Lab 2
Assembly Lab I

Video Link : https://youtu.be/Oli5gPcdjHM

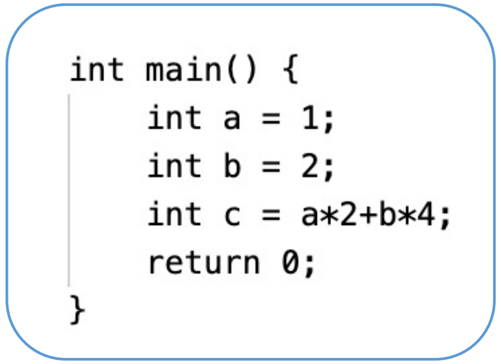
Outline

1. ISA Introduction
2. RISC-V Introduction
3. Assembly Introduction

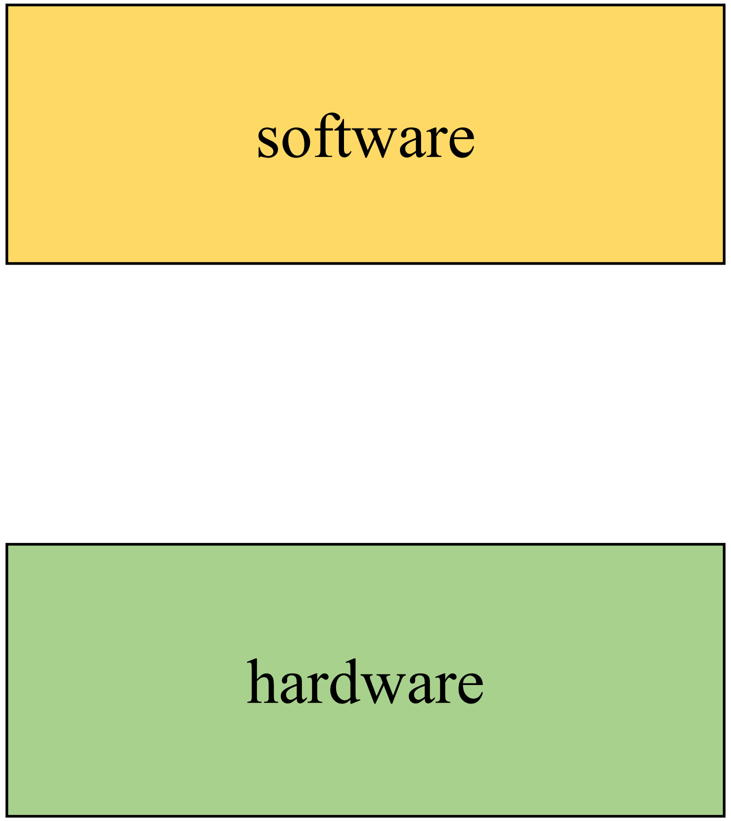
ISA Introduction

How the Hardware executes the C code ?

How can the hardware understand and execute this C code ?

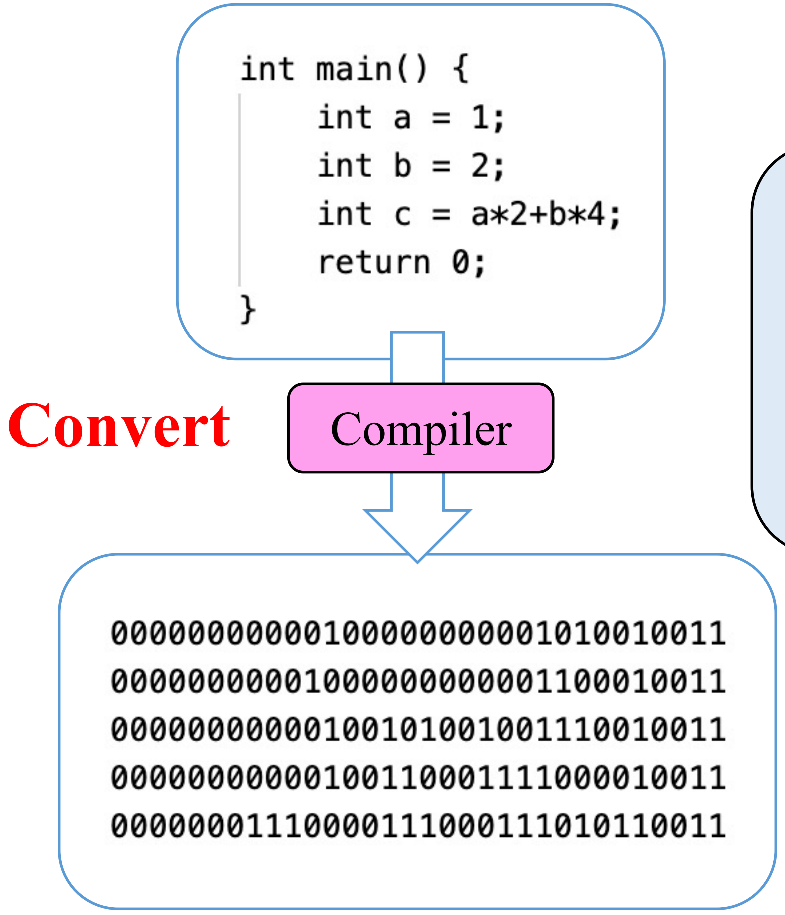


```
int main() {
    int a = 1;
    int b = 2;
    int c = a*2+b*4;
    return 0;
}
```



How the Hardware executes the C code ?

- Convert “C code” to “machine code” **according the instructions that hardware supports**

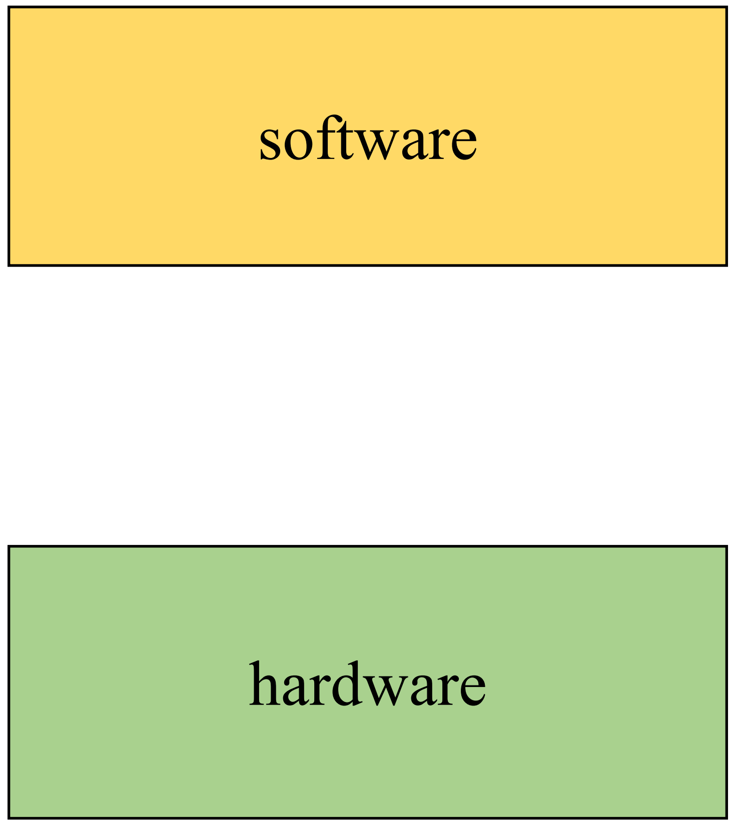


```
int main() {
    int a = 1;
    int b = 2;
    int c = a*2+b*4;
    return 0;
}
```

```
00000000000100000000001010010011
00000000001000000000001100010011
00000000000100101001001110010011
00000000000100110001111000010011
00000001110000111000111010110011
```

Think…

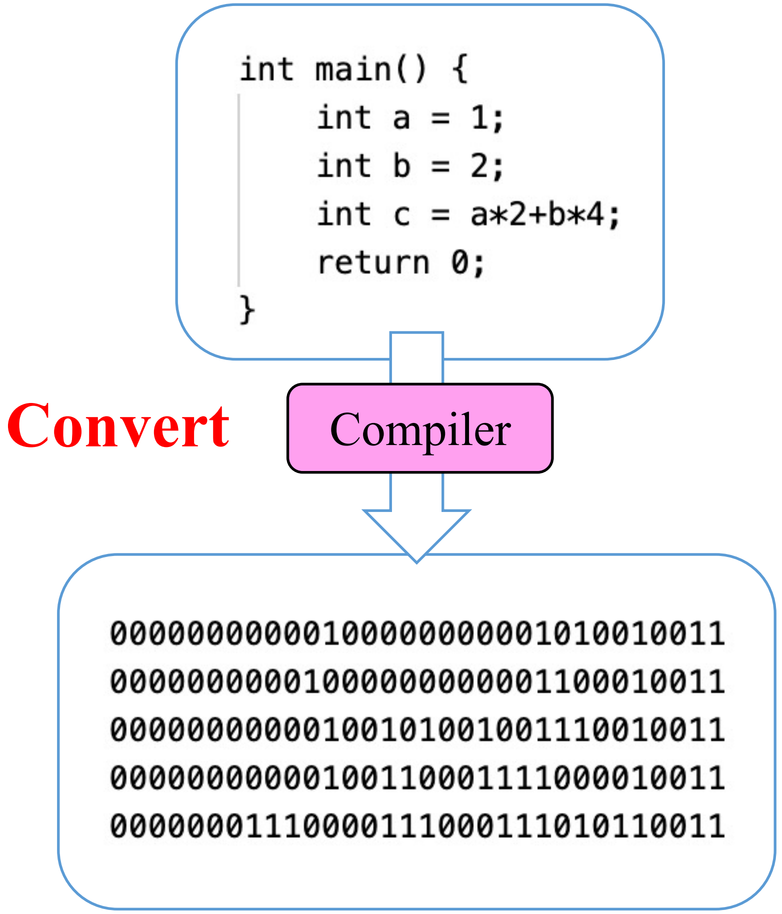
What will happen when different hardware supports different instructions ?



Hardware can only understand “machine instructions” which it supports

How the Hardware executes the C code ?

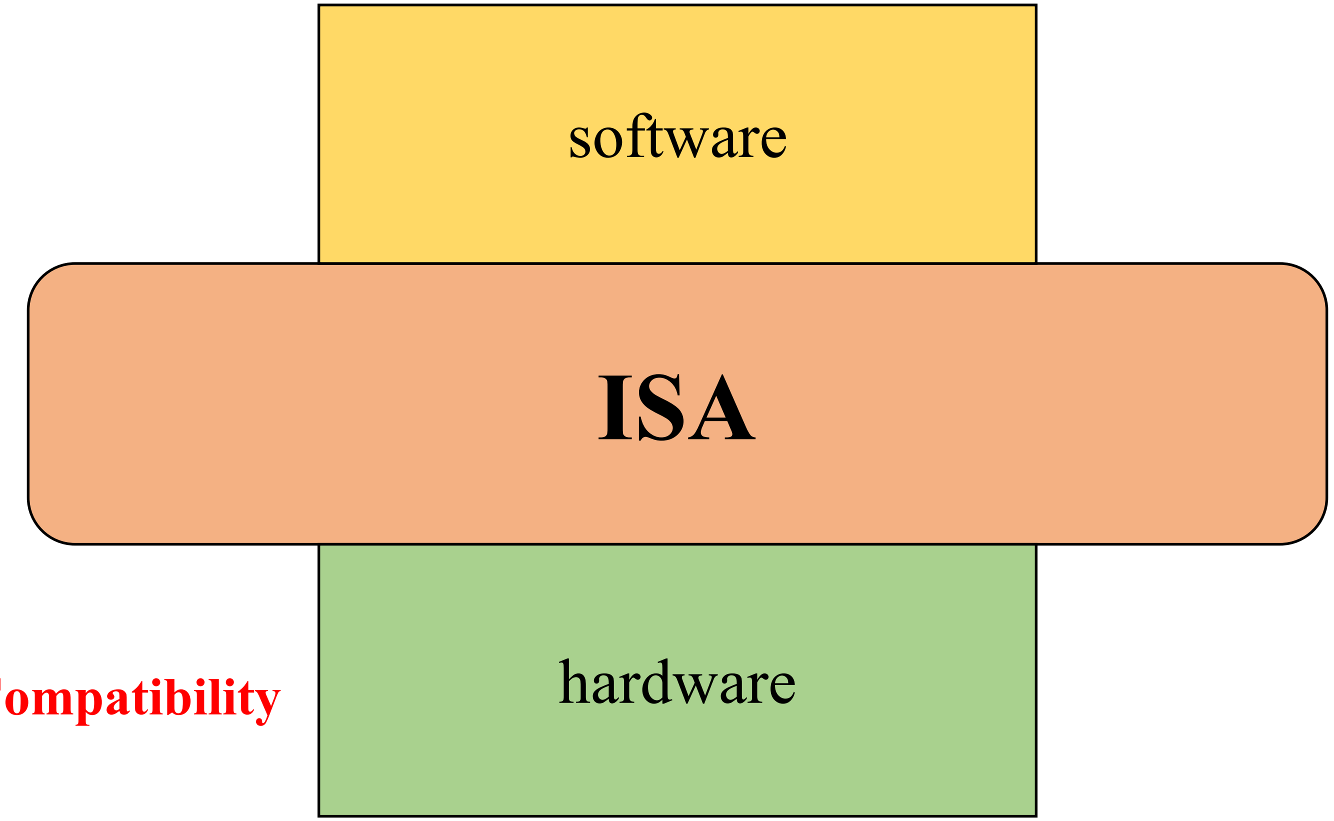
- Convert "C code" to "machine code" **according to ISA specifications**



```
int main() {
    int a = 1;
    int b = 2;
    int c = a*2+b*4;
    return 0;
}
```

```
00000000000100000000001010010011
00000000001000000000001100010011
00000000000100101001001110010011
00000000000100110001111000010011
00000001110000111000111010110011
```

