

Computer Architecture

Chapter 2

**Instructions: Language of the
computer**

1

Joonho Kong
School of EE, KNU

Introduction

- **How can we communicate to a computer's hardware?**

- Instruction: The words of a computer's language Ex) add, sub, ld, sd, and, or, xor, NOP, ...
- Instruction set: The vocabulary of the instruction Ex) MIPS, RISC-V, ARM, X86, ...

- **The reason of the similarity of instruction sets**

- All computers are constructed from hardware technologies based on similar underlying principles
- There are a few basic operations that all computers must provide
- Computer designers have a common goal

Programming languages

- ⌘ There are many programming languages, but they usually fall into two categories
 - ⌘ High-level languages
 - ⌘ Usually machine independent
 - ⌘ Instructions are often more expressive and easy-to-understand for human
 - ⌘ C, C++, Java, etc.
 - ⌘ Low-level languages
 - ⌘ Usually machine specific
 - ⌘ Offer much finer-grained instructions that closely match the machine language of the target processor
 - ⌘ one statements in high-level languages are generally translated into multiple low-level language instructions
 - ⌘ Assembly languages for MIPS, RISC-V x86, ARM, etc.
- ⌘ High-level languages will not be covered in this course

Assembly language

- ✧ Text representation of the machine language
 - ✧ By using an encoding rule, assembly language can be converted into machine language (binary)
- ✧ One statement represents one machine instruction
- ✧ Abstraction-layer between high-level programs and machine code
- ✧ Operation + operands
 - ✧ Operation: What will the computer do?
 - ✧ Operands: What data will the computer use for the operation?

RISC-V assembly language

- ⌘ A broad classification of the operations

- ⌘ Arithmetic/logical/shift

- ⌘ Performs basic arithmetic, logical, or shift operations

- ⌘ e.g., add, sub. etc.

- ⌘ Data transfer

- ⌘ Transfers data between memory-CPU (register)

- ⌘ e.g., ld, sd, etc.

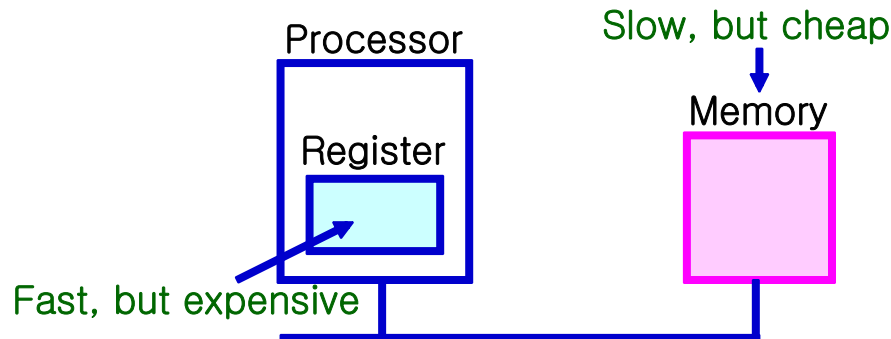
- ⌘ Branch

- ⌘ Controls program counter (PC)

- ⌘ e.g., beq, jal, etc.

Instruction Set Architecture

- ⌘ Text representation of ISA = Assembly language
- ⌘ Binary representation of ISA = Machine language
- ⌘ Interface between hardware and software
- ⌘ An Abstract Data Type
 - ⌘ Operands: Registers & Memory
 - ⌘ Operations: Instruction type



- ⌘ Goal of Instruction Set Architecture Design
 - ⌘ To allow high-performance & low-cost implementations while satisfying constraints imposed by applications including operating system and compiler

Operations of the Computer Hardware

Arithmetic operation of RISC-V

⌘ add instruction

⌘ add two operands from the registers and store the results in the register

⌘ add a, b, c

⌘ $a = b + c;$

⌘ sub instruction

⌘ subtract two operands from the registers and store the results in the register


⌘ sub a, b, c

⌘ $a = b - c;$

Operations of the Computer Hardware

Arithmetic operation of RISC-V

· RISC-V

· **C code**  `add a, b, c` // The sum of b and c in a
`a = b+c+d+e;` `add a, a, d` // The sum of b, c, and d is
now in a
`add a, a, e` // The sum of b, c, d, and e is
now in a

- In case of arithmetic operation like addition instruction of the RISC-V, exactly **three** variables are needed
- The two numbers being added together and placed into a register to save(put) the result of sum

Operations of the Computer Hardware

Example

How can we change below C code to RISC-V assembly language?

