

Chapter 2

Instructions: Language of the Computer

Outline

- Introduction
- Operations of the Computer Hardware
- Operands of the Computer Hardware
- Signed and Unsigned Numbers
- Representing Instructions in the Computer
- Logical Operations
- Instructions for Making Decisions

Instruction Set

- The repertoire of instructions of a computer
- Different computers have different instruction sets
 - But with many aspects in common
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets
核心用簡單的Instruction Set(RISC-V) -> 外部用較複雜的X86
將核心包覆
- Instruction sets define market segments

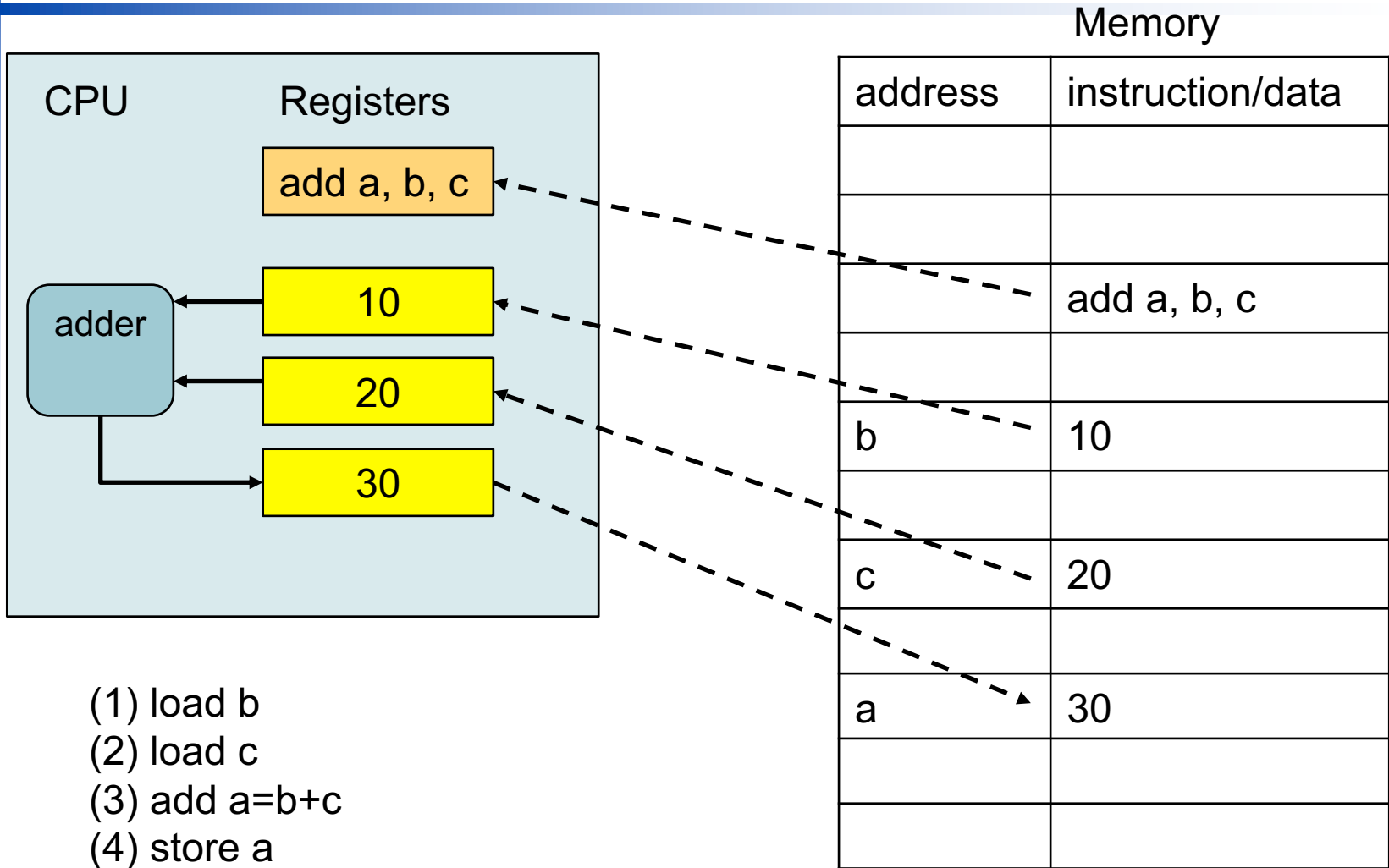
The RISC-V Instruction Set

- Used as the example throughout the course
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
 - See RISC-V Reference Data tear-out card
 - Or <https://www.ee.nthu.edu.tw/ee345000/downloads/doc/RISCVGreenCardv8-20151013.pdf>

Stored-program Concept

- Instructions and data are stored in and loaded from memory as binary codes.
- Assembler translate assembly programs into binary code for execution.
- A processor hardware will load instructions one-by-one and decode/execute them.

Stored Program Illustration



- (1) load b
- (2) load c
- (3) add a=b+c
- (4) store a

所有程式、資料都以Binary形式存在記憶體裡面
CPU則把這些Binary資料取出來使用後存回去

...

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination 長的運算分解成小部分

add a, b, c # a gets b + c

- All arithmetic operations have this form
 - Long expressions are broken into small ops

- *Design Principle 1:* **Simplicity favors**

regularity **較簡單去Implement** 簡化CPU架構的設計
一致性

- Regularity makes implementation simpler
- Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled pseudo assembly code:

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

- Note that we break down one-line of C into several basic arithmetic operations.

- Each handles only two inputs and one output
- Pseudo assembly --- codes similar to assembly but not executable

↙ CPU跟Memory是分開的!!!

- Program cannot access variables in memory directly
- Need to load from memory to CPU registers

Register Operands

- Arithmetic instructions use register operands
- RISC-V has 32 registers (each stores 64-bit)
 - x0 to x31: 32 general purpose registers
 - Use for frequently accessed data
 - Can also implement 32-bit version
 - 64-bit data is called a “double word”
 - 32-bit data is called a “word”

