CS 110
Computer Architecture
RISC-V

Instructors:
Siting Liu & Yuan Xiao
Course website: https://faculty.sist.shanghaitech.edu.cn/liust/courses/CS110.html
School of Information Science and Technology (SIST)
ShanghaiTech University

2025/3/17

Administrative

- HW2, due Mar. 24th and Lab 3 (RISC-V and Venus, to be checked next week) will be released later today
- Discussion this week on memory management & valgrind by TA Jianjun Lu at SPST 4-122 on Friday, 19:50-21:00
- The similarity check is conducted automatically for each HW, proj., etc. Please comply with the course rules!
- We grade only on your most recent activated submissions before ddl for all assignments!

Outline

- Intro. to ISA
- Intro. to RISC-V
- Assembly instructions in RISC-V (RV32I)
    - R-type
    - I-type arithmetic and logic
    - I-type load
    - S-type
    - Desicion-making instructions

Intro to ISA

- Part of the abstract model of a computer that defines how the CPU is controlled by the software; interface between the hardware and the software;
- Programmers' manual because it is the portion of the machine that is visible to the assembly language programmers, the compiler writers, and the application programmers.
- Defines the supported data types, the registers, how the hardware manages main memory, key features, instructions that can be executed (instruction set), and the input/output model of multiple ISA implementations
- ISA can be extended by adding instructions or other capabilities

-by ARM

ISA vs. Microarchitecture

- ISA: Manual for building microarchitecture or processors
- Microarchitecture: Implementation of an ISA

Popular ISAs

- X86/AMD64
  - Dominant architecture for personal computers and servers
  - Name derived from Intel 8086/80186/80286...
  - Multiple version: X86-16, X86-32 (IA-32), X86-64 (AMD64)
  - Extensions such as MMX, SSE, etc.
- Major vendors
  - Intel, AMD, VIA, Zhaoxin, DM&P, RDC
  - To OEM (original equipment manufacturer)
- CISC (complex instruction set computer)
  - Variable-length instructions
  - Allow memory access with instructions other than load or store
  - VAX architecture had an instruction to multiply polynomials!

add: X86 integer addition
Syntax
```
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <mem>,imm*
... ...
```

Popular ISAs

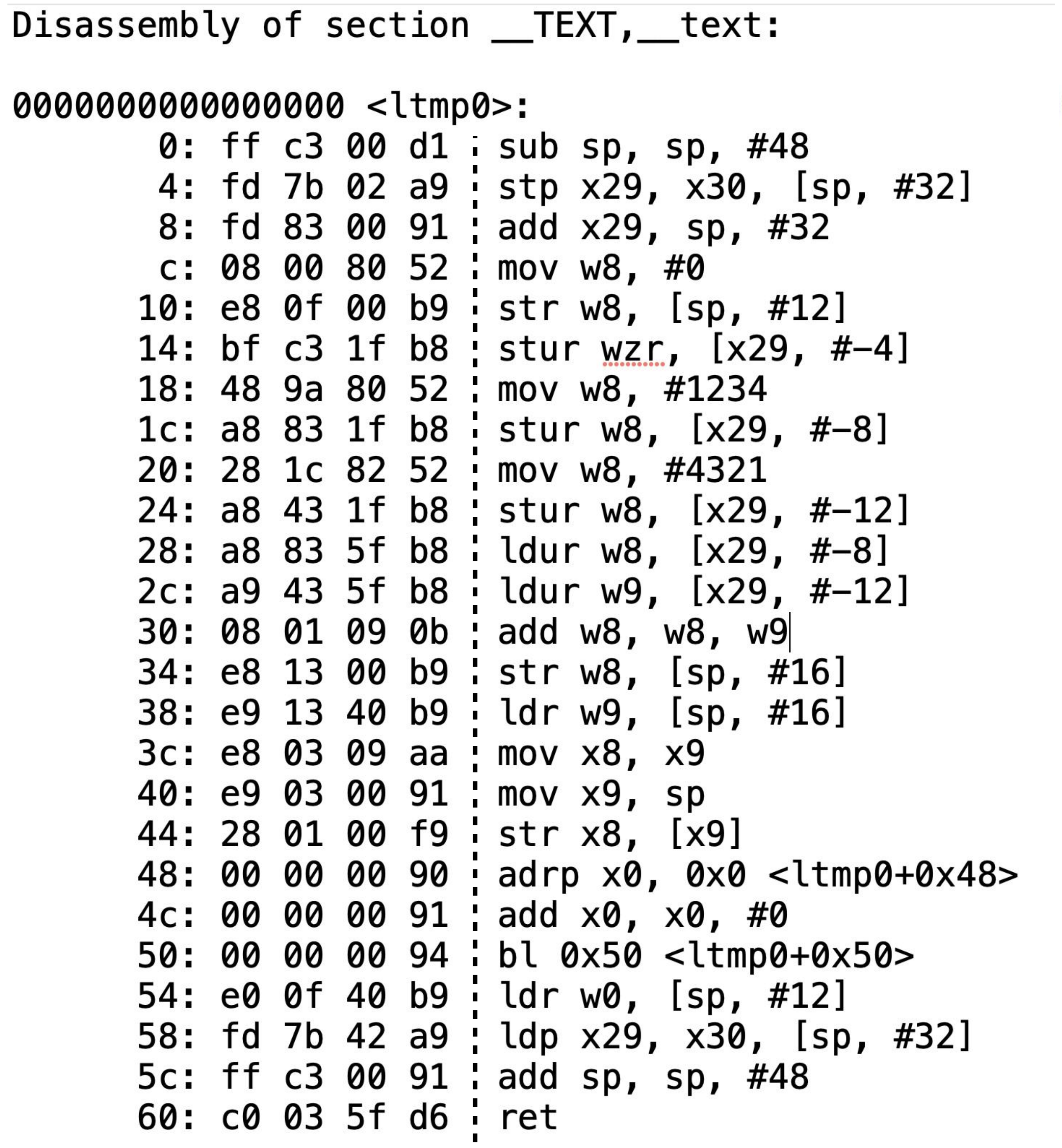
- ARM
  - Dominant architecture for embedded devices
  - Advanced RISC Machine
  - Multiple version: ARMv1-ARMv9
- Major vendors
  - Apple, Huawei, Qualcomm, Xilinx, etc.
  - ARM sells IP cores to IC vendors (core licence)
  - IC vendors sell MCU/CPU/SoC to OEM or for self use
- RISC (reduced instruction set computer)
  - 32-bit fixed-length instructions (not actually for Thumb-16)
  - Allow memory access with only `load` or `store` instructions
  - Simpler to design hardware. Generally generate smaller heat