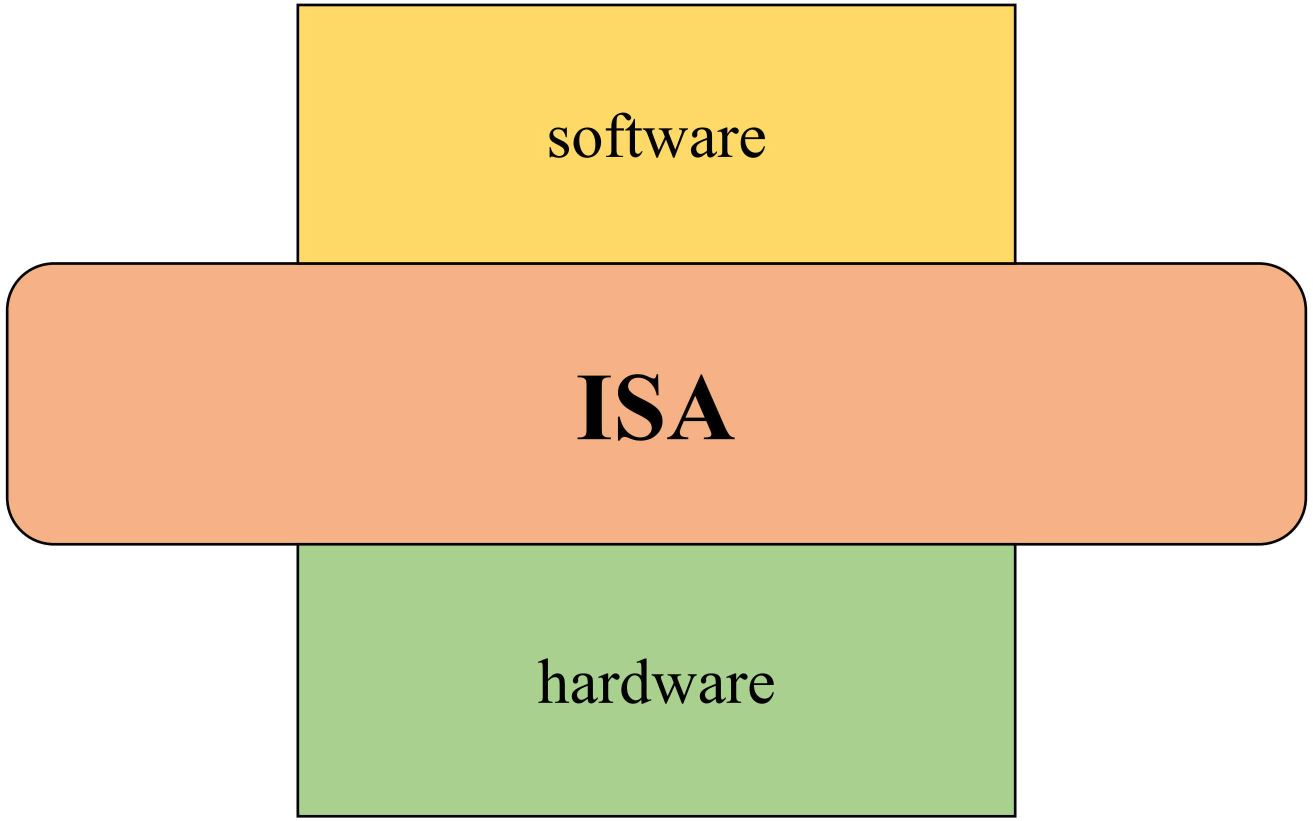
Binary Compatibility



What's ISA

ISA : Instruction-Set Architecture

- It's an **interface** between the hardware and the software

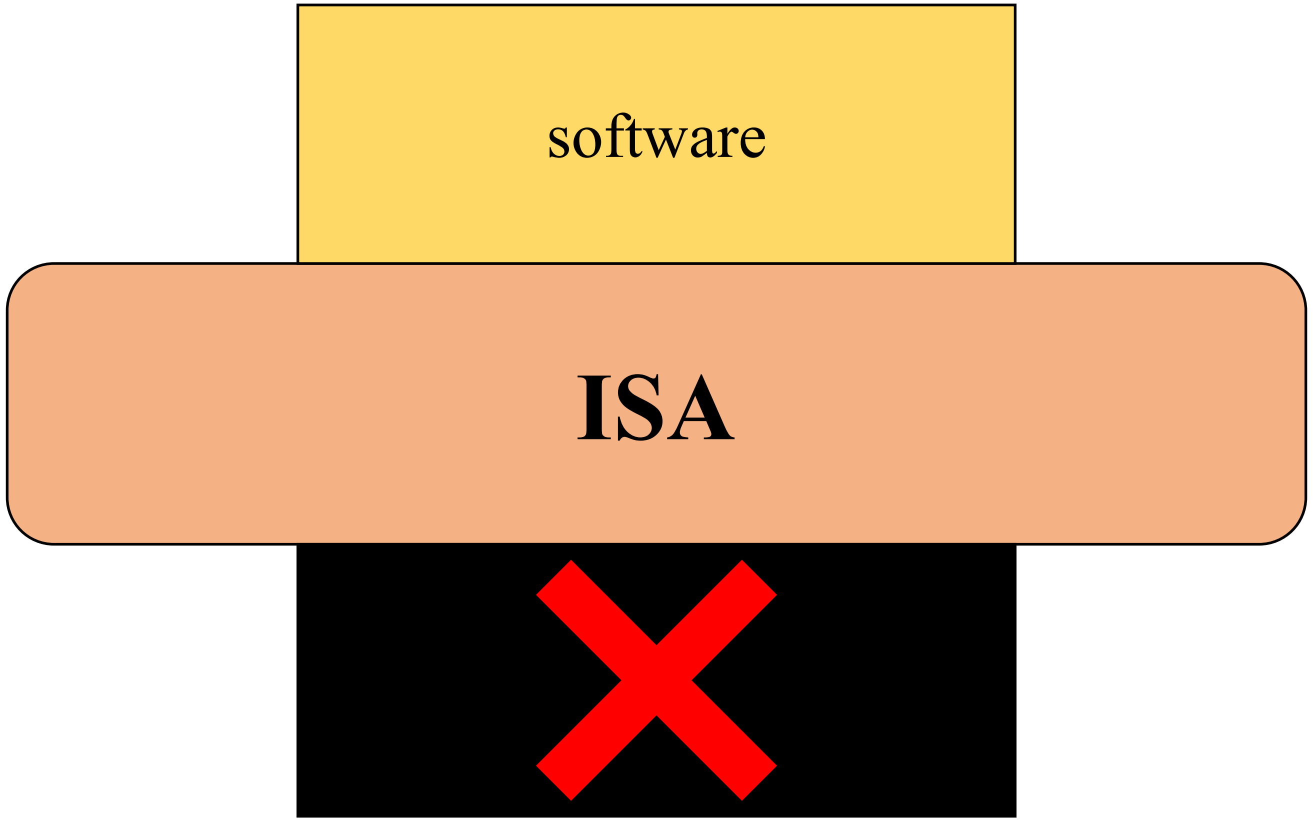


What’s ISA

ISA : Instruction-Set Architecture

- It’s an **interface** between the hardware and the software
- It’s **a set of instructions** that defines how the hardware is controlled by the software
- **For Software**, it’s an abstraction of the hardware

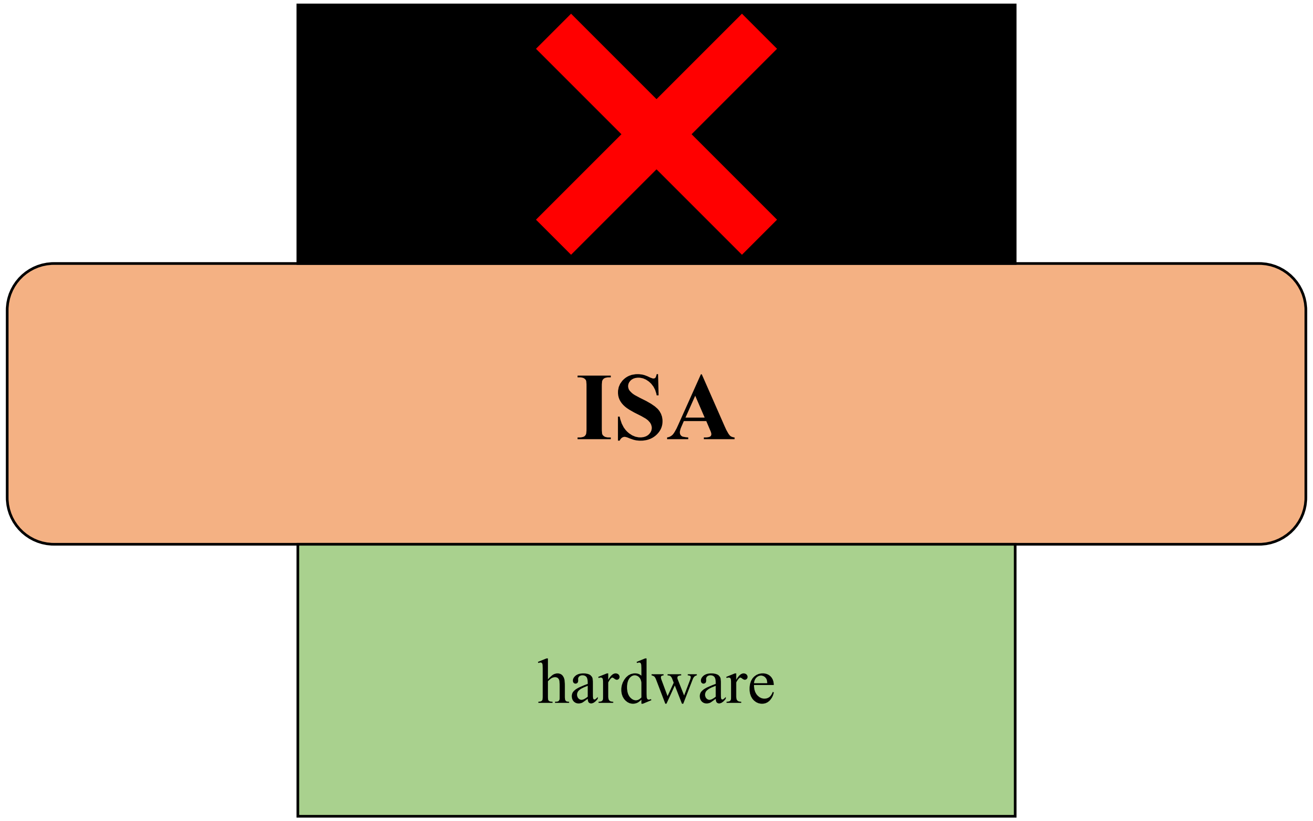
```
00000000000100000000001010010011
00000000001000000000001100010011
00000000000100101001001110010011
00000000000100110001111000010011
00000001110000111000111010110011
```



What's ISA

ISA : Instruction-Set Architecture

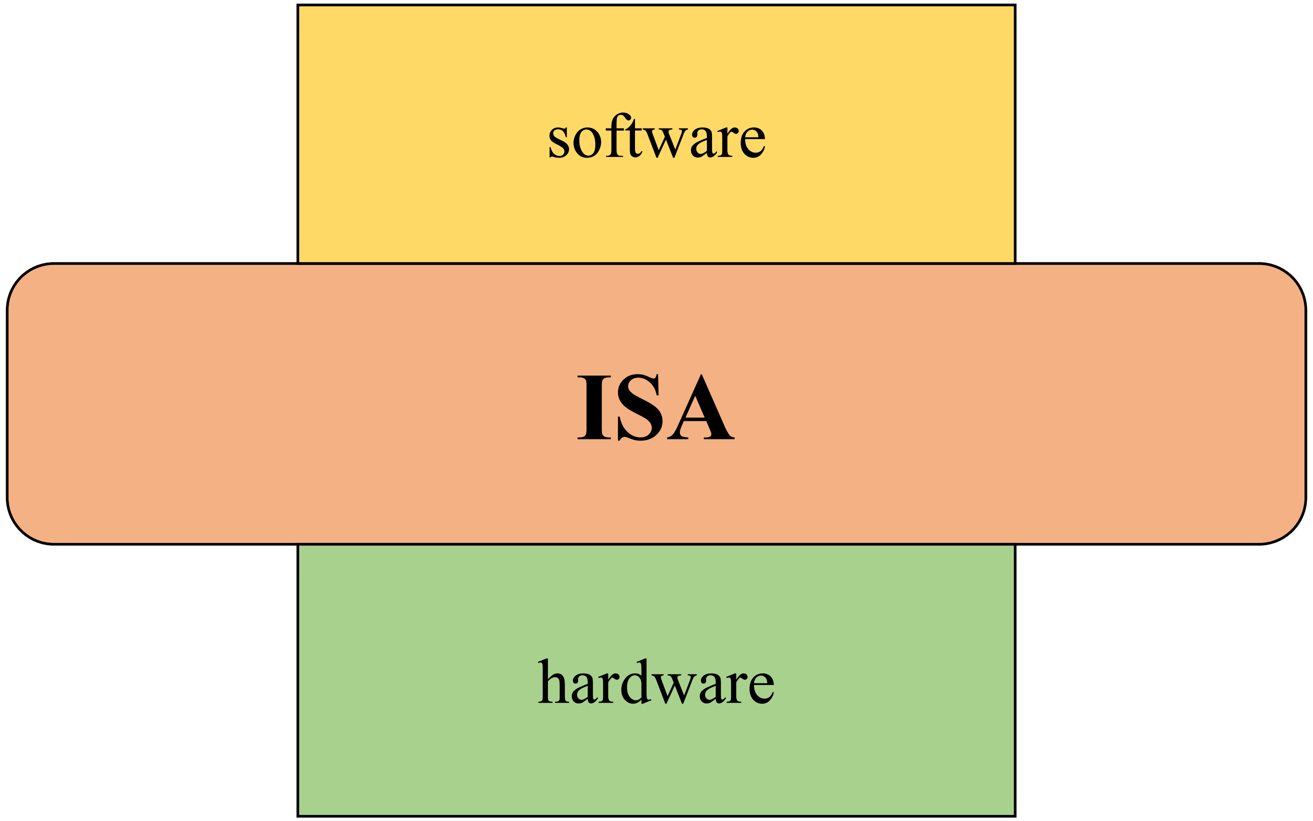
- It's an **interface** between the hardware and the software
- It's **a set of instructions** that defines how the hardware is controlled by the software
- **For Software**, it's an abstraction of the hardware
- **For Hardware**, it's an implementation target



What’s ISA

ISA : Instruction-Set Architecture

- It’s an **interface** between the hardware and the software
- It’s **a set of instructions** that defines how the hardware is controlled by the software
- **For Software**, it’s an abstraction of the hardware
- **For Hardware**, it’s an implementation target
- ISA can be viewed as a **programmer’s manual** which is referenced by the assembly language programmer and the compiler writer.