簡化設計，將Arithmetic部分放在Registers處理，另外增加Load/Store的Instructions去做資料搬運

(1) Instruction本身限制Register Bits的個數 (2) 沒有必要 (3) 電路變慢

- *Design Principle 2: **Smaller is faster***

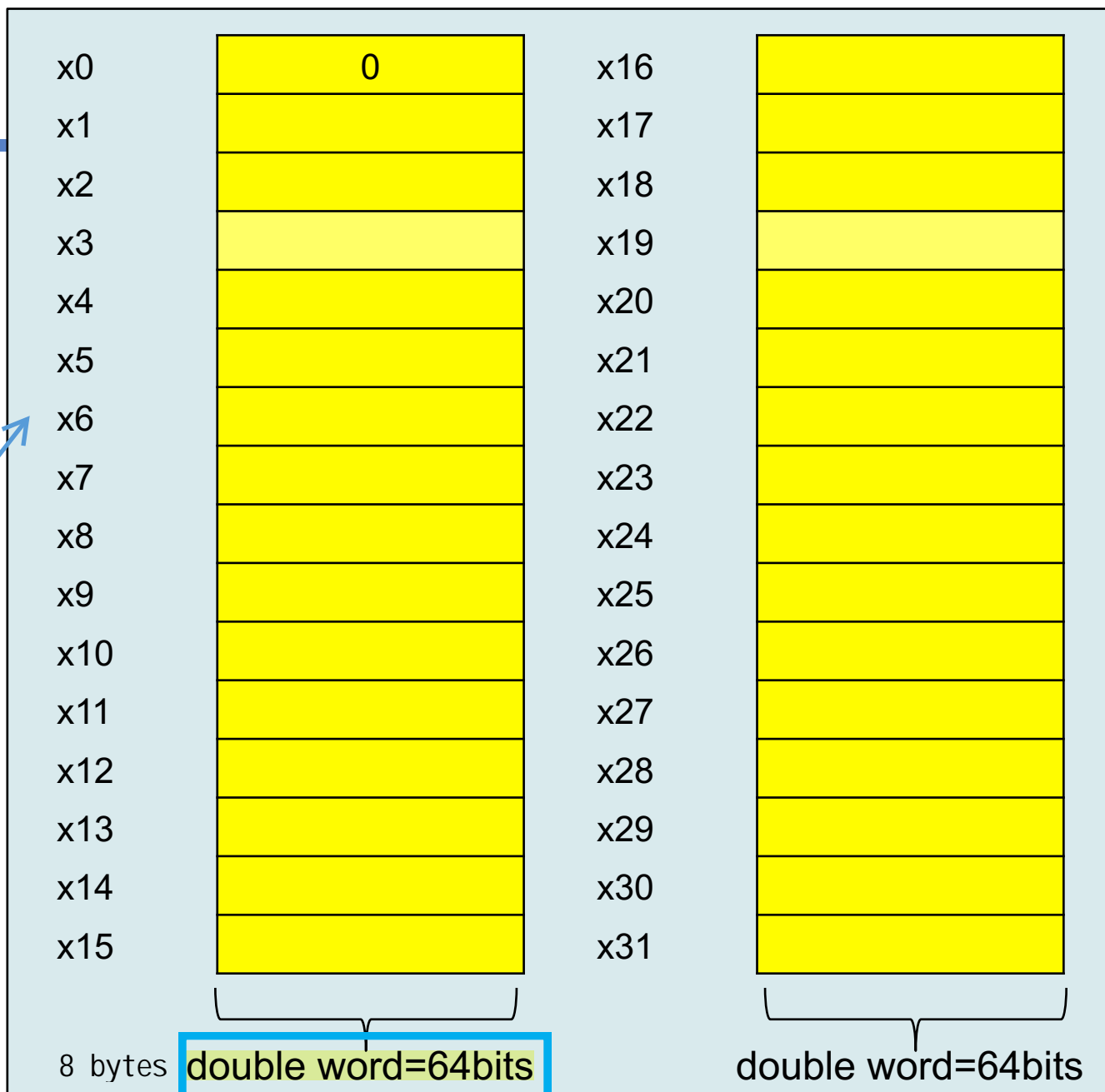
- More registers → complex hardware → slower clocks
- More registers → more bits to represent in ISA format
- c.f. main memory: millions of locations

↑ 占掉大部分bit的空間，使得可使用部分減少

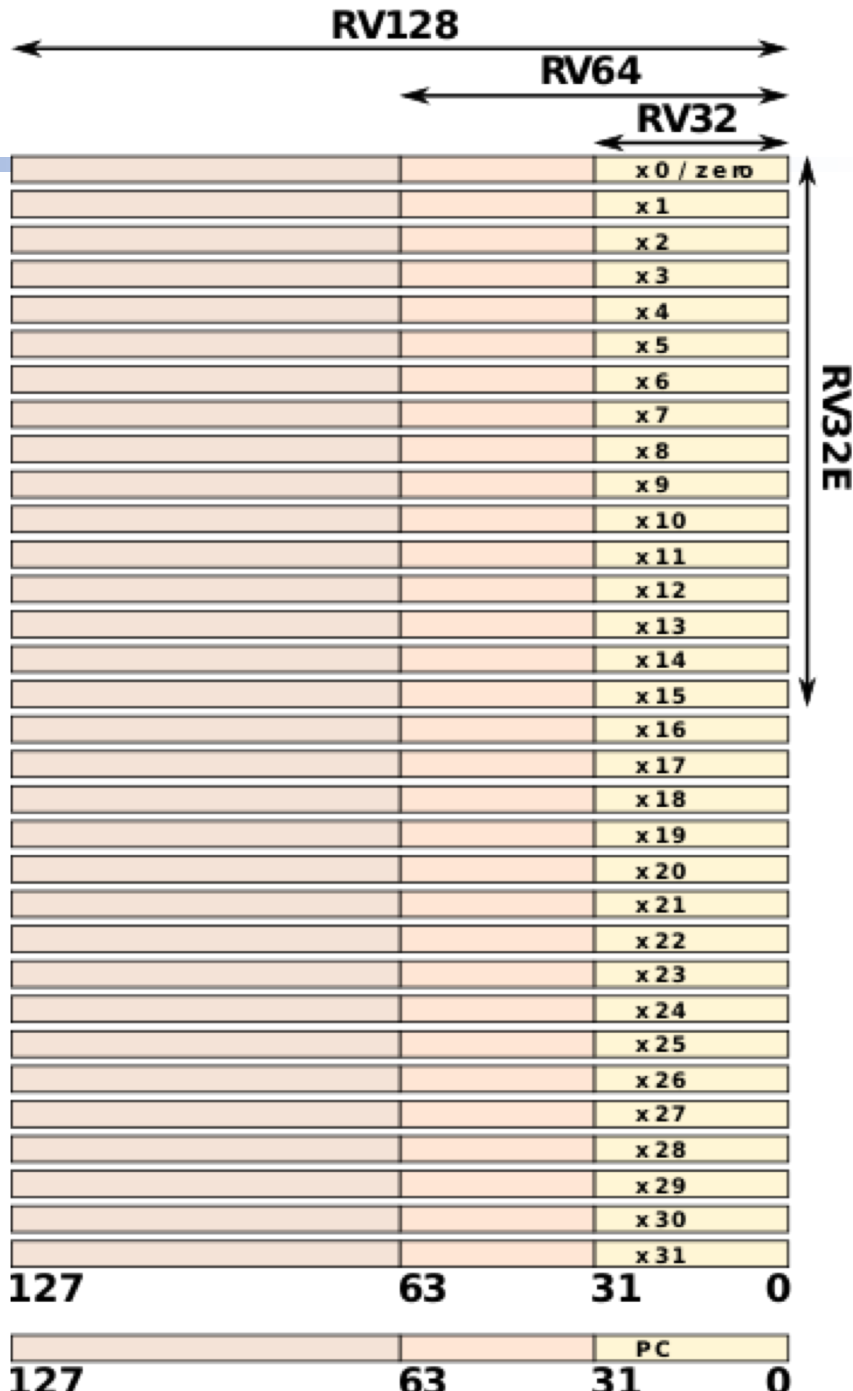
舉前面add運算，add指令會被占存在Register內。但其需要兩個Load、一個Store的Address要用，會使得可用的bit只剩下一個 (32個Reg(用5 bits表示)、每個16bit -> $16-5*3=1$)

