FISF130020.01: Introduction to Computer Science

Lecture 2: Computer Architecture

Hui Xu

xuh@fudan.edu.cn

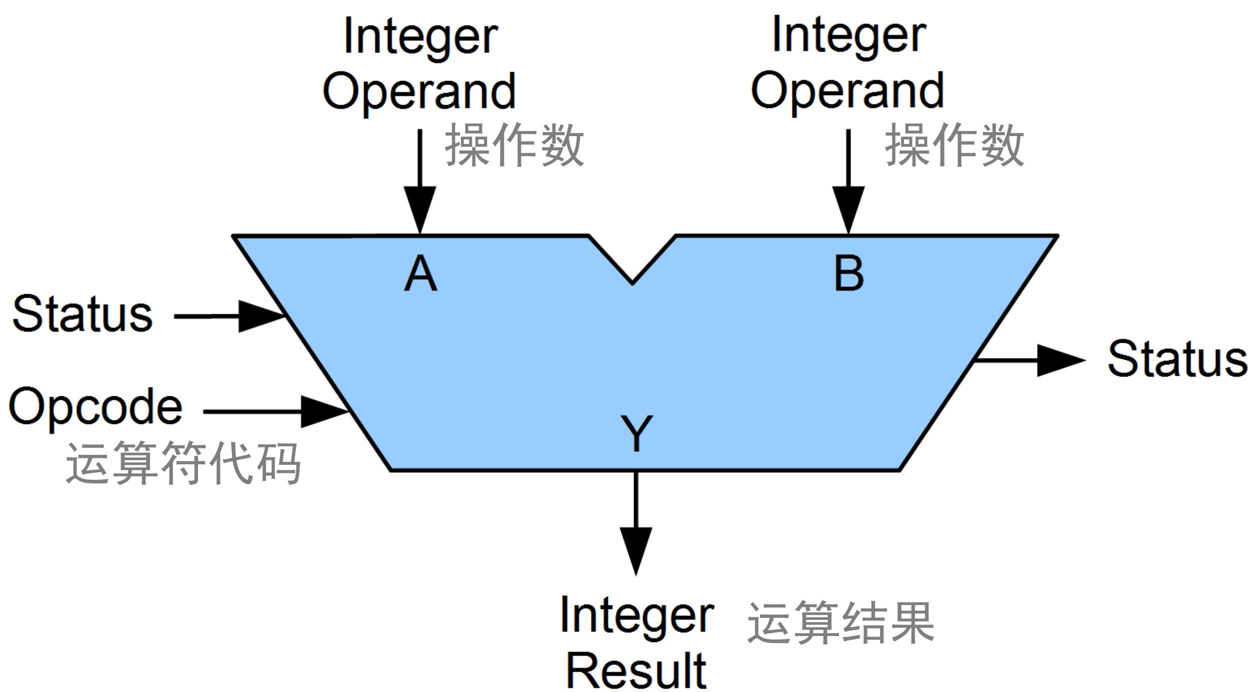
Outline

- 1. CPU and ISA
- 2. I/O and More
- 3. In-class Practice

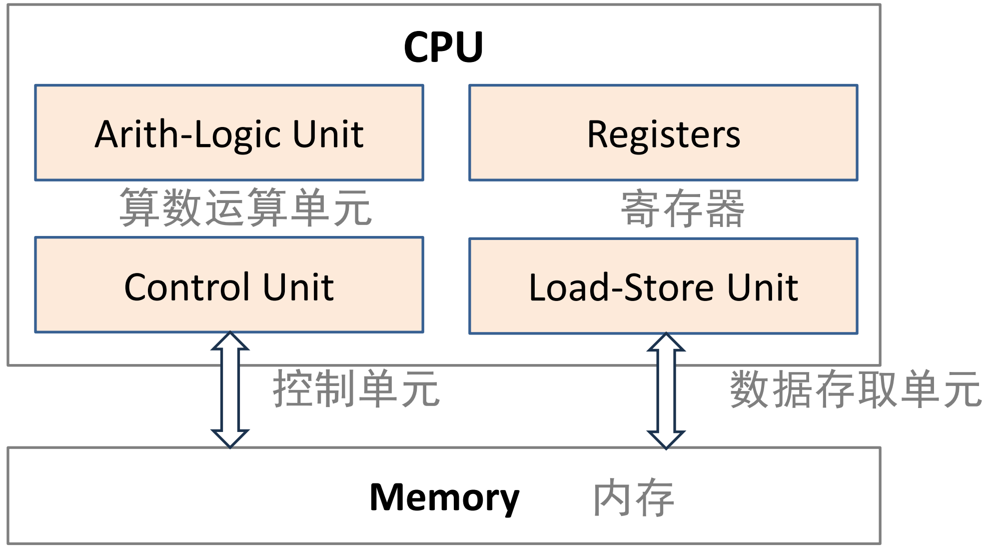
1. CPU and ISA

Recall: Transistor => Logic Gate => ALU

晶体管 逻辑门 算数运算单元

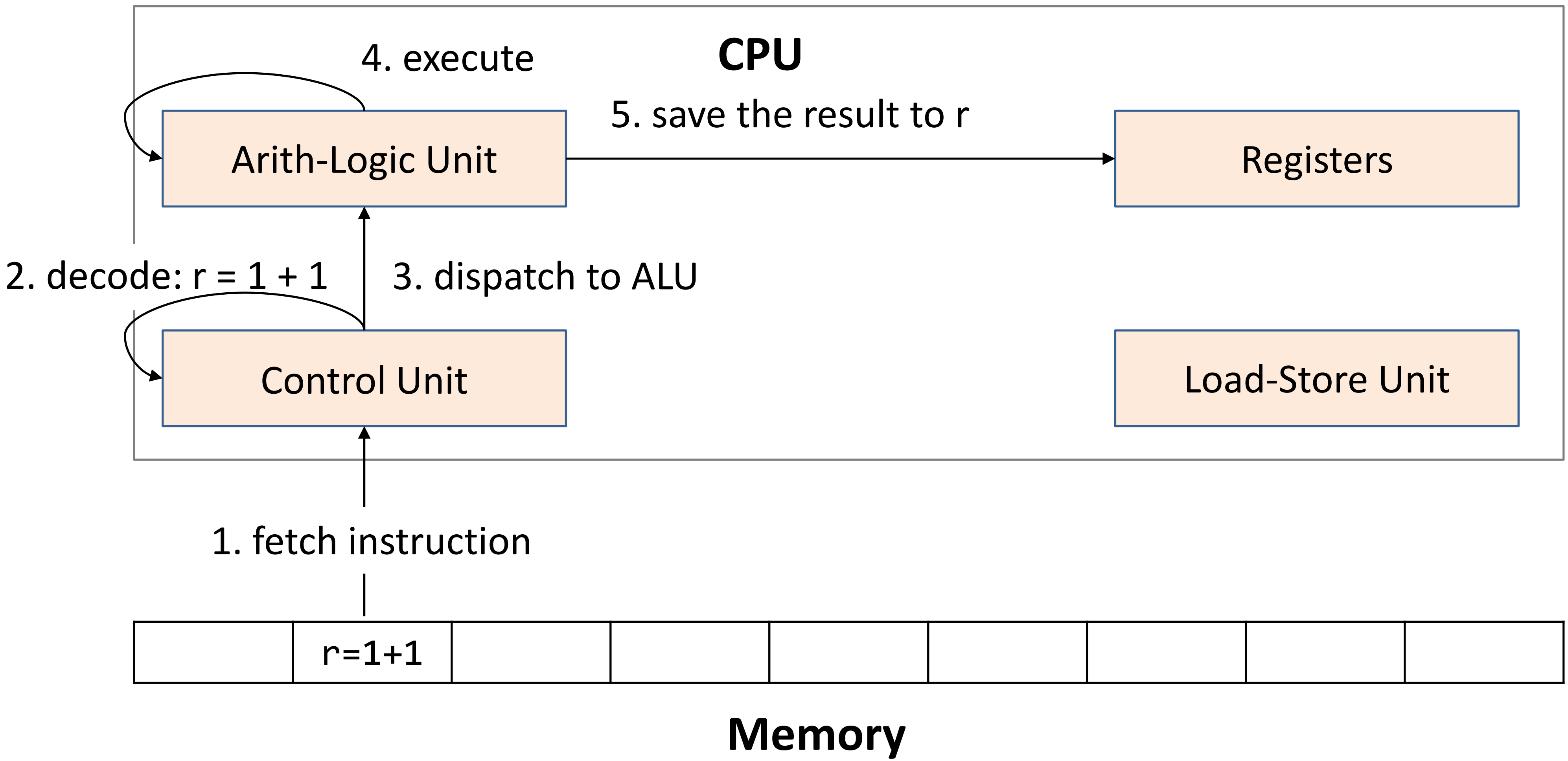


CPU: Central Processing Unit 中央处理器



- Arithmetic-logic Unit: perform arithmetic/logic operations
- Control unit: fetch instructions from memory; decode and dispatch
- Load-store unit: load/store data from/to memory
- Registers: supply operands to the ALU and save the execution result
- Memory: save data and instructions

Example: r = 1 + 1



CPU
4. execute
5. save the result to r
Arith-Logic Unit
Registers
2. decode: r = 1 + 1
3. dispatch to ALU
Control Unit
Load-Store Unit
1. fetch instruction
r=1+1
Memory

Memory (Random-Access Memory or RAM)

- Read or write a data item based on its address in memory
- Each data unit generally takes 1 byte or 8 bits
- Access any data with the same latency irrespective of their locations

