# Chapter 1
Introduction to ARM Processor and ARM Assembly

Zonghua Gu

Fall 2025

1

Why ARM processor

▸ ARM: **A**cron **R**ISC **M**achine, founded in 1990

▸ Public company, Headquarter at Cambridge, England, UK, 2023 Revenue: US$2.68 billion

▸ Arm processors are used as the main CPU for most mobile phones and handhelds.

▸ The world's second fastest supercomputer (previously fastest) in 2022, the Japanese Fugaku is based on Arm AArch64 architecture
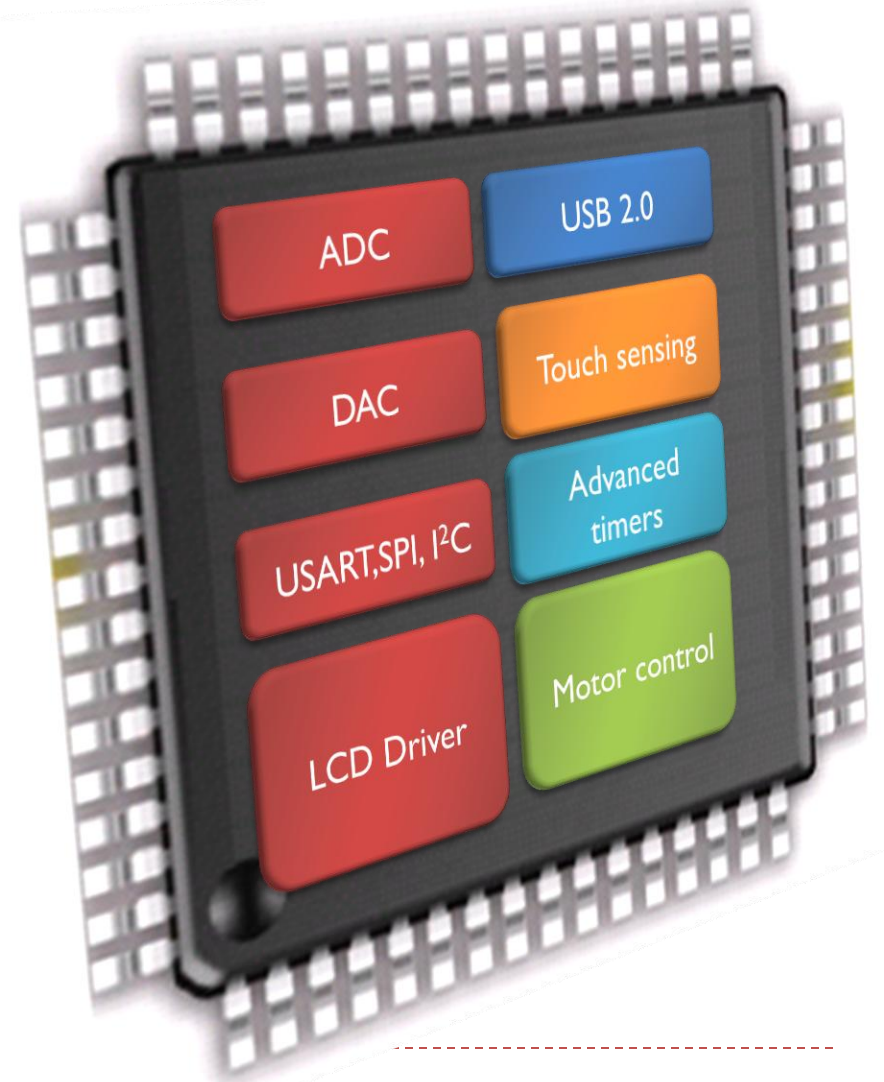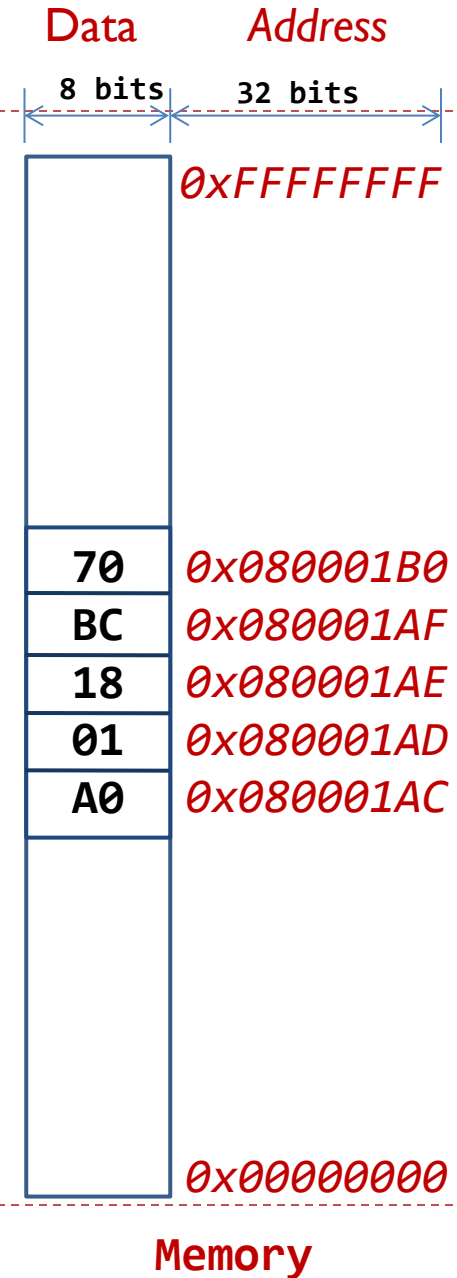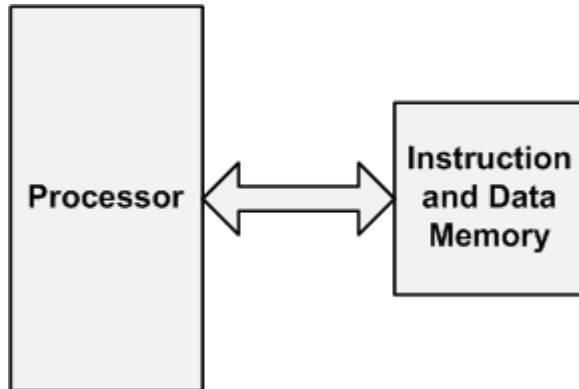
Embedded Systems

Memory

‣ Memory is arranged as a series of "locations"
  ‣ Each location has a unique "address"
  ‣ Each location holds a byte (**byte-addressable**)
  ‣ e.g. the memory location at address 0x080001B0 contains the byte value 0x70, i.e., 112

‣ The number of locations in memory is limited
  ‣ e.g. 4 GB of RAM
  ‣ 1 Gigabyte (GB) = 2^{30} bytes
  ‣ 2^{32} locations ➜ 4,294,967,296 locations!

‣ Values stored at each location can represent either program data or program instructions
  ‣ e.g. the value 0x70 might be the code used to tell the processor to add two values together

Data	Address
	0xFFFFFFFF
70	*0x080001B0*
BC	*0x080001AF*
18	*0x080001AE*
01	*0x080001AD*
A0	*0x080001AC*
	0x00000000

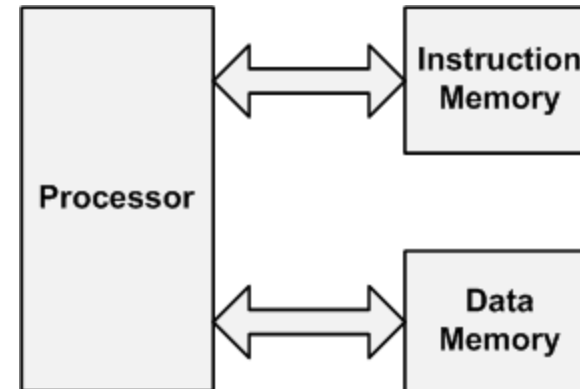**Memory**

Computer Architecture

Von-Neumann

Instructions and data are stored in the same memory.

Harvard

Data and instructions are stored into separate memories.