add: ARM addition

Syntax

```
add(S) <reg>,<reg>,<reg or imm>
```

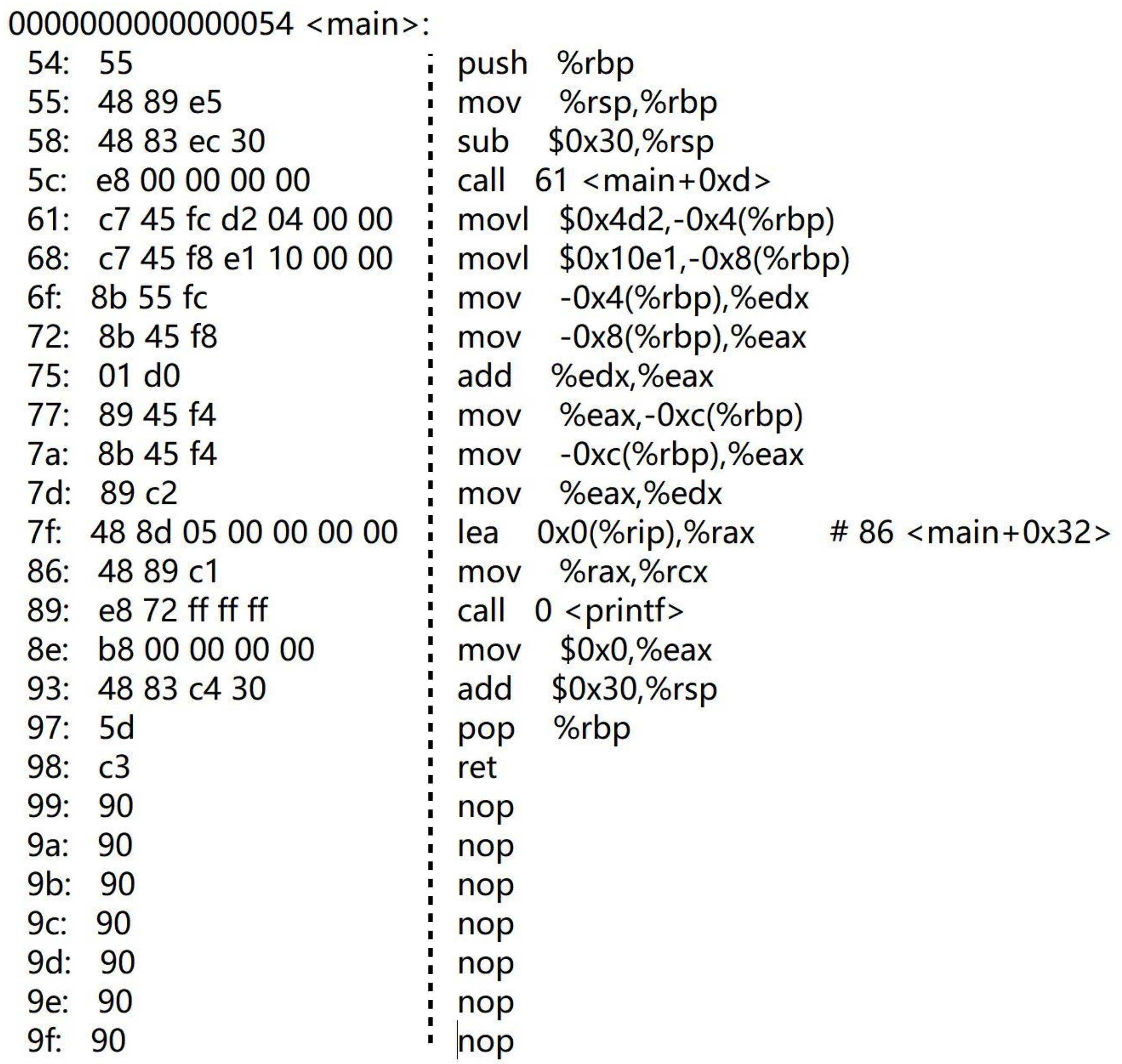
RISC vs. CISC



```
Disassembly of section __TEXT,__text:

0000000000000000 <ltmp0>:
       0: ff c3 00 d1 ¦ sub sp, sp, #48
       4: fd 7b 02 a9 ¦ stp x29, x30, [sp, #32]
       8: fd 83 00 91 ¦ add x29, sp, #32
       c: 08 00 80 52 ¦ mov w8, #0
      10: e8 0f 00 b9 ¦ str w8, [sp, #12]
      14: bf c3 1f b8 ¦ stur wzr, [x29, #-4]
      18: 48 9a 80 52 ¦ mov w8, #1234
      1c: a8 83 1f b8 ¦ stur w8, [x29, #-8]
      20: 28 1c 82 52 ¦ mov w8, #4321
      24: a8 43 1f b8 ¦ stur w8, [x29, #-12]
      28: a8 83 5f b8 ¦ ldur w8, [x29, #-8]
      2c: a9 43 5f b8 ¦ ldur w9, [x29, #-12]
      30: 08 01 09 0b ¦ add w8, w8, w9
      34: e8 13 00 b9 ¦ str w8, [sp, #16]
      38: e9 13 40 b9 ¦ ldr w9, [sp, #16]
      3c: e8 03 09 aa ¦ mov x8, x9
      40: e9 03 00 91 ¦ mov x9, sp
      44: 28 01 00 f9 ¦ str x8, [x9]
      48: 00 00 00 90 ¦ adrp x0, 0x0 <ltmp0+0x48>
      4c: 00 00 00 91 ¦ add x0, x0, #0
      50: 00 00 00 94 ¦ bl 0x50 <ltmp0+0x50>
      54: e0 0f 40 b9 ¦ ldr w0, [sp, #12]
      58: fd 7b 42 a9 ¦ ldp x29, x30, [sp, #32]
      5c: ff c3 00 91 ¦ add sp, sp, #48
      60: c0 03 5f d6 ¦ ret
```

Assembly
Compiled on a Mac machine using ARM CPU



```
0000000000000054 <main>:
 54:  55                    ¦ push   %rbp
 55:  48 89 e5              ¦ mov    %rsp,%rbp
 58:  48 83 ec 30           ¦ sub    $0x30,%rsp
 5c:  e8 00 00 00 00        ¦ call   61 <main+0xd>
 61:  c7 45 fc d2 04 00 00  ¦ movl   $0x4d2,-0x4(%rbp)
 68:  c7 45 f8 e1 10 00 00  ¦ movl   $0x10e1,-0x8(%rbp)
 6f:  8b 55 fc              ¦ mov    -0x4(%rbp),%edx
 72:  8b 45 f8              ¦ mov    -0x8(%rbp),%eax
 75:  01 d0                 ¦ add    %edx,%eax
 77:  89 45 f4              ¦ mov    %eax,-0xc(%rbp)
 7a:  8b 45 f4              ¦ mov    -0xc(%rbp),%eax
 7d:  89 c2                 ¦ mov    %eax,%edx
 7f:  48 8d 05 00 00 00 00  ¦ lea    0x0(%rip),%rax        # 86 <main+0x32>
 86:  48 89 c1              ¦ mov    %rax,%rcx
 89:  e8 72 ff ff ff        ¦ call   0 <printf>
 8e:  b8 00 00 00 00        ¦ mov    $0x0,%eax
 93:  48 83 c4 30           ¦ add    $0x30,%rsp
 97:  5d                    ¦ pop    %rbp
 98:  c3                    ¦ ret
 99:  90                    ¦ nop
 9a:  90                    ¦ nop
 9b:  90                    ¦ nop
 9c:  90                    ¦ nop
 9d:  90                    ¦ nop
 9e:  90                    ¦ nop
 9f:  90                    ¦ nop
```

Assembly
Compiled on a Windows machine using Intel CPU

Popular ISAs

- RISC philosophy (John Cocke IBM, John Hennessy Stanford, David Patterson Berkeley, 1980s)
- Hennessy & Patterson won ACM A.M. Turing Award (2017)

Reduced Instruction Set Computer (RISC)

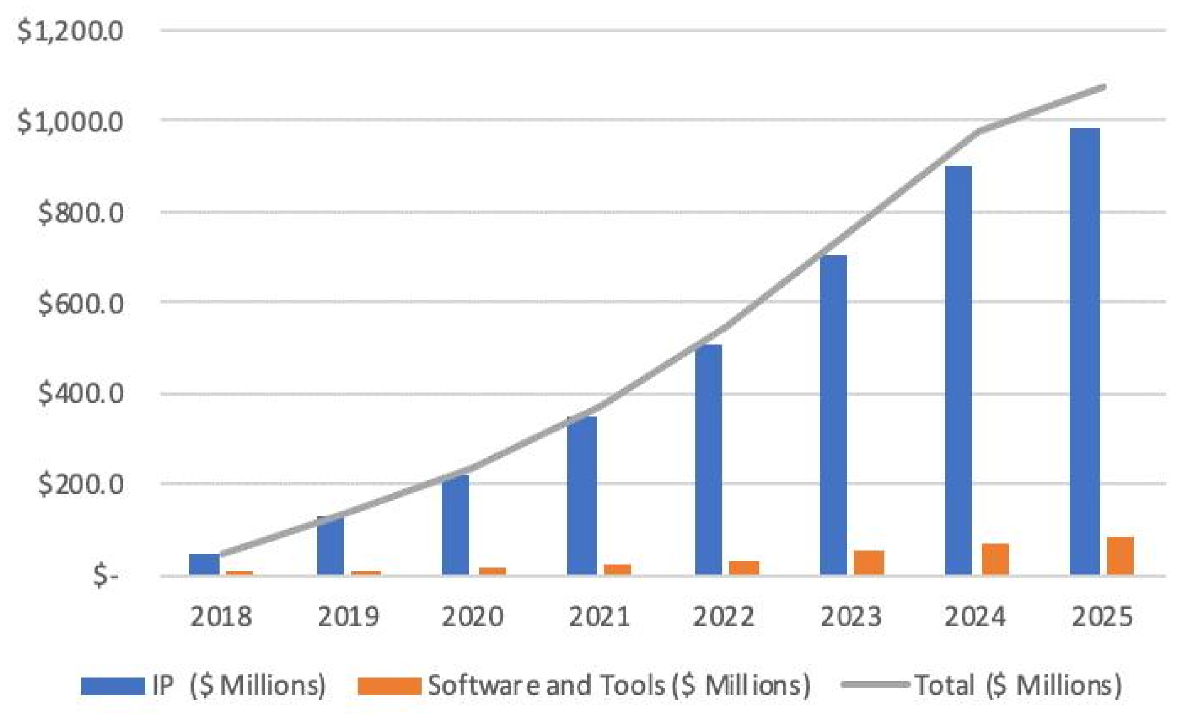
- Keep the instruction set small and simple, makes it easier to build fast hardware.
- Let software do complicated operations by composing simpler ones.