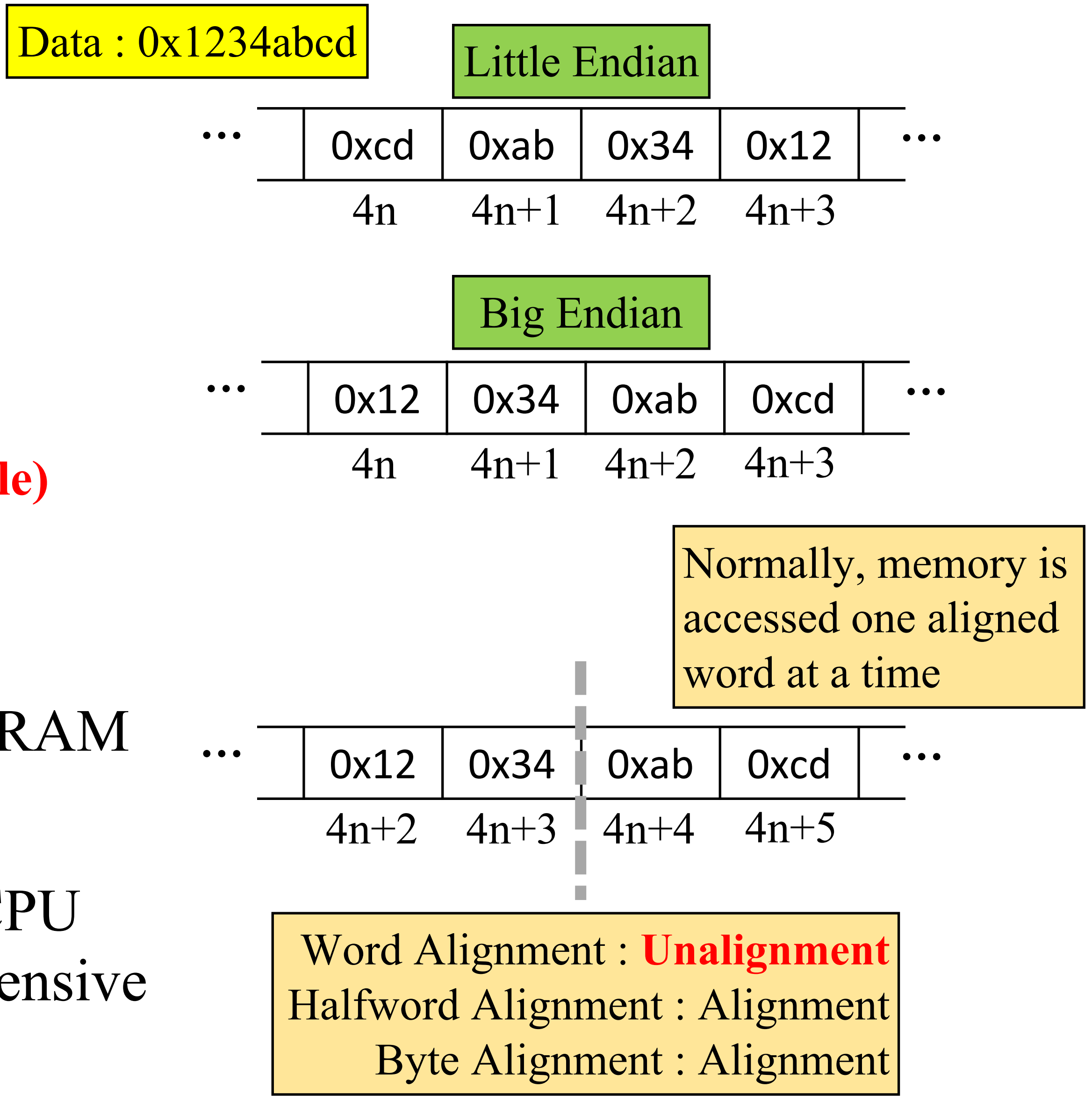


What’s ISA

Besides defines Instruction Set…

ISA also defines

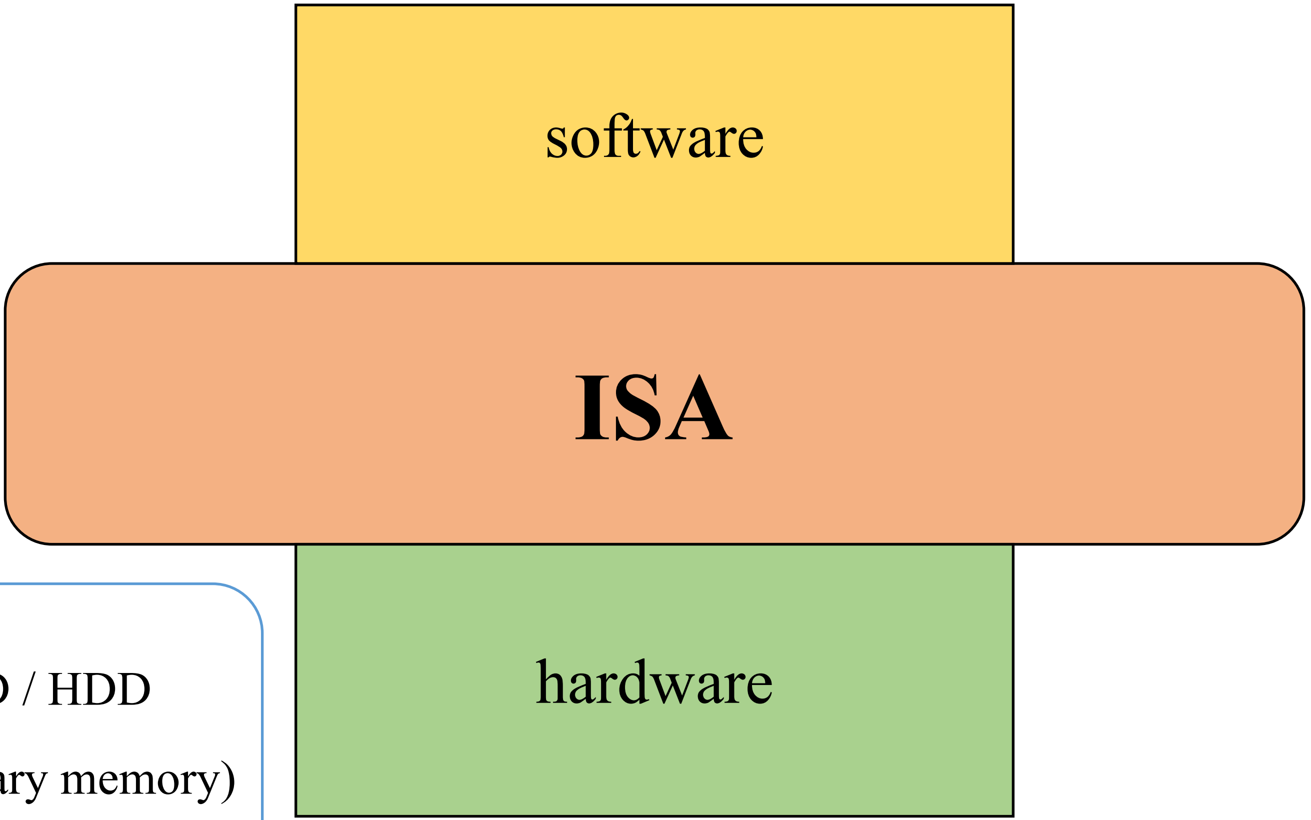
- **Data types**
- **Main Memory (endian / alignment …)**
  - RAM (Random Access Memory) **(volatile)**
  - Store Data & Instructions
- **Addressing Modes**
  - Define how to get the address to access RAM
- **Registers (amounts / width)**
  - They are container to store data inside CPU
  - They are faster than RAM but more expensive
- …

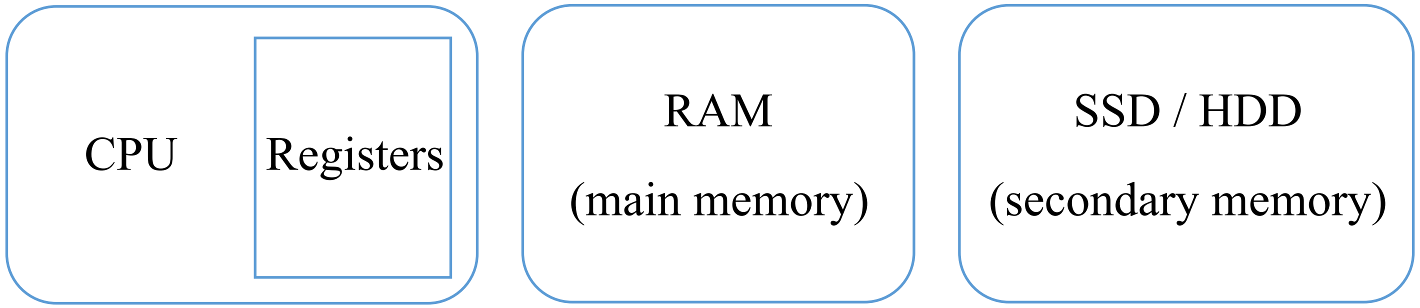


Now, we have …

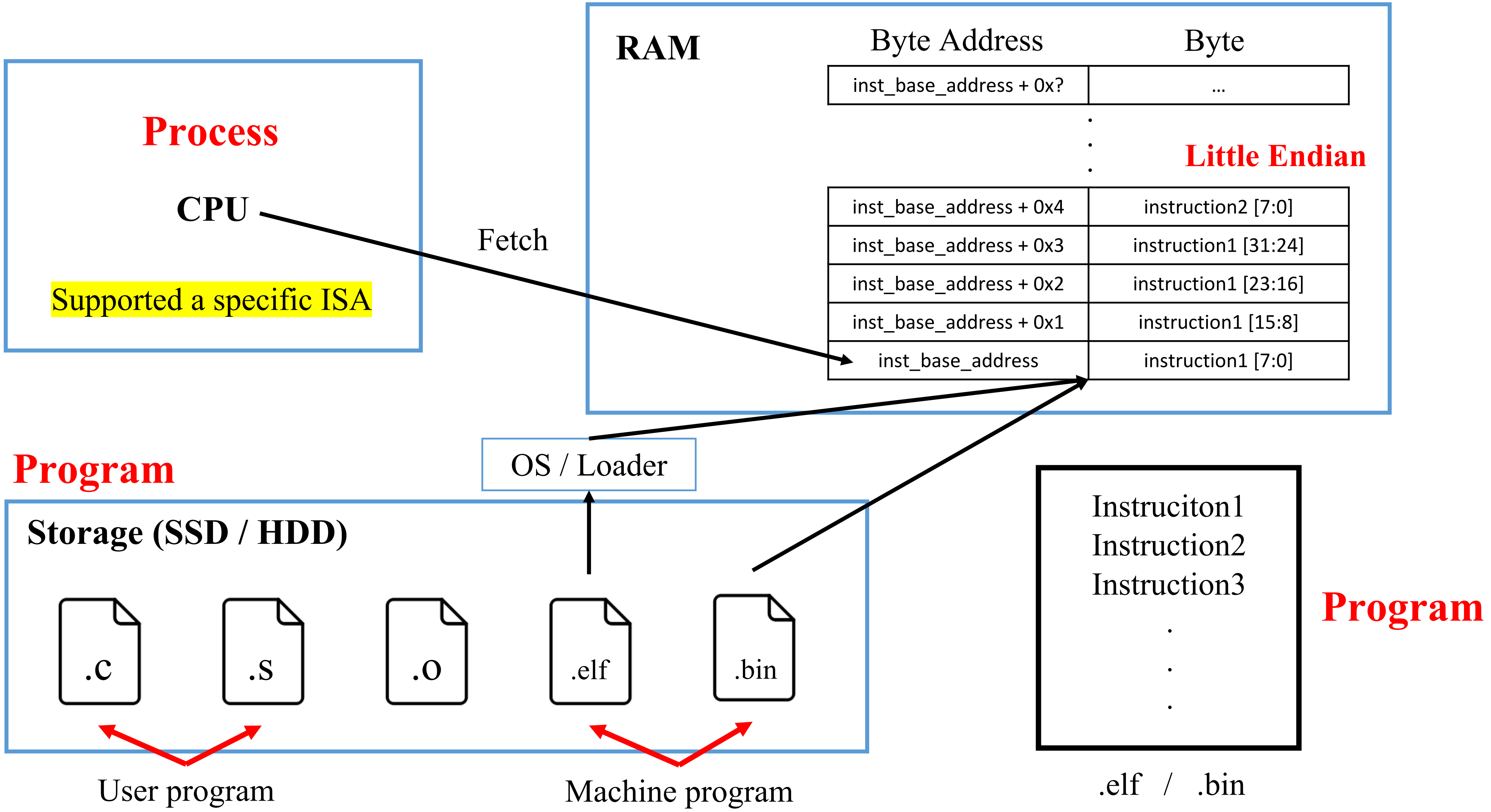
```
00000000000100000000001010010011
00000000001000000000001100010011
00000000000100101001001110010011
00000000000100110001111000010011
00000001110000111000111010110011
```

How a CPU execute a program ?