RISC-V I64 CPU
Registers (RV64)

因為有32個
Registers, 要用
5個bits去表示它



user-level base integer registers



RV128: 32 registers are 128 bits

RV64: 32 registers are 64 bits

RV32: 32 registers are 32 bits

RV32E: 16 registers are 32 bits

Functions of RISC-V

Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Operand Example

- C code:

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

- f, ..., j in x19, x20, ..., x23

- Compiled RISC-V code:

```
add x5, x20, x21
```

```
add x6, x22, x23
```

```
sub x19, x5, x6
```

Register Names

Recommended

- In assembly codes both names are the same.
- For example, x5 is t0, x20 is s4, and x21 is s5
- Following two codes are the same:

```
add x5, x20, x21
    =
add t0, s4, s5
```
- Assembly names are used by assembler, e.g., gas (GNU) across different architecture.

Register	Assembly or ABI name
x0	zero
x1	ra
x2	sp
x3	gp
x4	tp
x5-7	t0-2
x8	s0/fp
x9	s1
x10-11	a0-1
x12-17	a2-7
x18-27	s2-11
x28-31	t3-6

Memory Operands

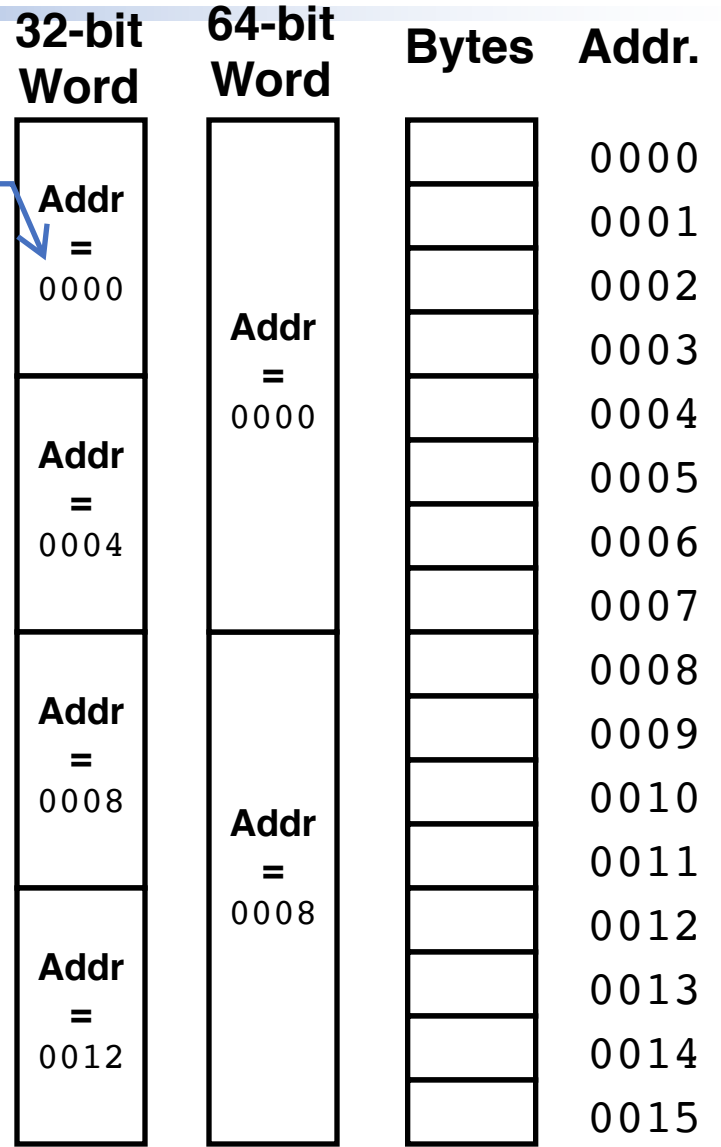
- Main memory used for composite data
 - Arrays, structures, dynamic data, etc.
- Register values are transferred to/from memory
 - To apply arithmetic operations, first load values from memory into registers
 - After calculation, store results from register to memory
- Memory is **byte addressed**
 - Each address identifies an 8-bit data (byte)

用Byte當作Address的單位

Word-Oriented Memory Organization

- Addresses specify byte locations
- Word address**
 - Address of first byte in word
- Addresses of successive words differ by 4 (Sun) or 8 (Alpha)
 - Aligned** word address
- RISC-V does not require words to be aligned in memory
 - Unlike some other ISAs, where address are usually a multiple of 4

