

Chapter I

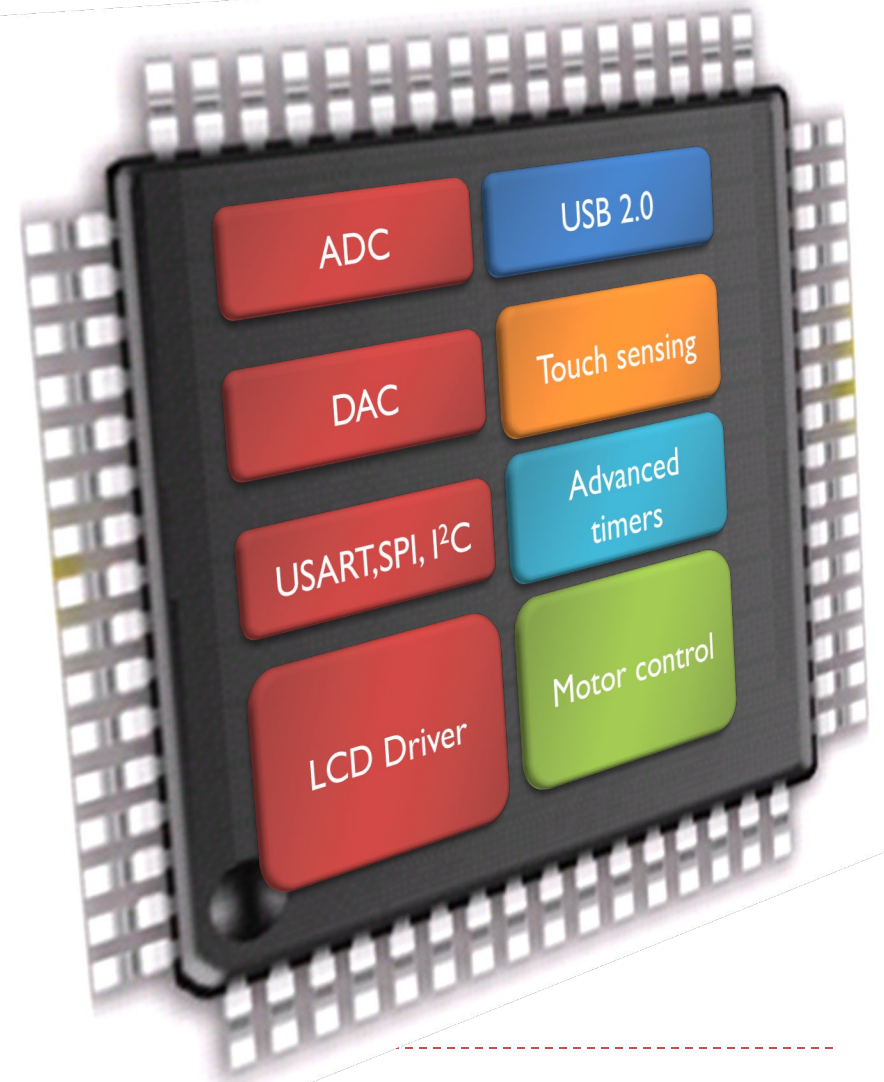
Computer and Assembly Language

Z. Gu

Fall 2026

Why ARM processor

- ▶ ARM: **A**corn **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 in 2022, the Japanese Fugaku is based on Arm AArch64 architecture