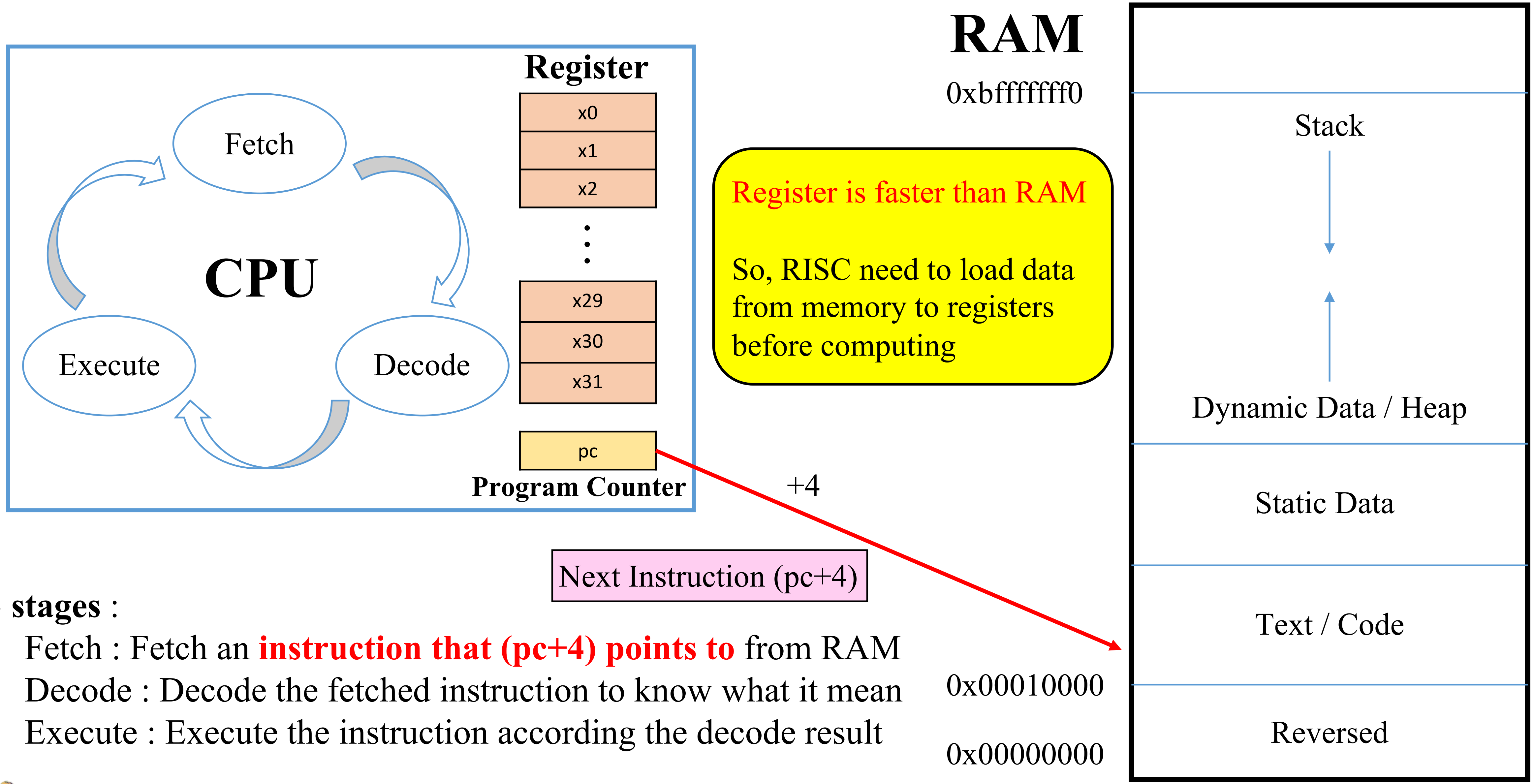


How a CPU execute a program ?



Byte Address	Byte
inst_base_address + 0x?	...
.	
.	
.	
inst_base_address + 0x4	instruction2 [7:0]
inst_base_address + 0x3	instruction1 [31:24]
inst_base_address + 0x2	instruction1 [23:16]
inst_base_address + 0x1	instruction1 [15:8]
inst_base_address	instruction1 [7:0]

Register & Main Memory



Register is faster than RAM

So, RISC need to load data from memory to registers before computing

Next Instruction (pc+4)

3 stages :

- Fetch : Fetch an **instruction that (pc+4) points to** from RAM
- Decode : Decode the fetched instruction to know what it mean
- Execute : Execute the instruction according the decode result

RISC-V Introduction

History of ISA : RISC versus CISC

- The instruction set of processors can be simply divided into two types, RISC (reduced instruction set computer) and CISC (complex instruction set computer).
- In the beginning, there was no distinction between RISC and CISC. At that time, the technology of compilers was not mature, and programs were written directly in machine code or assembly languages.
  In order to reduce the design time of programmers, a single instruction code with complex operations was gradually developed, so that programmers only had to write simple instructions.
- The research indicates that only about 20% of the instructions in the entire instruction set are often used, accounting for about 80% of the program; the remaining 80% of the instructions account for only 20% of the program.
- Because the more instructions supported will make the circuit more complex and increase the cost and energy consumption. In 1979, Professor David Patterson of the University of California, Berkeley, proposed the idea of RISC, suggesting that hardware should focus on accelerating commonly used instructions, while more complex instructions should be combined with commonly used instructions, and then divided into RISC and CISC.
- However, modern processors are not clearly categorized as CISC or RISC.

Comparison of RISC & CISC

	RISC (Reduced Instruction Set Computer)	**CISC (Complex Instruction Set Computer)**
Instruction Count	Less (Reduced)	Plentiful (Complex)
Instruction Ability	Simplicity and Less ability (1 operation per instruction)	More complex and strong ability (> 1 operation per instruction)
Instruction Length	Fixed -> easy for CPU to decode	Variable
CPU Microarchitecture	Simple Design easy Debug easy Low power	Complex Design hard Debug hard High energy consumption
Addressing Mode	Supports only a few types	Supports multiple types
Operands	Registers or Immediate (main memory is slow, register is fast) Data in memory needs to be loaded into registers before processing	Register or Immediate or Memory Data in memory can be processed directly without loading into registers
Code size	Large (complex instructions = many small instructions)	Short (complex calc = small amounts of instructions)
Example	MIPS, ARM, RISC-V	X86

RISC-V - Introduction

- RISC-V (pronounced "risk-five") is started by students from UC Berkeley in May 2010 as part of the Parallel Computing Laboratory (Par Lab), of which Prof. David Patterson was Director.

- RISC-V is a free and Open ISA.
  Open ISA delivers easier support from a broad range of operating systems, software vendors and tool developers.

- The conventional approach to computer architecture is incremental ISAs, where new processors must implement not only new ISA extensions but also all extensions of the past.

- RISC-V is modular. At the core is a base ISA, called RV32I, which will never change.
  The modularity comes from optional standard extensions that hardware can include or not depending on the needs of the application.

- The goal of the RISC-V Foundation is to maintain the stability of RISC-V, evolve it slowly and carefully, solely for technical reasons, and try to make it as popular for hardware as Linux is for operating systems.

RISC-V - Feature

- **Open**
  RISC-V ISA is provided under open source licenses that do not require fees to use.
  Deliver easier support from a broad range of operating systems, software vendors and tool developers.

- **Modulization & Extensibility**
  RISC-V allows users to select the modules they need to add as extensions to meet the needs from low power consumption to high performance. This also reduces the cost of redundant instruction support.

- **Stable**
  Base and first standard extensions are already ratified. There is no need to worry about updates.

- **Simple**
  Only few number of instructions in RISC-V.

- **Elegant**
  - Each instruction is the same length.
  - Instruction format is fixed.
  - The signed bit is always on the leftmost side of the instruction.

RISC-V - ISA

We learn this base ISA this time

- Format : **one base + optional extensions**
- Naming Convention (規範)

extension has order convention

RISC-V defines the order of ISA Subset：

XLEN

RV [32, 64, 128]

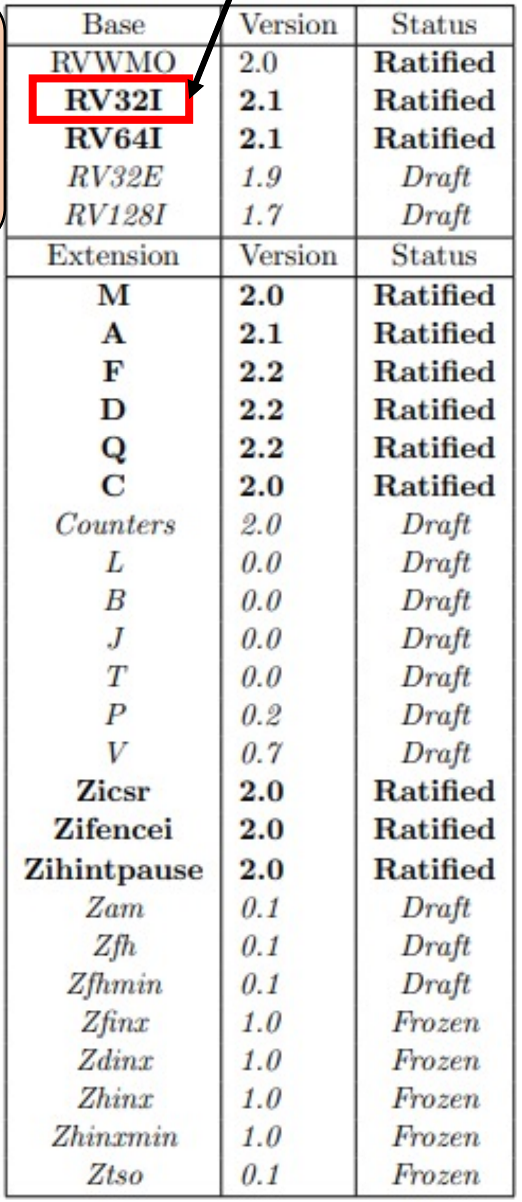
Subset

I , E , M , A , F , D , G , Q , C , B , K , J , P ,V

base

Must choose one

- For Example
  ① RV32IMACV ✓
  ② RV32IMAVC ✗



# Registers	Base	Version	Status	
	RVWMO	2.0	**Ratified**	Weak Memory Ordering
32	**RV32I**	**2.1**	**Ratified**	Base Integer Instruction Set, 32-bit
32	**RV64I**	**2.1**	**Ratified**	Base Integer Instruction Set, 64-bit
16	*RV32E*	*1.9*	*Draft*	Base Integer Instruction Set (embedded), 32-bit
32	*RV128I*	*1.7*	*Draft*	Base Integer Instruction Set, 128-bit

Extension	Version	Status	
M	**2.0**	**Ratified**	Integer Multiplication and Division
A	**2.1**	**Ratified**	Atomic Instructions
F	**2.2**	**Ratified**	Single-Precision Floating-Point
D	**2.2**	**Ratified**	Double-Precision Floating-Point
Q	**2.2**	**Ratified**	
C	**2.0**	**Ratified**	Compressed Instructions
Counters	*2.0*	*Draft*	
L	*0.0*	*Draft*	
B	*0.0*	*Draft*	
J	*0.0*	*Draft*	
T	*0.0*	*Draft*	
P	*0.2*	*Draft*	
V	*0.7*	*Draft*	
Zicsr	**2.0**	**Ratified**	
Zifencei	**2.0**	**Ratified**	
Zihintpause	**2.0**	**Ratified**	
Zam	*0.1*	*Draft*	
Zfh	*0.1*	*Draft*	
Zfhmin	*0.1*	*Draft*	
Zfinx	*1.0*	*Frozen*	
Zdinx	*1.0*	*Frozen*	
Zhinx	*1.0*	*Frozen*	
Zhinxmin	*1.0*	*Frozen*	
Ztso	*0.1*	*Frozen*	

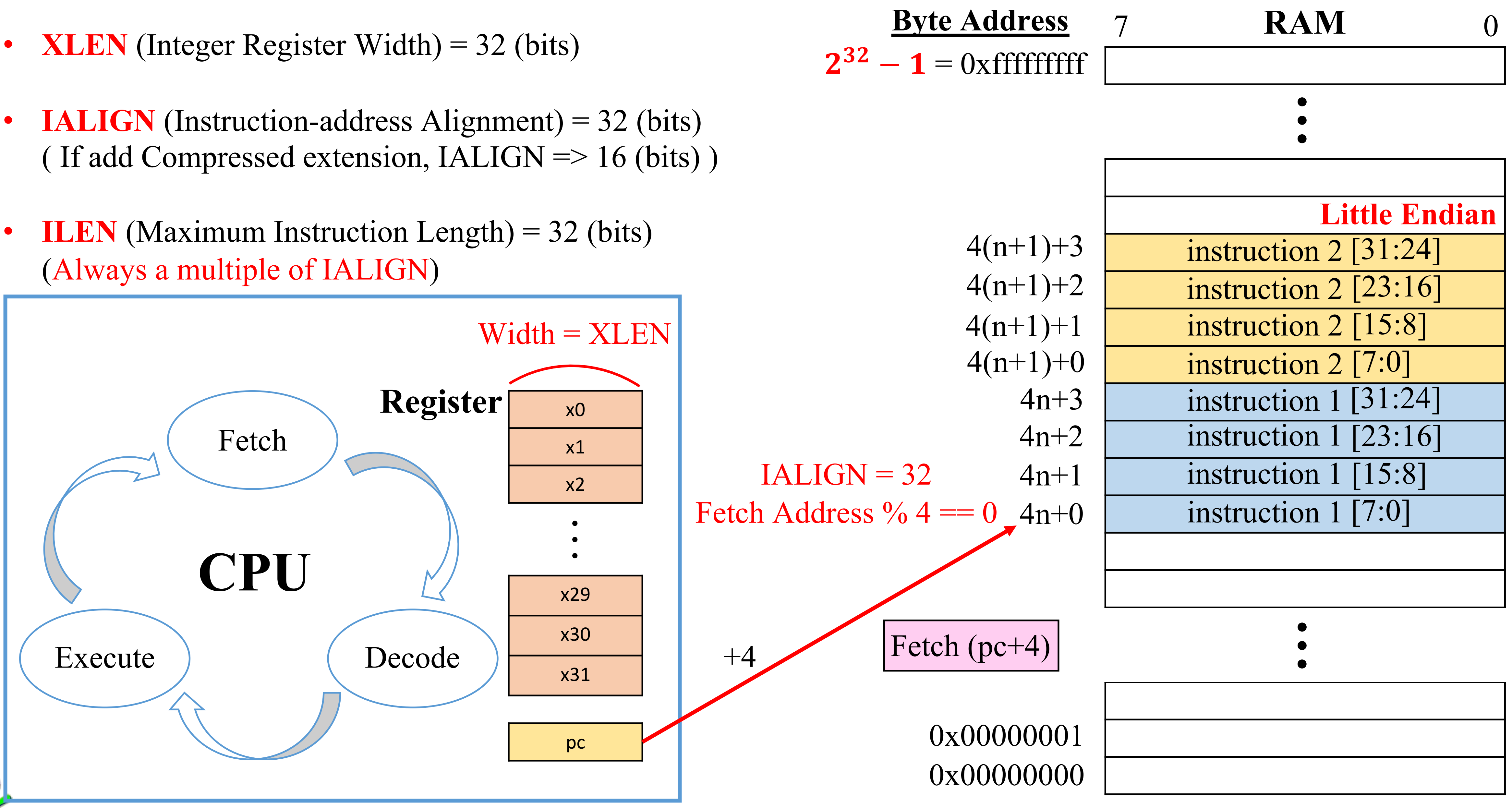
XLEN = 32 / 64 / 128
IALIGN = 16 / 32
ILEN = 16 / 32 / 48 / 64 / …

Base ISA

Ratified	已被批准
Draft	預計在批准前會有變化
Frozen	被批准之前預計不會有重大變化

RISC-V - RV32I

- **XLEN** (Integer Register Width) = 32 (bits)
- **IALIGN** (Instruction-address Alignment) = 32 (bits)
  (If add Compressed extension, IALIGN => 16 (bits))
- **ILEN** (Maximum Instruction Length) = 32 (bits)
  (Always a multiple of IALIGN)



RISC-V - ISA Questions !