用第一個Byte的Address當作Addr



RISC-V ld/sd Instructions

- ld: load double word data from memory
- sd: store double word data to memory

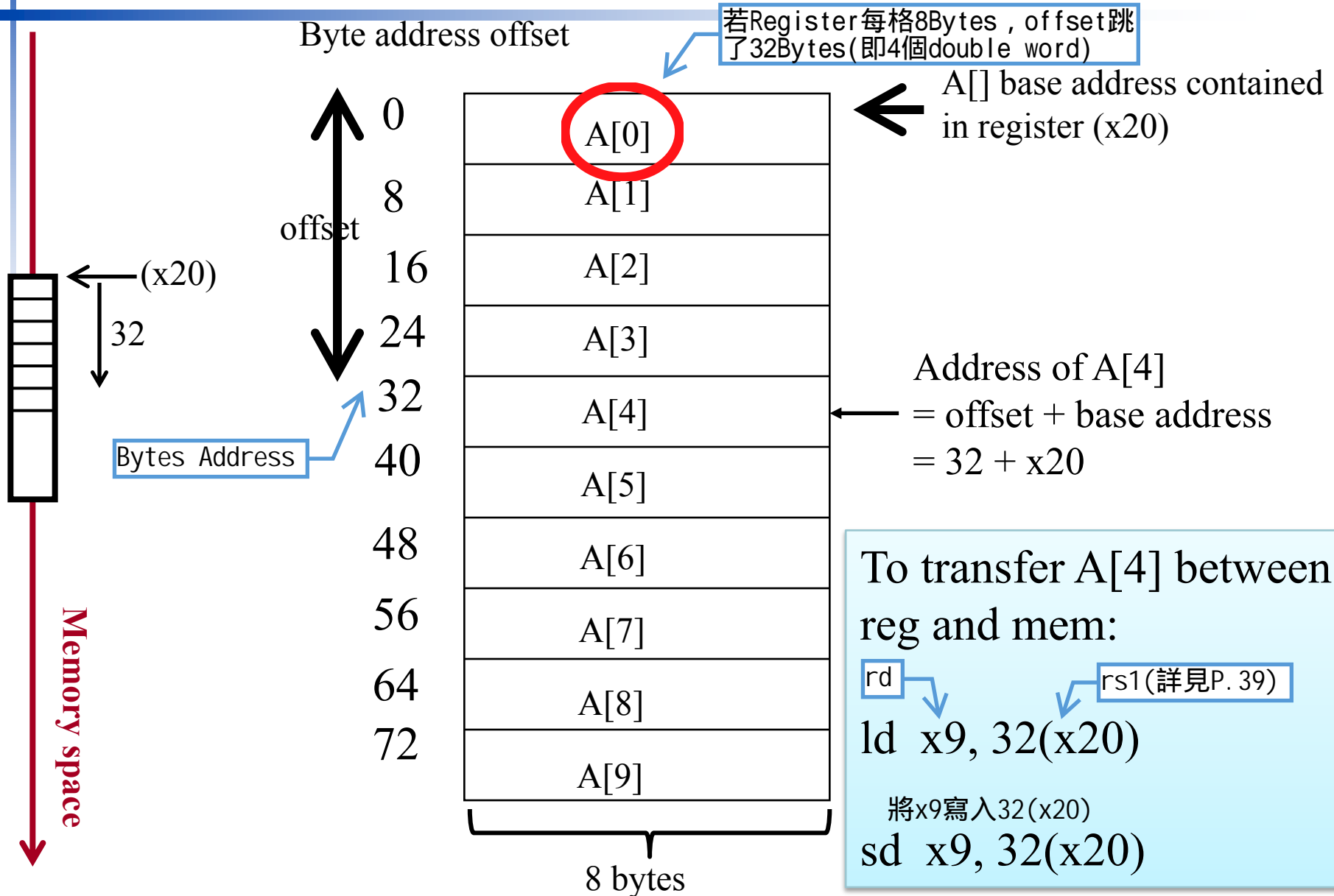
- Example:

ld x9, 32(x20)

(1) 0(x20) 將CPU內x20裡面存的address值，對應到Memory裡面抓取該address內存的數值
(2) 32(x20) 將CPU內x20裡面存的address值+32Bytes後，對應到Memory裡面抓取新計算address內存的數值

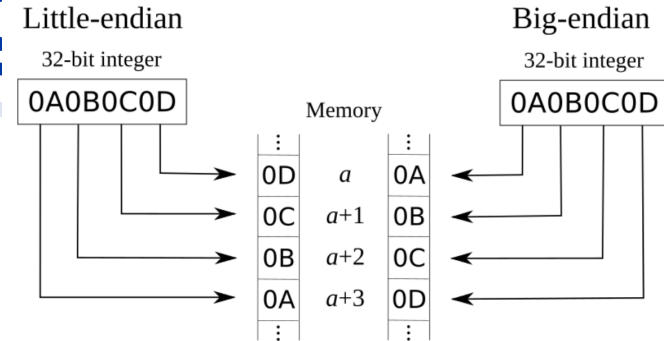
- Load a double word data from 32(x20) to x9 register
- 32(x20) is an address of *value of $x20 + 32$*
- Above is a relative addressing mode, where we usually put an array address at x20
- 32 bytes = 4 double words
- 32(x20) is to access the 5th double word in an array

ld/sd with Double Word Address



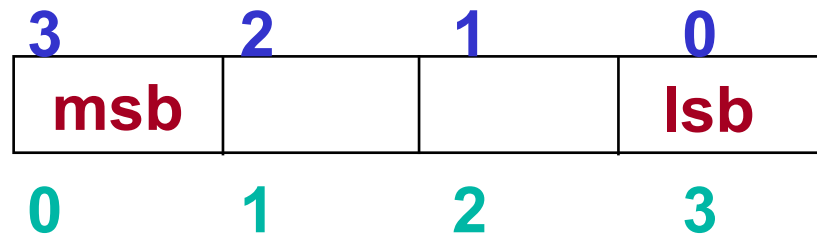
Little Endian vs. Big Endian

- RISC-V is Little Endian
 - Order to store bytes in a word
 - Least-significant byte at least address of a word
 - c.f. Big Endian: most-significant byte at least address



Little Endian byte order把
LSB存在較小的address、MSB
存在較大的address

← **little endian byte order**



big endian byte order →

Endian Example

- To store a 4-byte word “6789ABCD” in hexadecimal into a byte address of 0X1000

0X1000	CD
0X1001	AB
0X1002	89
0X1003	67
0X1004	
0X1005	
0X1006	
0X1007	

Little-endian

0X1000	67
0X1001	89
0X1002	AB
0X1003	CD
0X1004	
0X1005	
0X1006	
0X1007	

Big-endian

Memory Operand Example 1

- C code:

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

- g in x20, h in x21, base address of A in x22

- Compiled RISC-V code:

- Index 8 requires offset of 64
 - 8 bytes per double word

```
ld x9, 64(x22) # load double word  
add x20, x21, x9
```

offset

base register

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

- h in x21, base address of A in x22

- Compiled RISC-V code:

```
ld      x9, 64(x22) #load double word
add     x9, x21, x9
sd      x9, 96(x22) #store double word
```

Register可以重複使用

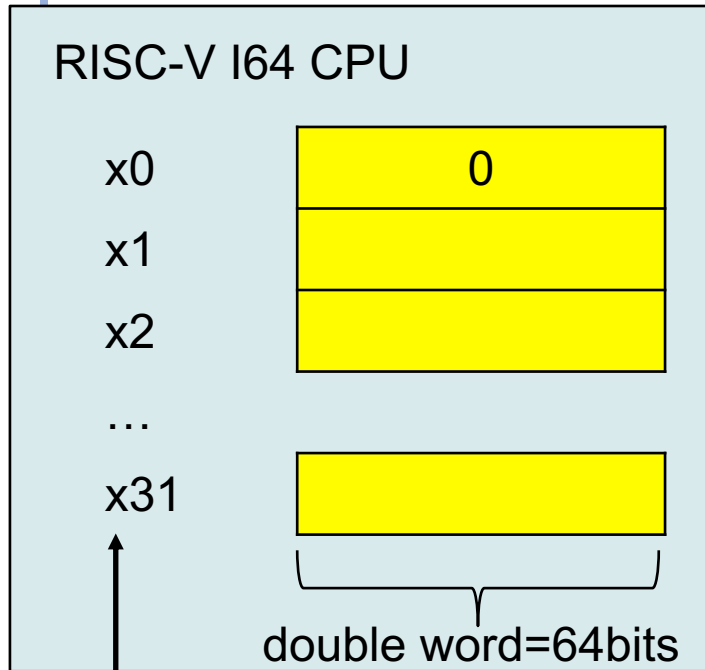
Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must keep variables in registers as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