Computer Architecture

Von-Neumann

Instructions and data are stored in the same memory.
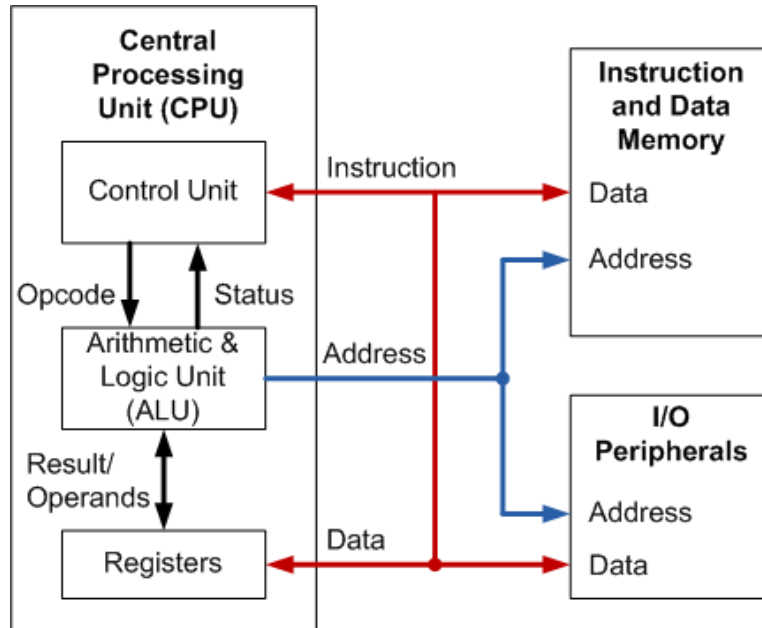
Harvard

Data and instructions are stored into separate memories.

ARM Cortex-M Series Family

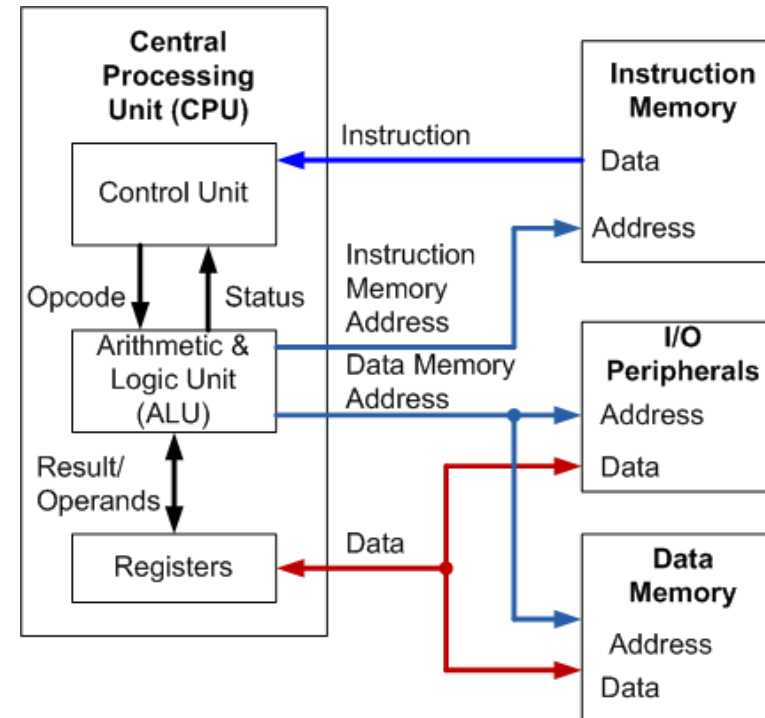## Von-Neumann

Instructions and data are stored in the same memory.

ARM
Cortex-M0

ARMv6-M

ARM
Cortex-M0+

ARMv6-M

ARM
Cortex-M1

ARMv6-M

ARM
Cortex-M23

ARMv8-M

Harvard

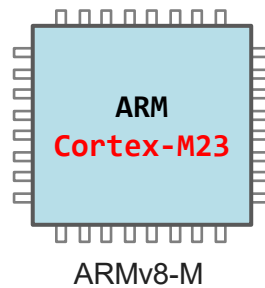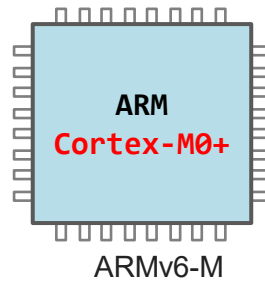**Data and instructions are stored into separate memories.**

ARM
Cortex-M3

ARMv7-M

ARM
Cortex-M4

ARMv7E-M

ARM
Cortex-M7

ARMv7E-M

ARM
Cortex-M33

ARMv8-M
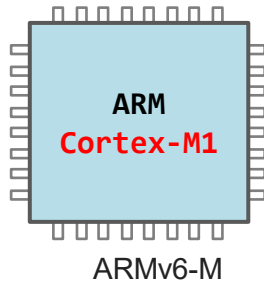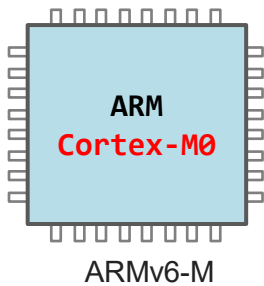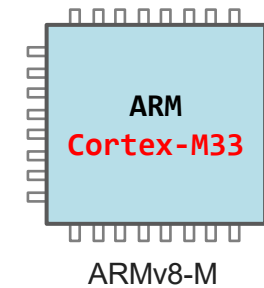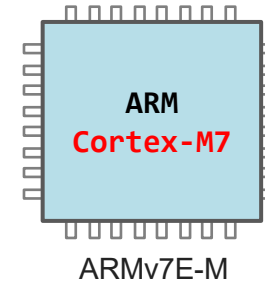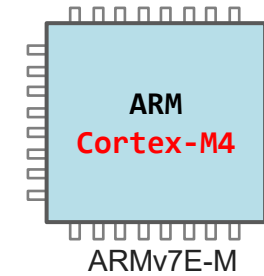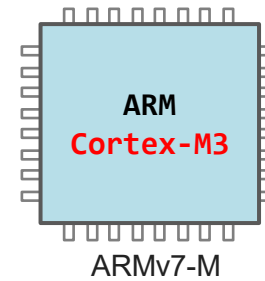
Levels of Program Code

C Program

```
int main(void){
  int i;
  int total = 0;
  for (i = 0; i < 10; i++) {
    total += i;
  }
  while(1); // Dead loop
}
```

Compile →

Assembly Program

```
        MOVS  r1, #0
        MOVS  r0, #0
        B     check
loop    ADD   r1, r1, r0
        ADDS  r0, r0, #1
check   CMP   r0, #10
        BLT   loop
self    B     self
```

Assemble →

Machine Program

```
0010000100000000
0010000000000000
1110000000000001
0100010000000001
0001110001000000
0010100000001010
1101110011111011
1011111100000000
1110011111111110
```

▸ **High-level language**

- ▸ Level of abstraction closer to problem domain
- ▸ Provides for productivity and portability

▸ **Assembly language**

- ▸ Textual representation of instructions
- ▸ Human-readable format instructions

▸ **Hardware representation**

- ▸ Binary digits (bits)
- ▸ Encoded instructions and data
- ▸ Computer-readable format instructions

See a Program Runs

C Code

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

compiler →

Assembly Code

```
MOVS r1, #0x00        ; int a = 0
MOVS r2, #0x01 ; int b = 1
ADDS r3, r1, r2       ; c = a + b
MOVS r0, 0x00  ; set return value
BX lr            ; return
```

assembler ↘

Machine Code

```
0010000100000000
0010001000000001
0001100010001011
0010000000000000
0100011101110000
```

```
2100
2201
188B
2000
4770
```

```
; MOVS r1, #0x00
; MOVS r2, #0x01
; ADDS r3, r1, r2
; MOVS r0, #0x00
; BX lr
```

In Binary   In Hex

Processor Registers

32 bits

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12

Low Registers (R0-R7)

High Registers (R8-R12)

General Purpose Register

R13 (SP)
R14 (LR)
R15 (PC)

R13 (MSP)

R13 (PSP)

32 bits

xPSR
BASEPRI
PRIMASK
FAULTMASK
CONTROL

Special Purpose Register

- ▸ Fastest way to read and write
- ▸ Registers are within the processor chip
- ▸ A register stores 32-bit value
- ▸ ARM Cortex-M has
  - ▸ **R0-R12**: 13 general-purpose registers
  - ▸ **R13**: Stack pointer (Shadow of MSP or PSP)
  - ▸ **R14**: Link register (LR)
  - ▸ **R15**: Program counter (PC)
  - ▸ Special registers (xPSR, BASEPRI, PRIMASK, etc)