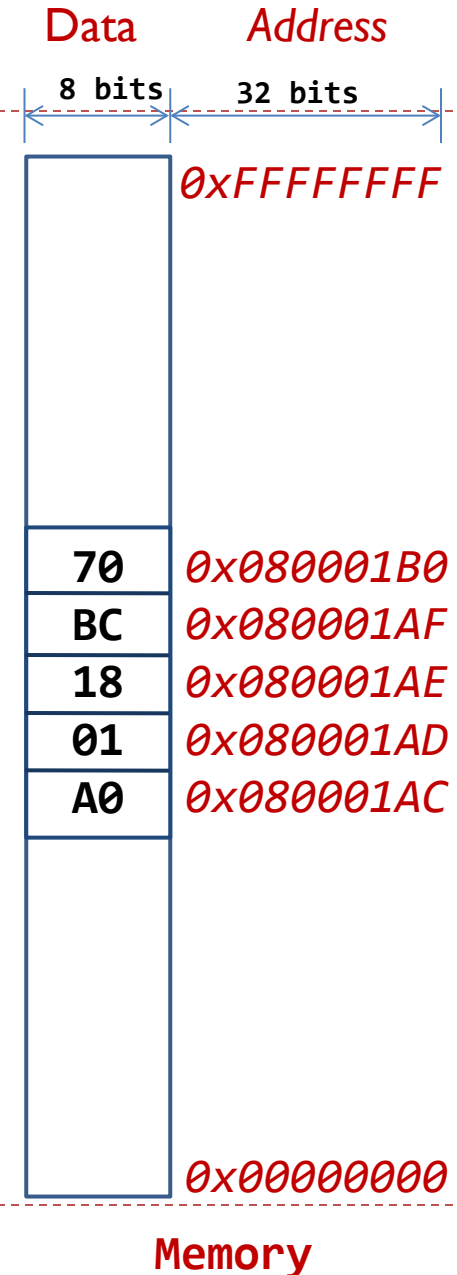


Embedded Systems



Memory

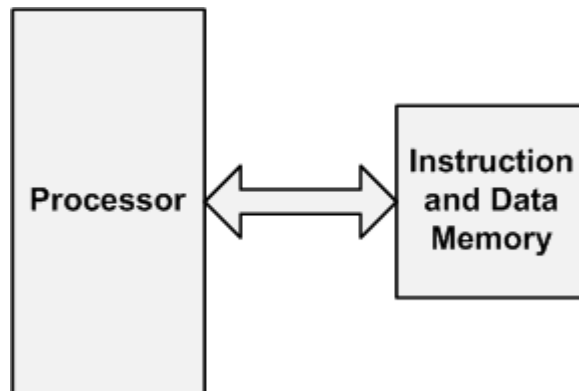
- ▶ Memory is arranged as a series of “locations”
 - ▶ Each location has a unique “address”
 - ▶ Each location holds a byte (**byte-addressable**)
 - ▶ e.g. the memory location at address `0x080001B0` contains the byte value `0x70`, i.e., 112
- ▶ The number of locations in memory is limited
 - ▶ e.g. $4\text{ GB} = 2^{32}\text{ bytes} \rightarrow 4,294,967,296\text{ locations}$
- ▶ 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



Computer Architecture

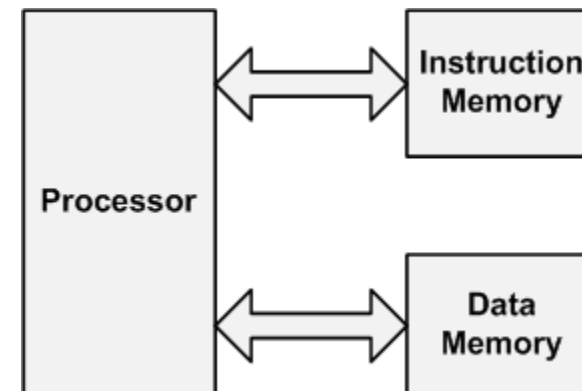
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