		r=1+1							
Address:	0x00	0x04	0x08	0x0B	0x10	0x14	0x18	0x1B	0x20

Example: z = x + y

```
z = x + y
```

Compile

编译

```
r1 = load x
r2 = load y
r2 = add r1, r2
store r2, z
```

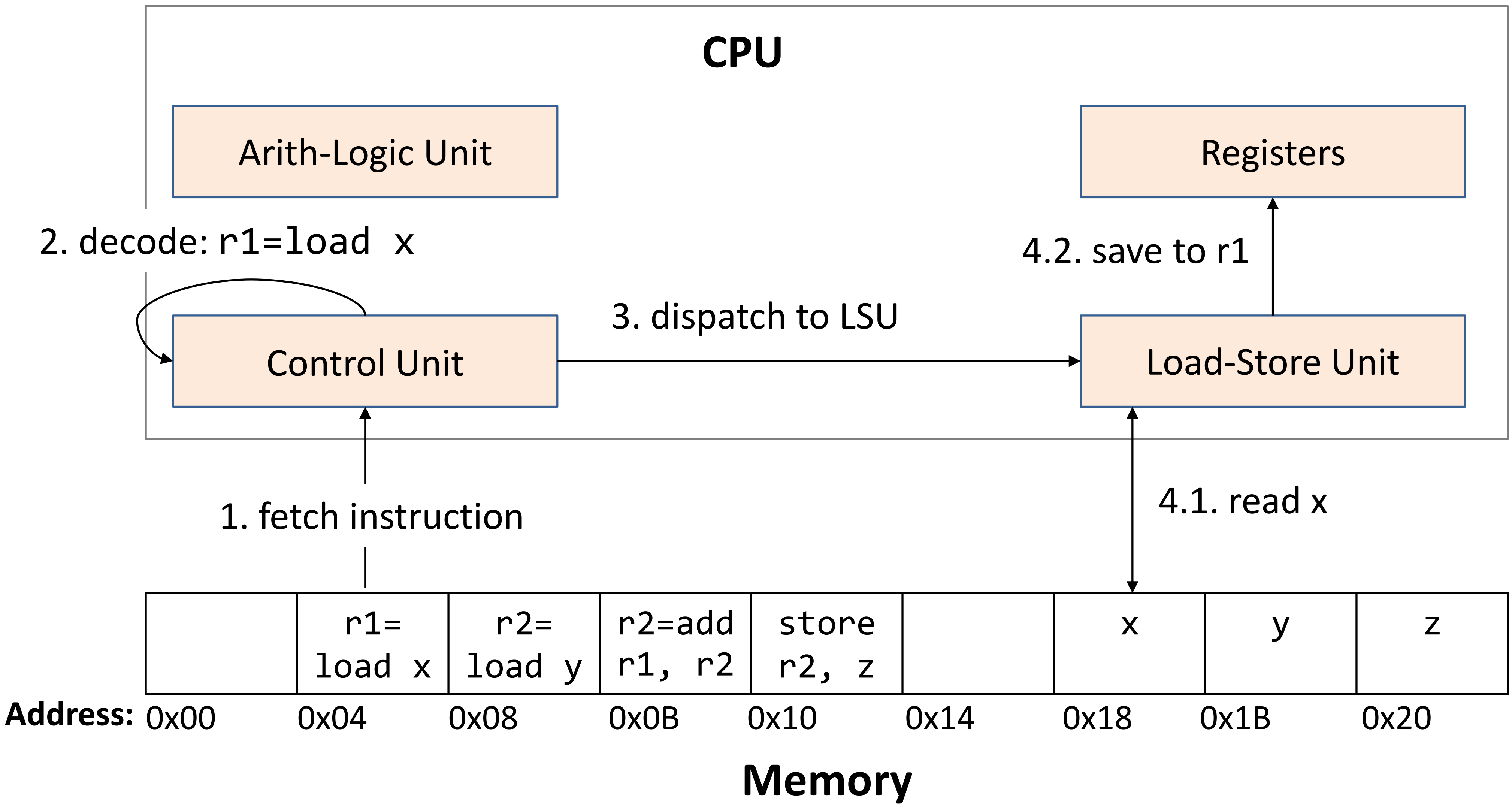
High-level Code

代码

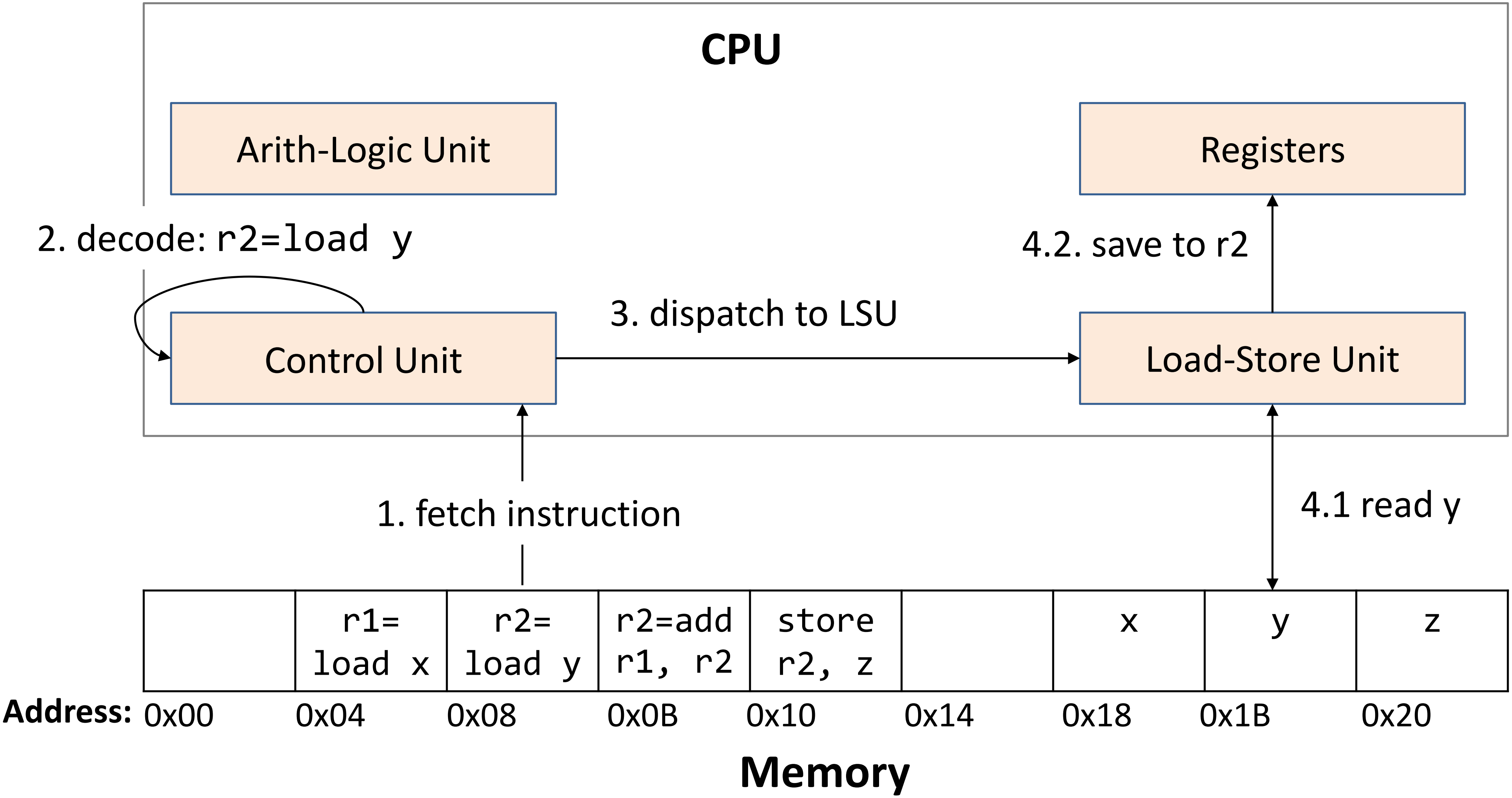
Low-level Instructions

指令