Program Execution

▸ **Program Counter** (**PC**) is a register that holds the memory address of the next instruction to be fetched from the memory.
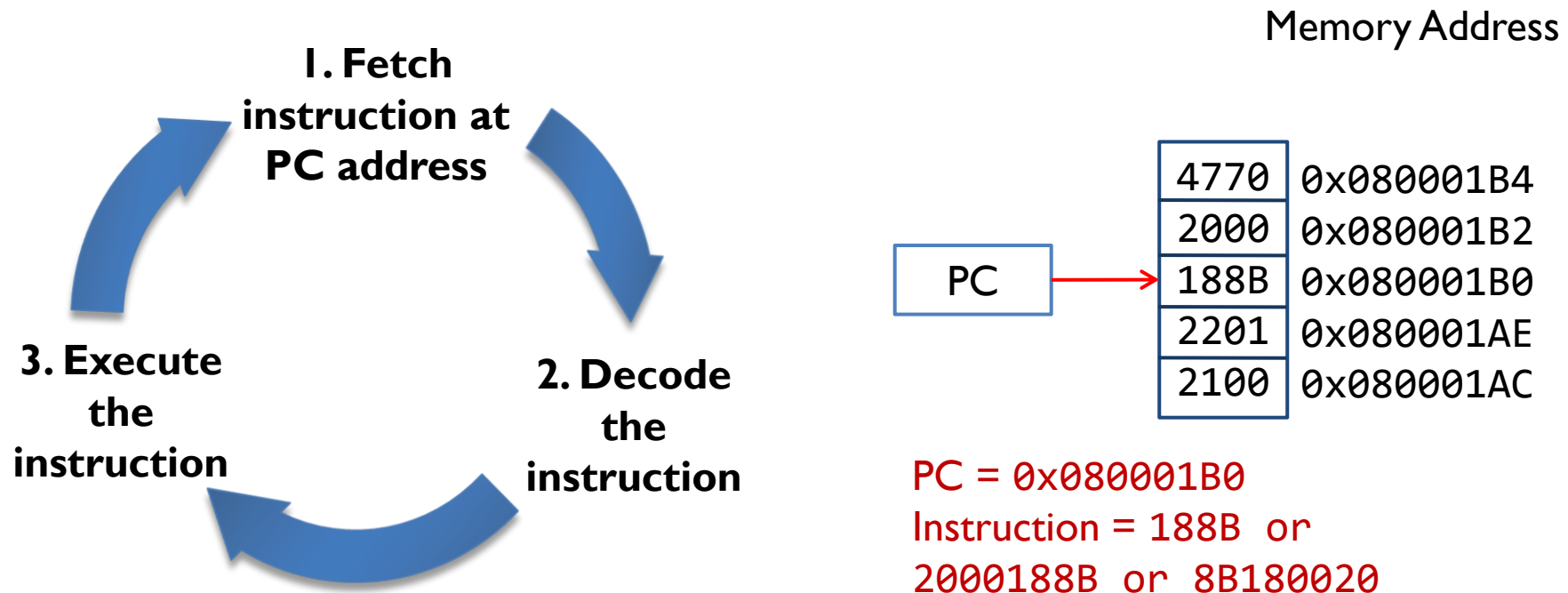
Memory Address

4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC

PC →

1. **Fetch instruction at PC address**
2. **Decode the instruction**
3. **Execute the instruction**

PC = 0x080001B0
Instruction = 188B or
2000188B or 8B180020

Three-state pipeline: Fetch, Decode, Execution

▸ **Pipelining** allows hardware resources to be fully utilized
▸ One 32-bit instruction or two 16-bit instructions can be fetched.
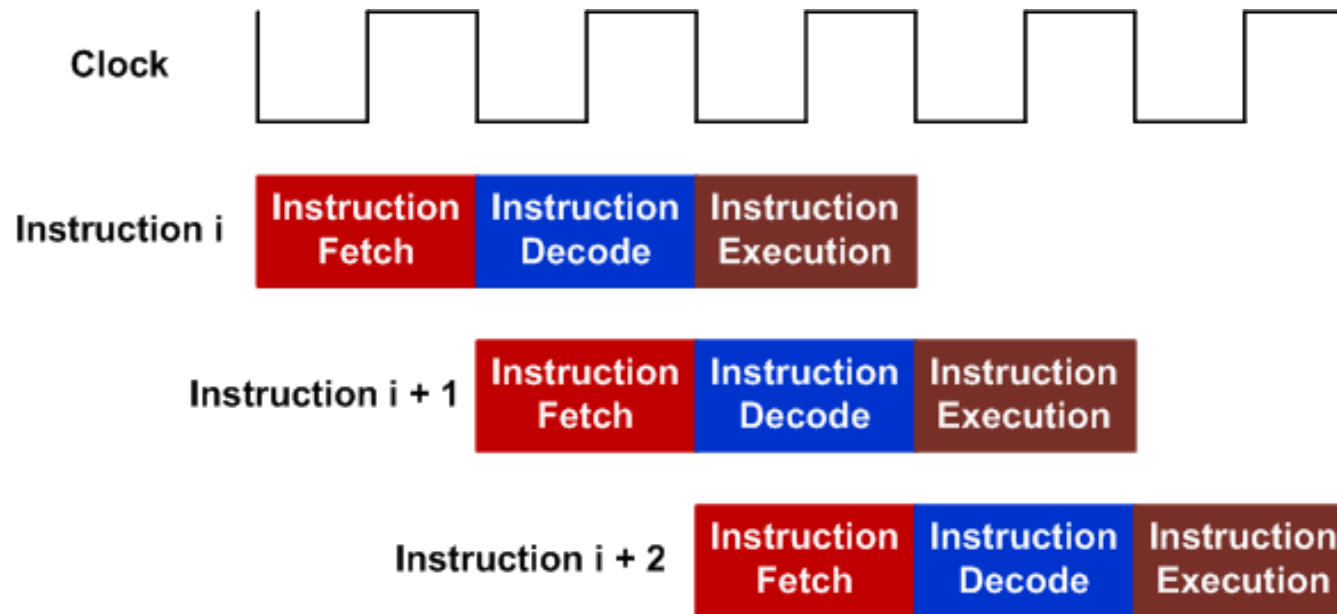
Pipeline of 32-bit instructions

Three-state pipeline: Fetch, Decode, Execution

▸ **Pipelining** allows hardware resources to be fully utilized
▸ One 32-bit instruction or two 16-bit instructions can be fetched.
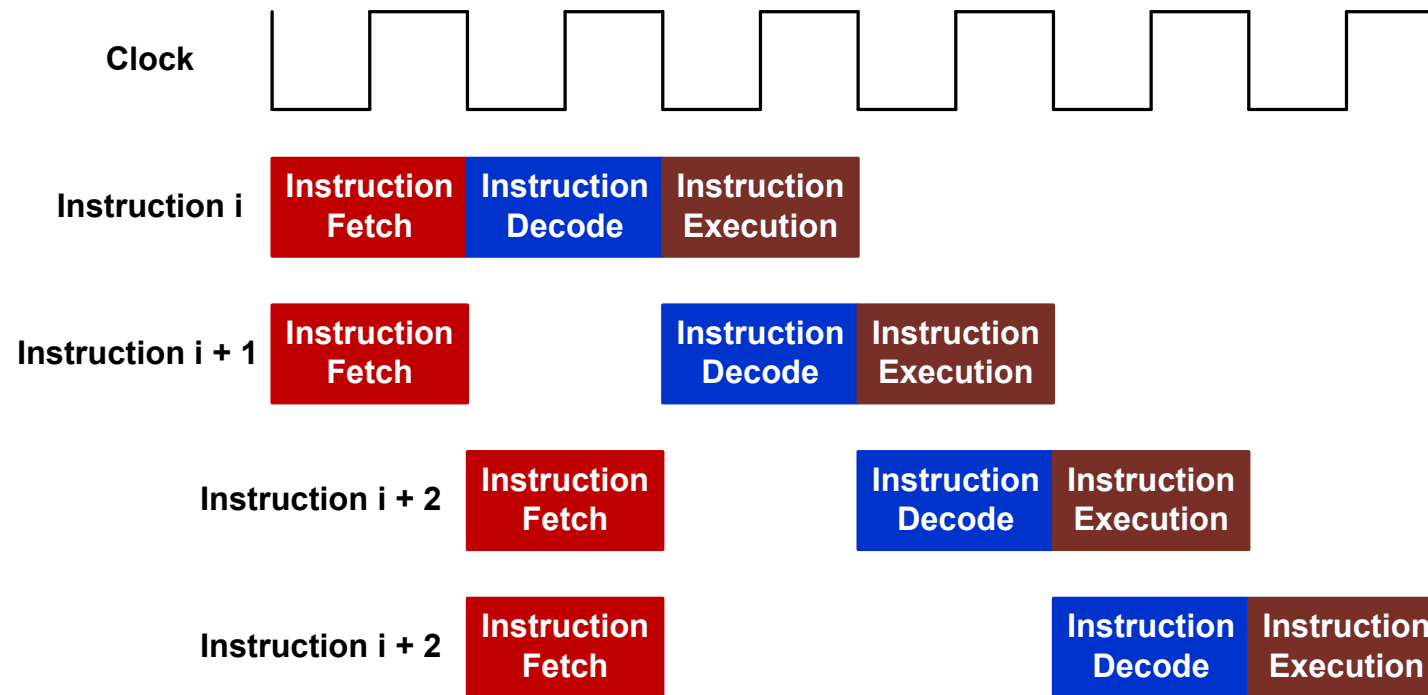
Pipeline of 16-bit instructions