Popular ISAs

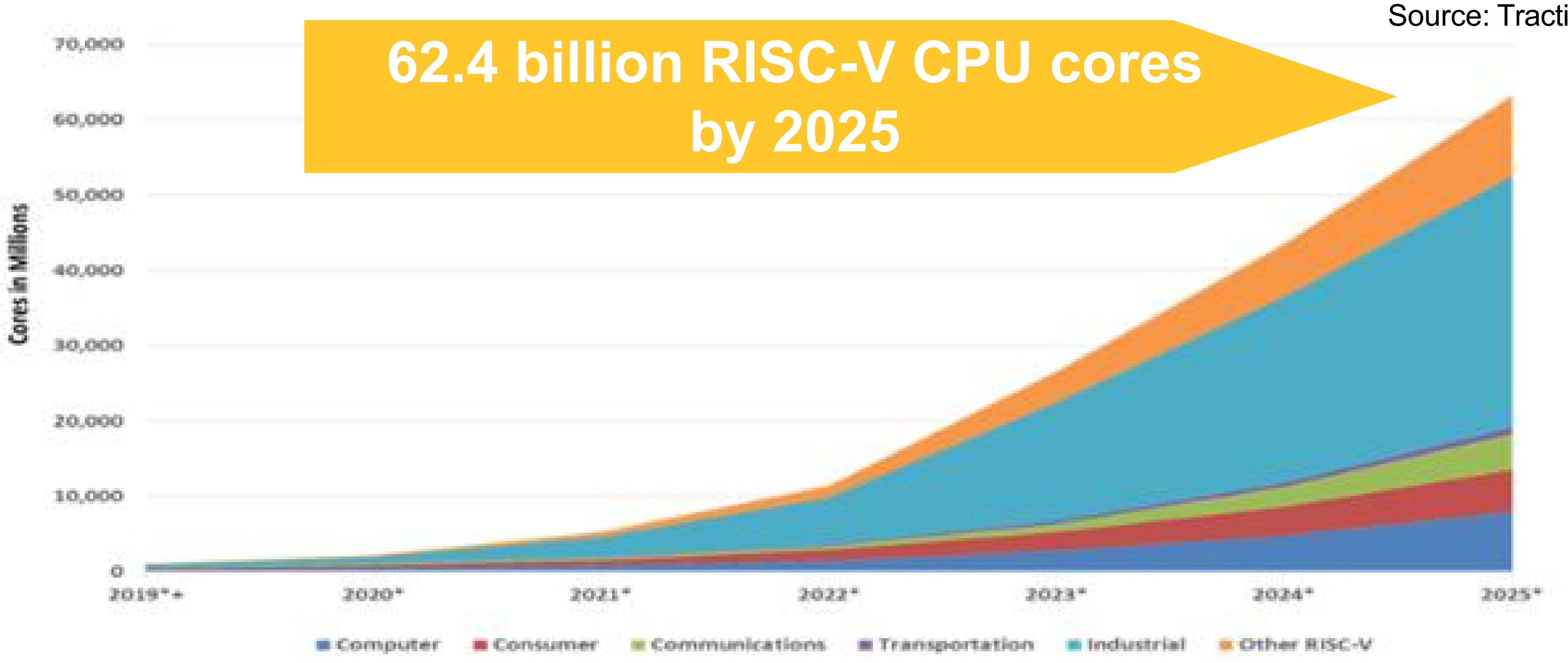
- RISC-V (pronounced “risk-five”)
  - Started as a summer project in UC Berkeley, 2010
  - The ISA itself is published in 2011 as open source
  - RISC-V foundation formed 2015 to own, maintain and publish IP related to RISC-V’s definition (a nonprofit business association)
- More than 3,100 members and still growing
  - Alibaba Cloud: T-Head (玄铁) C series; E series, and R series
  - Huawei: Hi3861V100 SoC for IoT/smart home
  - Tencent: a premier member
  - Intel, Google, Meta, SiFive, AMD/Xilinx, etc.
  - ShanghaiTech hold several RISC-V Summits China recent years!
- Other ISA examples: MIPS, IBM/Motorola PowerPC (quite old Mac), Intel IA64, ...

More than 3,100 RISC-V Members

The total market for RISC-V IP and Software is expected to grow to $1.07 billion by 2025 at a CAGR of 54.1%



Source: Tractica



From riscv.org

Source: Semico Research Corp

RISC-V

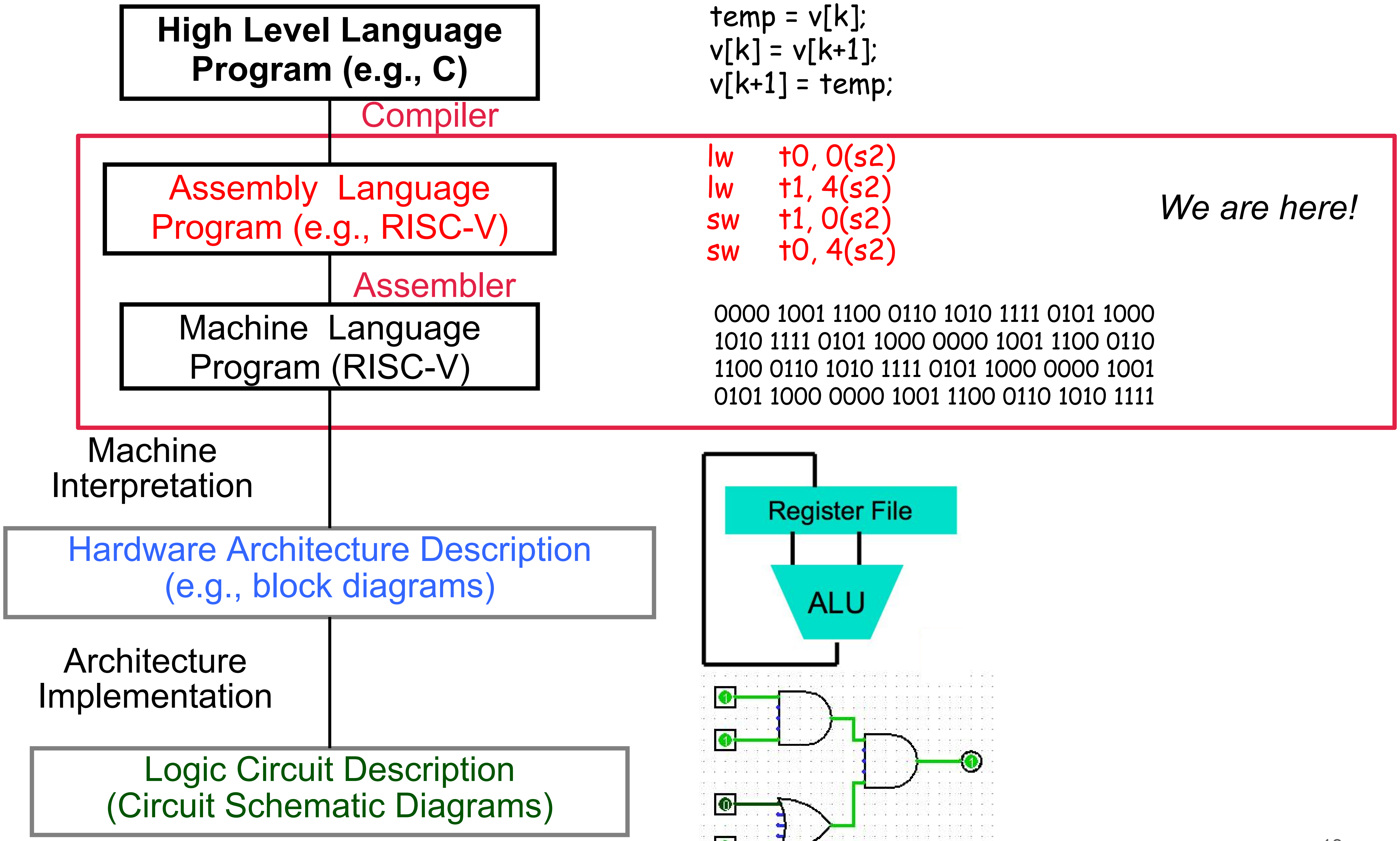
- Why RISC-V instead of Intel x86?
  – RISC-V is simple, elegant and open-source. Don’t want to get bogged down in gritty details.
- It is flexible/extendable
  – Enabled by different extensions

Name	**Description**
Base	
RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 register
RV64I	Base Integer Instruction Set, 64-bit
RV64E	Base Integer Instruction Set (embedded), 64-bit
RV128I	Base Integer Instruction Set, 128-bit
Extension	
M	Standard Extension for Integer Multiplication and Division
A	Standard Extension for Atomic Instructions
F	Standard Extension for Single-Precision Floating-Point
D	Standard Extension for Double-Precision Floating-Point

RVG = RVI + M + A + F + D

Manual available: https://riscv.org

Where are we?


High Level Language Program (e.g., C)
Compiler
Assembly Language Program (e.g., RISC-V)
Assembler
Machine Language Program (RISC-V)
Machine Interpretation
Hardware Architecture Description (e.g., block diagrams)
Architecture Implementation
Logic Circuit Description (Circuit Schematic Diagrams)
We are here!
Register File
ALU