Code 1)

```
a = b + c;  
d = a - e;
```

Solution 1)

```
add a, b, c  
sub d, a, e
```

Code 2)

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

Solution 2)

```
add t0, g, h  
add t1, i, j  
sub f, t0, t1
```

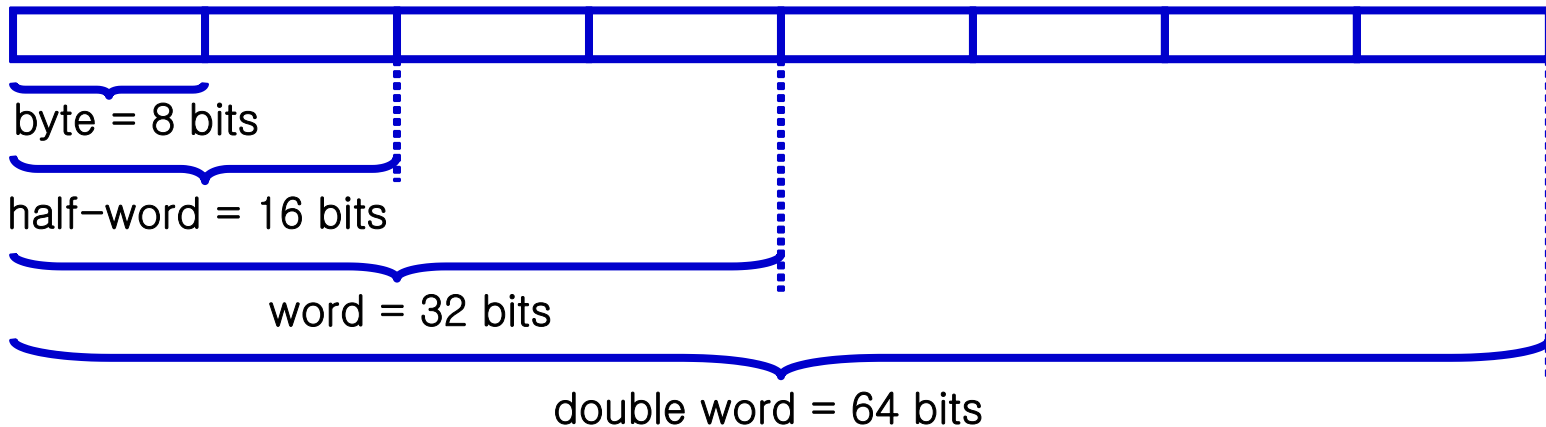
Design Principle 1: Simplicity favors regularity

- Generally use three operands for each instruction

- Make hardware design simpler

Operands of the Computer Hardware

Size of the register



- The size of a register in the RISC-V architecture is 64-bit (double word)
- Arithmetic operations occur only with **registers** or **immediate values** in RISC-V
- The number of the available registers = 32 (x0~x31)