用到越多 register 越好, 減少 Memory I/O 的次數 (減少 Instruction)

Registers and Memory Size

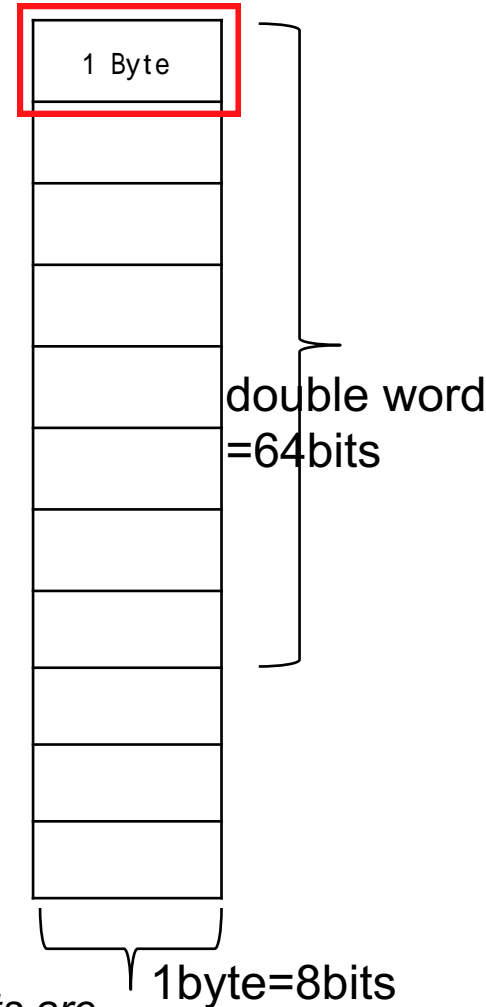
Memory使用64bit去定址，每個8bytes
Register共32個，每個8bytes



64bit address

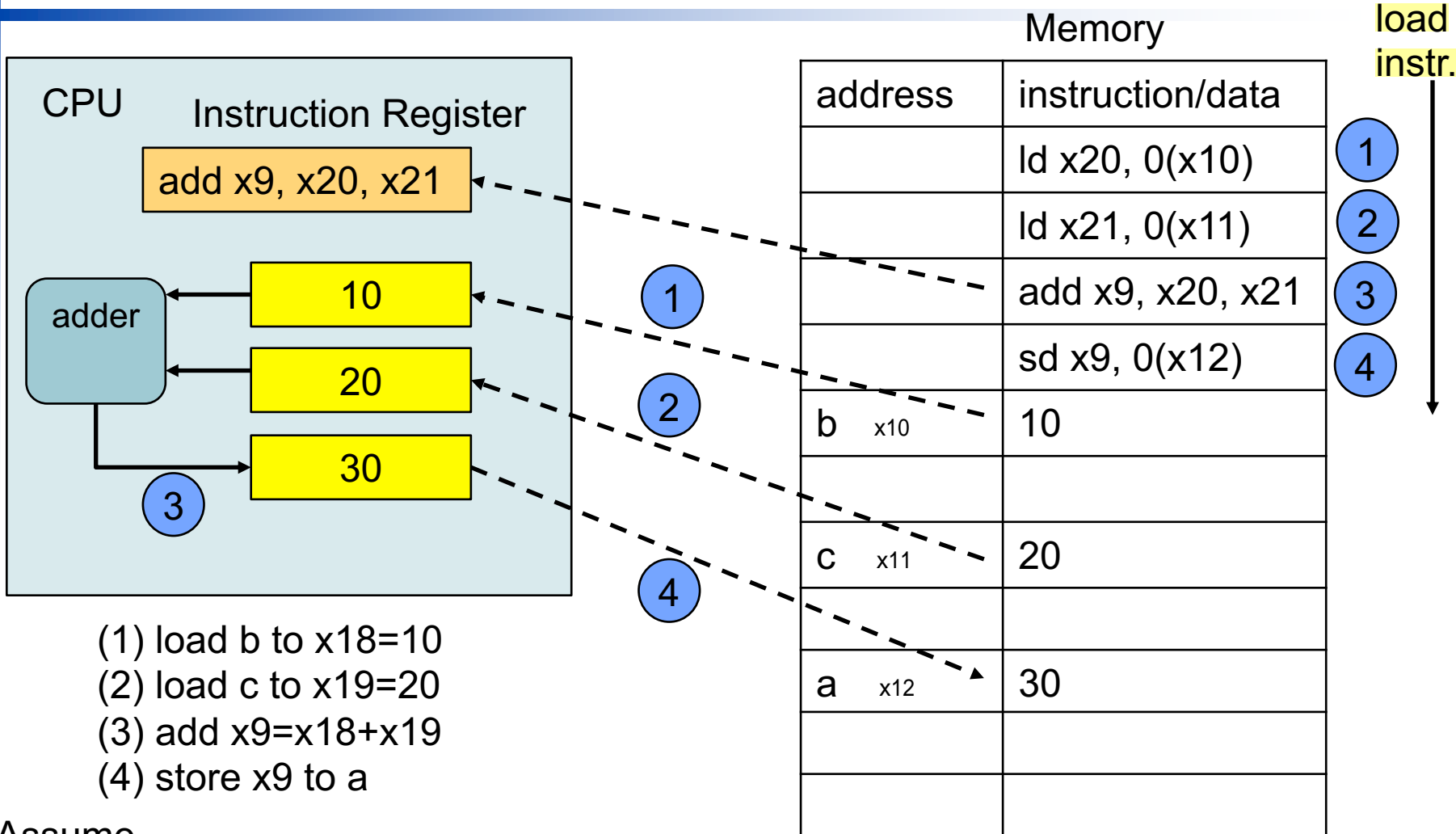
0x 0000 0000 0000 0000
0x 0000 0000 0000 0001
0x 0000 0000 0000 0002
0x 0000 0000 0000 0003
0x 0000 0000 0000 0004
0x 0000 0000 0000 0005
0x 0000 0000 0000 0006
0x 0000 0000 0000 0007
...
0x FFFF FFFF FFFF FFFF

data Memory



Note: actual physical address bits are less than 64, e.g., 42 for Intel x86_64.