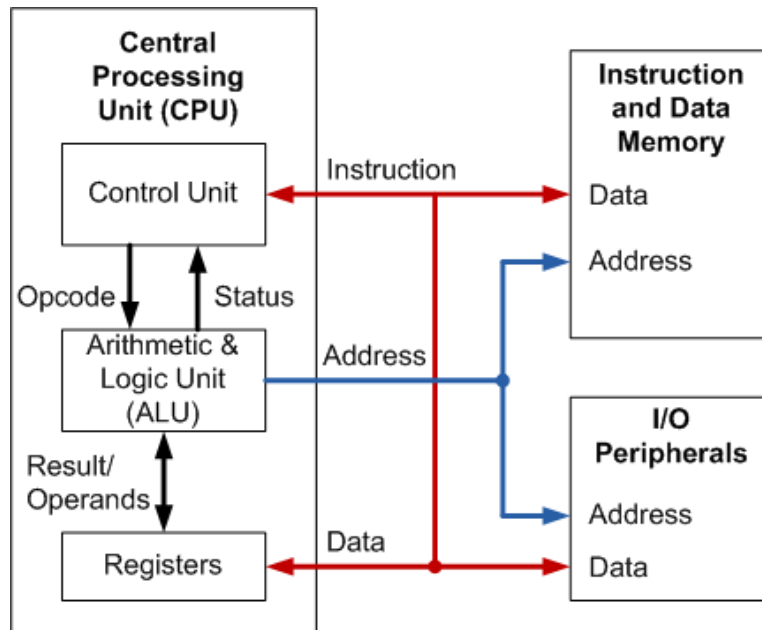
Data and instructions are stored into separate memories.



Computer Architecture

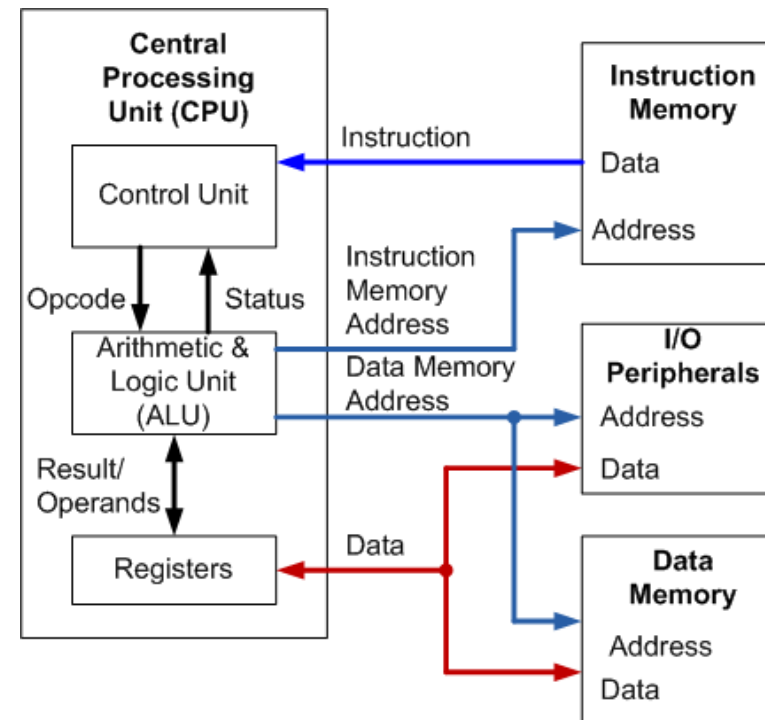
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

Data and instructions are stored into separate memories.



von Neumann vs. Harvard

- ▶ **Von Neumann Architecture:** One memory, one address space, one main bus
 - ▶ Instructions and data are stored in the same memory
 - ▶ CPU fetches instructions and accesses data over the same bus
 - ▶ Simple and flexible, but instruction fetch and data access cannot happen at the same time
 - ▶ Advantages: more efficient memory space utilization, with more flexible placement of instructions and data in memory
 - ▶ Mental model: Code and data live together, competing for attention.
- ▶ **Harvard Architecture:** Separate memories, separate buses
 - ▶ Instructions stored in instruction memory; Data stored in data memory
 - ▶ CPU can fetch an instruction and access data simultaneously
 - ▶ Advantages: Higher and more predictable performance
 - ▶ Common in DSPs (Digital Signal Processors)
 - ▶ Mental model: Code and data live in different houses, with separate roads.
- ▶ **ARM Cortex-M** uses a **modified Harvard architecture**: a unified address space with separate instruction and data paths.
 - ▶ Unified address space (programmer sees one memory map)
 - ▶ Physically separate instruction and data buses inside the CPU
 - ▶ Instruction memory (Flash) and data memory (SRAM) are distinct; allows instruction fetch and data access in parallel
 - ▶ Why “modified”? Instructions and data have different physical paths, but they share a single, consistent address space