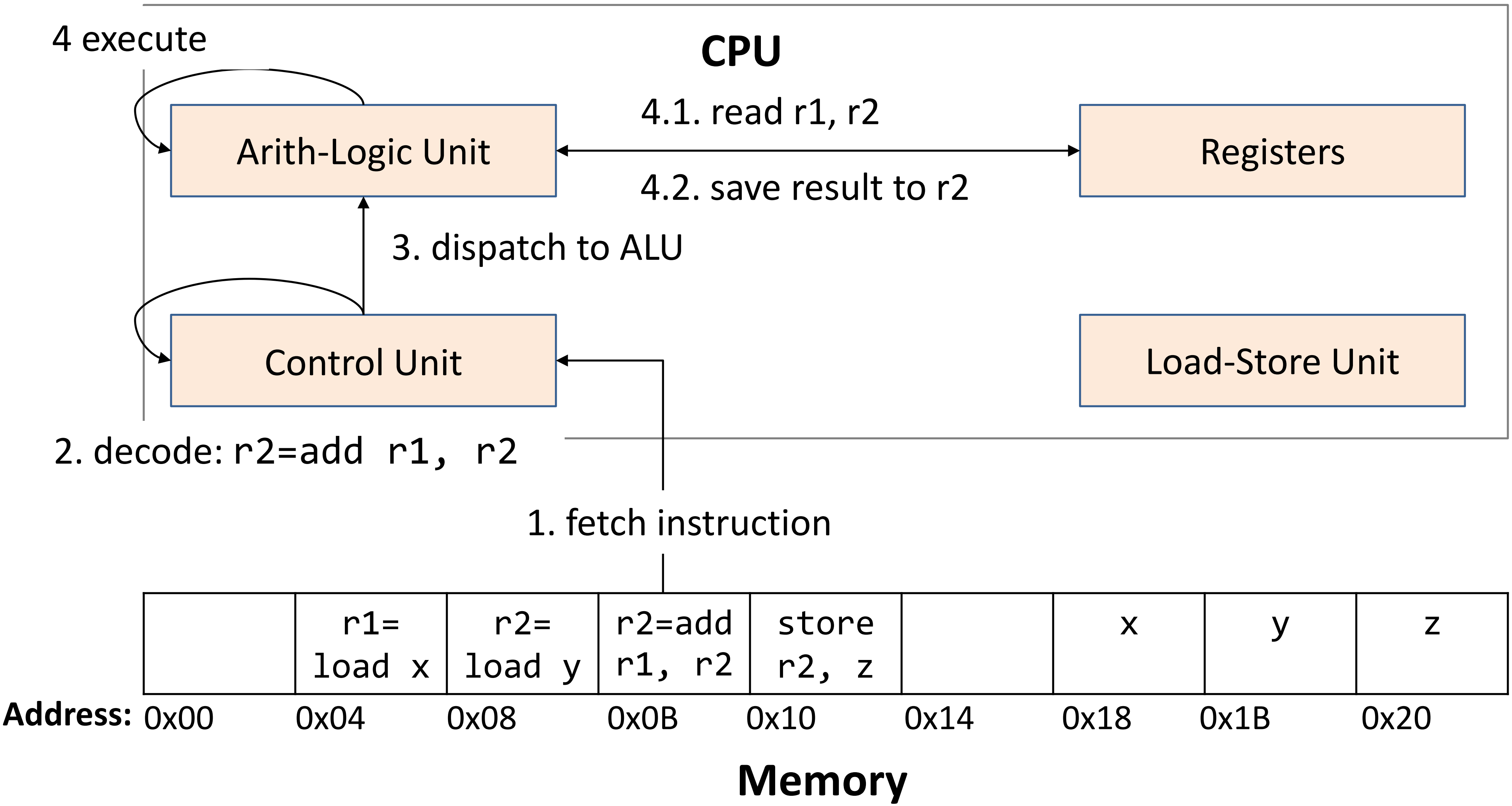
Step 1: r1 = load x



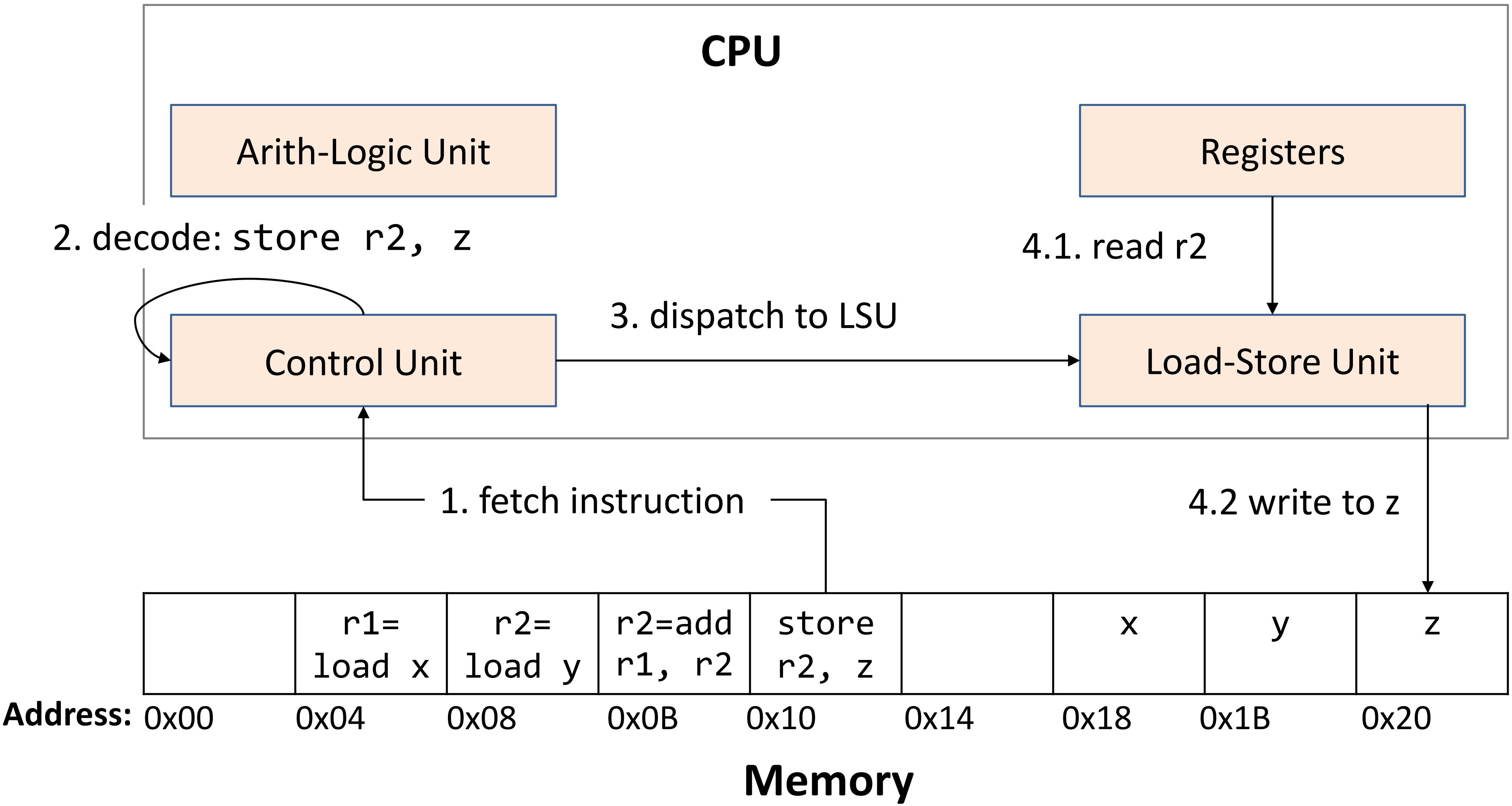
Step 2: r1 = load y



Step 3: r1 = load y



Step 4: store r2, z



Question

- If a CPU employs 32 bits to represent a memory address, what is the maximum memory space the CPU can support?

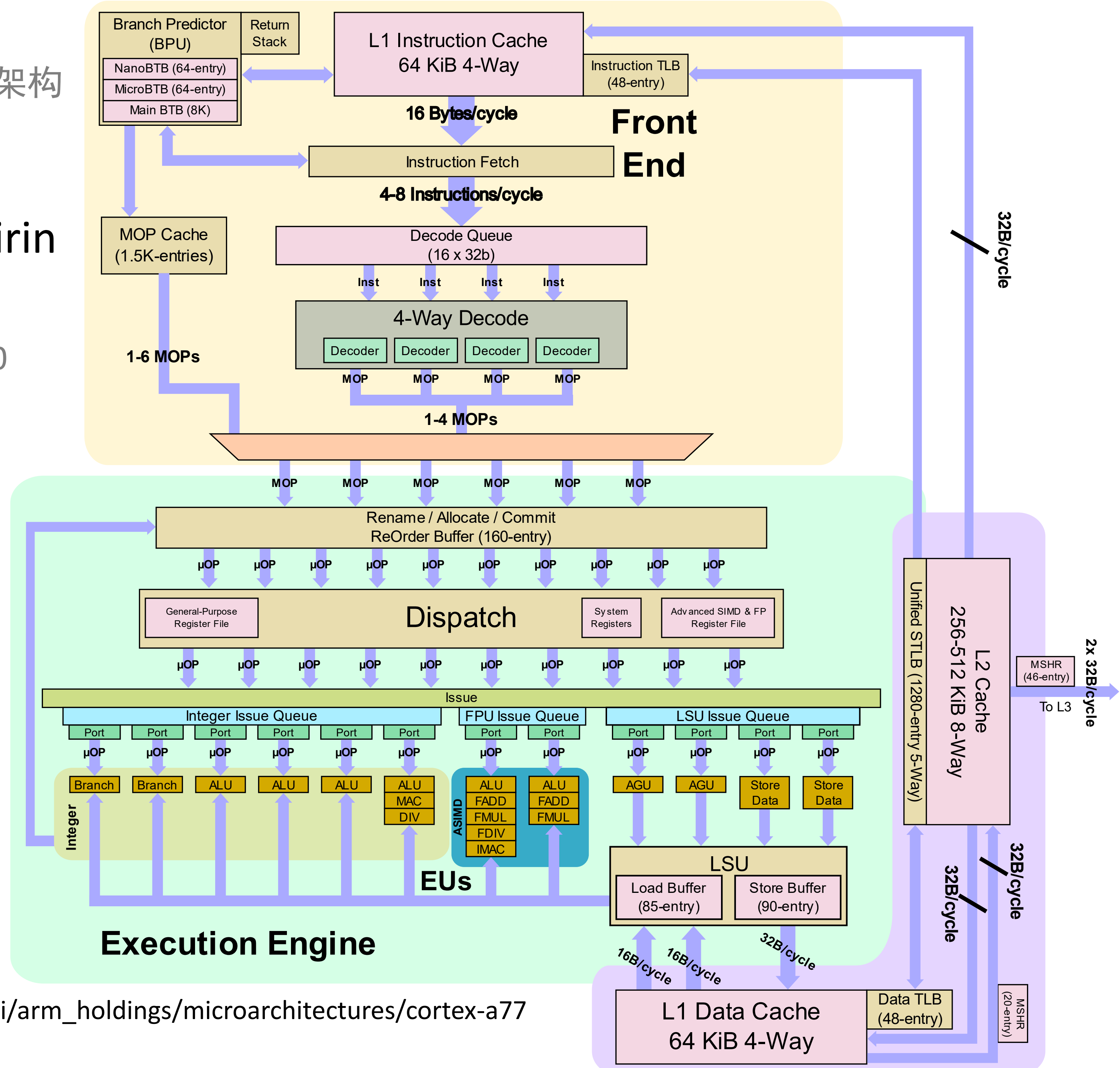
ISA: Instruction Set Architecture 指令集架构

- RISC: Reduced Instruction Set Computer 精简指令集
  - Separate data load/store and computation into different instructions
  - *e.g.,* AArch/ARM、RISC-V
- CISC: Complex Instruction Set Computer 复杂指令集
  - An instruction can do both data load/store and computation
  - *e.g.,* X86, X86-64