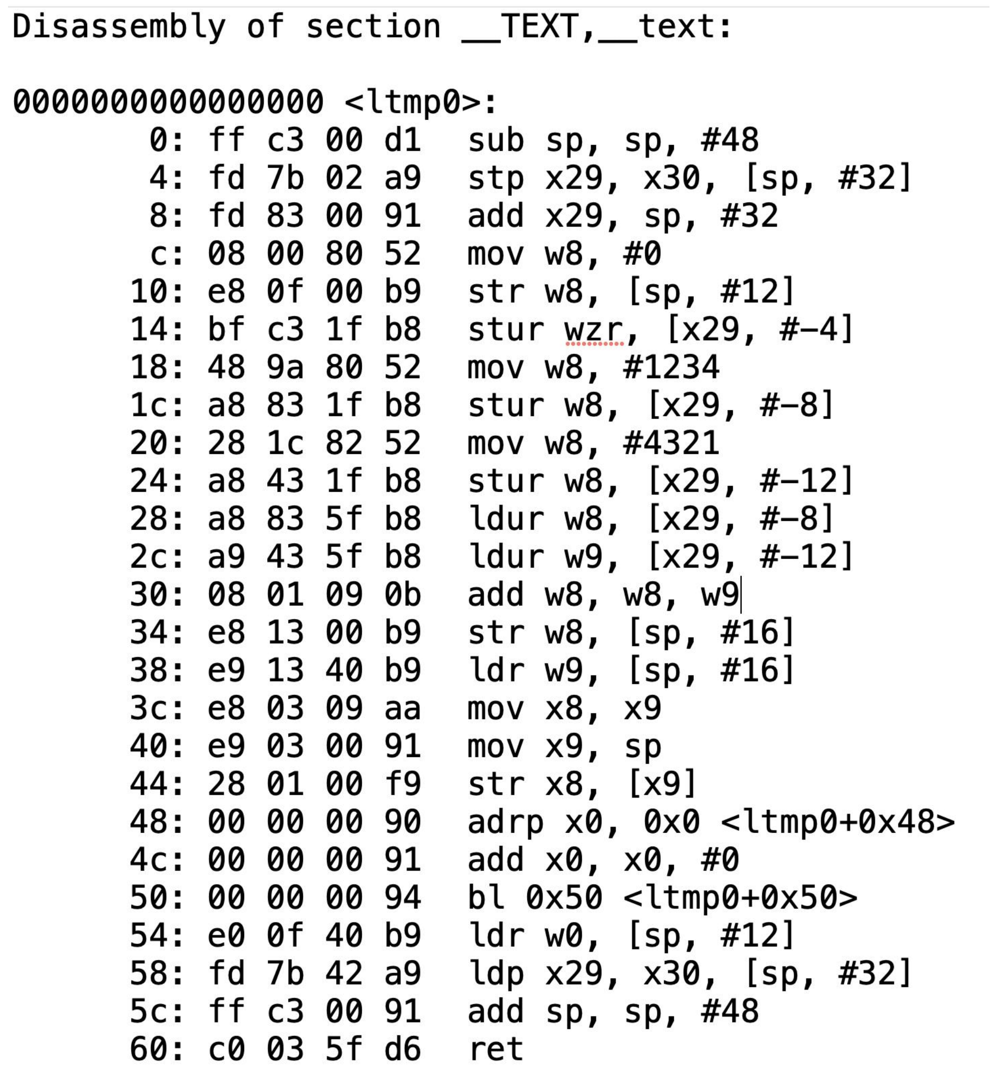
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw    t0, 0(s2)
lw    t1, 4(s2)
sw    t1, 0(s2)
sw    t0, 4(s2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

Assembly Language

- Basic job of a CPU: execute a series of instructions.
- Instructions are the primitive operations that the CPU may execute.
- Basic job of an instruction: change the state of a computer.



```
Disassembly of section __TEXT,__text:

0000000000000000 <ltmp0>:
       0: ff c3 00 d1   sub sp, sp, #48
       4: fd 7b 02 a9   stp x29, x30, [sp, #32]
       8: fd 83 00 91   add x29, sp, #32
       c: 08 00 80 52   mov w8, #0
      10: e8 0f 00 b9   str w8, [sp, #12]
      14: bf c3 1f b8   stur wzr, [x29, #-4]
      18: 48 9a 80 52   mov w8, #1234
      1c: a8 83 1f b8   stur w8, [x29, #-8]
      20: 28 1c 82 52   mov w8, #4321
      24: a8 43 1f b8   stur w8, [x29, #-12]
      28: a8 83 5f b8   ldur w8, [x29, #-8]
      2c: a9 43 5f b8   ldur w9, [x29, #-12]
      30: 08 01 09 0b   add w8, w8, w9
      34: e8 13 00 b9   str w8, [sp, #16]
      38: e9 13 40 b9   ldr w9, [sp, #16]
      3c: e8 03 09 aa   mov x8, x9
      40: e9 03 00 91   mov x9, sp
      44: 28 01 00 f9   str x8, [x9]
      48: 00 00 00 90   adrp x0, 0x0 <ltmp0+0x48>
      4c: 00 00 00 91   add x0, x0, #0
      50: 00 00 00 94   bl 0x50 <ltmp0+0x50>
      54: e0 0f 40 b9   ldr w0, [sp, #12]
      58: fd 7b 42 a9   ldp x29, x30, [sp, #32]
      5c: ff c3 00 91   add sp, sp, #48
      60: c0 03 5f d6   ret
```

ARM Assembly
Compiled on Mac machine using ARM CPU

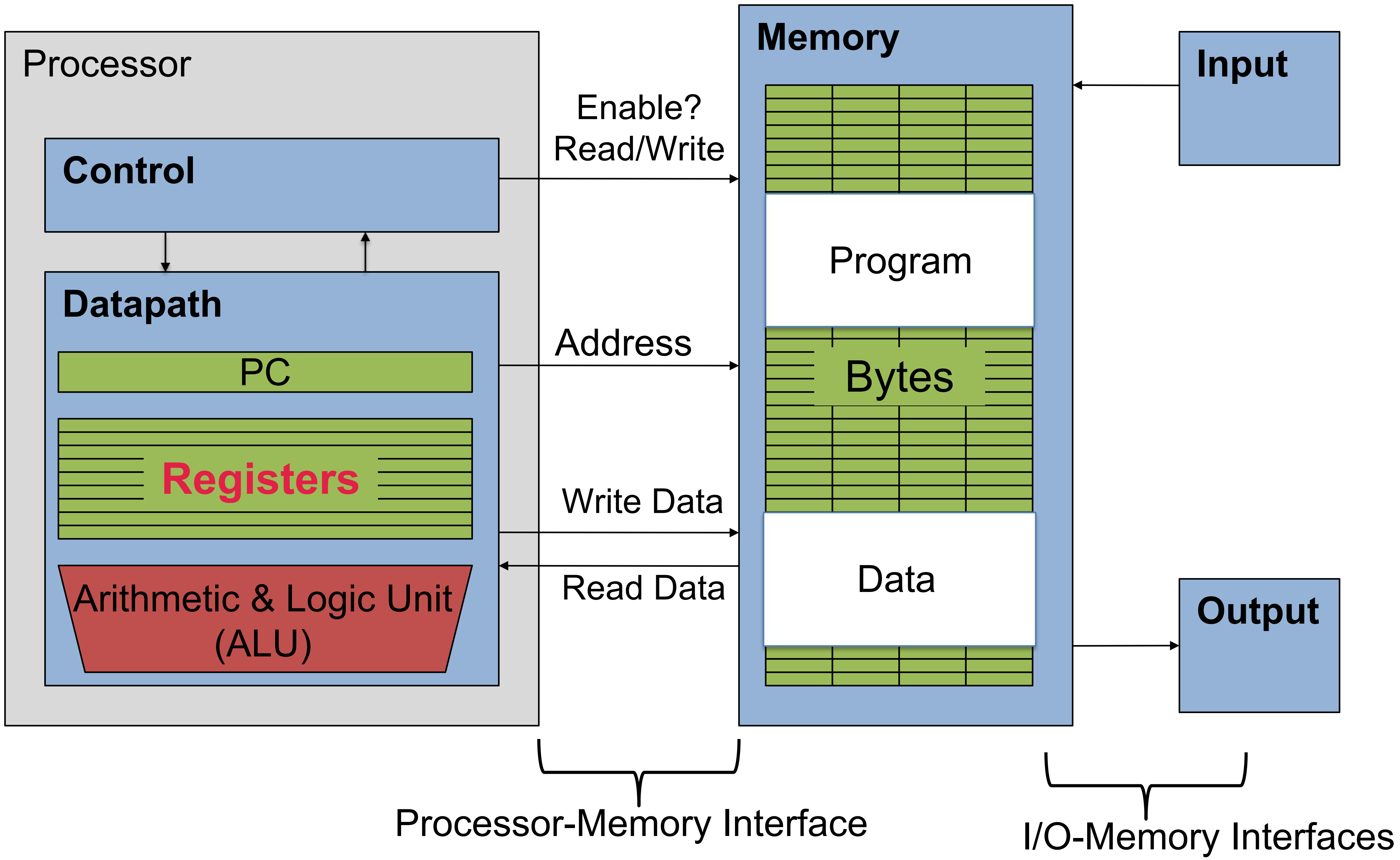
Exercise 1: Can we execute this assembly on a X86 CPU?

Exercise 2: Can we use a Linux OS to run ARM assembly?