ARM refers to an instruction set architecture (ISA);
Cortex-M is a microarchitecture implementing it.

von Neumann vs. Harvard: Summary

- ▶ **Von Neumann:** Instructions and data share the *same memory and bus* → no parallel access
- ▶ **Harvard:** Instructions and data use *separate memories and buses* → parallel access
- ▶ **ARM Cortex-M: Modified Harvard:** *ARM Cortex-M uses a modified Harvard architecture: a unified address space with separate instruction and data paths.*

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

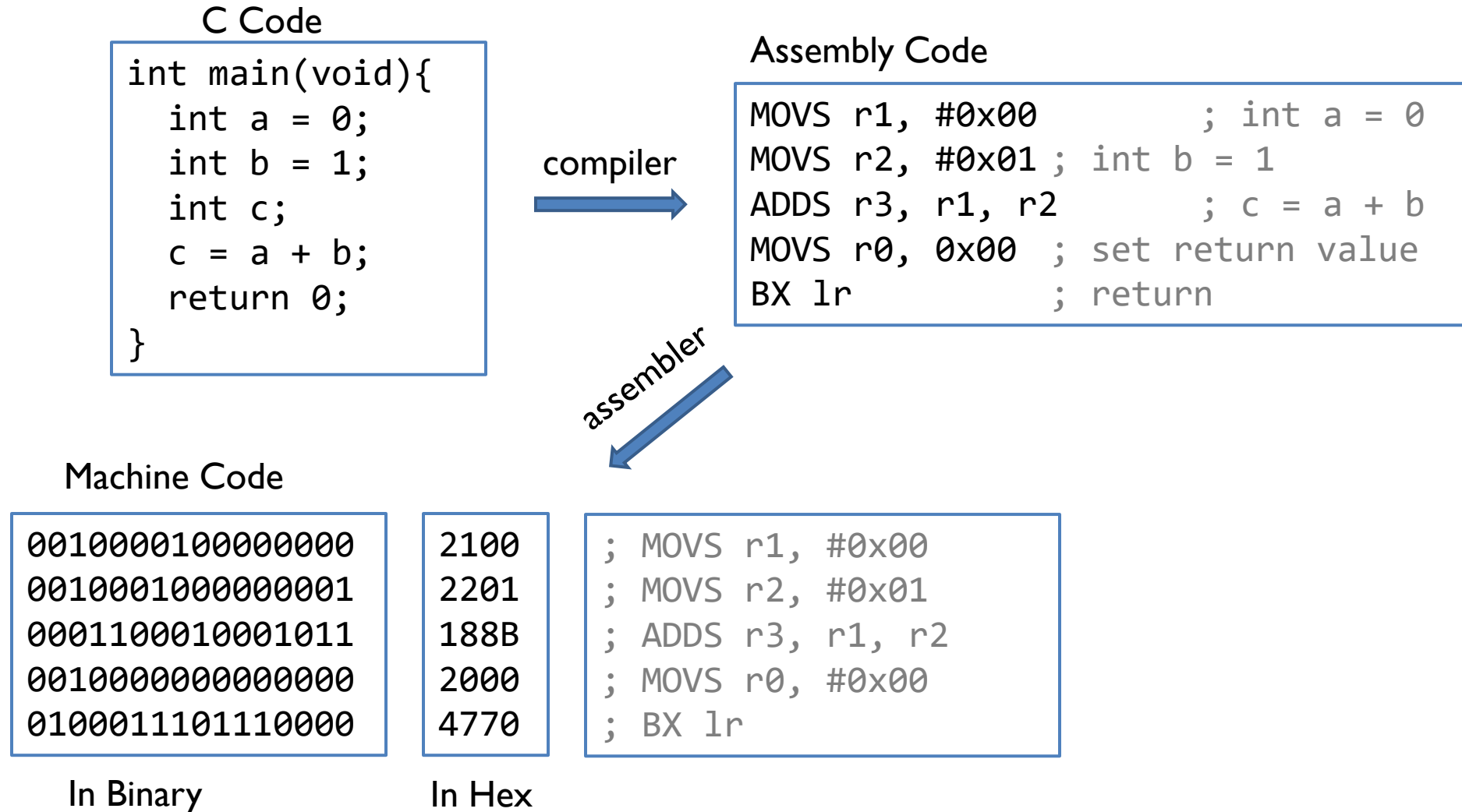
▶ Assembly language