Stored Program Example with Memory Access



- (1) load b to x18=10
- (2) load c to x19=20
- (3) add x9=x18+x19
- (4) store x9 to a

Assume

x10=b's address

x11=c's address

x12=a's address

Immediate Operands

- Constant data specified in an instruction

`addi x22, x22, 4` ← 加一個常數

- No subtract immediate instruction

- Just use a negative constant

`addi x22, x22, -4`

- Make the common case fast

- Small constants are common

- Immediate operand avoids a load instruction

The Constant Zero

- RISC-V register 0 (x0) is the constant 0
 - The value of x0 is hard wired to 0.
 - Don't use x0 for storing regular data.
- Useful for common operations
 - E.g., move between registers

```
add x21, x20, x0
```

=

```
move x21, x20
```

← 會被Assembler轉譯成
add的形式去執行

用加法器加0去取代移動暫存器資料

move is a **pseudo instruction**, which will be translated by assembler to add

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

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

- Example

- $0000\ 0000\ \dots\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 64 bits: 0 to
 $+18,446,774,073,709,551,615$

2s-Complement Signed Integers

- Given an n-bit number

將MSB變成負數

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

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

- Example

- $1111\ 1111\ \dots\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$


- Using 64 bits:

$-9,223,372,036,854,775,808$

to $9,223,372,036,854,775,807$

2s-Complement Signed Integers

- Bit 63 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111



非負數在unsigned
和2s-complement有
一樣的代表法

2's Complement Numbers (32bits)

0000 0000 0000 0000 0000 0000 0000 0000 = 0

0000 0000 0000 0000 0000 0000 0000 0001 = 1

0000 0000 0000 0000 0000 0000 0000 0010 = 2

...

0111 1111 1111 1111 1111 1111 1111 1101 = 2147483645

0111 1111 1111 1111 1111 1111 1111 1110 = 2147483646

0111 1111 1111 1111 1111 1111 1111 1111 = 2147483647

1000 0000 0000 0000 0000 0000 0000 0000 = -2147483648

1000 0000 0000 0000 0000 0000 0000 0001 = -2147483647

1000 0000 0000 0000 0000 0000 0000 0010 = -2147483646

...

1111 1111 1111 1111 1111 1111 1111 1101 = -3

1111 1111 1111 1111 1111 1111 1111 1110 = -2

1111 1111 1111 1111 1111 1111 1111 1111 = -1

Signed Negation

- Fast conversion to 2's complement
 - Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111\dots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: find 2's complement for +2
 - $+2 = 0000\ 0000 \dots 0010_2$
 - $-2 = 1111\ 1111 \dots 1101_2 + 1$
 $= 1111\ 1111 \dots 1110_2$

Sign Extension

當把32bits的資料放到64bits的空間內，
必須要做bit extension才不會發生2's complement
用0延伸產生的錯誤

- Representing a number using more bits
 - Preserve the numeric value
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- 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
unsigned (補0)

將MSB做延伸

Representing Instructions

- Instructions are encoded in binary
 - Called machine code
- **RISC-V instructions**
 - Encoded as **32-bit** instruction words
 - Formats to encode operation code (opcode), register numbers, immediate values, etc...
 - Regularity!
- RISC-V has a RVC (compressed extension) to store instructions in 16-bit format

Actual instruction formats please see Chapter 19 RV32/64G Instruction Set Listings of Volume I: RISC-V User-Level ISA

Hexadecimal

- Base 16

Instruction共有32bits, 用Hex(Base16, 4bits)做表示, 則需要8個

- Compact representation of bit strings
- 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: 0x eca8 6420

- 1110 1100 1010 1000 0110 0100 0010 0000
- Usually in programming language **0x** indicates **Hex format**.

RISC-V R-format Instructions



■ Instruction fields

- opcode: operation code (op group)
- 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)

R-format Example



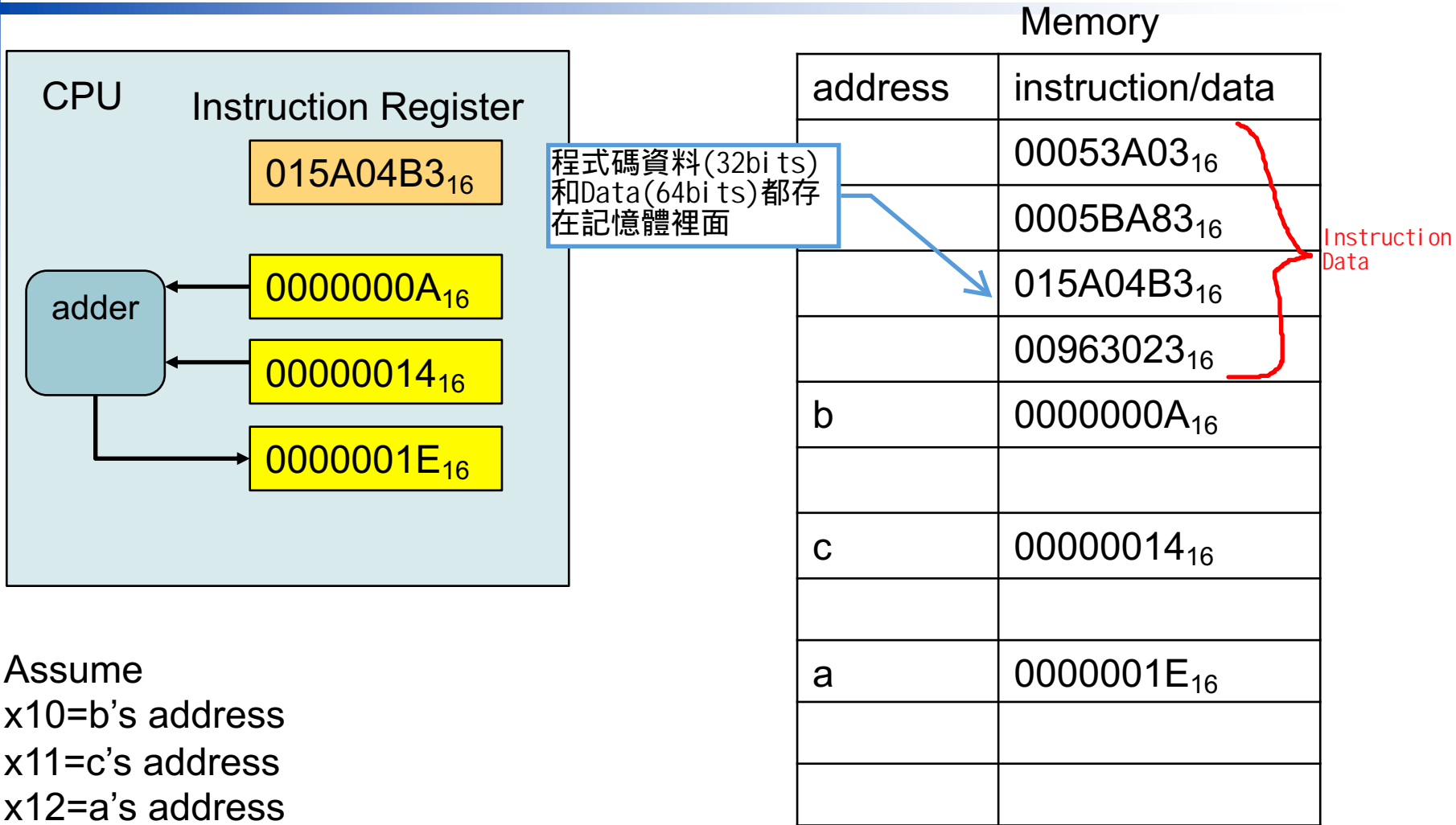
add x9, x20, x21

0	21	20	0	9	51
---	----	----	---	---	----

0000000	10101	10100	000	01001	0110011
---------	-------	-------	-----	-------	---------