Machine codes are stored in memory

Registers

CPU

r15		pc
r14		lr
r13		sp
r12		
r11		
r10		
r9		
r8		
r7		
r6		
r5		
r4		
r3		
r2		
r1		
r0		

ALU

Data	Address
	0xFFFFFFFF
4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC
	0x00000000

Memory
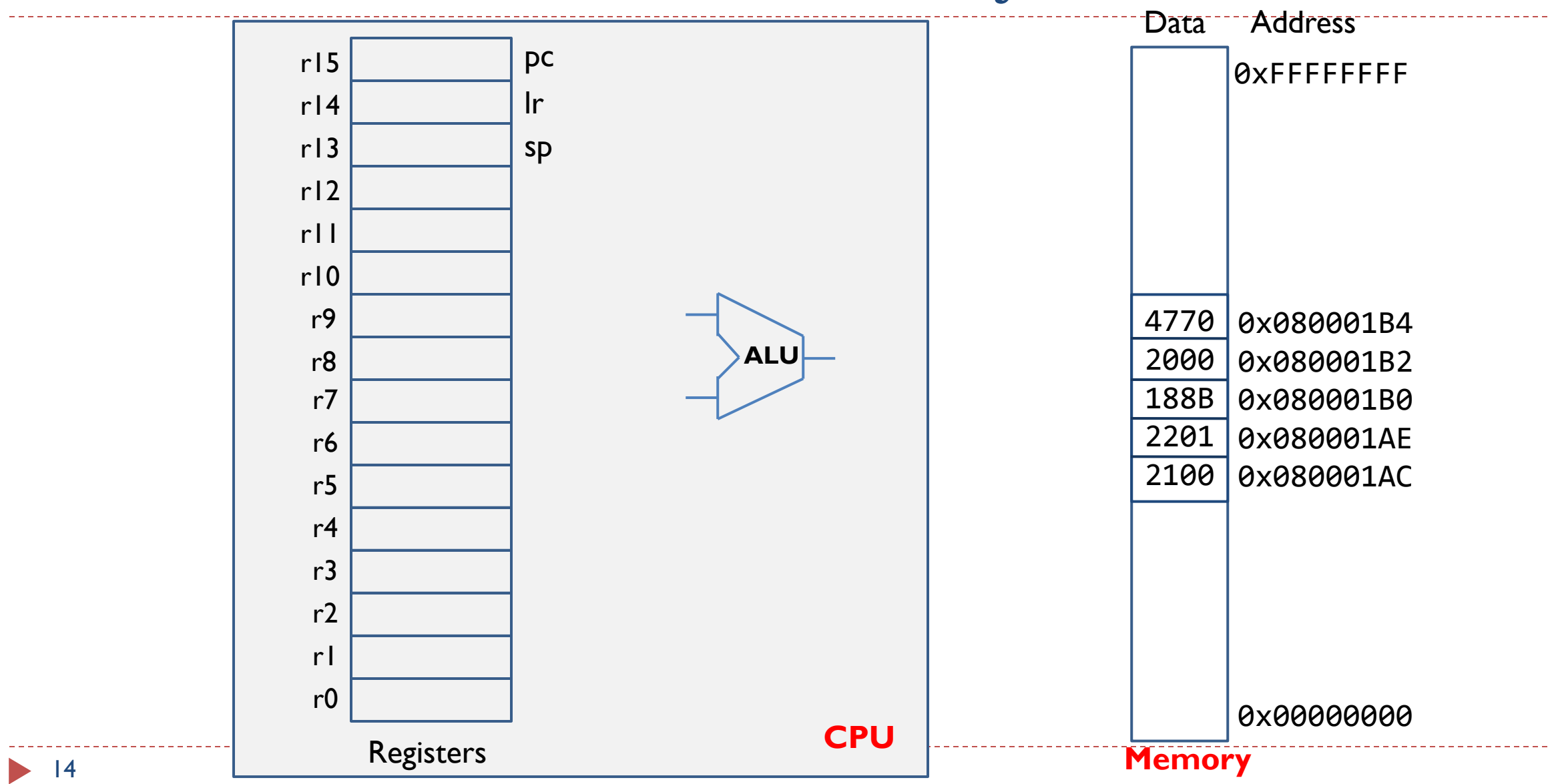
Fetch Instruction: pc = 0x08001AC
Decode Instruction: 2100 = MOVS r1, #0x00
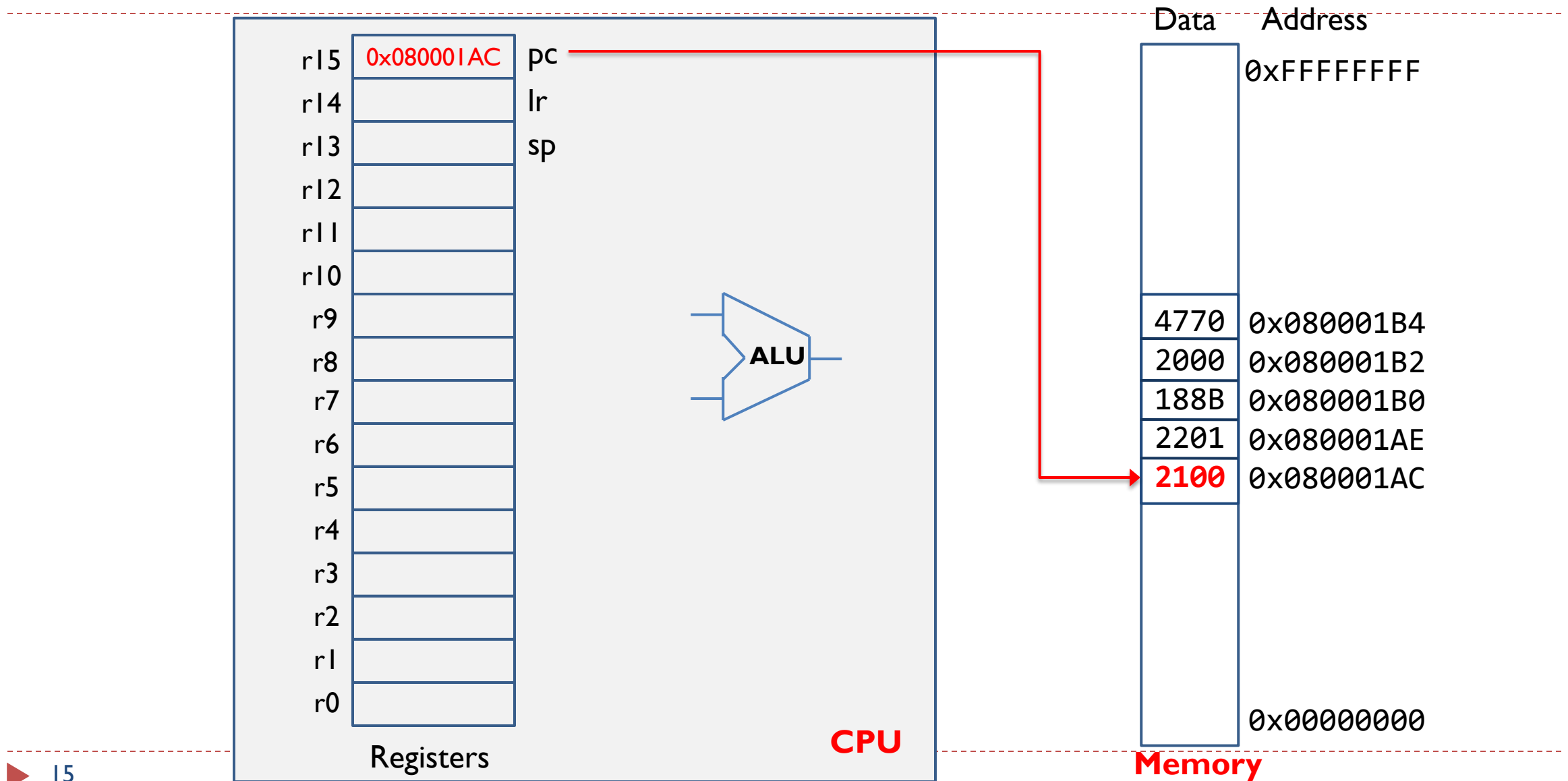
Execute Instruction:
MOVS r1, #0x00
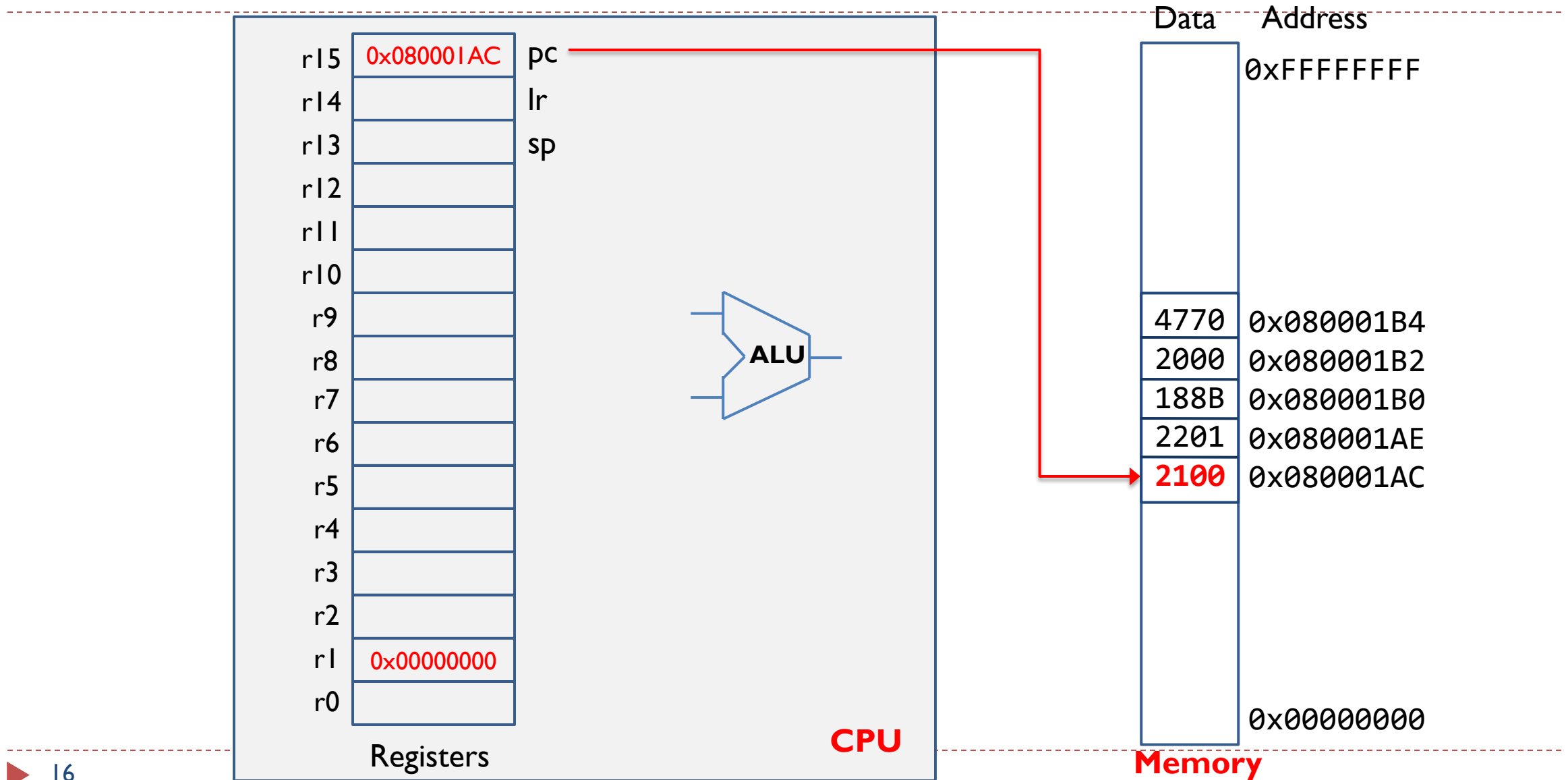
Fetch Next Instruction: **pc = pc + 2**

