instruction fields
  - **`opcode`**: operation code
  - **`rd`**: destination register number
  - **`funct3`**: 3-bit function code (additional opcode)
  - **`rs1`**: the first source register number
  - **`rs2`**: the second source register number
  - **`funct7`**: 7-bit function code (additional opcode)

RISC-V R-format Instructions




```
add x9, x20, x21
```

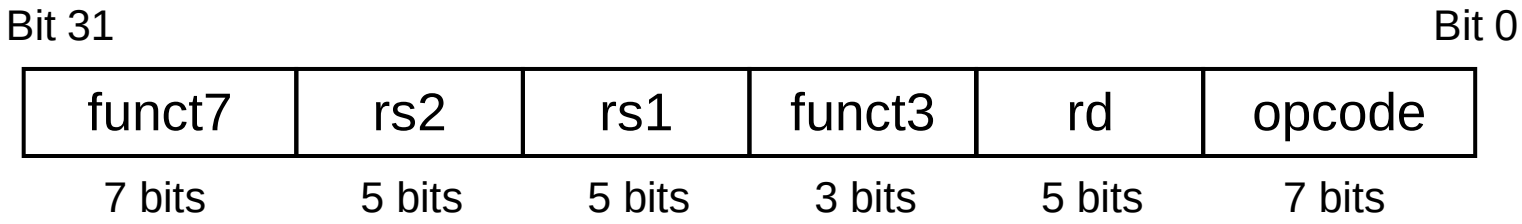
funct7	x21	x20	funct3	x9	Arithmetic
0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

$0000000\ 10101\ 10100\ 000\ 01001\ 0110011_2$

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_2$

$= 0x015A04B3$

RISC-V R-format Instructions




```
sub x9, x20, x21
```

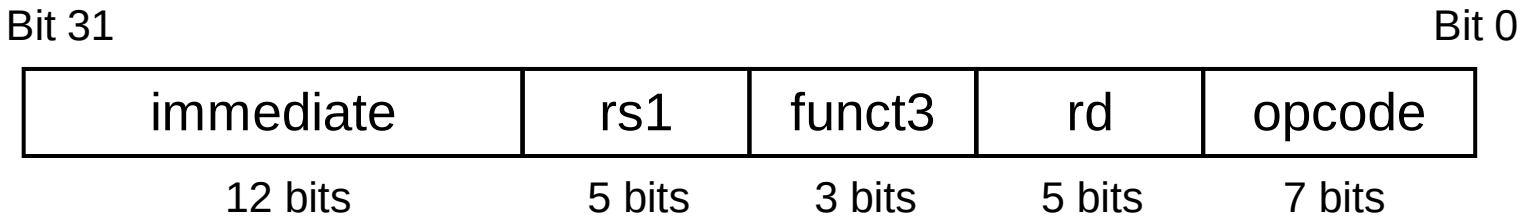
funct7	x21	x20	funct3	x9	Arithmetic
32	21	20	0	9	51
0100000	10101	10100	000	01001	0110011

$0100000\ 10101\ 10100\ 000\ 01001\ 0110011_2$

$0100\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_2$

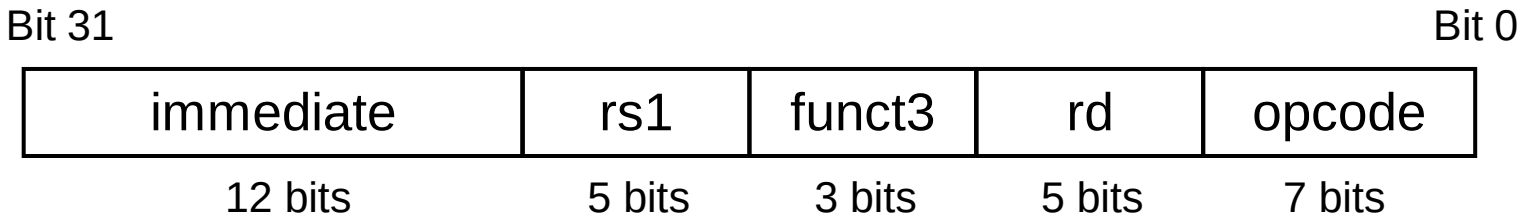
$= 0x415A04B3$

RISC-V I-format Instructions



- Immediate arithmetic and load instructions
  - **`rs1`**: source or base address register number
  - **`immediate`**: constant operand, or offset added to base address
    - 2’s-complement
    - Will be sign extended to 32-bit