0000 0001 0101 1010 0000 0100 1011 0011_{two} =
015A04B3₁₆

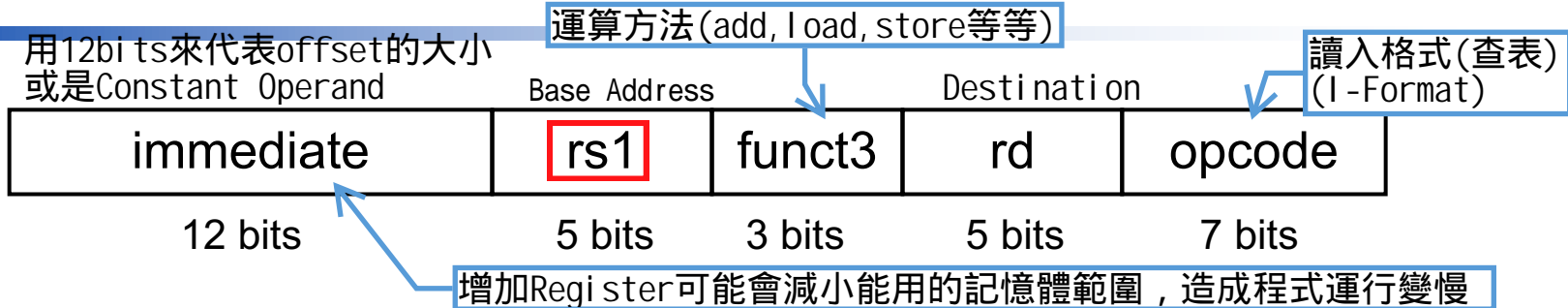
Binary Codes of Stored Program Example (in Hexadecimal)



Assume
 x10=b's address
 x11=c's address
 x12=a's address

...

RISC-V I-format Instructions



Immediate arithmetic and load instructions

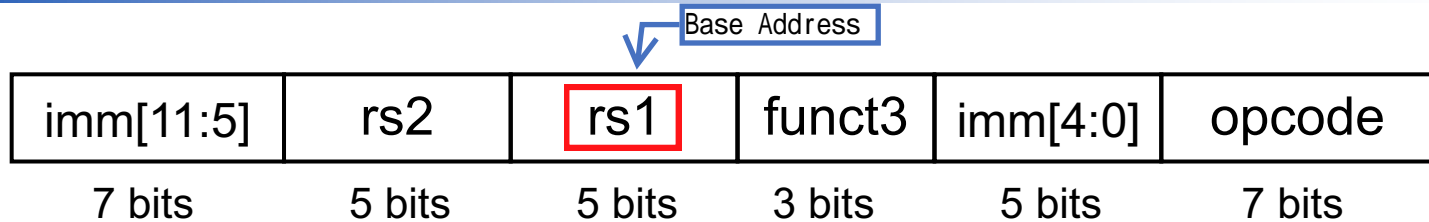
- rs1: source or base address register number
- Constant: -2^{11} to $+2^{11} - 1$
- immediate: constant operand, or offset added to base address
 - 2s-complement, sign extended

Design Principle 3: Good design demands good compromises

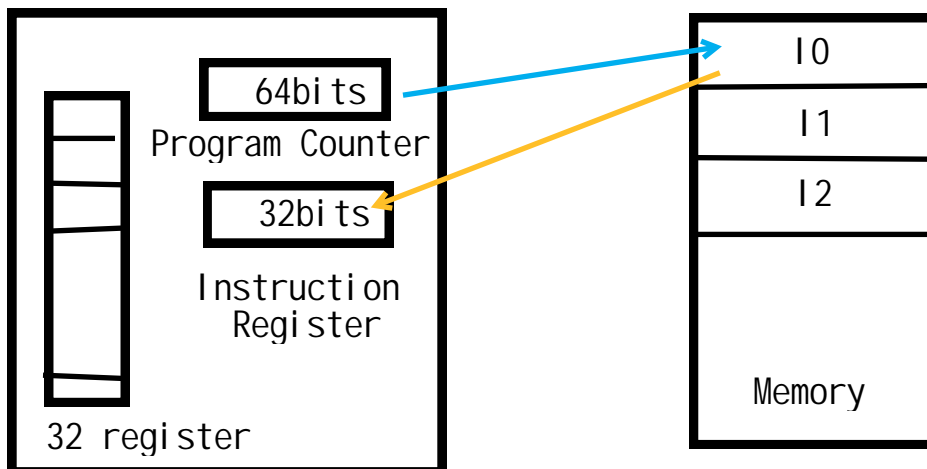
- Different formats complicate decoding, but allow 32-bit instructions uniformly
- Keep formats as similar as possible

該Instruction會存在記憶體內部(32bit)，非Register內部

RISC-V S-format Instructions

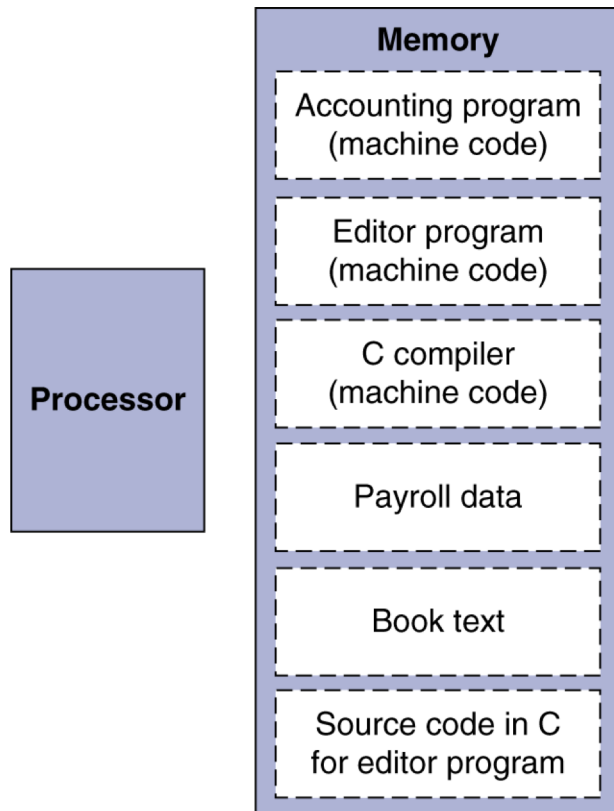


- Different immediate format for **store** instructions
 - rs1: base address register number
 - rs2: source operand register number
 - immediate: offset added to base address
 - Split so that rs1 and rs2 fields always in the same place