- Thumb-2 consists of a mix of 16- & 32-bit instructions
- In reality, we always fetch 4 bytes from the instruction memory (either one 32-bit instruction or two 16-bit instructions)
- To simplify the demo, we assume we only fetch 2 bytes from the instruction memory in this example.

Data   Address

	0xFFFFFFFF

Data	Address
4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC

0x00000000

Memory

CPU

Registers

Register	Value	
r15	0x080001AE	pc
r14		lr
r13		sp
r12		
r11		
r10		
r9		
r8		
r7		
r6		
r5		
r4		
r3		
r2		
r1	0x00000000	
r0		

ALU
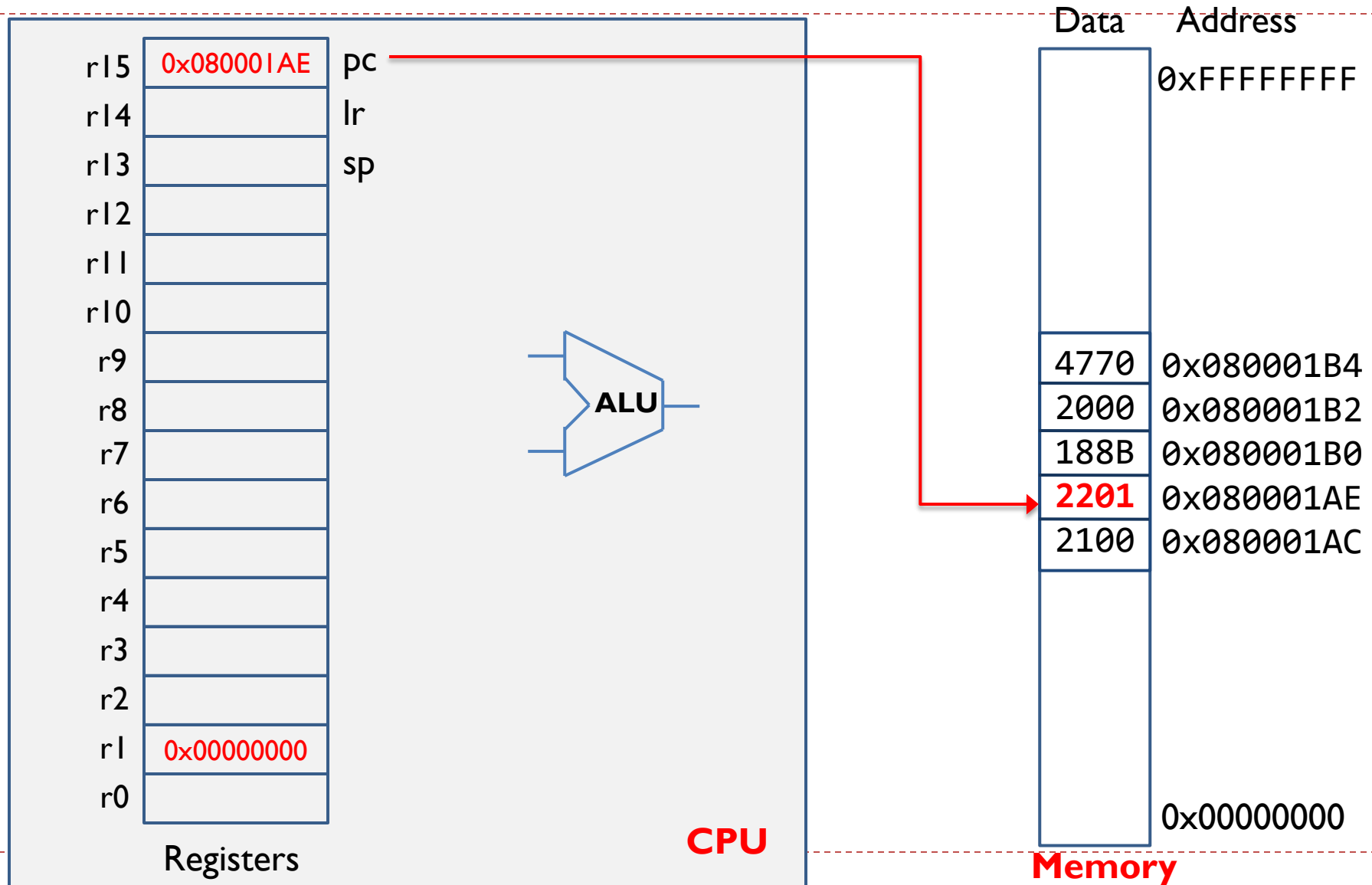
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2201 = MOVS r2, #0x01
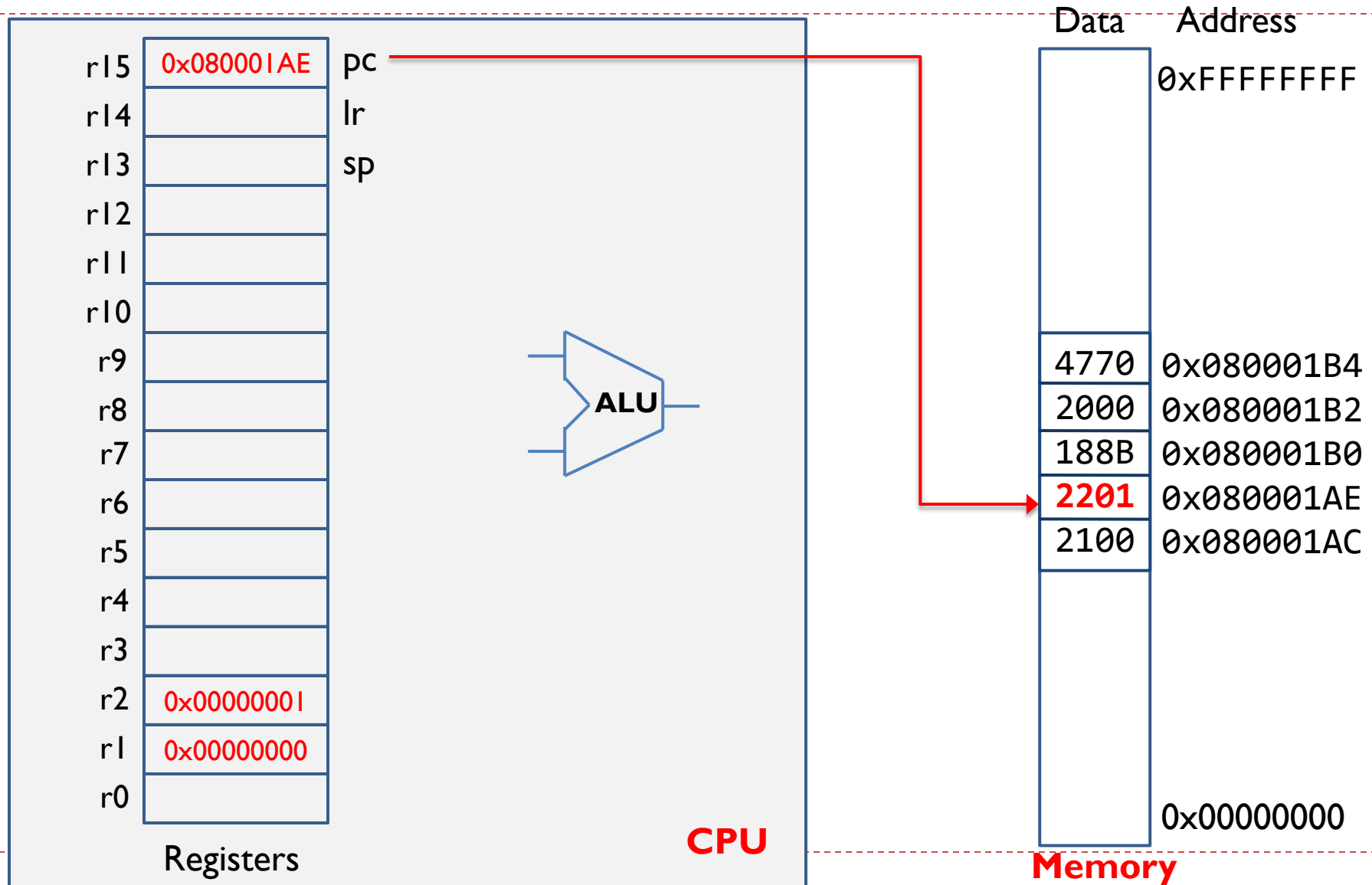
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 188B = ADDS r3, r1, r2
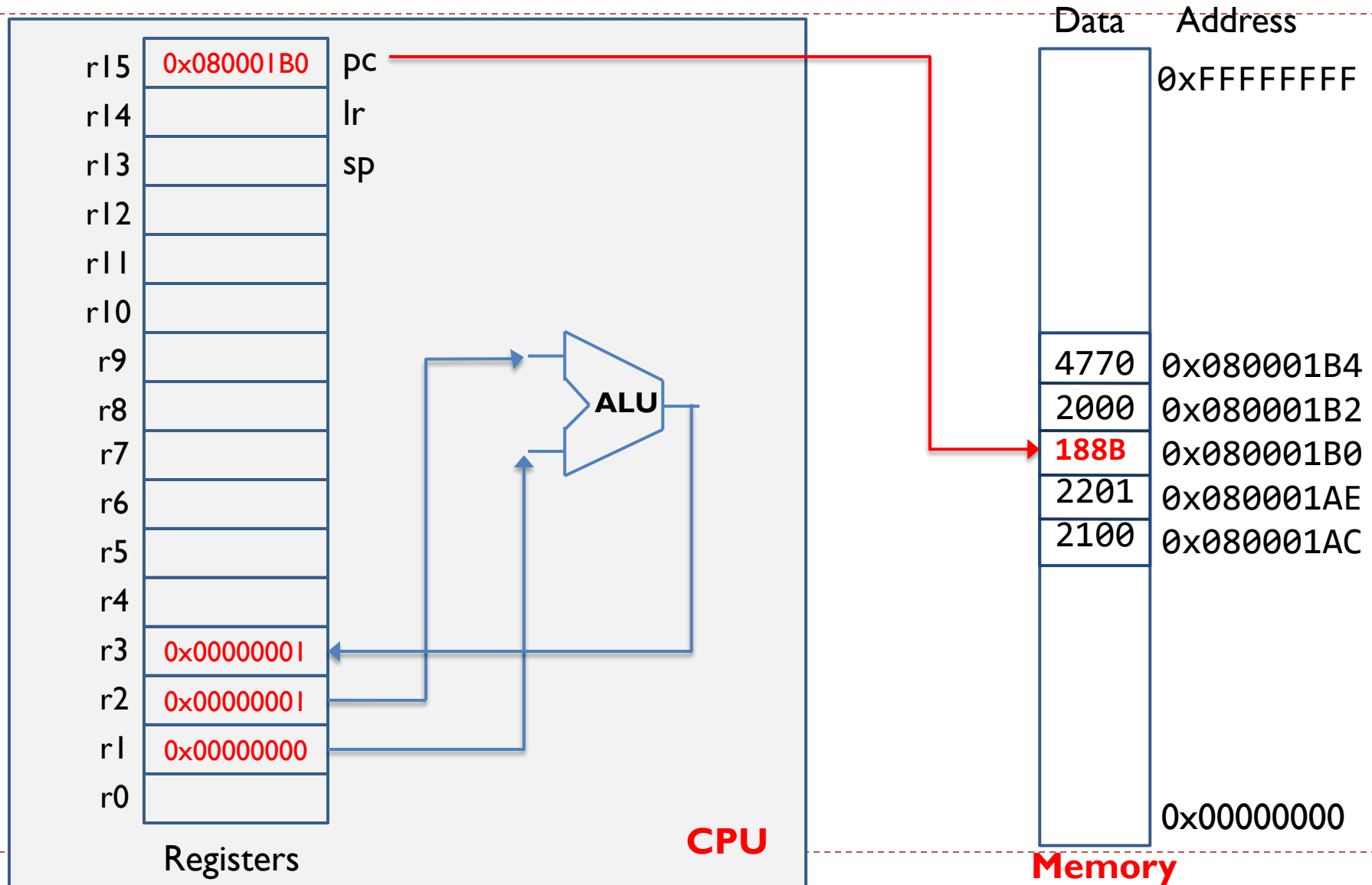
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2000 = MOVS r0, #0x00
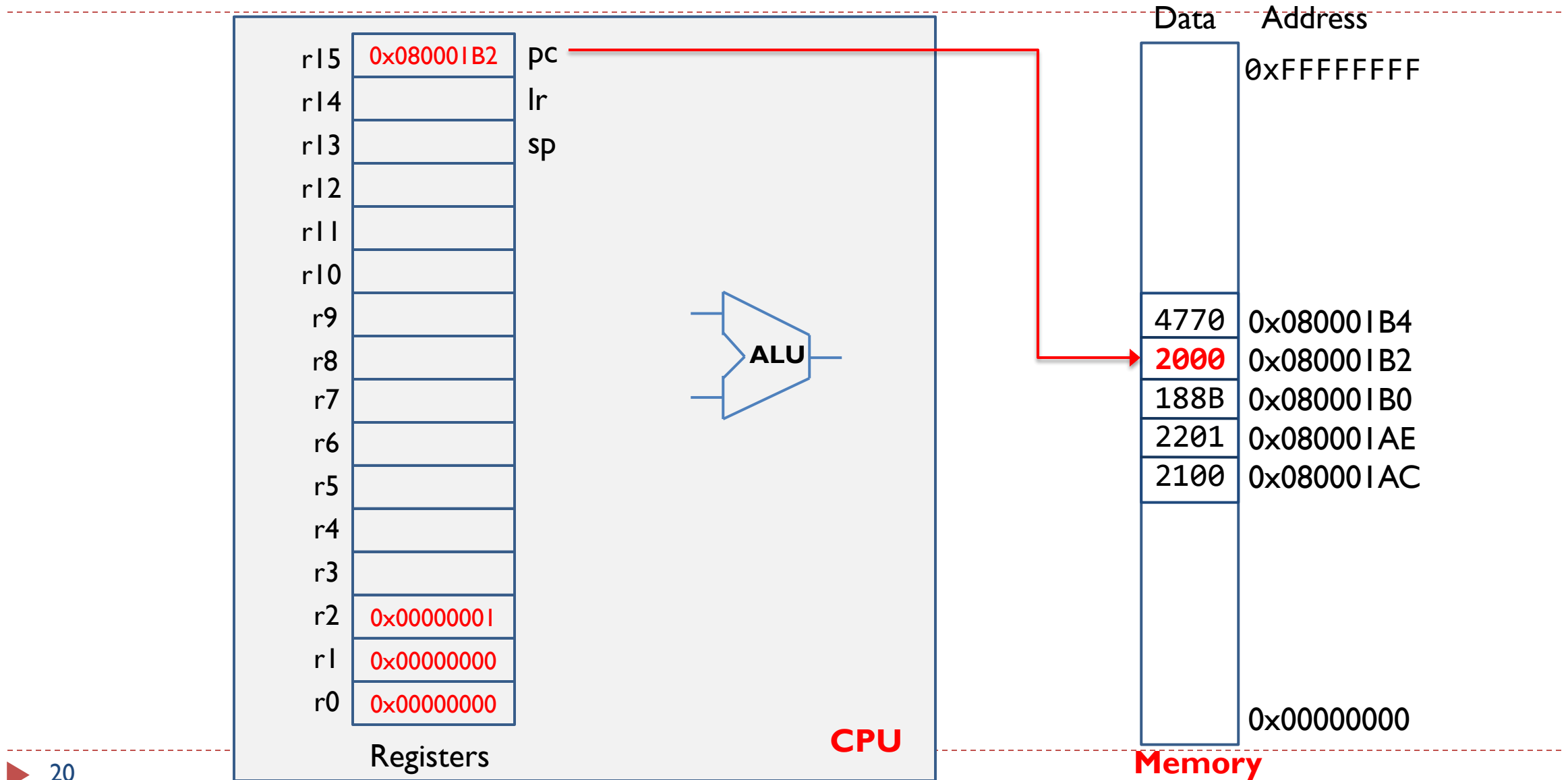
Fetch Next Instruction: pc = pc + 2
Decode & Decode: 4770 = BX lr
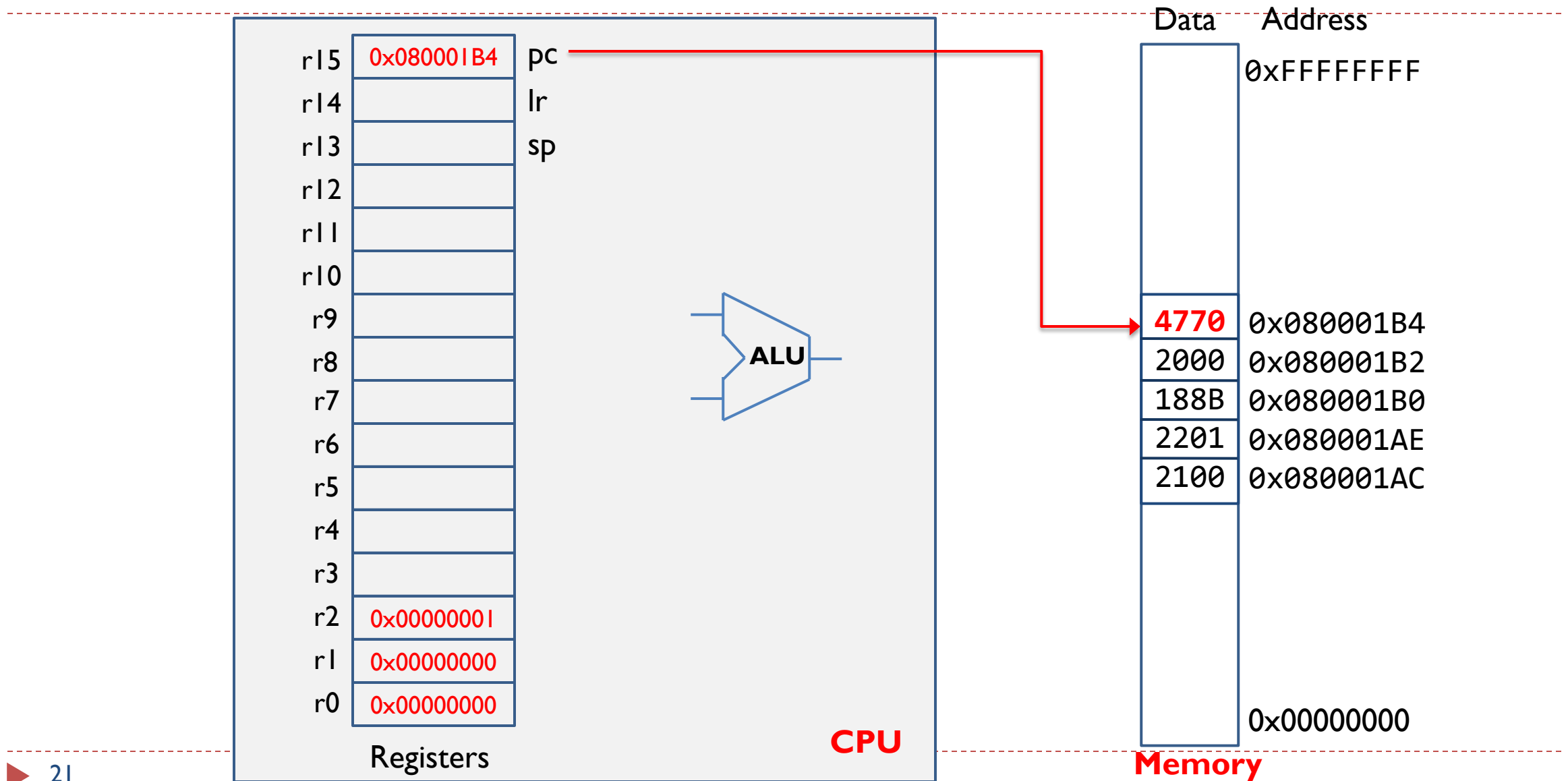
Realities