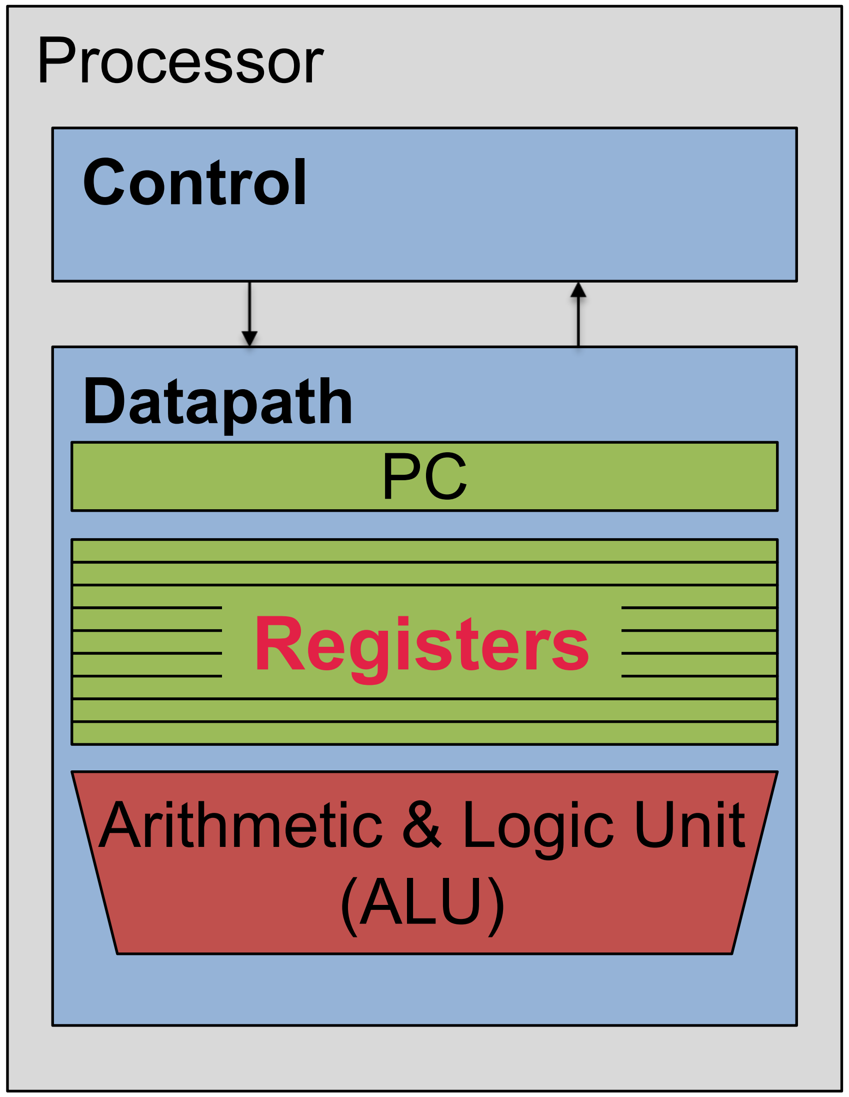
CPU State: Assembly Registers (hardware/variable)

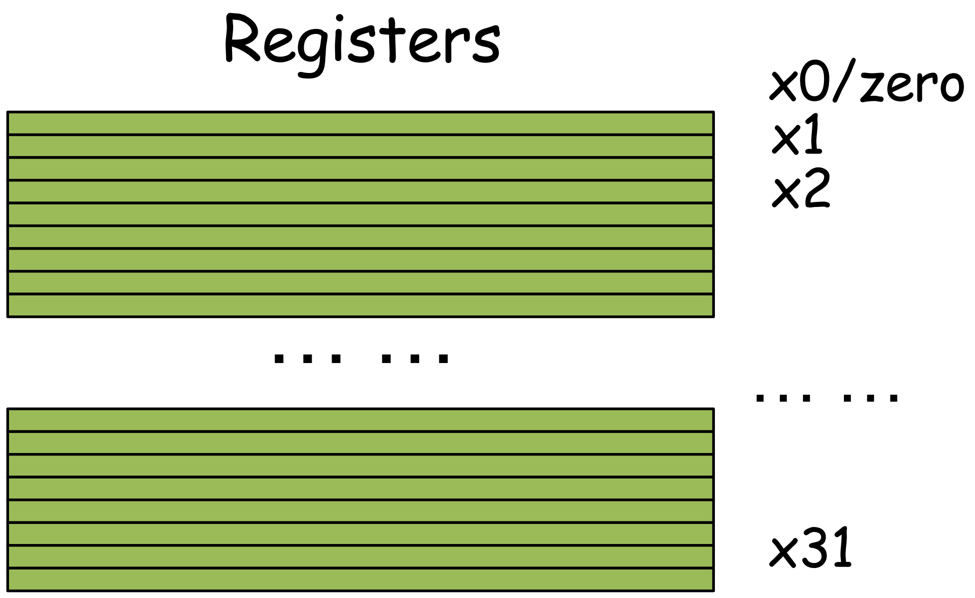
- Unlike C or Java, assembly cannot use variables
  - Keep assembly/computer hardware abstract simple
- Assembly operands are registers
  - Limited number of special locations/memory built directly into the CPU
  - Operations can only be performed on these registers in RISC-V
- Benefit: Since registers are directly in hardware (CPU), they are very fast

Registers, inside the Processor



RV32I Registers





- Similar to memory, use “address” to refer to specific location

- Hold address of the current instruction

```
1c: a8 83 1f b8   stur w8, [x29, #-8]
20: 28 1c 82 52   mov w8, #4321
24: a8 43 1f b8   stur w8, [x29, #-12]
28: a8 83 5f b8   ldur w8, [x29, #-8]
2c: a9 43 5f b8   ldur w9, [x29, #-12]
30: 08 01 09 0b   add w8, w8, w9
```

- 32 registers in RISC-V
  – Why 32? Smaller is faster, but too small is bad.
- Each RV32 register is 32-bit wide
  – Groups of 32 bits called a word in RV32; P&H textbook uses 64-bit variant RV64 (doubleword)

C, Java variables vs. registers

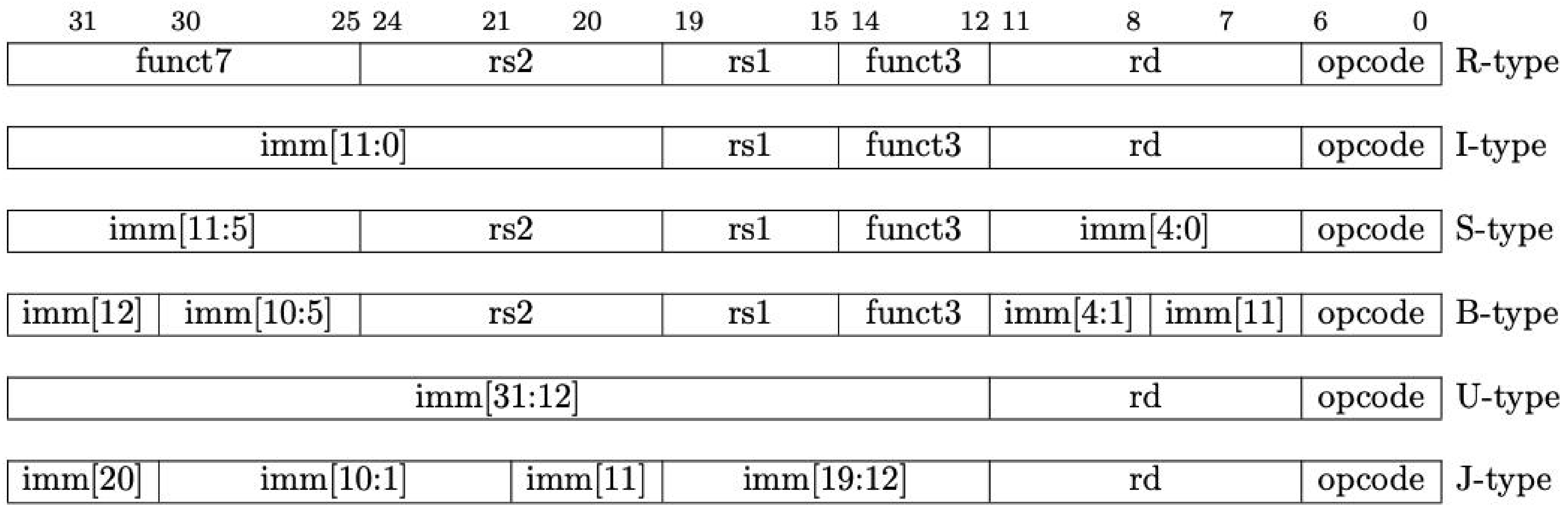
- In C (and most high level languages) variables declared first and given a type
  - Example: `int fahr, celsius;`
    `char a, b, c, d, e;`
- Each variable can ONLY represent a value of the type it was declared as (cannot mix and match int and char variables).
- In Assembly Language, registers have no type, simply stores 0's and 1's; operation determines how register contents are treated (think about the hardware)

Assembly Instructions

- In assembly language, each statement (called an instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other high hevel hanguages), each line of assembly code contains at most 1 instruction
- Another way to make your code more readable: comments!
- Hash (#) is used for RISC-V comments
  - anything from hash mark to end of line is a comment and will be ignored

Assembly Instructions

- Different types of instructions (4 core types + B/J based on the handling of immediate)



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

- Different types have different format but "rs1", "rs2" and "rd" are at the same position (hardware friendly)
- As an ID number, the machine code of the instructions has different fields; format depends on their operands/type

Assembly Instructions

- Different types of instructions (4 core types + B/J based on the handling of immediate)

31 30 25	24 21 20	19 15	14 12	11 8 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type

- **R-type**
  - Register-register operation, mainly for arithmetic & logic
  - Has two operands (accessed from the source registers, `rs1` & `rs2`) and one output (saved to the destination register, `rd`)
  - Cannot access main memory (instruction executed by CPU alone, no data exchange with main memory)

RV32I R-type Arithmetic

- Syntax of instructions
  - Addition: add rd,rs1,rs2 (operation rd,rs1,rs2)