RISC-V I-format Instructions




```
addi x9, x20, 123
```

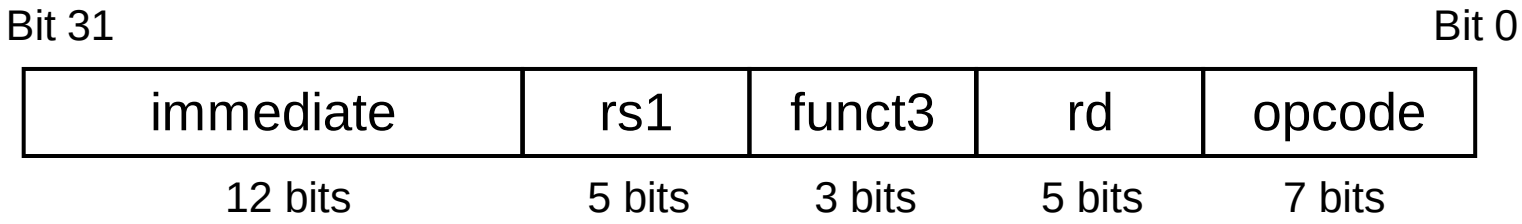
immediate	x20	funct3	x9	Imm. Arithmetic
123	20	0	9	19
0000 0111 1011	10100	000	01001	0010011

$000001111011\ 10100\ 000\ 01001\ 0010011_2$

$0000\ 0111\ 1011\ 1010\ 0000\ 0100\ 1001\ 0011_2$

$= 0x7ba0493$

RISC-V I-format Instructions




```
andi x9, x20, -123
```

immediate	x20	funct3	x9	Imm. Arithmetic
-123	20	**7**	9	19
1111 1000 0101	10100	**111**	01001	0010011

$$111110000101\ 10100\ 111\ 01001\ 0010011_2$$

$$1111\ 1000\ 0101\ 1010\ 0111\ 0100\ 1001\ 0011_2$$

$$= 0xF85A7493$$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value

- Replicate the sign bit to the left
- 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

Sign Extension For Logic Operations

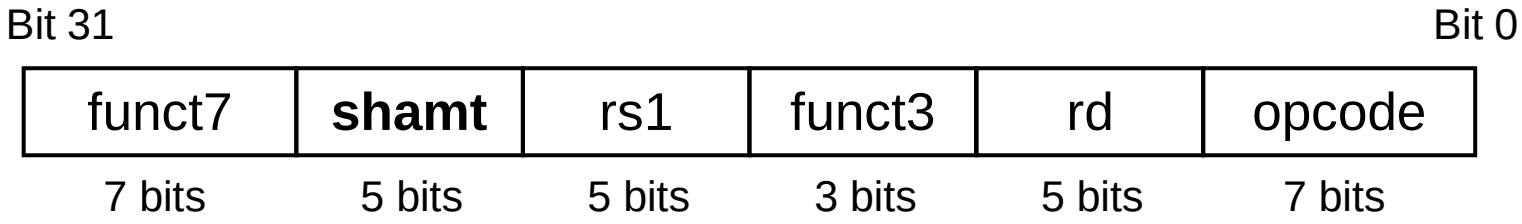
```
li x20, 0x12345   //x20 = 0000 0000 0000 0001 0010 0011 0100 0101₂

andi x9, x20, -123 // -123 = 1111 1000 0101₂

                   //      → 1111 1111 1111 1111 1111 1111 1000 0101₂
```

x20	0000 0000 0000 0001 0010 0011 0100 0101
-123	1111 1111 1111 1111 1111 1111 1000 0101
x9	0000 0000 0000 0001 0010 0011 0000 0101

0x12305

Shift Operations

- `slli, srli, srai`
  - `shamt`: shift amount in number of bit positions



- `sll, srl, sra`
  - `rd` ← value in `rs1` shifted by shift amount in `rs2[4:0]`