- In the previous example,
  - PC is incremented by 2

Well, I lied!

Realities

▸ PC is always incremented by **4**.

  ▸ Each time, 4 bytes are fetched from the instruction memory

  ▸ It is either two 16-bit instructions or one 32-bit instruction
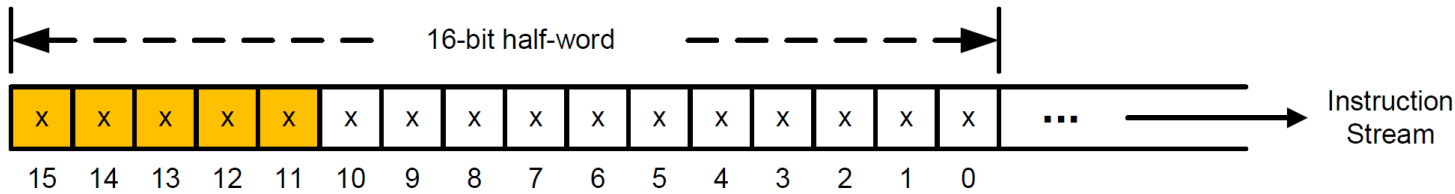
If bit [15-11] = 11101, 11110, or 11111, then, it is the first half-word of a 32-bit instruction.

Otherwise, it is a 16-bit instruction.