    Adds the value stored in register rs1 to that of rs2 and stores the sum into register rd, similar to a = b+c, a ⇔ rd, b ⇔ rs1, c ⇔ rs2
  - Example: add x5, x2, x1

    add x6, x0, x5

    add x4, x1, x3

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
	x4
	x5
	x6
	x7

RV32I R-type Arithmetic

- Syntax of instructions
  - Subtraction: sub rd, rs1, rs2

Subtract the value stored in register rs2 from that of rs1 and stores the difference into register rd, equivalent to a = b-c, a ⇔ rd, b ⇔ rs1, c ⇔ rs2

- Example:
  ```
  sub x5, x2, x1
  sub x6, x5, x0
  ```

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
	x4
	x5
	x6
	x7

RV32I R-type Logic Operation

- Syntax of instructions:
  - AND/OR/XOR: and/or/xor rd, rs1, rs2

Logically bit-wise and/or/xor the value stored in register rs1 and that of rs2 and stores the result into register rd, equivalent to a = b (&/|/^) c, a ⇔ rd, b ⇔ rs1, c ⇔ rs2

  - Example: and x5, x2, x1

    xor x6, x1, x5

    and x4, x1, x3

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
	x4
	x5
	x6
	x7

RV32I R-type Logic Operation

- Syntax of instructions:
  - AND/OR/XOR: and/or/xor rd, rs1, rs2

Logically bit-wise and/or/xor the value stored in register rs1 and that of rs2 and stores the result into register rd, equivalent to a = b (&/|/^) c, a ⇔ rd, b ⇔ rs1, c ⇔ rs2

  - Used for bit-mask

```
and x5, x7, x4
or x6, x7, x4
```

  - XOR can be used for bit-wise negation

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0xFFFF0000	x4
	x5
	x6
0x12345678	x7

RV32I R-type Compare

- Syntax of instructions
  - SLT/SLTU: slt/sltu rd, rs1, rs2

  Compare the value stored in register rs1 and that of rs2, sets rd=1, if rs1<rs2 otherwise rd=0, equivalent to a = b < c ? 1 : 0, a ⇔ rd, b ⇔ rs1, c ⇔ rs2. Treat the numbers as signed/unsigned with slt/sltu.
  - Example:

```
slt x5, x2, x1
slt x4, x3, x1
sltu x5, x3, x1
```

  - Overflow detection (unsigned)

```
add x5, x3, x3
sltu x6, x5, x3
```

  - Overflow detection (signed)?

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

Registers

Registers	
0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
	x4
	x5
	x6
	x7

RV32I R-type Shift

- Syntax of instructions:
  - Shift left/right (arithmetic): sll/srl/sra rd, rs1, rs2

    Left/Right shifts the value stored in register rs1 by lower 5 bits of rs2, equivalent to a = b <</>> />>>c, a ⇔ rd, b ⇔ rs1, c ⇔ rs2.

    arithmetic: sign extended.
  - Example: sll x5, x2, x4
    srl x6, x1, x4
    sra x7, x3, x4

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

RV32I R-type Shift

- Syntax of instructions:
  - Shift left/right (arithmetic): sll/srl/sra rd, rs1, rs2

    Left/Right shifts the value stored in register rs1 by lower 5 bits of rs2, equivalent to a = b <</>> />>>c, a ⇔ rd, b ⇔ rs1, c ⇔ rs2.

    arithmetic: sign extended.
  - Example: sll x5, x2, x4
    srl x6, x1, x4
    sra x7, x3, x4
  - What is the arithmetic effect by shifting?

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Assembly Instructions

- Different types of instructions

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

- I-type
  - Register-Immediate type
  - Has two operands (one accessed from source register, another a constant/immediate, sign-extended) and one output (saved to destination register)
  - Can do arithmetic, logic and load from main memory

RV32I I-type Arithmetic

- Syntax of instructions
  - Addition: addi rd, rs1, imm

    Adds imm to rs1, stores the result to rd, and imm is a signed number.
  - Example: addi x5, x4, 10

    addi x6, x4, -10

- Similarly, andi/ori/xori/slti/sltui
  - All the imm's are sign-extended (details will be covered in later lectures)

- slli/srli/srai shift rs1 by the lower 5-bits of imm, srai is distinguished by using one of the higher bit of the imm (or funct7 field)

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x3	x4
	x5
	x6
	x7

RV32I Excercise

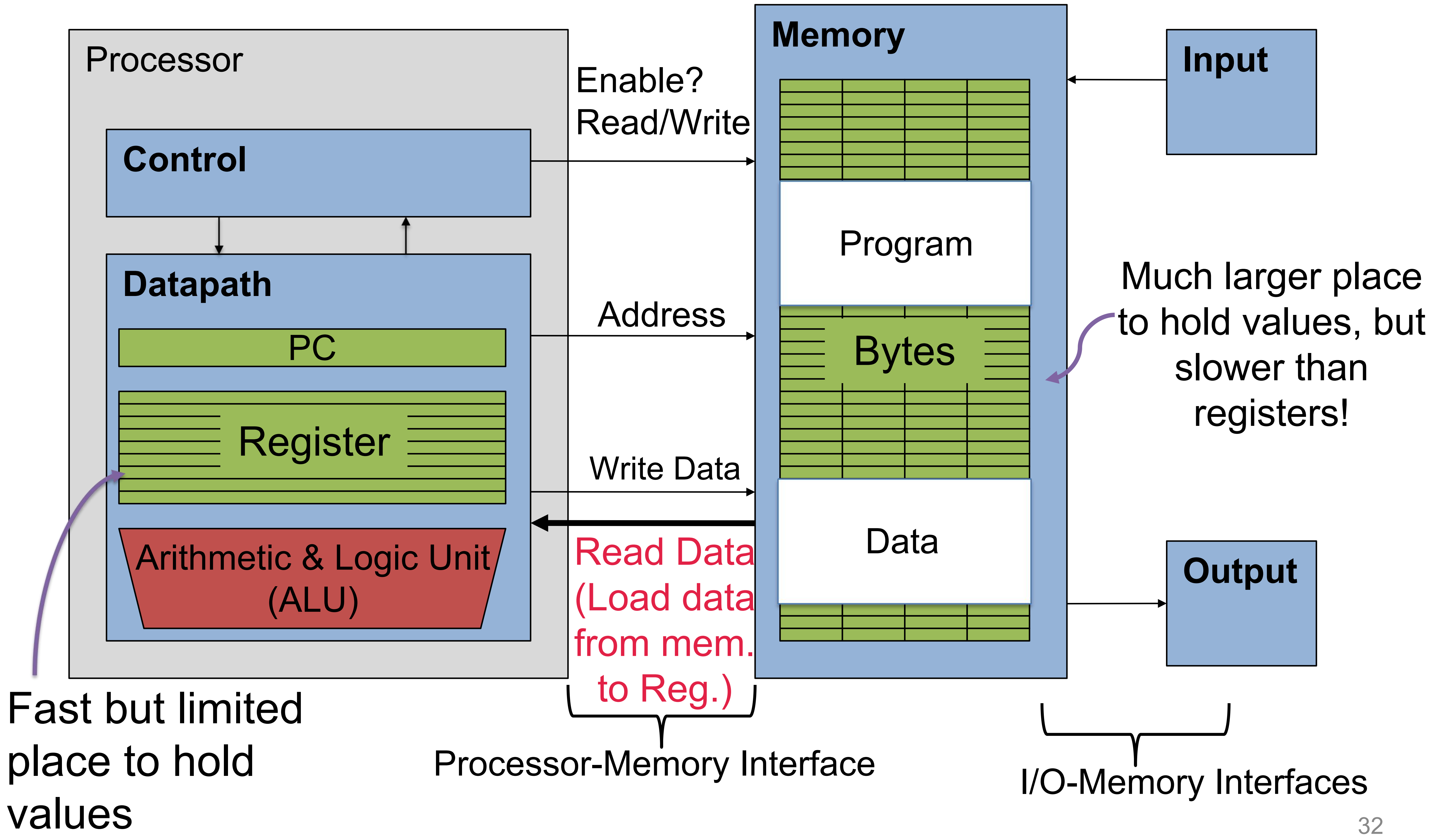
```
addi x1, x0, -1
or   x2, x2, x1
add  x3, x1, x2
slt  x4, x3, x1
sra  x5, x3, x4
sub  x0, x5, x4
```

Registers

Value	Register
0	x0/zero
0	x1
0	x2
0	x3
0	x4
0	x5
0	x6
0	x7

- Register zero (**x0**) is 'hard-wired' to 0;
- By convention RISC-V has a specific **no-op** instruction...
  - **addi x0 x0 0**
  - You may need to replace code later: No-ops can fill space, align data, and perform other options
  - Practical use in jump-and-link operations (covered later)

RV32I I-type Load


Processor
Control
Datapath
PC
Register
Arithmetic & Logic Unit (ALU)
Enable?
Read/Write
Address
Write Data
Read Data (Load data from mem. to Reg.)
Memory
Program
Bytes
Data
Input
Output
Much larger place to hold values, but slower than registers!
Fast but limited place to hold values
Processor-Memory Interface
I/O-Memory Interfaces