Cortex-a77 微架构

- ISA: ARMv8-A
- *e.g.*, HiSilicon Kirin 9000

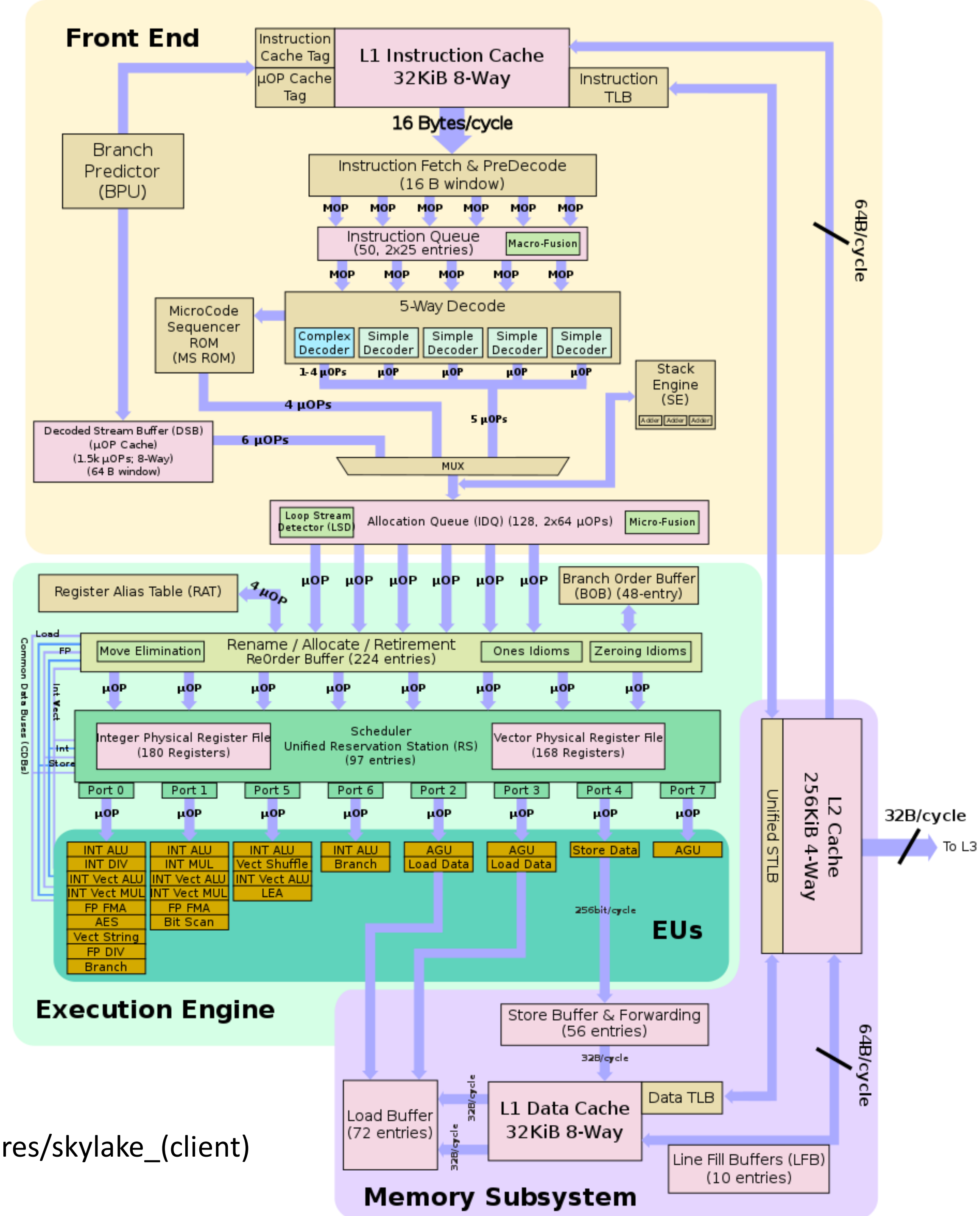
海思公司的麒麟9000



https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a77

Skylake 微架构

- ISA: X86-64
- *e.g.*, Intel i7, i9



https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)

ARMv8-A Instructions

```
mov x0, #5
mov x1, #10
add x2, x0, x1
sub  x3, x0, x1
mul  x4, x0, x1
sdiv x5, x0, x1
```

```
x0 = 5
x1 = 10
x2 = x0 + x1
x3 = x0 - x1
x4 = x0 * x1
x5 = x0 / x1
```

ARMv8-A Instructions: Data Load/Store

`ldr x1, [x0]`	x1 = [x0]
`ldr x2, [x0, #12]`	x2 = [x0 + 12]
`add x3, x1, x2`	x3 = x1 + x2,
`str x3, [x0, #8]`	[x0 + 8] = x3

```
add x2, x1, [x0]
```

Not allowed (CISC operation)

ARMv8-A Instructions: Control 控制指令

address	Instructions
0x20	`cmp x0, x1`
0x24	`ble .addr1`
...	...
0x104	`.addr1:`
...	...

```
[flag] = compare(x0, x1)
if(flag:<=) PC = 0x104
...
```

PC: 下一条指令地址

Instructions in Memory

- 32bit fixed-length for ARMv8-A　　每条指令占固定内存空间
- Limitation: cannot employ some large numbers as operands

		r1= load x	r2= load y	r2=add r1, r2	store r2, z		x	y	z
Address:	0x00	0x04	0x08	0x0B	0x10	0x14	0x18	0x1B	0x20

31	30	29	28 27 26 25 24	23 22	21	20 … 16	15 … 10	9 … 5	4 … 0
sf	0	0	0 1 0 1 1	shift	0	Rm	imm6	Rn	Rd
	op	S							

```
add rn, rm, rd
```

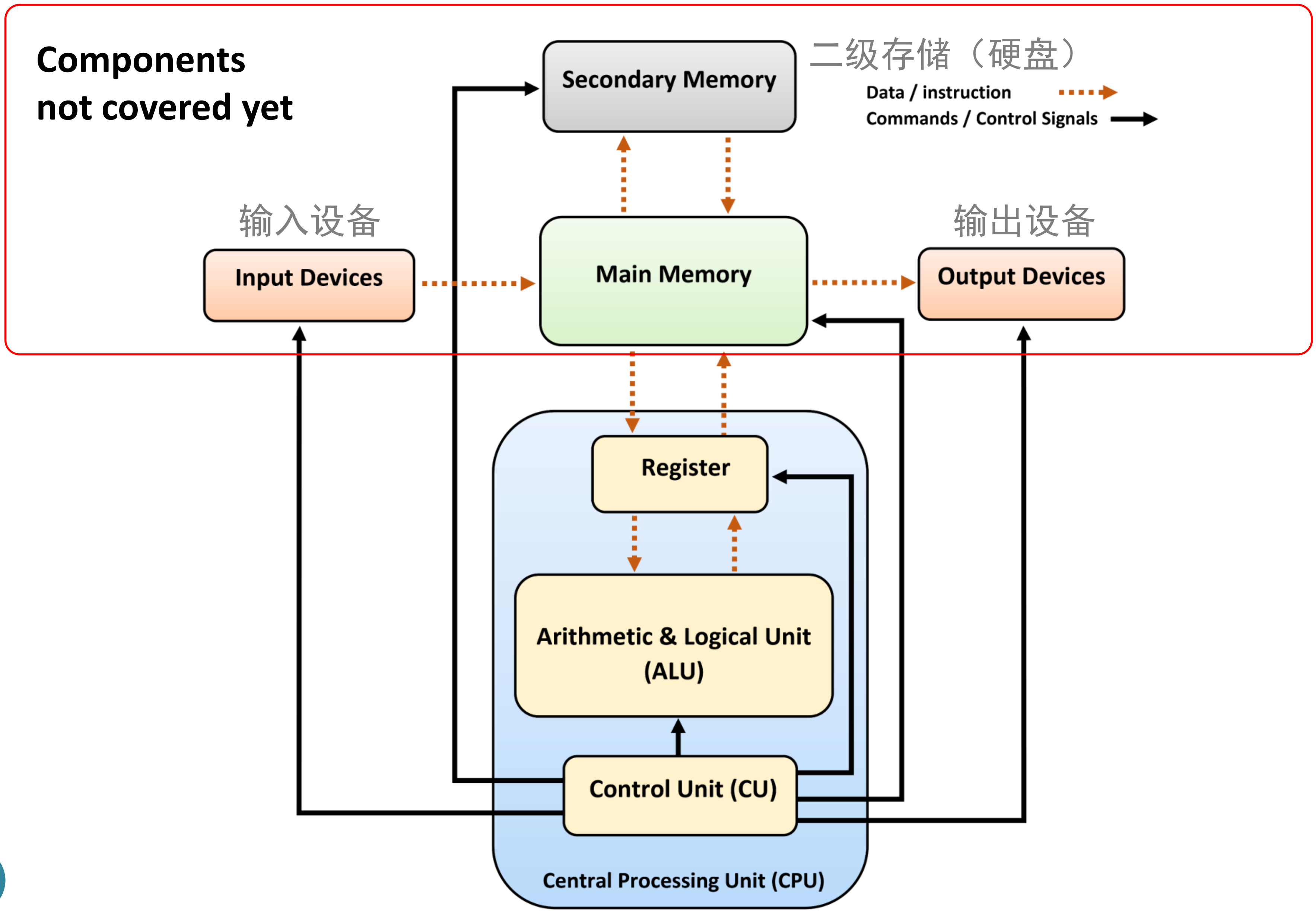
31	30	29	28 27 26 25 24 23	22	21 … 10	9 … 5	4 … 0
sf	0	0	1 0 0 0 1 0	sh	imm12	Rn	Rd
	op	S					

```
add rd, rn, imm12
add rd, rn, #4095
add rd, rn, #4097
```