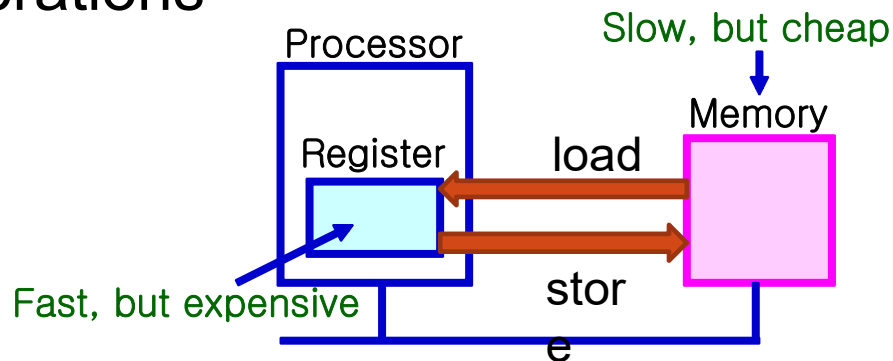
Design Principle 2: Smaller is faster

- **Only use 32 registers**

Operands of the Computer Hardware

⌘ Accessing operands in memory

⌘ Data in memory must be loaded into registers for operations

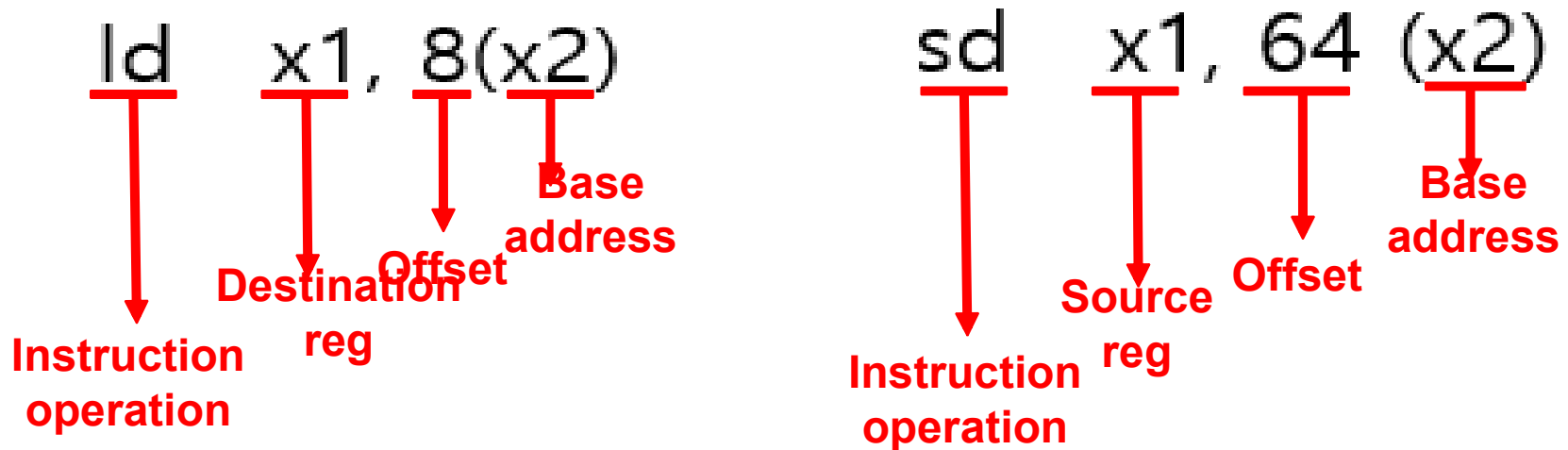


⌘ Load: transfer data from memory to processor (register)

⌘ Store: transfer data from processor (register) to memory

Operands of the Computer Hardware

load and store instruction



Be careful! : a unit of the offset is byte, not index

What is the meaning of above instructions?

Operands of the Computer Hardware

Example

Q. There is a doubleword array $A[100]$ and the compiler has associated the variables g and h with the register $x20$ and $x21$ as before. Let's assume that the starting address ($A[0]$) of the array is in $x22$. Compile this C assignment statement (hint: use **ld** and **add**).

```
g = h + A[8];
```

Operands of the Computer Hardware

Memory operands

- The data accesses are much faster if data are in registers instead of memory
 - Keeping data in the register is better for the performance of the system, but the number of registers is limited
 - We need the **spilling** that tries to put less frequently used variables into memory
 - Registers take less time to access and have higher throughput than memory

Operands of the Computer Hardware

⌘ Constant or immediate operands

⌘ Operands from the instructions, not from the registers

⌘ addi instruction

⌘ add two operands (one from register and the other from immediate) and store the result to the register

⌘ addi a, b, c // a and b are registers c is constant or immediate value

⌘ $a = b + c$

⌘ Example

⌘ addi x22, x22, 4 // $x22 = x22 + 4$

Signed and Unsigned Numbers

2's complement

To represent signed numbers, we use 2's complement method

Let's see an example of 2's complement representation

Q. How to modify the number from N to -N?

A. Negate bits in the number, and then add 1 (ignore a carry bit).

$2_{\text{ten}} = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000010_{\text{two}}$

↓
2's
complement

$$\begin{array}{r} 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111101_{\text{two}} \\ + \\ \hline = 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111111\ 11111110_{\text{two}} \\ = -2_{\text{ten}} \end{array}$$

Representing instructions in the computer language?