Assembly Instructions—Load

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

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

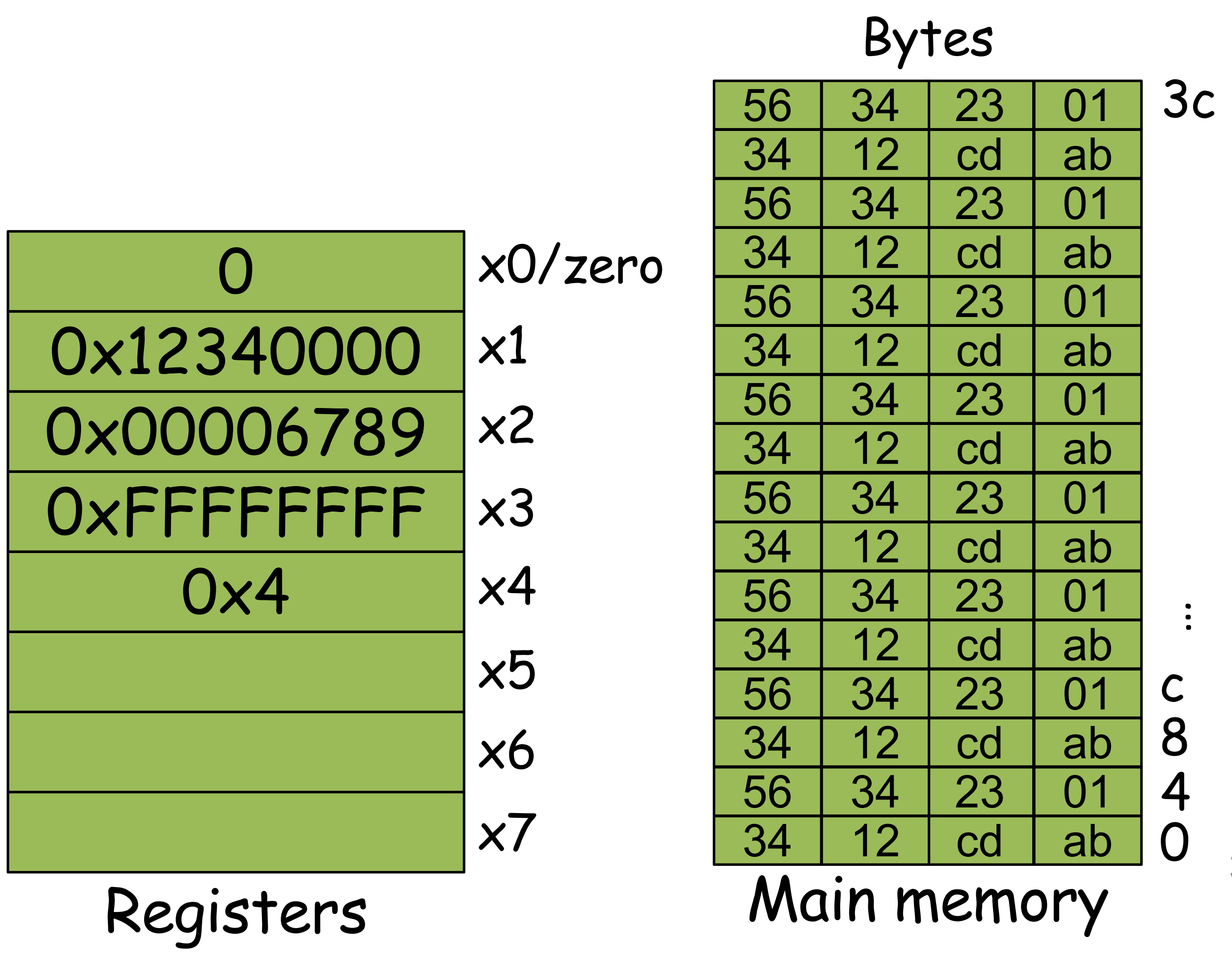
- lw rd, imm(rs1) : Load word at addr. to register rd

addr.= (number in rs1) + imm

- Example

lw x1, 12(x4)

addr.= 4 + 12 = (10)HEX



Big Endian vs. Little Endian

Big-endian and little-endian from Jonathan Swift's *Gulliver's Travels*

- **The order in which BYTES are stored in memory**
- **Bits always stored as usual. (E.g., 0xC2=0b 1100 0010)**

Consider the number 1025 as we normally write it:

BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Big Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE0	BYTE1	BYTE2	BYTE3
00000001	00000100	00000000	00000000

Examples

Names in the West (e.g. Siting, Liu)

”Network Byte Order”: most network protocols

IBM z/Architecture; very old Macs

Little Endian

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Examples

Names in China (e.g. LIU Siting)

CANopen

Intel x86; **RISC-V** (can also support big-endian)

en.wikipedia.org/wiki/Big_endian

big-endian: MIPS, IA-64, PowerPC

Assembly Instructions—Load

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

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

- lw rd, imm(rs1) : Load word at addr. to register rd

addr.= (number in rs1) + imm

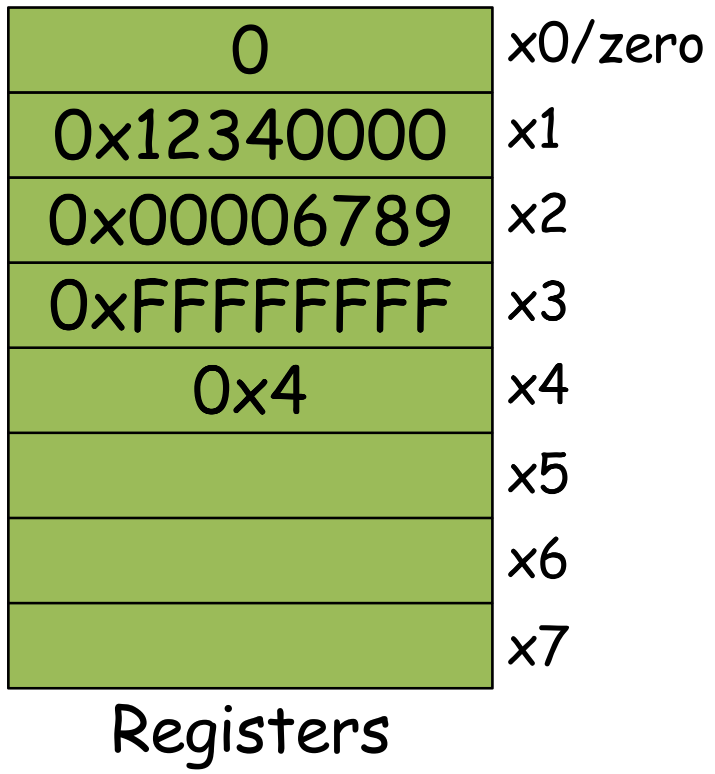
- Example

lw x1, 12(x4)

addr.= 4 + 12 = (10)HEX

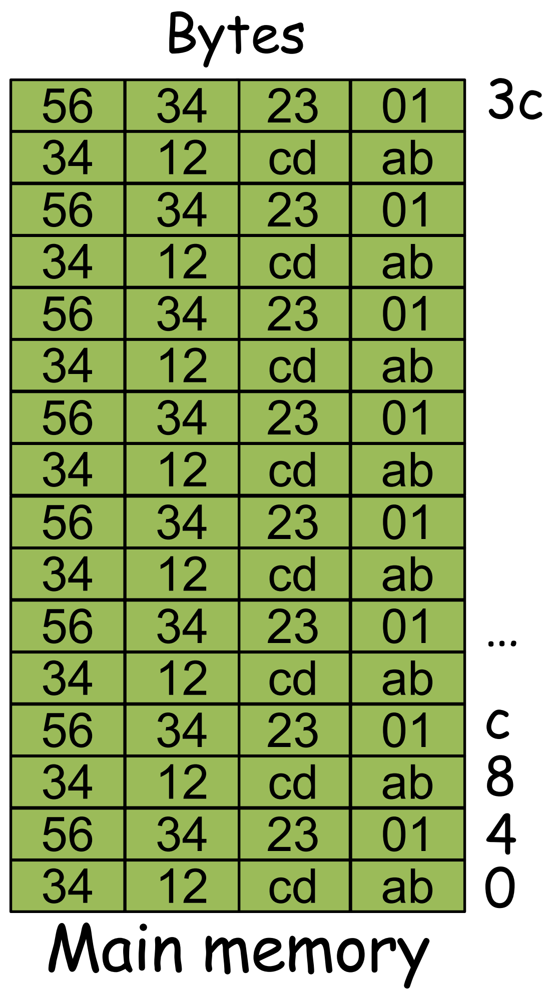
- C code example

```
int A[100];
/*assume &A[0] = 4*/
G = A[3];
/*load G/A[3] to x1/
```



Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7



Bytes

56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	…
34	12	cd	ab	
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Main memory

Assembly Instructions—Load

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

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

- lb/lbu rd, imm(rs1): Load signed/unsigned byte at addr. to register rd

addr.= (number in rs1) + imm

- Example

lb x1, 12(x4)

addr.= 4 + 12 = $(10)_{\text{HEX}}$

lbu x1, 12(x4)

Registers	
0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Bytes				
56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	…
34	12	cd	ab	
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Assembly Instructions—Load

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

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

- lh/lhu rd, imm(rs1): Load signed/unsigned halfword at addr. to register rd (similar to lb/lbu)