Q1 :
Why IALIGN of RV32I is 32 bits ?
Why IALIGN need to support 16 bits ?
Why IALIGN have no greater than 32 bits (e.g. 64 / 128 bits) ?

RISC-V - Register

Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

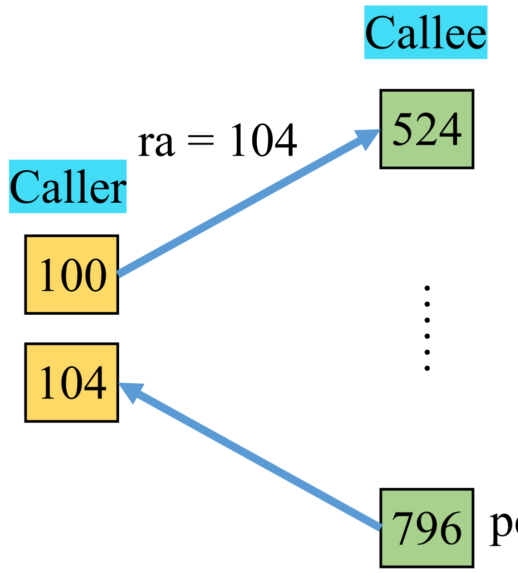
Additional Register : **Program Counter**
points to the last fetched instruction

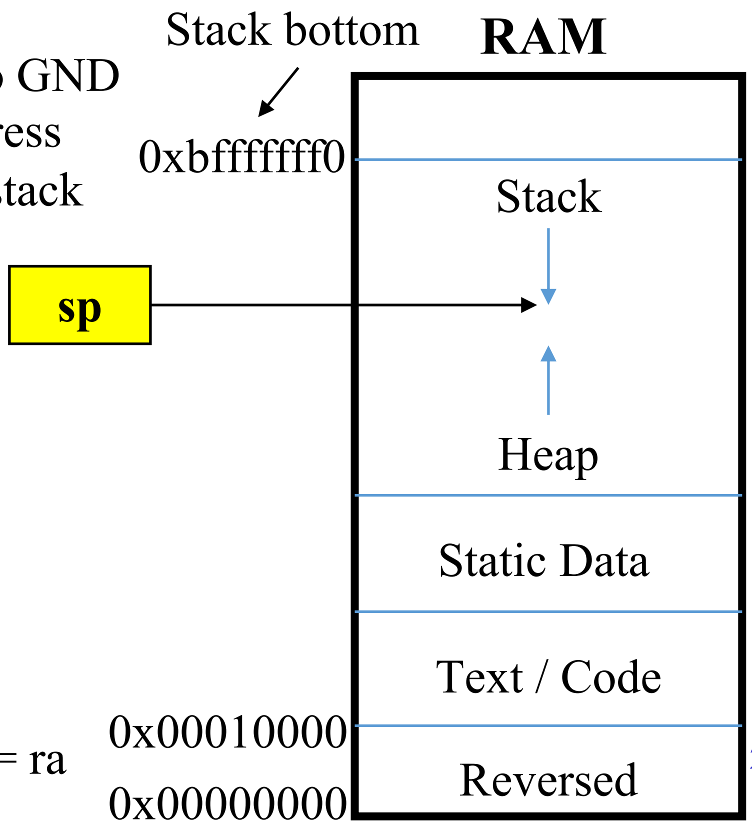
The same

```
add   x5, x9, x17
add   t0, s1, a7
```

RV32I

- 32 registers, each has 32 bits
- You can use these registers at will when you write your own programs. It will work fine.
- **But if you need to use others code or co-work with others, then you need to follow the convention of RISC-V !!!**
- x : integer registers
  - x0 : always 0, connect to GND
  - x1 (ra) : save return address
  - x2 (sp) : point to top of stack





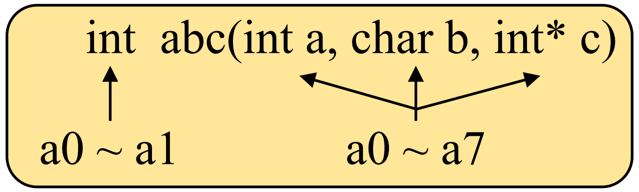
RISC-V - Register

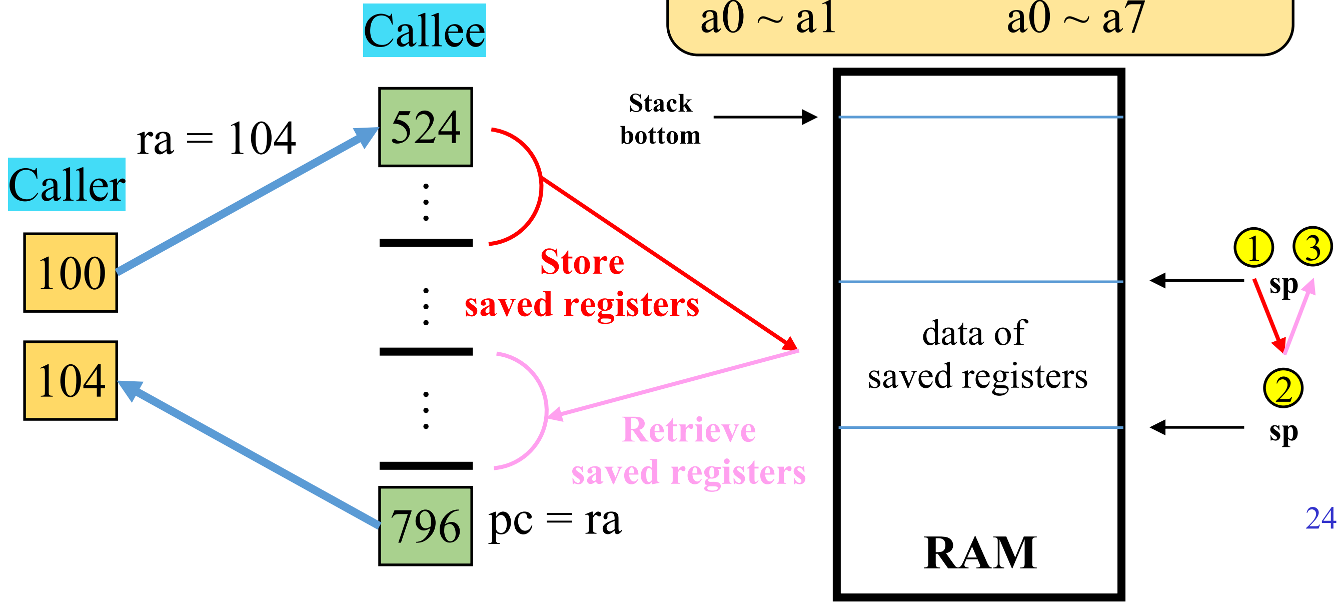
Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

Additional Register : **Program Counter**
points to the last fetched instruction

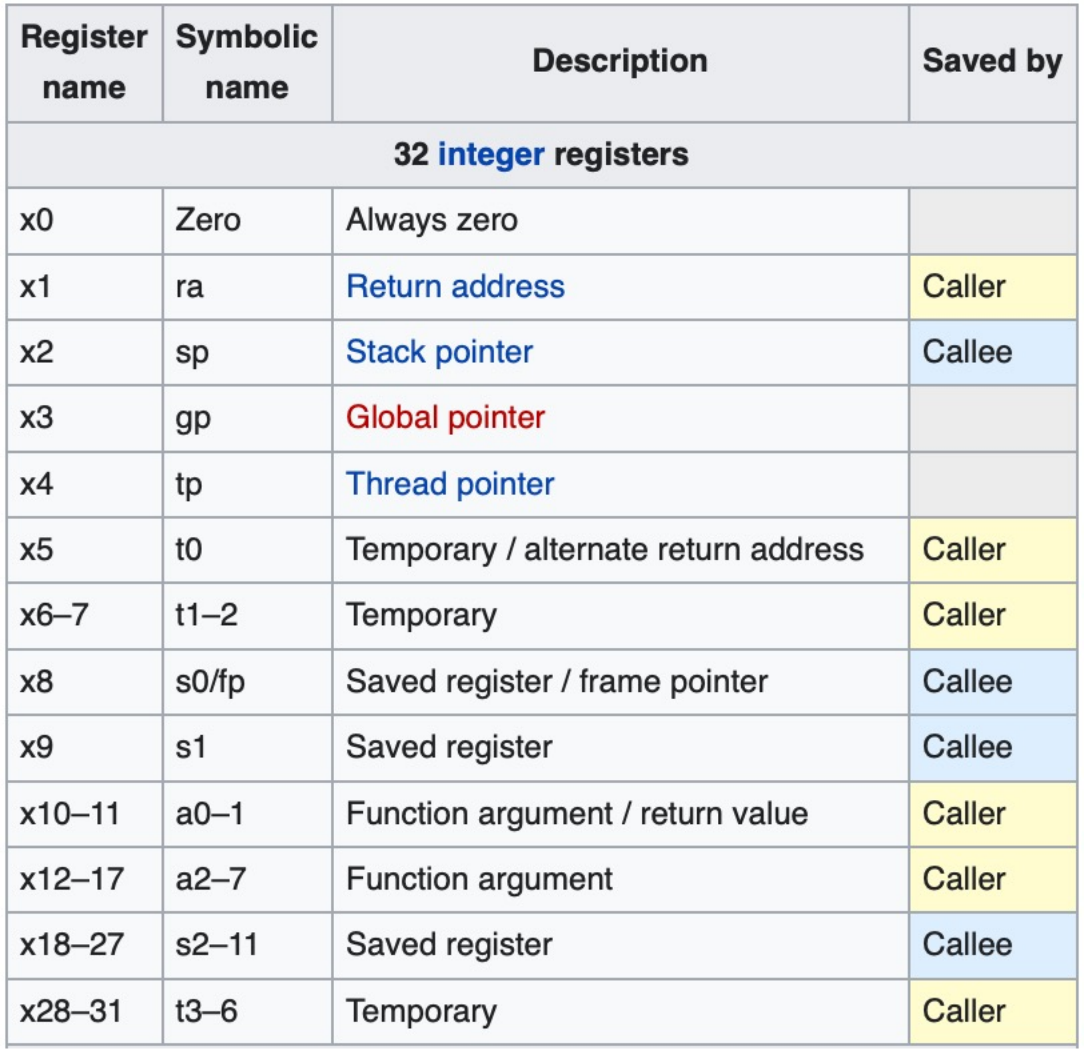
RV32I

- t0 ~ t6 : temporary register
  - **Caller saved** : The programmer does not guarantee that the register will not be changed after the call function returns
- s0 ~ s11 : saved register
  - **Callee saved** : The programmer guarantee that the register will not be changed after the call function returns
- a0 ~ a7 : argument register





RISC-V - Register Questions !

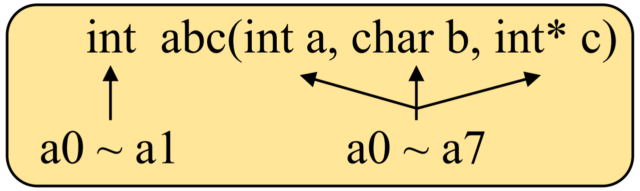


Register name	Symbolic name	Description	Saved by
32 integer registers			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

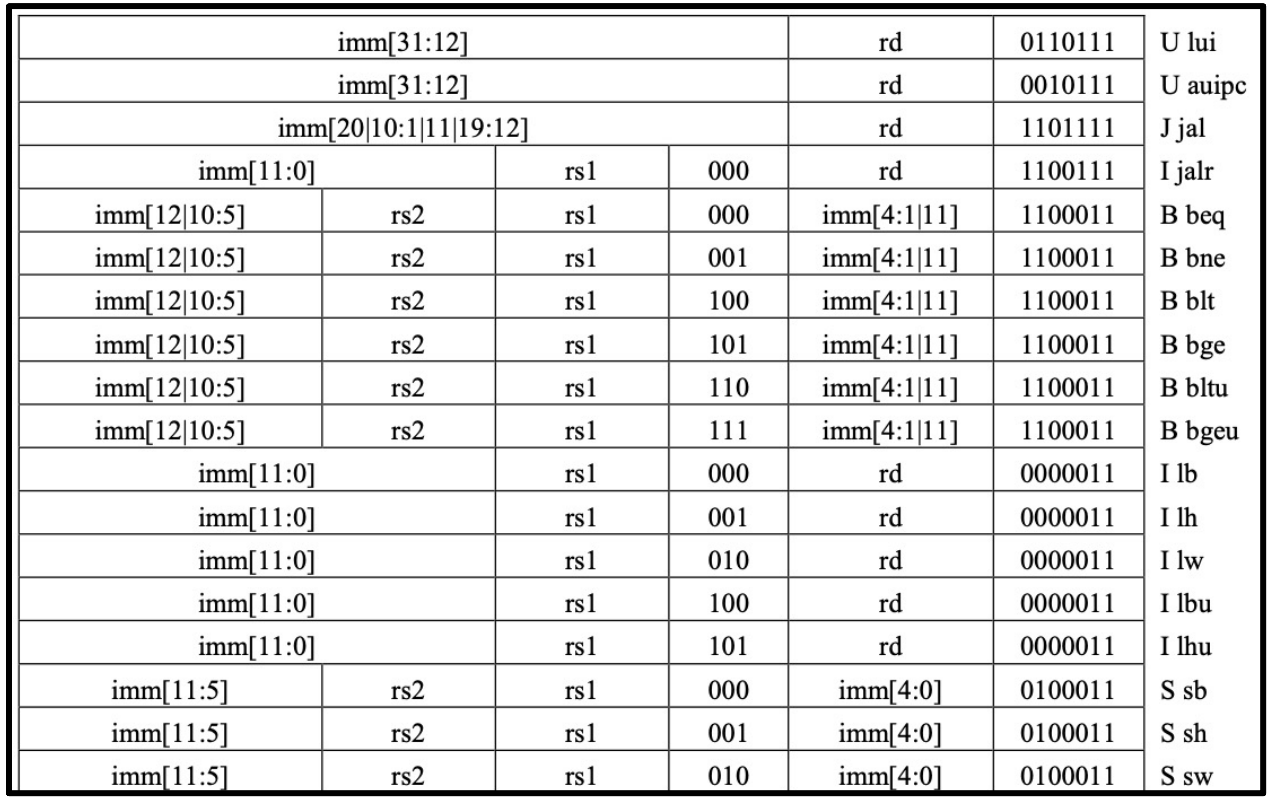
Additional Register : **Program Counter**
points to the last fetched instruction

Q2 :
Why temporary registers and saved registers are not numbered sequentially ?