invalid

2. I/O and More

Computer Architecture



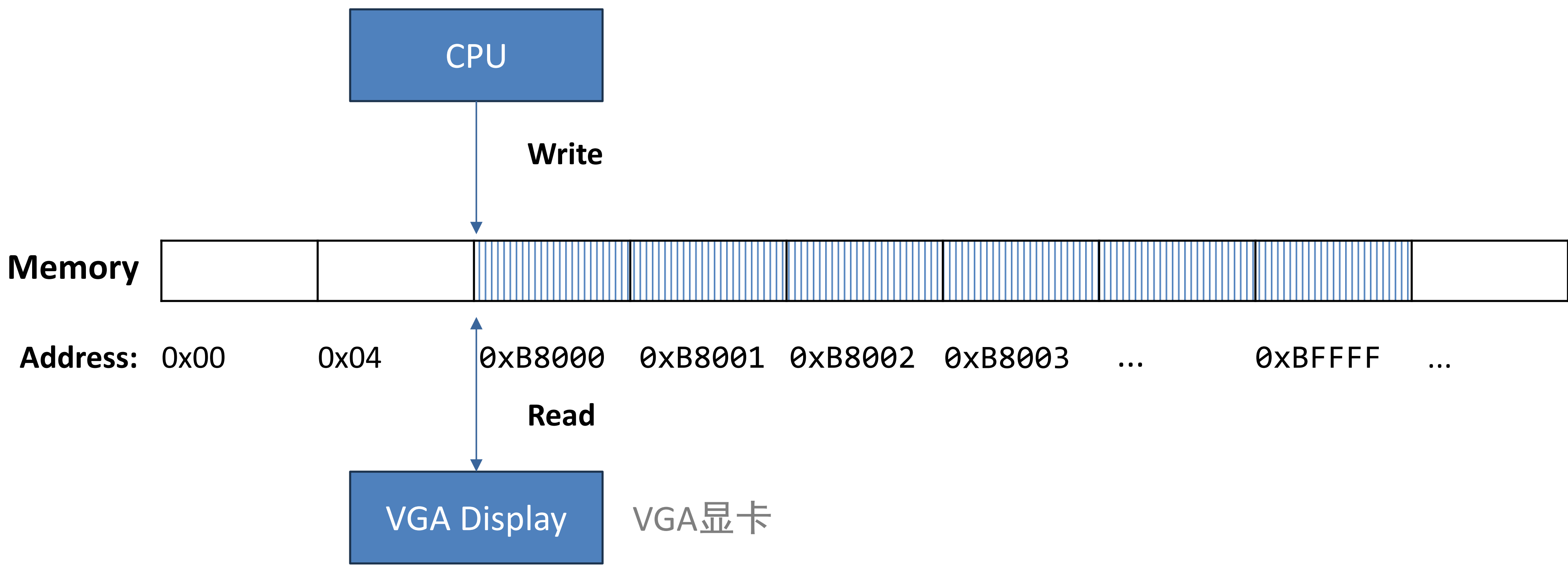
I/O Devices

- Input device: keyboard, mouse, *etc*
- Output device: display, printer, *etc*
- Question: how does a CPU handle I/O operations?

Option 1: Memory-mapped I/O

- Read/write data to I/O devices via memory access
- Addressing achieved via PCI (peripheral component interconnect)

外设互联总线



Output: Video as an Example

- VGA (Video Graphics Array): video display controller 15-pin
- Mapped to the PC's memory: 0xA0000 - 0xBFFFF:
- 0xB8000 for color text mode (32 KB)

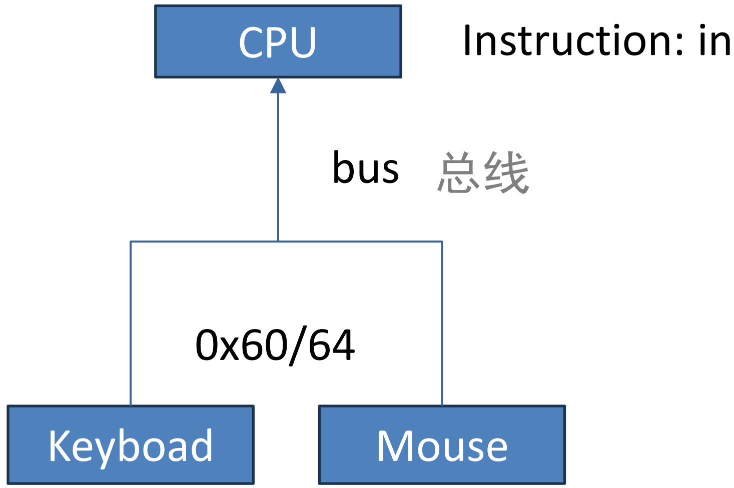
```
mov ax, 0xB8000    ; 将显卡地址0xB8000保存到ax寄存器
mov es, ax         ; 将ax寄存器的值保存到es寄存器

mov di, 0          ; 将显示位置0保存到di寄存器
mov al, 'A'        ; 将显示字符'A'保存到al寄存器
mov ah, 0x0F       ; 将显示颜色0x0F保存到ah寄存器
mov [es:di], al    ; 将al的值写如内存地址es+di
mov [es:di+1], ah  ; 将ah的值写如内存地址es+di+1
```

Question

- Why does the available physical memory not match the original purchase capacity?

Option 2: Port-based I/O

- Read/write data to I/O devices via port numbers.



Input is More Complicated

- Challenge: when/how many data arrives is not controlled by the CPU
  - Polling: let the CPU check particular signals periodically? How frequently?
  - By interrupt (trigger-based) via particular CPU pins

```
wait_for_input:
    in al, 0x64    ; 读取0x64端口信号到al
    test al, 0x01  ; 测试al是否为1
    jz wait_for_input ; 如果是0则继续轮询

read_input:
    in al, 0x60    ; 从0x60端口读入字符
```