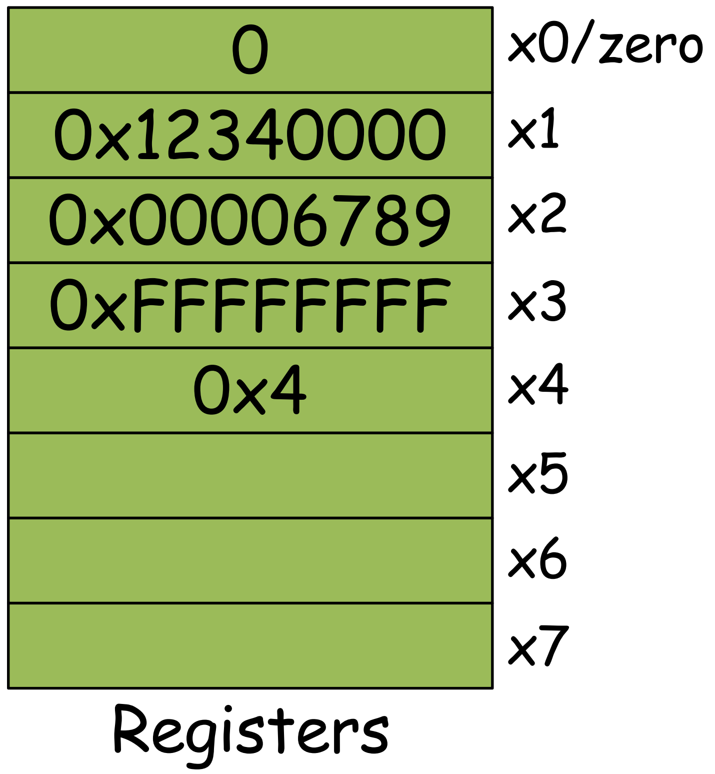
addr.= (number in rs1) + imm

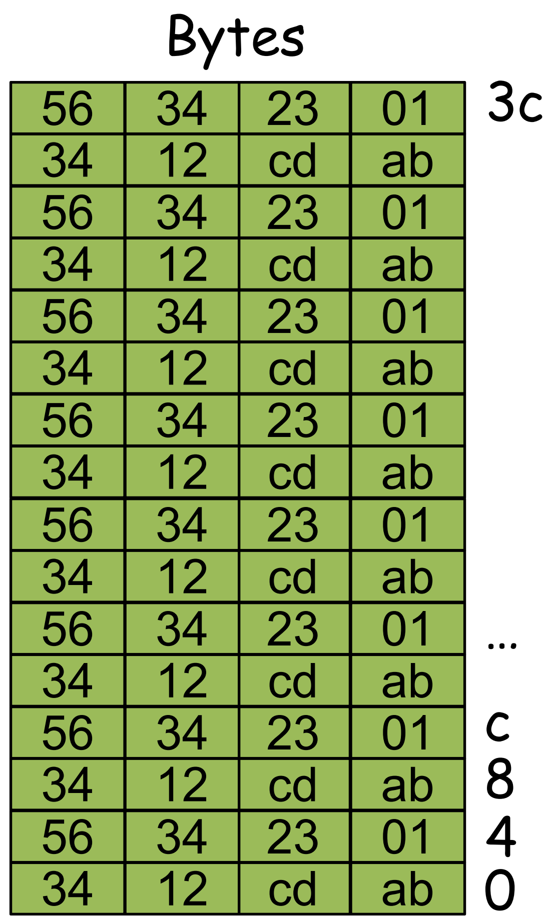
- Example

lh x1, 12(x4)

addr.= 4 + 12 = (10)HEX

lhu x1, 12(x4)





Assembly Instructions—S-Type Store

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

- sw rs2, imm(rs1): Store word at rs2 to memory addr.

addr.= (number in rs1) + imm

- Example

sw x1, 12(x4)

addr.= 4 + 12 = $(10)_{HEX}$

- C code example

```
int A[100];
/* &A[0] => x4 */
A[3] = h;
/* h in rs2 => A[3] */
```

Registers	
0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7

Bytes

56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	...
34	12	cd	ab	
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Registers

Assembly Instructions—S-Type Store

- RV32I is a load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on CPU registers.

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

- sw rs2, imm(rs1) : Store word at rs2 to memory addr.

addr.= (number in rs1) + imm

- Example

sw x1, 12(x4)

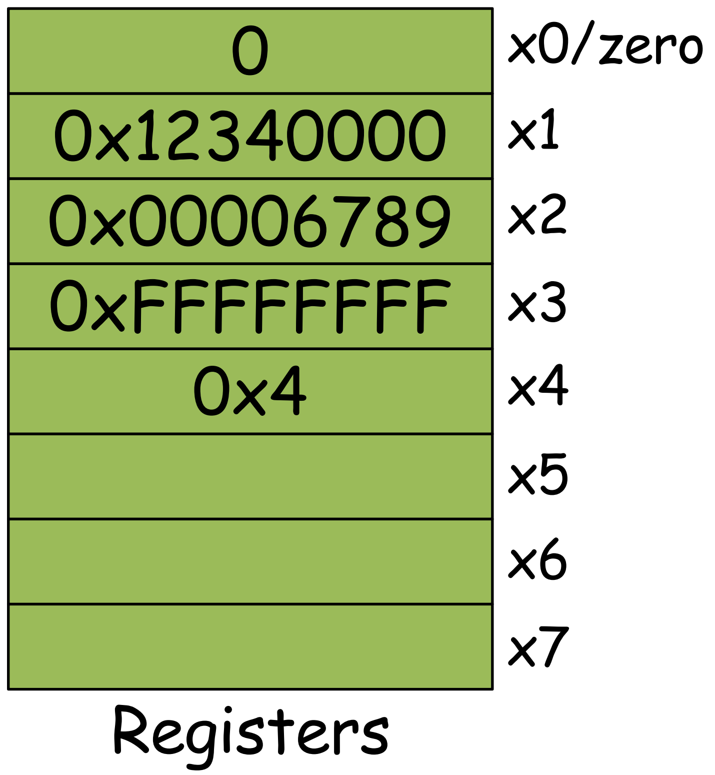
addr.= 4 + 12 = (10)HEX

- Similarly,

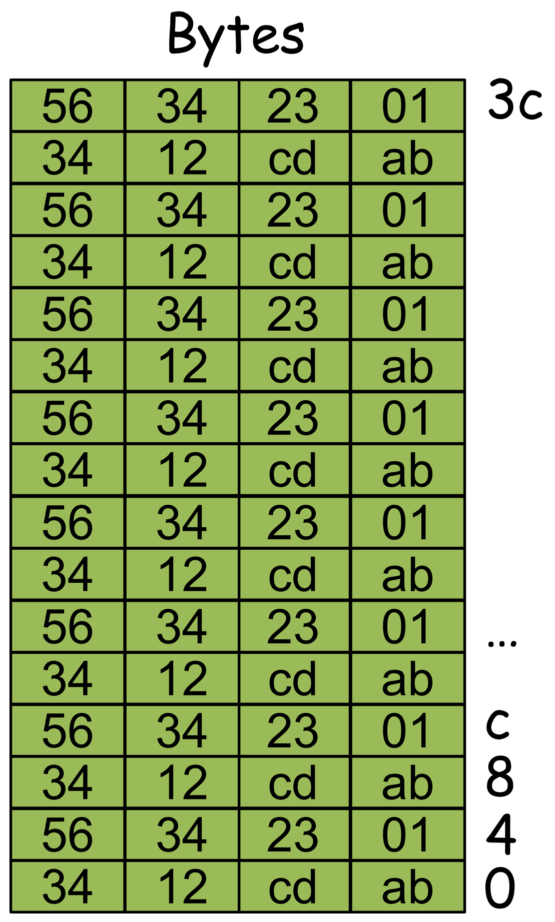
sh: Store lower 16 bits at rs2

sb: Store lower 8 bits at rs2

No shu/sbu?



Registers	
0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7



Bytes				
56	34	23	01	3c
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	
34	12	cd	ab	
56	34	23	01	...
34	12	cd	ab	
56	34	23	01	c
34	12	cd	ab	8
56	34	23	01	4
34	12	cd	ab	0

Memory Alignment

- RISC-V standard does not require that integers be word aligned...
  - But it can be very very bad if you don't make sure they are...
- Consequences of unaligned integers
  - Slowdown: The processor is allowed to be a lot slower when it happens
    - In fact, a RISC-V processor may natively only support aligned accesses, and do unaligned-access in software!
      An unaligned load could take hundreds of times longer!
    - Lack of atomicity: The whole thing doesn't happen at once... can introduce lots of very subtle bugs
- So in practice, RISC-V recommends integers to be aligned on 4- byte boundaries; halfword 2-byte boundaries

Excercise! What's in x12?

```
addi x11,x0,0x4F6
sw x11,0(x5)
lb x12,1(x5)
```

A: 0x0

[✓ B: 0x4]

C: 0x6

D: 0xF

E: 0xFFFFFFFF

Excercise! What's in x12?

```
addi x11,x0,0x85F6
sw x11,0(x5)
lb x12,1(x5)
```

A:	0x8
B:	0x85
C:	0xC
D:	0xBC
E:	$\boxed{\text{0XFFFFFF85}}$
F:	0XFFFFFFF8
G:	0XFFFFFFFC
H:	0XFFFFFFBC

Summary

- RISC-V ISA basics: (32 registers, referred to as x0-x31, x0=0)
- Simple is better
- One instruction (simple operation) per line (RISC-V assembly)
- Fixed-length instructions (for RV32I)
- 6 types of instructions (depending on their format/function)
- Instructions for arithmetics, logic operations, register-memory data exchange (load/store word/halfword/byte)
- RISC-V is little-endian
- Load-store architecture