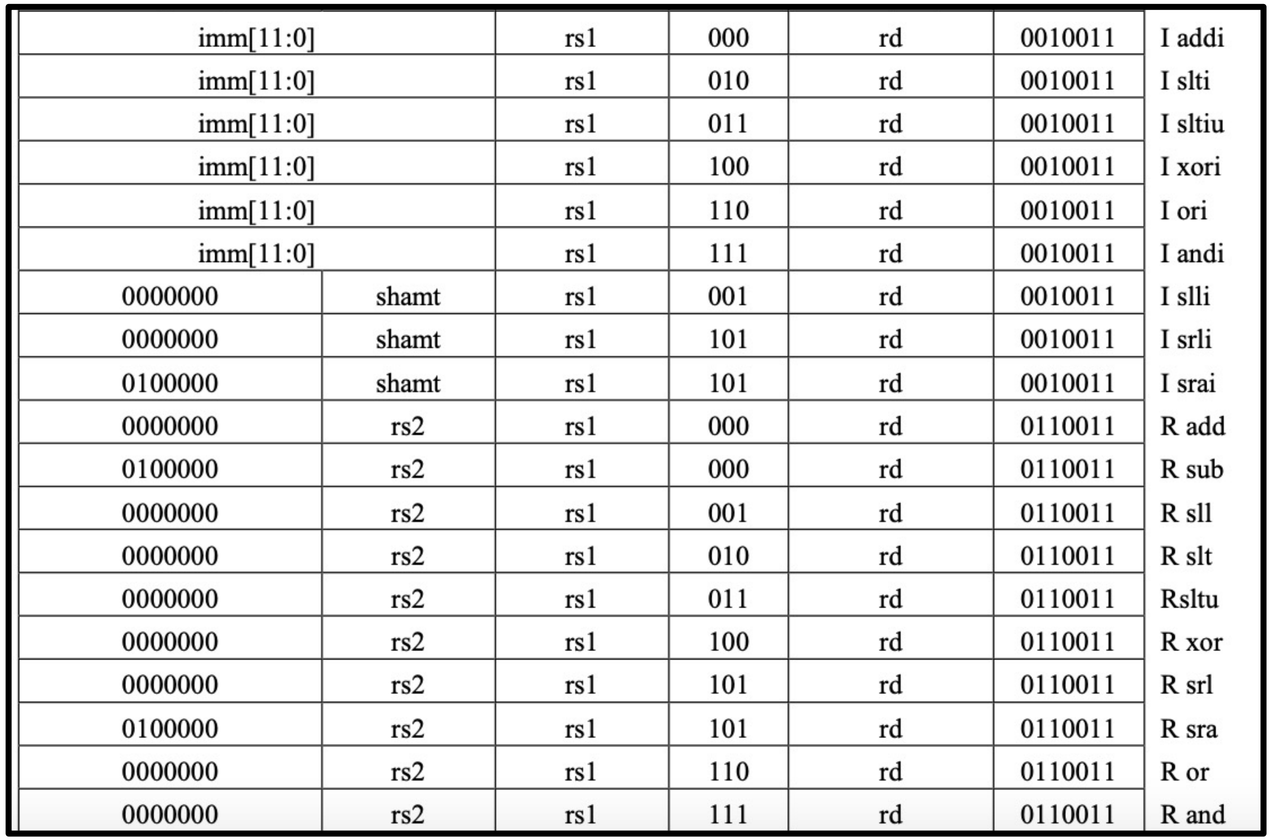
Q3 :
Why return value needs 2 registers (a0, a1) ?



RISC-V - Overview of RV32I Instructions



imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw



imm[11:0]		rs1	000	rd	0010011	I addi
imm[11:0]		rs1	010	rd	0010011	I slti
imm[11:0]		rs1	011	rd	0010011	I sltiu
imm[11:0]		rs1	100	rd	0010011	I xori
imm[11:0]		rs1	110	rd	0010011	I ori
imm[11:0]		rs1	111	rd	0010011	I andi
0000000	shamt	rs1	001	rd	0010011	I slli
0000000	shamt	rs1	101	rd	0010011	I srli
0100000	shamt	rs1	101	rd	0010011	I srai
0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu
0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

RISC-V - Instruction Formats

- According to the used source & destination registers, we can divide the RISC-V ISA to 4 types of formats

4 types : R / I / S / U

R : rd / rs1 / rs2	(Register)
I : rd / rs1	(Immediate)
S : rs1 / rs2	(Store)
U : rd	(Upper)

```
int  a = 10
int  b = a + 1
int  c = a + b
```

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

- In advance, according the bit position of immediate, we can extend it to 6 types of formats (4 + 2 variants)

6 types : R / I / S / (B) / U / (J)

	S : rs1 / rs2 / imme[11:0]
(Branch)	(B) : rs1 / rs2 / imme[12:1]
	U : rd / imme[31:12]
(Jump)	(J) : rd / imme[20:1]

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

RISC-V - Instruction Formats

In order to simplify hardware implementation

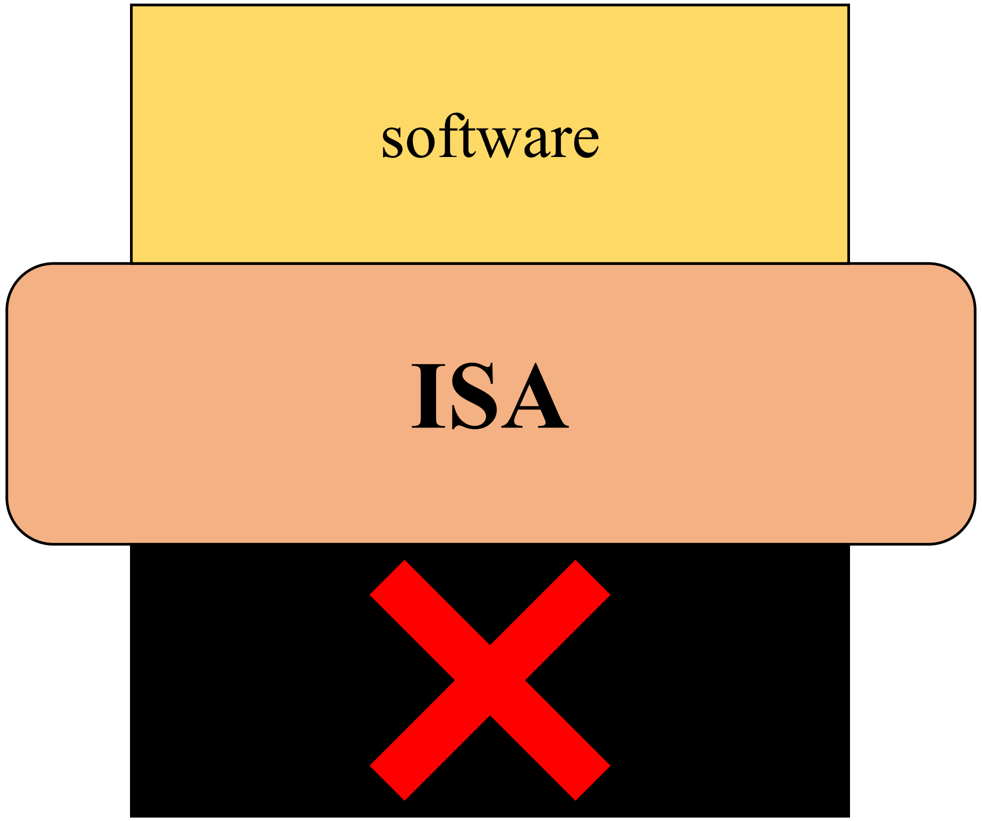
- Fixed the position of opcode / func3 / func7
- Fixed the position of used register (rs1 / rs2 / rd)
- Signed bits of immediate is always at inst[31]
- Let the bits of the immediate overlap as much as possible

opcode : the operation of the instruction
func3 : support opcode to express the operation
func7 : support opcode to express the operation
rs1 : source register 1
rs2 : source register 2
rd : destination register

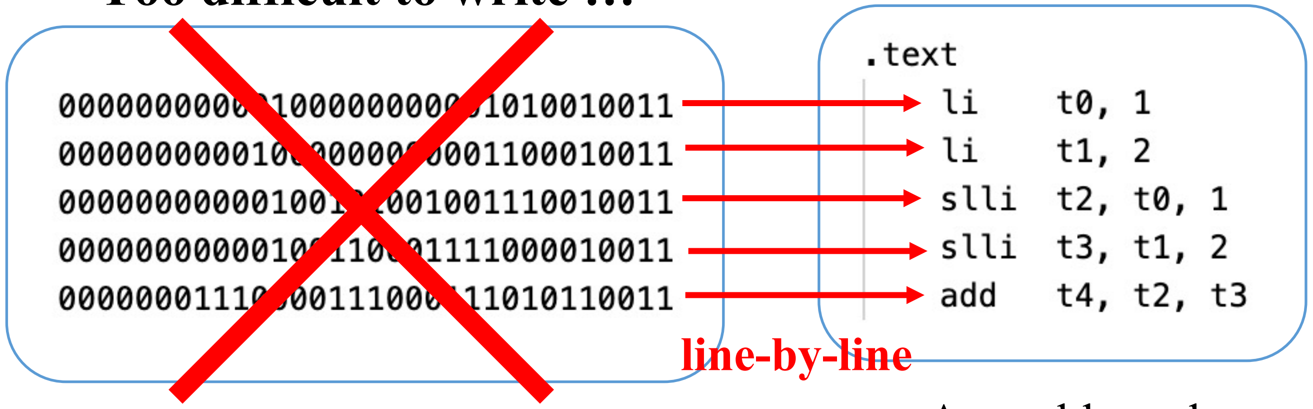
Formats	32 Bits (RV32I)																															
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R type	func7							rs2					rs1					func3			rd					opcode						
I type	imme[11:0]												rs1					func3			rd					opcode						
S type	imme[11:5]							rs2					rs1					func3			imme[4:0]					opcode						
B type	{ imme[12], imme[10:5] }							rs2					rs1					func3			{ imme[4:1], imme[11] }					opcode						
U type	imme[31:12]																				rd					opcode						
J type	{ imme[20], imme[10:1], imme[11], imme[19:12] }																				rd					opcode						

Assembly Introduction

What’s Assembly Language



Too difficult to write !!!



```
00000000000100000000001010010011
00000000001000000000001100010011
00000000000100101001001110010011
00000000001000110001111000010011
00000001110000111000111010110011
```

```
.text
    li    t0, 1
    li    t1, 2
    slli  t2, t0, 1
    slli  t3, t1, 2
    add   t4, t2, t3
```

Machine Code

Assembly code

Machine code is difficult to write and has poor readability.

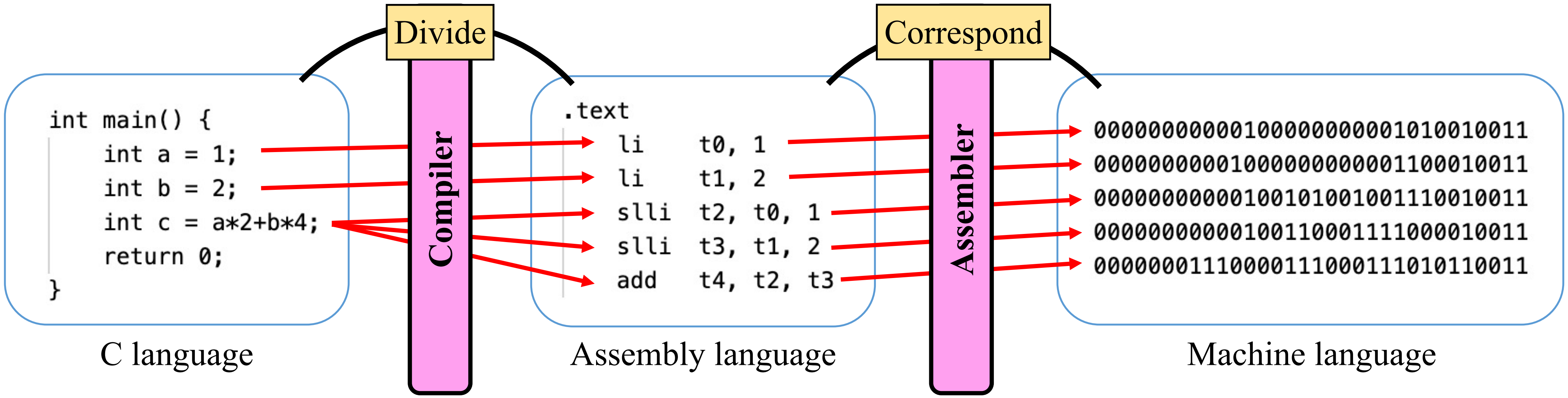
Assembly Language

- A type of low-level programming language
- Correspond almost line by line with machine code
- Communicate directly with a computer’s hardware
- Readable by humans
- Also Often known as “symbolic machine code”

Why we need to learn assembly code

Reference : https://www.techopedia.com/why-is-learning-assembly-language-still-important/7/32268

- Despite the prevalence(流行度) of high-level languages that are mainly used for the development of applications and software programs, the importance of assembly language in today's world cannot be understated(不可小覷).
- Compared to high-level languages, assembly language is bare(裸露) and transparent(透明).
- It communicates hardware directly and see how the processor and memory work.
- Assembly language is the gateway(途徑) to optimization in speed, thereby offering great efficiency and performance.



```
int main() {
    int a = 1;
    int b = 2;
    int c = a*2+b*4;
    return 0;
}
```

```
.text
    li    t0, 1
    li    t1, 2
    slli  t2, t0, 1
    slli  t3, t1, 2
    add   t4, t2, t3
```

```
00000000000100000000001010010011
00000000001000000000001100010011
00000000000100101001001110010011
00000000000100110001111000010011
00000001110000111000111010110011
```

So, Let us learn how to write assembly language !

```
00000000000100000000001010010011
00000000001000000000001100010011
00000000000100101001001110010011
00000000001000110001111000010011
00000000011100001110001110101110011
```

```
.text
    li    t0, 1
    li    t1, 2
    slli  t2, t0, 1
    slli  t3, t1, 2
    add   t4, t2, t3
```

Signed-extension :

An instruction has only 32 bits.