Example:
Calculate the Sum of an Array

```c
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;

int main(void){
  int i;
  total = 0;
  for (i = 0; i < 10; i++) {
    total += a[i];
  }
  while(1);
}
```

Example:
Calculate the Sum of an Array

CPU

Instruction Memory (Flash)

```
int main(void){
    int i;
    total = 0;
    for (i = 0; i < 10; i++) {
        total += a[i];
    }
    while(1);
}
```

Starting memory address 0x08000000

Data Memory (RAM)

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;
```

Starting memory address 0x20000000

I/O Devices
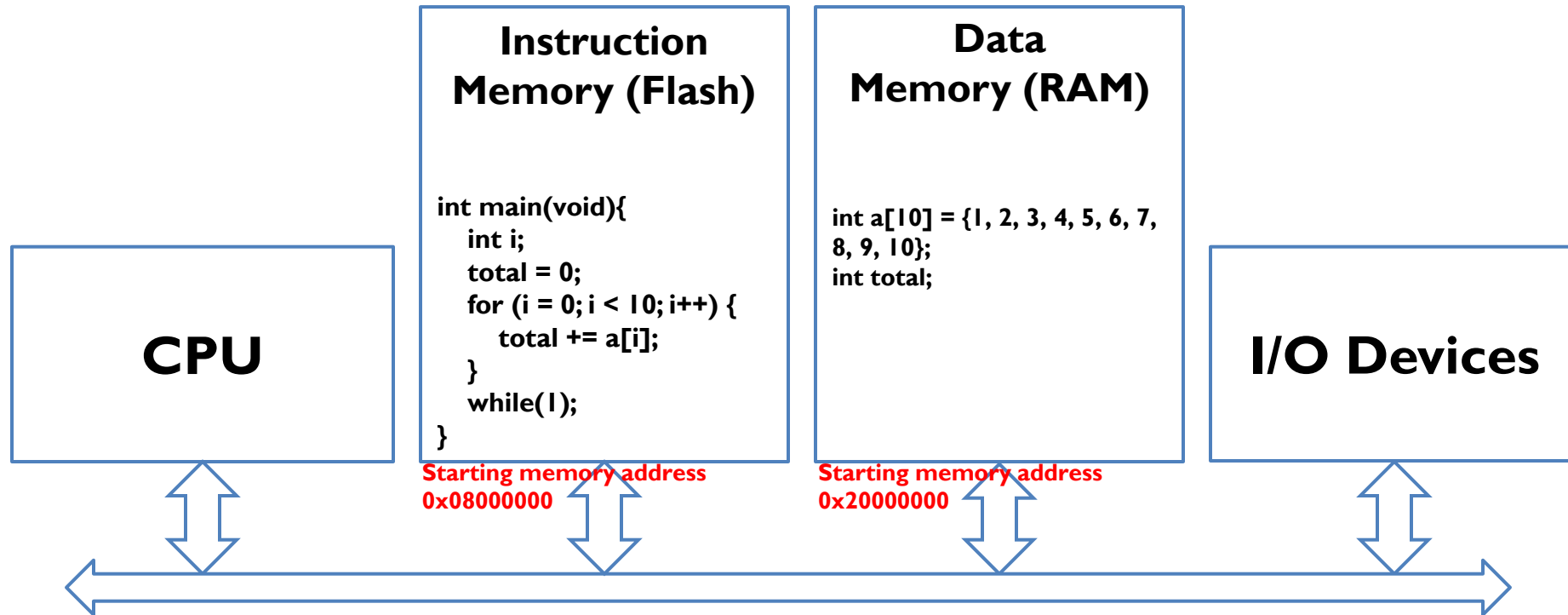
Example:
Calculate the Sum of an Array

Instruction Memory (Flash)

```
int main(void){
    int i;
    total = 0;
    for (i = 0; i < 10; i++) {
        total += a[i];
    }
    while(1);
}
```

Starting memory address 0x08000000

```
0010 0001 0000 0000
0100 1010 0000 1000
0110 0000 0001 0001
0010 0000 0000 0000
1110 0000 0000 1000
0100 1001 0000 0111
1111 1000 0101 0001
0001 0000 0010 0000

0100 1010 0000 0100
0110 1000 0001 0010
0100 0100 0001 0001
0100 1010 0000 0011
0110 0000 0001 0001
0001 1100 0100 0000
0010 1000 0000 1010
1101 1011 1111 0100
1011 1111 0000 0000
1110 0111 1111 1110
```

```
        MOVS   r1, #0x00
        LDR    r2, = total_addr
        STR    r1, [r2, #0x00]
        MOVS   r0, #0x00
        B      Check
Loop:   LDR    r1, = a_addr
        LDR    r1, [r1, r0, LSL #2]
        LDR    r2, = total_addr
        LDR    r2, [r2, #0x00]
        ADD    r1, r1, r2
        LDR    r2, = total_addr
        STR    r1, [r2, #0x00]
        ADDS   r0, r0, #1
Check:  CMP    r0, #0x0A
        BLT    Loop
        NOP
Self:   B      Self
```

Example:
Calculate the Sum of an Array

Data Memory (RAM)

int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;

Assume the starting memory address of the data memory is 0x20000000

Memory address in bytes	Memory content	
0x20000054	0x00000000	
0x20000050	0x00000000	
0x2000004C	0x00000000	
0x20000048	0x00000000	
0x20000044	0x00000000	
0x20000040	0x00000000	
0x2000003C	0x00000000	
0x20000038	0x00000000	
0x20000034	0x00000000	
0x20000030	0x00000000	
0x2000002C	0x00000000	
0x20000028	0x00000000	total= 0x00000000
0x20000024	0x0000000A	a[9] = 0x0000000A
0x20000020	0x00000009	a[8] = 0x00000009
0x2000001C	0x00000008	a[7] = 0x00000008
0x20000018	0x00000007	a[6] = 0x00000007
0x20000014	0x00000006	a[5] = 0x00000006
0x20000010	0x00000005	a[4] = 0x00000005
0x2000000C	0x00000004	a[3] = 0x00000004
0x20000008	0x00000003	a[2] = 0x00000003
0x20000004	0x00000002	a[1] = 0x00000002
0x20000000	0x00000001	a[0] = 0x00000001

Loading Code and Data into Memory


```
int counter;

int a[5] = {1, 2, 3, 4, 5};

int main(void){
    int i;

    int b[5];

    counter = 0;
    for (i = 0; i < 5; i++){
        b[i] = a[i];
        counter++
    }
    while(1);
}
```

C Program:
Copying an Array

0xFFFF,FFFF

Data Memory
(SRAM)

SRAM Start Address

Instruction
Memory
(Flash Memory)

Instruction Memory
Start Address

0x0000,0000

Loading Code and Data into Memory


```
int counter;

int a[5] = {1, 2, 3, 4, 5};

int main(void){
    int i;

    int b[5];

    counter = 0;
    for (i = 0; i < 5; i++){
        b[i] = a[i];
        counter++
    }
    while(1);
}
Dissection of a C Program:
Copying an Array
```

0xFFFF,FFFF

Data Memory
(SRAM)

SRAM Start Address

Instruction
Memory
(Flash Memory)

Instruction Memory
Start Address

0x0000,0000

Loading Code and Data into Memory

To improve performance, some variables are not stored in memory. Variable i will be stored in a register.

```c
int counter;

int a[5] = {1, 2, 3, 4, 5};

int main(void){
    int i;

    int b[5];

    counter = 0;
    for (i = 0; i < 5; i++){
        b[i] = a[i];
        counter++
    }
    while(1);
}
```

Dissection of a C Program:
Copying an Array

SRAM End Address

Main Stack *int b[5];* ← Stack Start
← Stack Pointer (SP)

Stack grows down

← Stack End
← Heap End

Heap grows up

← Heap Start

Zero-Initialized Data *counter = 0;*

Initialized Data *a[5] ={1,2,3,4,5};*

SRAM Start Address

Data Memory (SRAM)

SRAM Start Address

Instruction Memory (Flash Memory)

Instruction Memory Start Address

← Program Counter (PC)

0x0000,0000

- **Stack is mandatory**
- **Heap is used only if dynamic allocation (e.g. *malloc, calloc*) is used.**

View of a Binary Program

STM32L4

Parallel Interface

FSMC 8-/16-bit
(TFT-LCD, SRAM, NOR, NAND)

Display

LCD driver 8 x 40

Timers

17 timers including:
2 x 16-bit advanced motor control timers
2 x ULP timers
7 x 16-bit-timers
2 x 32-bit timers

I/Os

Up to 114 I/Os
Touch-sensing controller

Cortex-M4
80 MHz
FPU
MPU
ETM

DMA

ART Accelerator™

Up to 1-Mbyte Flash with ECC Dual Bank

128-Kbyte RAM

Connectivity

USB OTG,
1x SD/SDIO/MMC, 3 x SPI,
3 x I²C, 1x CAN, 1 x Quad SPI,
5 x USART + 1 x ULP

Digital

AES (256-bit), TRNG, 2 x SAI, DFSDM (8 channels)

Analog

3 x 16-bit ADC, 2 x DAC,
2 x comparators, 2 x op amps
1 x temperature sensor

from st.com

Memory Map

8 bits

Address	Region
0xFFFFFFFF	
0xE0100000	Cortex-M3 Internal Peripherals (64 KB)
0xE0000000	
0xC0000000	External Device
0xA0000000	
0x80000000	External RAM
0x60000000	
0x40026400	Peripheral (153KB)
0x40000000	
0x20000000	SRAM (Data Memory)
0x08000000	Flash Memory (Instruction Memory)
0x00000000	

512MB

1GB

1GB

512MB

512MB

512MB

128MB

SRAM end address

0x20000000

SRAM start address

Main Stack — Stack start

Stack pointer (SP)

Stack grows down.

Stack end
Heap end

Heap grows up.

Heap start

Data