Polling

轮询模式读取键盘输入

```
in al, 0x60
...
out 0x20, 0x20 ; 中断响应结束
```

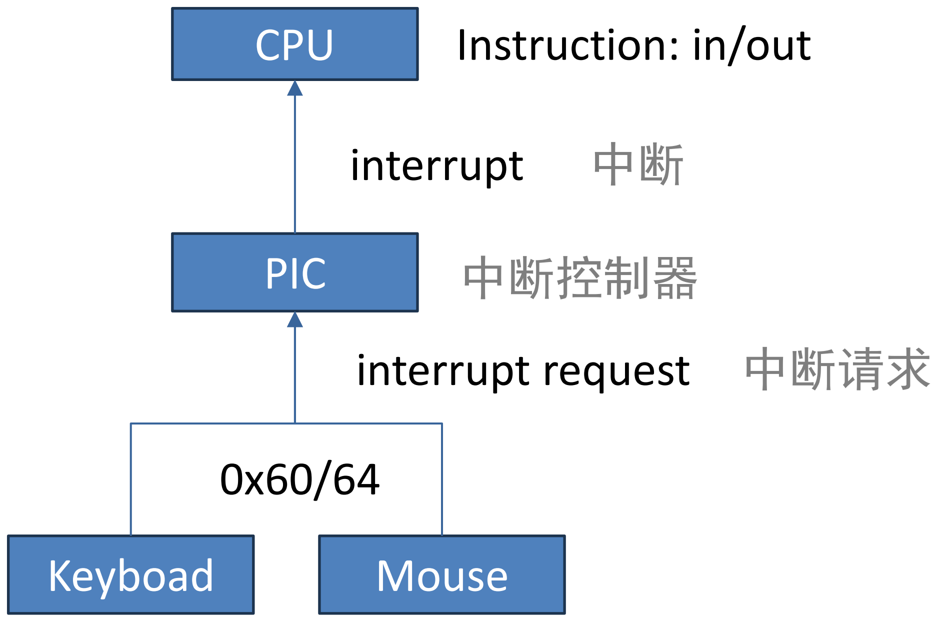
Interrupt

中断模式读取键盘输入

Input is More Complicated

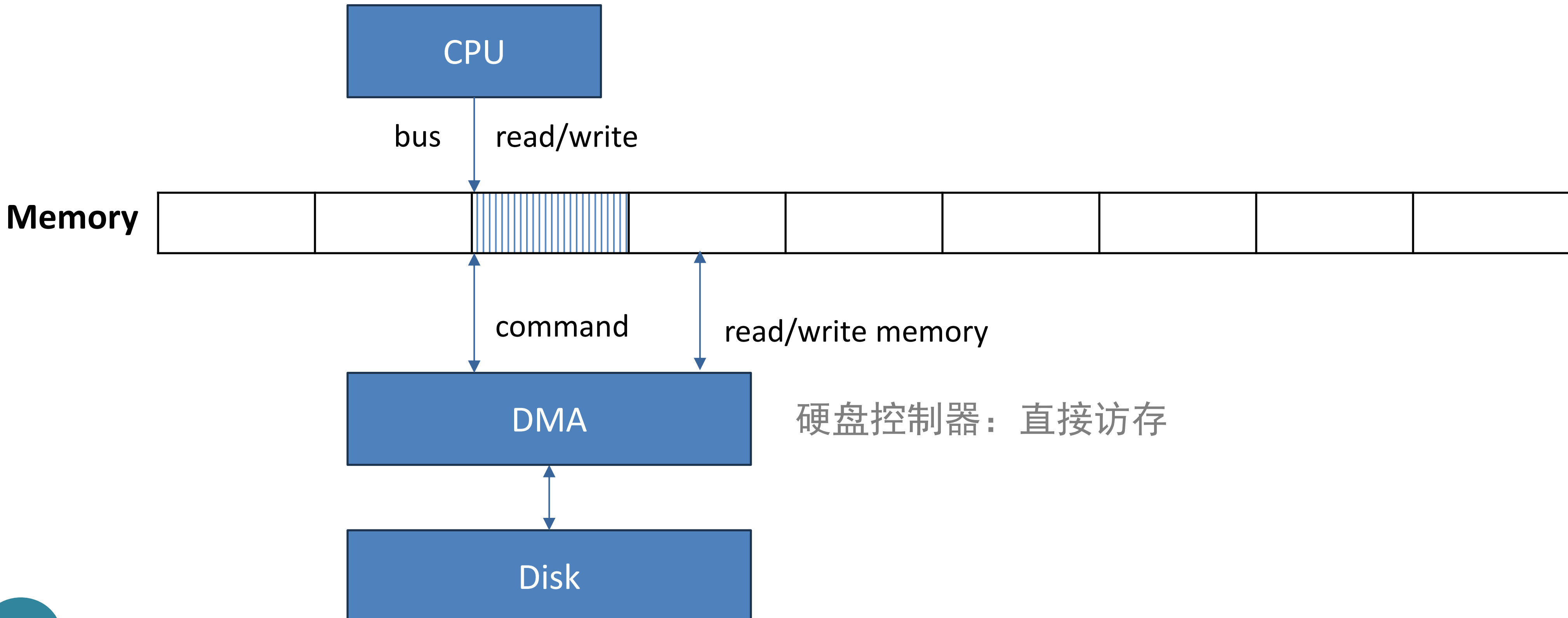
- What if data arrives from multiple input devices simultaneously?
  - First come first serve? May lead to denial-of-service
  - Priority/queuing via PIC (programmable interrupt controller)

优先级　排队



Disk I/O

- Disk read/write operations are much more slower than memory
- Offload the workload to disk controller via direct memory access