Take I-type for example :

imm[11:0]	rs1	funct3	rd	opcode	I-type

It can't fully express an immediate when it also need to include other info except immediate.

When CPU decodes the instruction, it will do **signed extension** for immediate.
Extend it to **XLEN** bits.

imme[11:0] = 1100_0010_0000
imme = 1111_1111_1111_1111_1111_1100_0010_0000 (XLEN)

imme[11:0] = 0100_0010_0000
imme = 0000_0000_0000_0000_0000_0100_0010_0000 (XLEN)

Instruction classification of RV32I depends on functionality

	Arithmetic	Set	Shift	Logical
Register – Register	add, **sub**	slt, sltu	sll, srl, sra	xor, or, and
Register – Immediate	addi	slti, sltiu	slli, srli, srai	xori, ori, andi

- Computation Instruction
  - Register – Register (add, **sub**, slt, sltu, sll, srl, sra, xor, or, and)
  - Register – Immediate (addi, slti, sltiu, slli, srli, srai, xori, ori, andi)
  - Long Immediate (lui, auipc)
- Load & Store Instruction
  - Load (lb, lh, lw, lbu, lhu)
  - Store (sb, sh, sw)
- Control Transfer Instruction
  - Unconditional Jump (jal, jalr)
  - Conditional Branch (beq, bne, blt, bge, bltu, bgeu)

Computation Instruction – (Register - Register)

R type operation

- **Arithmetic**
  - add / sub (Add / Subtract)

    rd = rs1 [+-] rs2
- **Set**
  - slt / sltu (Set Less Than (Unsigned))

    slt : rd = (rs1 < rs2) ? 1 : 0

    sltu : rd = $(rs1_{unsigned} < rs2_{unsigned})$? 1 : 0
- **Shift**
  - sll / srl / sra (Shift [Left / Right] [Logic / Arithmetic])

    sll : rd = rs1 << rs2[4:0]

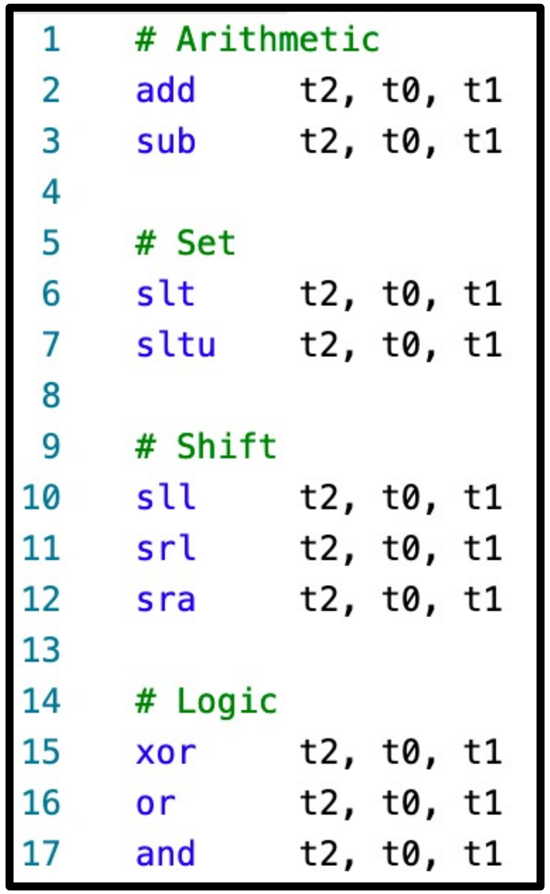
    srl : rd = rs1 >> rs2[4:0]

    sra : rd = rs1 >>> rs2[4:0]
- **Logical**
  - xor / or / and (Exclusive-OR / OR / AND)

    rd = rs1 [^|&] rs2

R type format

Rop rd, rs1, rs2



```
# Arithmetic
add     t2, t0, t1
sub     t2, t0, t1

# Set
slt     t2, t0, t1
sltu    t2, t0, t1

# Shift
sll     t2, t0, t1
srl     t2, t0, t1
sra     t2, t0, t1

# Logic
xor     t2, t0, t1
or      t2, t0, t1
and     t2, t0, t1
```

Computation Instruction – (Register - Immediate)

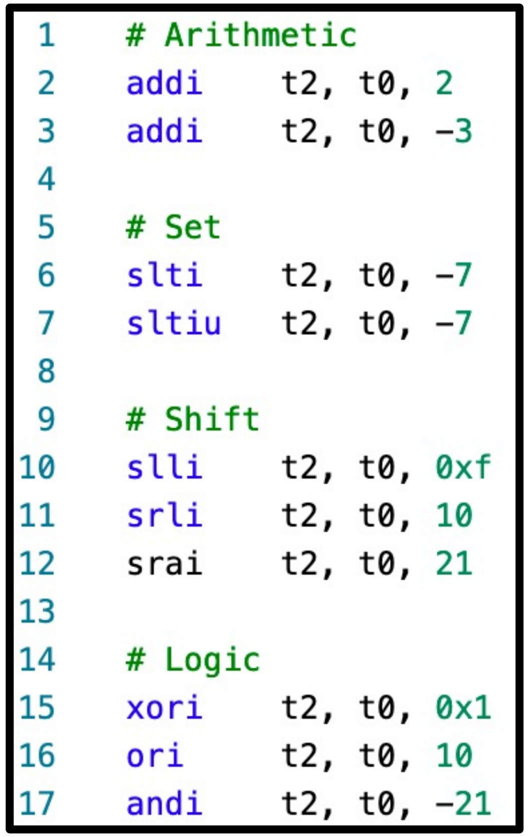
I type operation

- **Arithmetic**
  - addi (Add Immediate)
    rd = rs1 + simm12
- **Set**
  - slti / sltiu (Set Less Than Immediate (Unsigned))
    slti : rd = (rs1 < simm12) ? 1 : 0
    sltiu : rd = ($rs1_{unsigned}$ < $simm12_{unsigned}$) ? 1 : 0
- **Shift**
  - slli / srli / srai (Shift [Left / Right] [Logic / Arithmetic] Immediate)
    slli : rd = rs1 << uimm5
    srli : rd = rs1 >> uimm5
    srai : rd = rs1 >>> uimm5
- **Logical**
  - xori / ori / andi (Exclusive-OR / OR / AND)
    rd = rs1 [^|&] simm12

I type format

Shift : Iop rd, rs1, uimm5

Others : Iop rd, rs1, simm12



```
# Arithmetic
addi    t2, t0, 2
addi    t2, t0, -3

# Set
slti    t2, t0, -7
sltiu   t2, t0, -7

# Shift
slli    t2, t0, 0xf
srli    t2, t0, 10
srai    t2, t0, 21

# Logic
xori    t2, t0, 0x1
ori     t2, t0, 10
andi    t2, t0, -21
```

Computation Instruction – (Long Immediate)

U type operation

- lui (Load Upper Immediate)
  - rd = {uimm20, 12'b0}
- auipc (Add Upper Immediate to PC)
  - rd = pc + {uimm20, 12'b0}

U type format

Uop rd, uimm20

```
# lui
lui      t0, 0xc8763

# auipc
auipc    t0, 0xc8763
```

t0 = 0xc8763000

if pc = 0x666
t0 = 0xc8763666

Computation Instruction – Application

- **Load a large immediate (0xabcd1234) to x5**

0x234 = 0010_0011_0100

```
lui  x5, 0xabcd1
addi  x5, x5, 0x234
```

x5 = 0xabcd1000
0x234 = 0010_0011_0100
sext (0xbcd) = 0000_0000_0000_0000_0000_0010_0011_0100 = 0x00000234
x5 = 0xabcd1000 + 0x00000234 = 0xabcd1234 ✓

- **Load a large immediate (0x1234abcd) to x5**

0xbcd = 1011_1100_1101 = -0x433

lui x5, 0x1234a
addi x5, x5, ~~0xbcd~~

-0x433

lui x5, 0x1234b
addi x5, x5, ~~0xbcd~~

x5 = 0x1234a000
-0x433 = 1011_1100_1101
sext (-0x433) = 1111_1111_1111_1111_1111_1011_1100_1101 = 0xffffbcd
x5 = 0x1234a000 + 0xffffbcd = 0x12349bcd ✗

x5 = 0x1234b000
-0x433 = 1011_1100_1101
sext (-0x433) = 1111_1111_1111_1111_1111_1011_1100_1101 = 0xffffbcd
x5 = 0x1234b000 + 0xffffbcd = 0x1234abcd ✓

- **Multiplication / Division**
  - $x5 = x5 \times 2$

    ```
    slli  x5, x5, 1
    ```

  - $x5 = x5 \times 3$

    ```
    slli  x6, x5, 1
    addi  x5, x6, x5
    ```

imm[11:0]	rs1	funct3	rd	opcode	I-type

Only 12 bits
Decimal : -2048 ~ 2047
Hexadecimal : -0x800 ~ 0x7ff

0x234 ✓

0xbcd ✗ $\rightarrow$ -0x433 ✓

Computation Instruction – Application

- **Logical Operation - AND (Keep only the masked bits)**

and x5, x6, x7

x6	0	0	1	1	0	1	0	0	1	1	0	0	1	0	0	1	1	0	0	1	0	1	0	1	0	1	0	1	0	1	1
and (mask) x7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0
x5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0	0	0

- **Logical Operation - OR (Append on original data)**

or x5, x6, x7

x6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0
or x7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0
x5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0

- **Logical Operation - XOR**

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

1. (two operands) The same is “0", not same is “1”
2. (two operands) “1” can complement a bit whether it was 0 or 1
3. (multi operands) Odd number of 1 is “1”, even is “0”

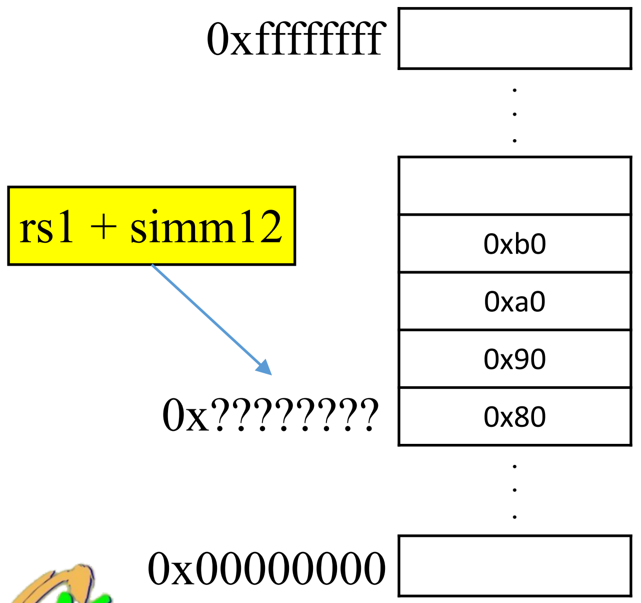
Load & Store Instruction

- lb / lh / lw (Load [byte / halfword / word]) — 8 bits, 16 bits, 32 bits
  - rd = signed-extension(RAM[rs1 + simm12] [7/15/31 : 0])
- lbu / lhu (Load [byte / halfword] unsigned)
  - rd = unsigned-extension(RAM[rs1 + simm12] [7/15/31 : 0])
- sb / sh / sw (Store [byte / halfword / word])
  - RAM[rs1 + simm12] = rs2[7/15/31 : 0]

format

Load : op rd, simm(rs1)

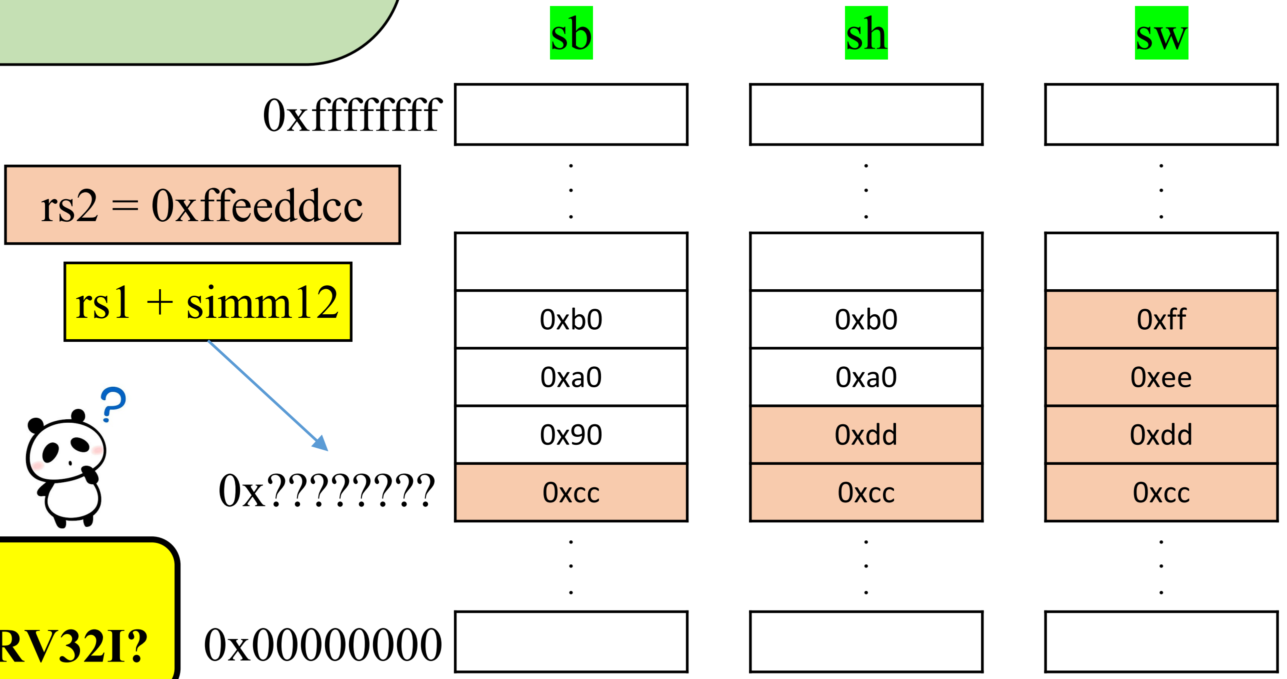
Store : op rs2, simm(rs1)



lb : rd = 0xffffff80
lh : rd = 0xffff9080
lw : rd = 0xb0a09080

lbu : rd = 0x00000080
lhu : rd = 0x00009080

Q4 :
Why there is no lwu in RV32I?