Stored Program Computers

The BIG Picture



- 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

Logical Operations

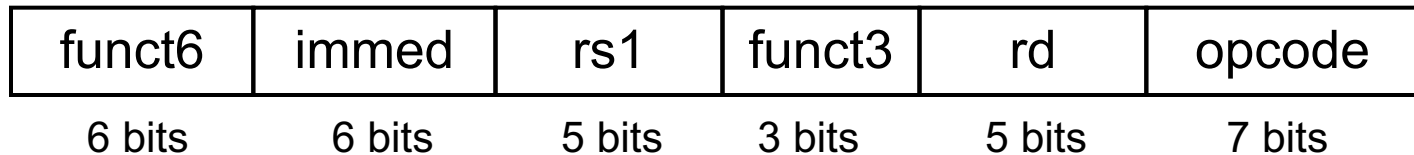
- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	slli
Shift right	>>	>>>	srlrli
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

- Useful for extracting and inserting groups of bits in a word

(1)	1,	1: 0
(2)	1,	0: 1
(3)	0,	1: 1
(4)	0,	0: 0

Shift Operations



- immed: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - `sll i by i bits multiplies by 2^i` `slli rd, rs1, immed`
- Shift right logical
 - Shift right and fill with 0 bits
 - `srl i by i bits divides by 2^i (unsigned only)`

因為補0，所以只有unsigned能用

AND Operations

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

and x9, x10, x11

x10 00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000

x11 00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000

x9 00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

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

or x9, x10, x11

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

- Differencing operation

← 不一樣時輸出1

- Set some bits to 1, leave others unchanged

xor x9, x10, x12 // XOR operation

x10	00000000	00000000	00000000	00000000	00000000	00000000	00001101	11000000
x12	11111111	11111111	11111111	11111111	11111111	11111111	11111111	11111111
x9	11111111	11111111	11111111	11111111	11111111	11111111	11110010	00111111

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq rs1, rs2, L1`
 - if (`rs1 == rs2`) branch to instruction labeled L1
- `bne rs1, rs2, L1`
 - if (`rs1 != rs2`) branch to instruction labeled L1
- There is no direct jump instruction

Compiling If Statements

- C code:

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

- f, g, ... in x19, x20, ...

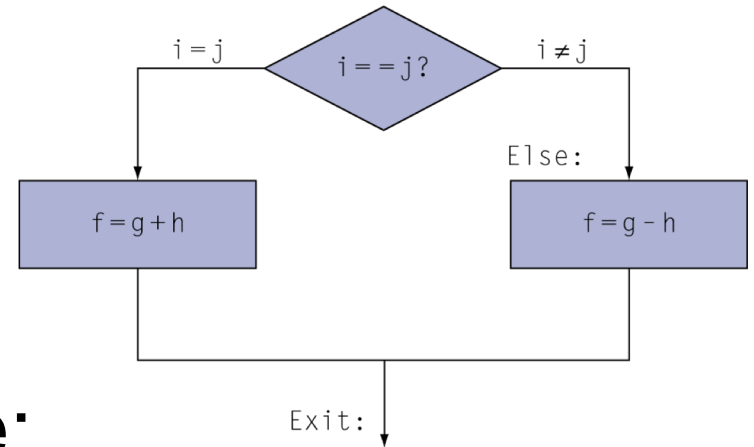
- Compiled RISC-V code:

```
    bne x22, x23, Else  
    add x19, x20, x21
```

```
    beq x0, x0, Exit // unconditional
```

```
Else: sub x19, x20, x21
```

```
Exit: ...
```



必要，讓Else不被執行

Assembler calculates addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

- i in x22, k in x24, address of save in x25

- Compiled RISC-V code:

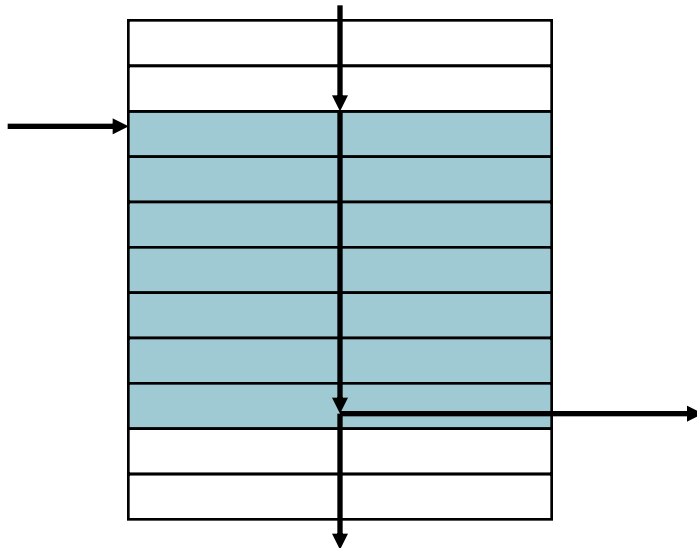
```
Loop: slli x10, x22, 3  往左3bits, 乘以8(dword為8Bytes)
      add  x10, x10, x25  找到save[i]的地址
      ld   x9, 0(x10)    讀取x10位址存的value至x9
      bne x9, x24, Exit
      addi x22, x22, 1
      beq x0, x0, Loop
Exit: ...
```

Basic Blocks

- A **basic block** is a **sequence of instructions** with

Basic Block為一連串的Instruction中間沒有任何branch，即可以從頭執行到尾，對於開發者而言，對Basic Block做優化的時候，不用考慮branch出去的情況。

- **No embedded branches** (except at end)
- **No branch targets** (except at beginning)



- **A compiler identifies basic blocks for optimization**
- **An advanced processor can accelerate execution of basic blocks**

More Conditional Operations

- `blt rs1, rs2, L1`
 - if ($rs1 < rs2$) branch to instruction labeled L1
- `bge rs1, rs2, L1`
 - if ($rs1 \geq rs2$) branch to instruction labeled L1
- Example
 - if ($a > b$) `a += 1`;
 - a in x22, b in x23
`bge x23, x22, Exit` // branch if $b \geq a$
`addi x22, x22, 1`

Exit:

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
 - $x_{22} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
 - $x_{23} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
 - $x_{22} < x_{23}$ // signed
 - $-1 < +1$
 - $x_{22} > x_{23}$ // unsigned
 - $+4,294,967,295 > +1$