- We follow the encoding rule defined in RISC-V architecture

Design Principle 3: Good design demands good compromises

Instruction format
RISC-V instructions are all 32 bits long

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7bits

I-type

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7bits

S-type

immediate [11:5]	rs2	rs1	funct3	immediate [4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7bits

Representing Instruction in the Computer

Instruction format

R-type

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7bits

- opcode: What kind of operations does the instruction do?
- rd: Destination register number
- funct3: An additional opcode field (function code enables different operations even with the same opcode instructions)
- rs1: 1st source register number
- rs2: 2nd source register number
- funct7: An additional opcode field (function code enables different operations even with the same opcode instructions)

Representing Instruction in the Computer

R-type example

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7bits

Let's suppose that compiled code is that

add x9, x20, x21

- This instruction can be represented like

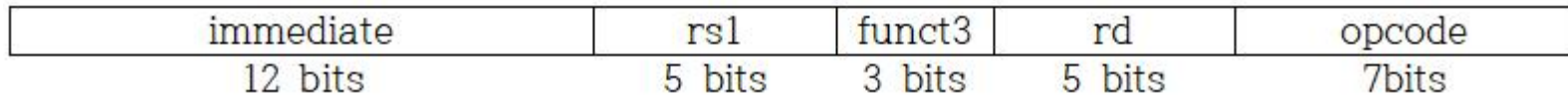
0000000	10101	10100	000	01001	0110011
7 bits	5 bits	5 bits	3 bits	5 bits	7bits

We don't need to memorize the numbers that will be filled in funct7, funct3, and opcode field!

Representing Instruction in the Computer

Instruction format

I-type



- I-type instruction is used by arithmetic operands with one constant operand, including **addi**, and by **load instructions**.
- When I-type is used for **addi** instruction , **immediate field is interpreted as a two's complement value (integer)**.
- When I-type is used for **load** instruction, **immediate field is interpreted as a byte offset**.

Example)

ld x9, 64(x22) // the data A[8] is loaded into register x9

- 22 (for x22) is placed in the rs1 field
- 64 is placed in the immediate field
- 9 (for x9) is placed in the rd field

Representing Instruction in the Computer

Instruction format

S-type

immediate [11:5]	rs2	rs1	funct3	immediate [4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7bits

- S-type format is a format for the store instructions
- The reason of splitting the immediate value field into two parts is rs2, rs1, and funct3 fields can be placed in the same field as other instructions
- **Keeping the instruction formats as similar as possible reduces hardware complexity**

Representing Instruction in the Computer

Instruction format

Summary (don't memorize opcode and function codes!)

R-type

instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

I-type

instruction	Format	Immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
ld (load doubleword)	I	address	reg	011	reg	0000011

S-type

instruction	Format	immediate	rs2	rs1	funct3	rd	opcode
sd (store doubleword)	S	address	reg	reg	011	add- ress	0100011

Logical Operations

Logical operators and corresponding RISC-V instructions

Logical operations	C operators	java operators	RISC-V instruction
shift left	<<	<<	sll, slli
shift right	>>	>>>	srl, srli
shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

Logical Operations

• Shifts

- Shift operations move all the bits in a doubleword to the left or right, filling empty bits with 0s

- For example, if register x19 contained

00000000 00000000 00000000 00000000 00000000 00000000 00000000 00001001_{two} = 9_{ten}

- After shift left by 4, the new value would be:

00000000 00000000 00000000 00000000 00000000 00000000 00000000 10010000_{two} = 14_{ten}

Example

slli x11, x19, 4 // reg x11 = reg x19 << 4 bits

Shift instruction use the I-type format.

Logical Operations

• AND

- AND is a bit-by bit operation that leaves a 1 in the result only if both bits of the operands are 1.

Example

if register x11 contains

00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000_{two}

and register x10 contains

00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000_{two}

then, after executing the RISC-V instruction:

and x9, x10, x11 // reg x9 = reg x10 & reg x11

the value of the register x9 would be

00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000_{two}

- AND can apply a bit pattern to a set of bits to force 0s where there is a 0 in the bit pattern (mask)

Logical Operations

- **OR**

- OR is a bit-wise operation that places a 1 in the result if either operand bit is a 1

Example

if register x10 and register x11 keep the values like the example above, the result of the RISC-V instruction:

or x9, x10, x11 //reg x9 = reg x10 | reg x11

and register x9 will obtain the result as follows:

Logical Operations

• XOR

- Exclusive OR (XOR) puts a 0 when bits are the same and an 1 if they are different.

Example

`xor x9, x10, x11` `//reg x9 = reg x10 ^ reg x11`

`xori x9, x10, 1` `// reg x9 = reg x10 ^ 1`

• NOT (not rd, rs)

- NOT takes one operand and places a 1 in the result if an operand bit is a 0, and vice versa
- **pseudo instruction**: actually it is not supported in RISC-V hardware but programmer can use this instruction and it is automatically converted into an equivalent instruction supported by hardware
- Actually uses XORI instruction (`xori rd, rs, -1`)