Load & Store Instruction – Example & Questions !

int array[6] = {0xddccbbaa, 0x44332211, 0x00ff8899, … , 0x12345678}

Suppose x26 = 0x8800

array[1] = array[2] + array[5]

```
lw  t0, 8(x26)
lw  t1, 20(x26)
add t2, t0, t1
sw  t2, 4(x26)
```

RAM Little Endian

	Offset 0	Offset 1	Offset 2	Offset 3
0xfffffffc				
⋮				
4n+20	0x78	0x56	0x34	0x12
⋮				
4n+8	0x99	0x88	0xff	0x00
4n+4	0x11	0x22	0x33	0x44
4n = 0x8800	0xaa	0xbb	0xcc	0xdd
⋮				
0x00000000				

Q5 : What is the addressing mode of Load & Store ?

Pseudo Instruction (偽指令)

- They are not real instructions.
- They provided by the assembler tools, which convert them into one or more real instructions.
- They make programmers more intuitive when writing programs.

Pseudo Instruction	Meaning	Base Instruction
nop	No operation	addi x0, x0, 0
snez rd, rs	rd = (rs != 0) ? 1 : 0	sltu rd, x0, rs
sltz rd, rs	rd = (rs < 0) ? 1 : 0	slt rd, rs, x0
sgtz rd,rs	rd = (rs > 0) ? 1 : 0	slt rd, x0, rs
li rd, immediate	rd = immediate	lui / addi
mv rd, rs	rd = rs (copy register)	addi rd, rs, 0
not rd, rs	rd = ~rs (one's complement)	xori rd, rs, -1
seqz rd, rs	rd = (rs == 0) ? 1: 0	sltiu rd, rs, 1

Assemble Directives

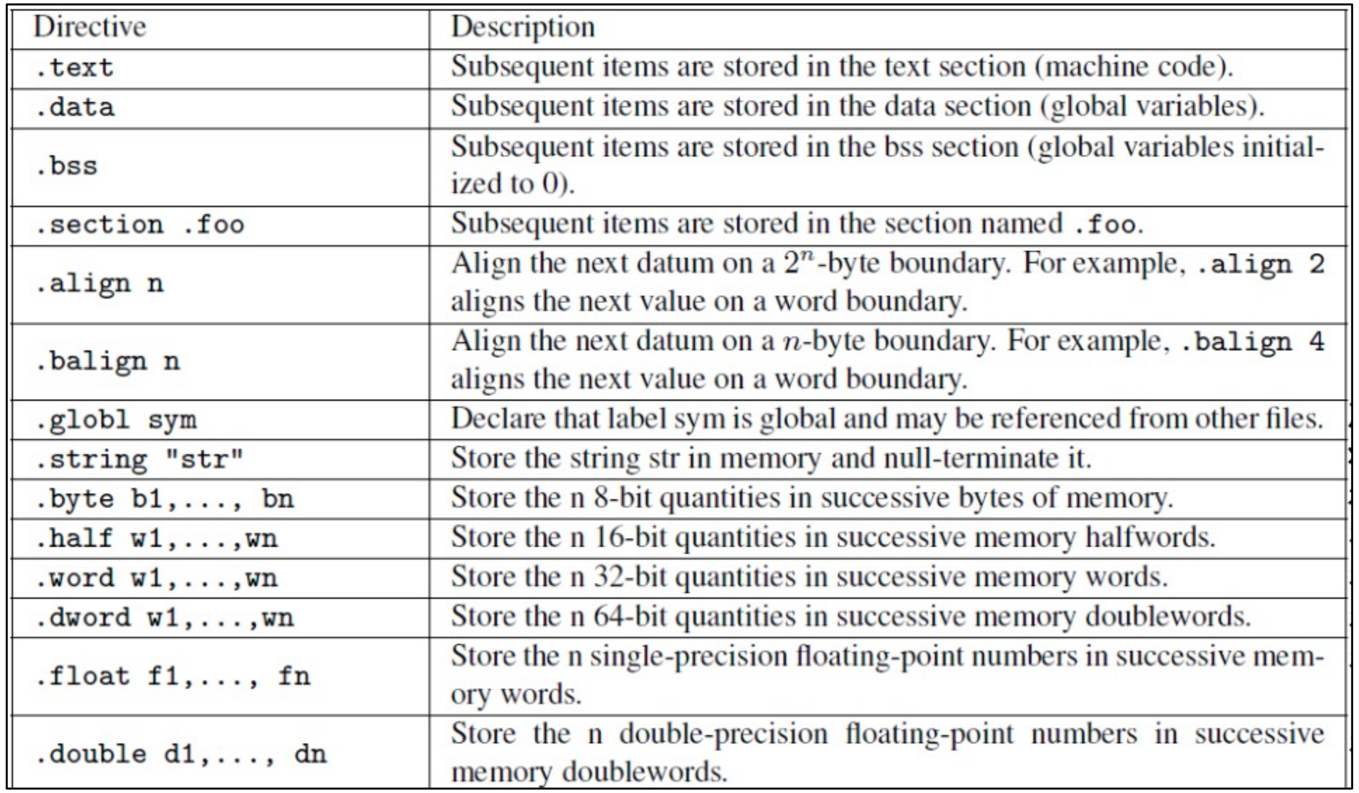
- **Directive is not the CPU instructions, there are two purposes :**
  1. To prompt the assembler to do something
  2. To inform the assembler of certain information

Reference :
http://godleon.blogspot.com/2008/01/machine-language-cpu-machine-language.html

- **Directive will not be converted into machine code by the assembler**
  There are several common uses of the directive :
  1. Define constants
  2. Define the memory location for storing data
  3. Include external source code as appropriate
  4. Include other files

```
.data
size:       .word   8
arr:        .word   7, 4, 10, -1, 3, 2, -6, 9
str1:       .string " Before Sort : "
str2:       .string " After Sort  : "
newline:    .string "\n"
comma:      .string " , "

.text
##############    main    ################
main:
    addi    sp, sp, -4
    sw  ra, 0(sp)
```



Directive	Description
`.text`	Subsequent items are stored in the text section (machine code).
`.data`	Subsequent items are stored in the data section (global variables).
`.bss`	Subsequent items are stored in the bss section (global variables initialized to 0).
`.section .foo`	Subsequent items are stored in the section named `.foo`.
`.align n`	Align the next datum on a 2^n-byte boundary. For example, `.align 2` aligns the next value on a word boundary.
`.balign n`	Align the next datum on a n-byte boundary. For example, `.balign 4` aligns the next value on a word boundary.
`.globl sym`	Declare that label sym is global and may be referenced from other files.
`.string "str"`	Store the string str in memory and null-terminate it.
`.byte b1,..., bn`	Store the n 8-bit quantities in successive bytes of memory.
`.half w1,...,wn`	Store the n 16-bit quantities in successive memory halfwords.
`.word w1,...,wn`	Store the n 32-bit quantities in successive memory words.
`.dword w1,...,wn`	Store the n 64-bit quantities in successive memory doublewords.
`.float f1,..., fn`	Store the n single-precision floating-point numbers in successive memory words.
`.double d1,..., dn`	Store the n double-precision floating-point numbers in successive memory doublewords.

Format Conversion Flow

```
int main() {
    int a = 1
    int b = a*3+1
    int c = 0x1234abcd
}
```

User program

Label

A **label** can be placed at the beginning of a statement, which represent current value of the active location counter and can be serves as an instruction operand.

User program

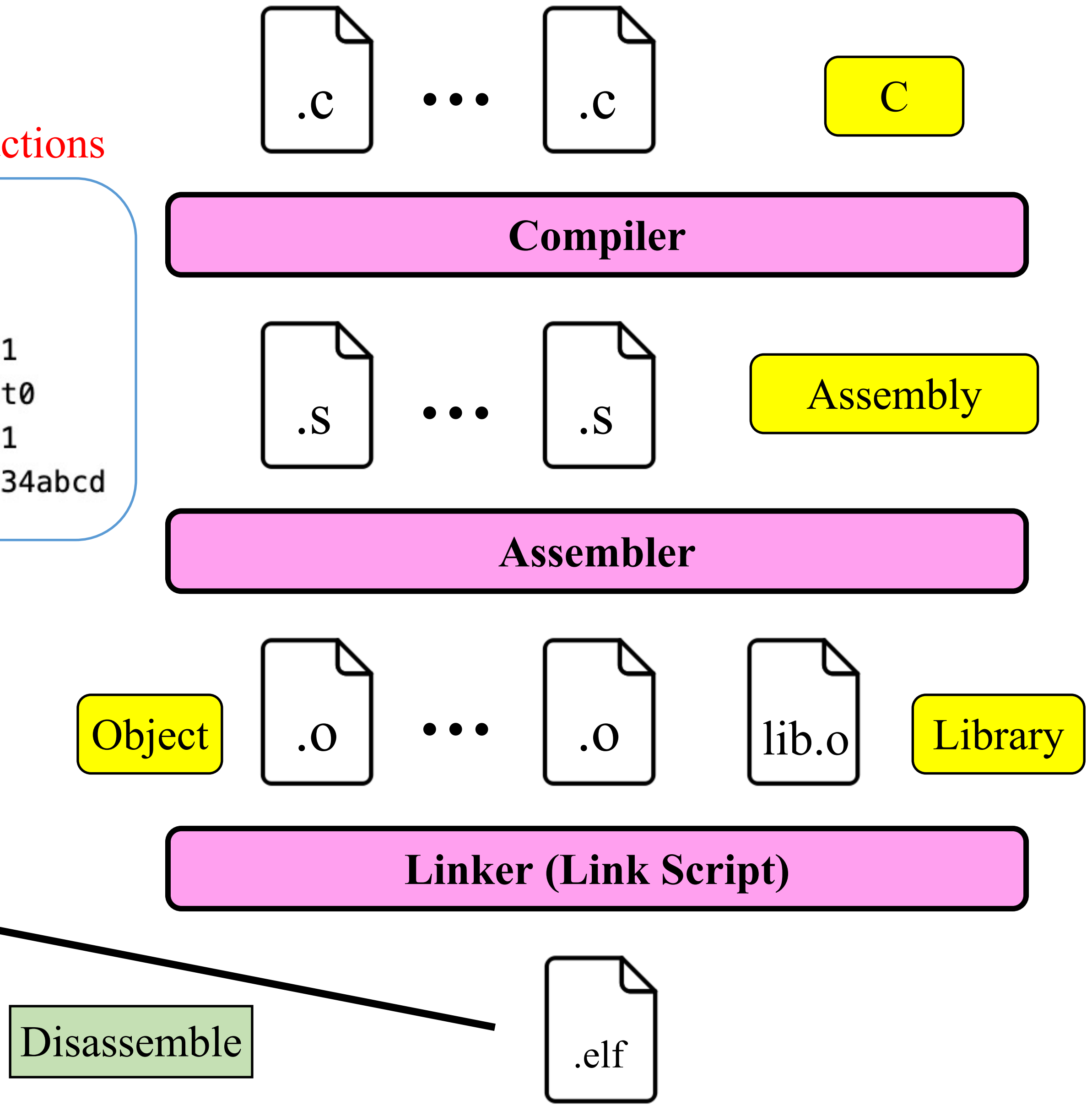
ISA + Pseudo Instructions

```
.text
main:
    li      t0, 1
    slli    t1, t0, 1
    add     t1, t1, t0
    addi    t2, t1, 1
    li      t5, 0x1234abcd
```

```
00010054 <main>:
   10054:  00100293        li      t0,1
   10058:  00129313        slli    t1,t0,0x1
   1005c:  00530333        add     t1,t1,t0
   10060:  00130393        addi    t2,t1,1
   10064:  1234bf37        lui     t5,0x1234b
   10068:  bcdf0f13        addi    t5,t5,-1075
```

Machine program

Only ISA

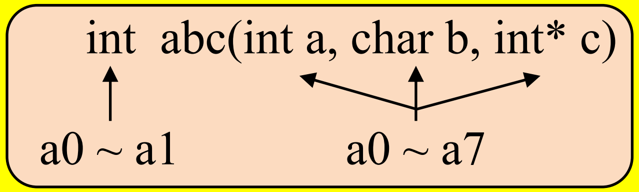


Question List

Q1 :
Why IALIGN of RV32I is 32 bits ?
Why IALIGN need to support 16 bits ?
Why IALIGN have no greater than 32 bits (e.g. 64 / 128 bits) ?

Q2 : Why temporary registers and saved registers are not numbered sequentially ?

Q3 : Why return value needs 2 registers (a0, a1) ?



Q4 : Why there is no lwu in RV32I?

Q5 : What is the addressing mode of Load & Store ?