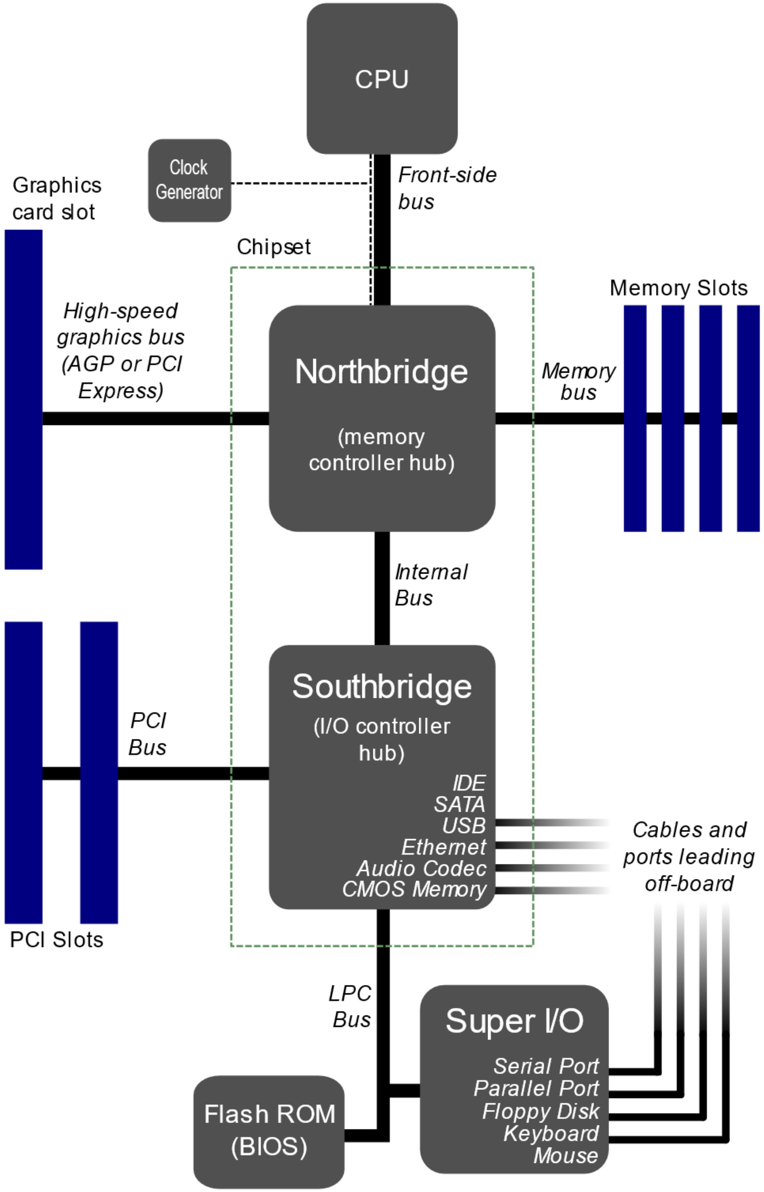


硬盘控制器：直接访存

Question

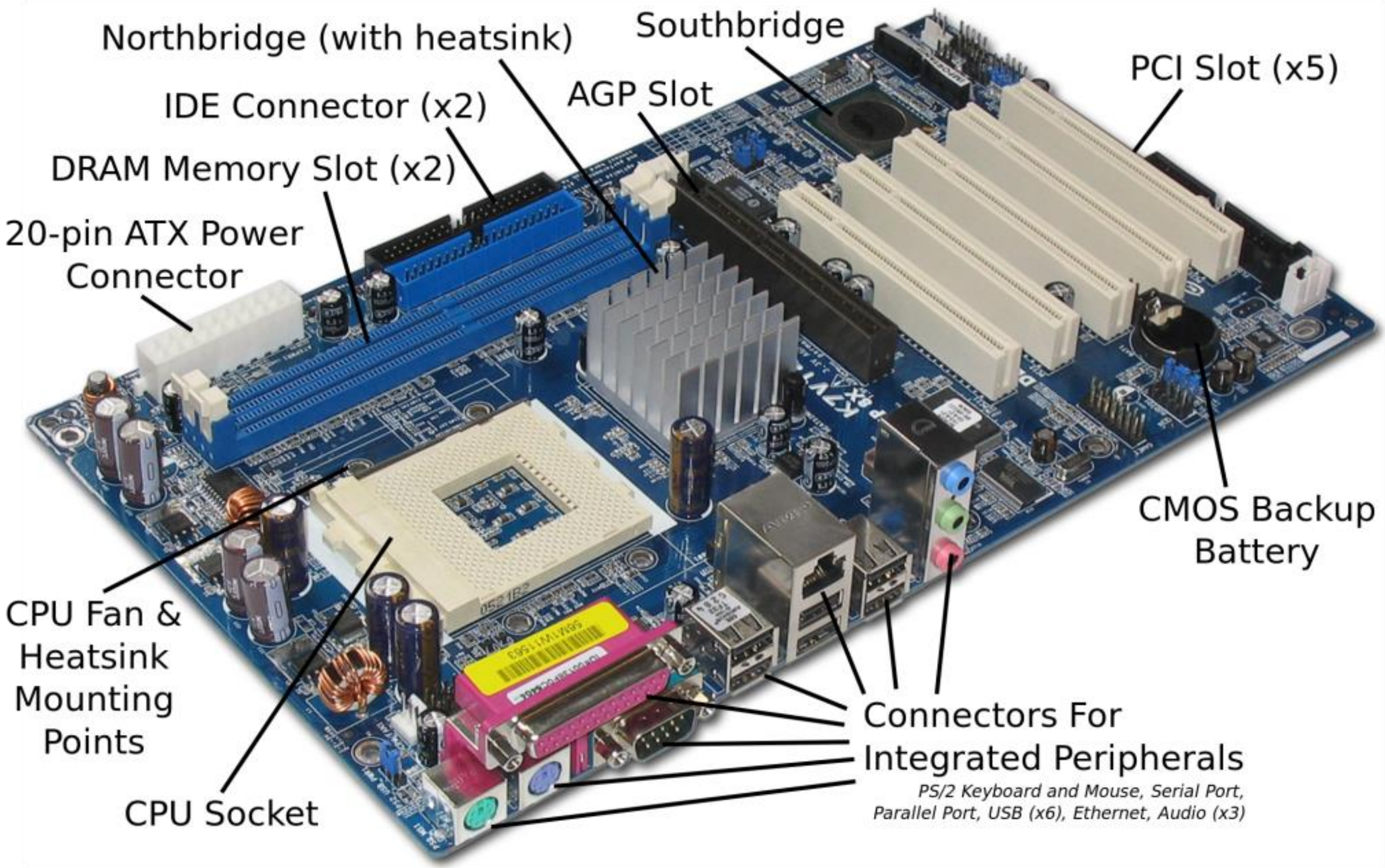
- Why does the available disk space not match the capacity at the time of purchase?

Motherboard 主板



- 北桥 Northbridge: handle high-performance tasks, connected directly to a CPU.
- 南桥 Southbridge: handle low speed tasks, connected via the northbridge

Motherboard 主板



Serveral Concepts

- Hardware: physical components of a computer system
  硬件
- Software: programs that run on the hardware
  软件
  - *e.g.*, operating systems and applications
- Firmware: specialized software embedded into hardware devices to
  固件
  control or manage the hardware directly
  - *e.g.*, BIOS (basic input/output system) for interrupt handling

3. In-class Practice

In-class Practice 1

- What is the CPU info and RAM size of your PC?
  - Show the screenshot and highlight the corresponding region

Windows: Win + R: msinfo32.exe

MacOS

In-class Practice 2

- The following code computes the largest Fibonacci number less than 100 and saves the result to the register x0 (aarch64) or eax (x86).
- Modify the X86 version to compute the largest accumulated value less than 100, starting from 1.

```
_main:
    mov x0, #1
    mov x2, #1

_loop:
    mov x1, x0
    mov x0, x2
    add x2, x2, x1
    cmp x2, #100
    ble _loop
    ret
```

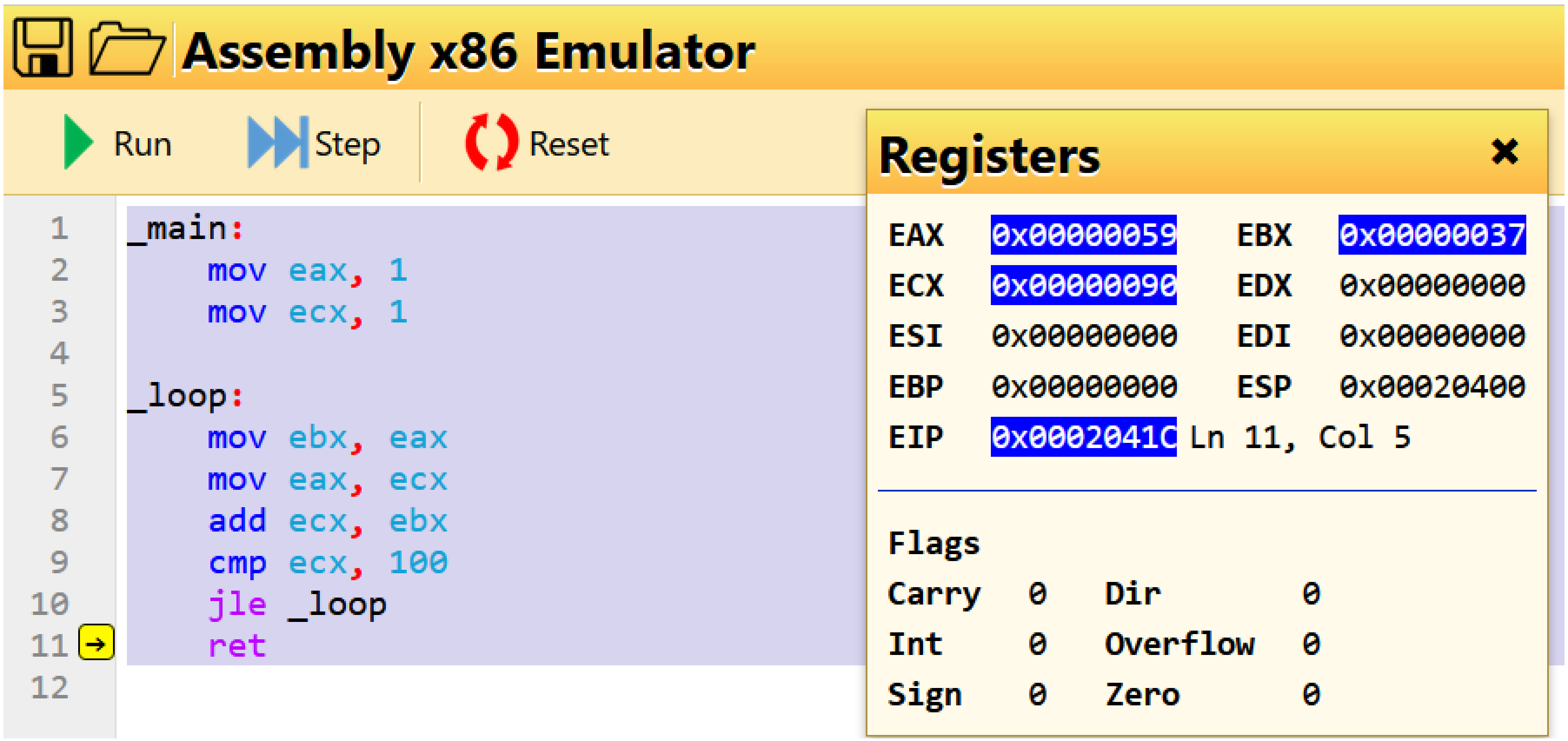
Aarch64 Version

```
_main:
    mov eax, 1
    mov ecx, 1

_loop:
    mov ebx, eax
    mov eax, ecx
    add ecx, ebx
    cmp ecx, 100
    jle _loop
    ret
```

ecx = ecx + ebx

X86 Version

In-class Practice 2



https://carlosrafaelgn.com.br/Asm86/