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

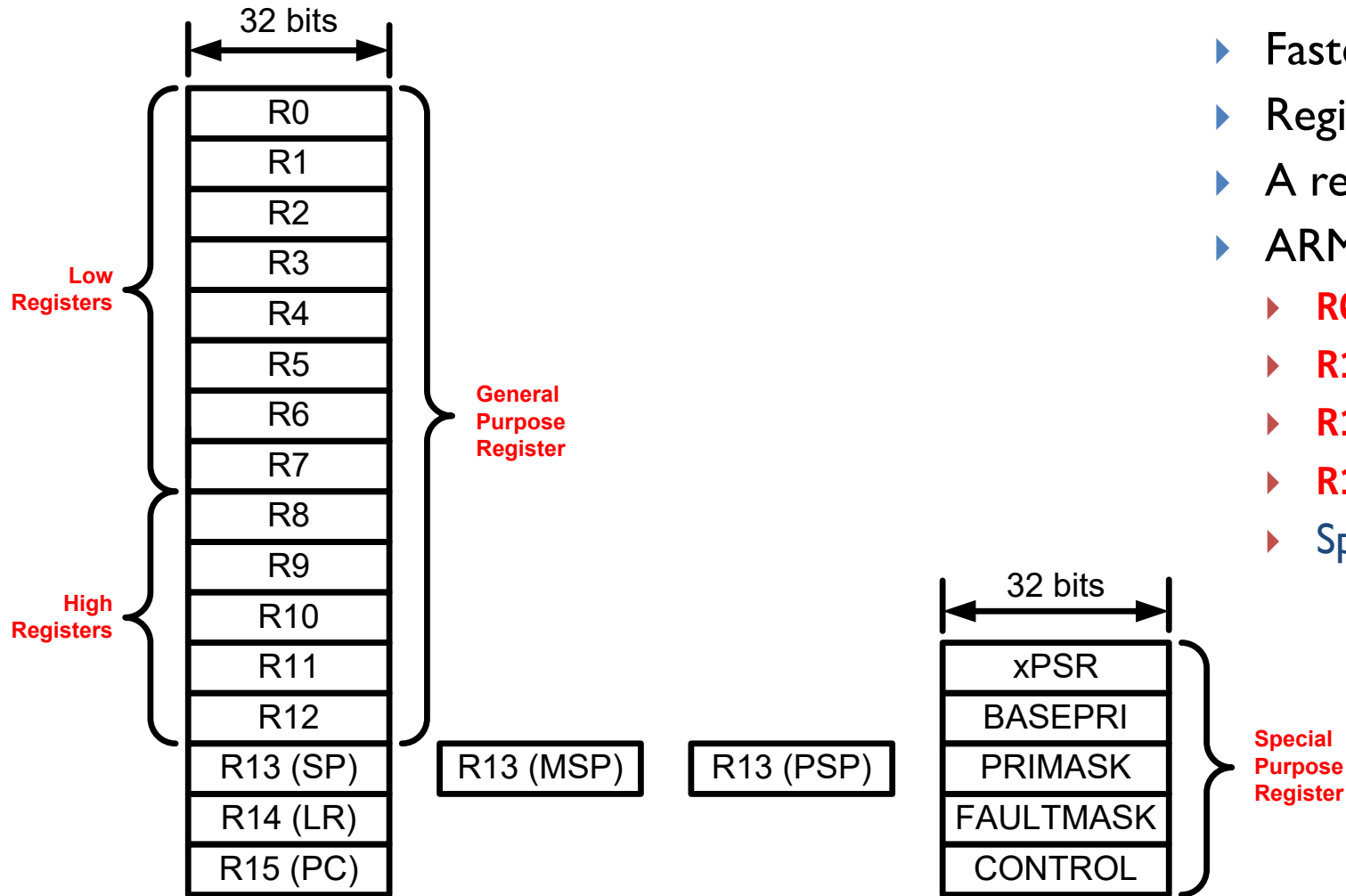
▶ Hardware representation

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

See a Program Runs



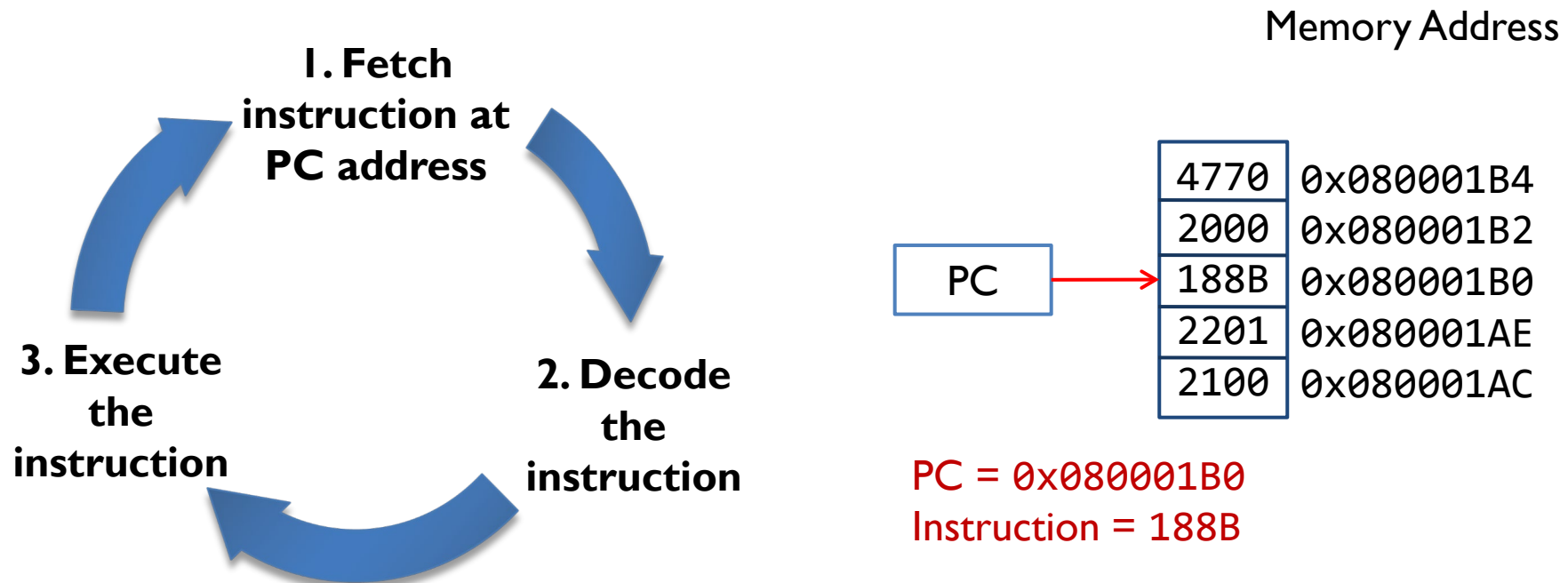
Processor Registers



- ▶ 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)

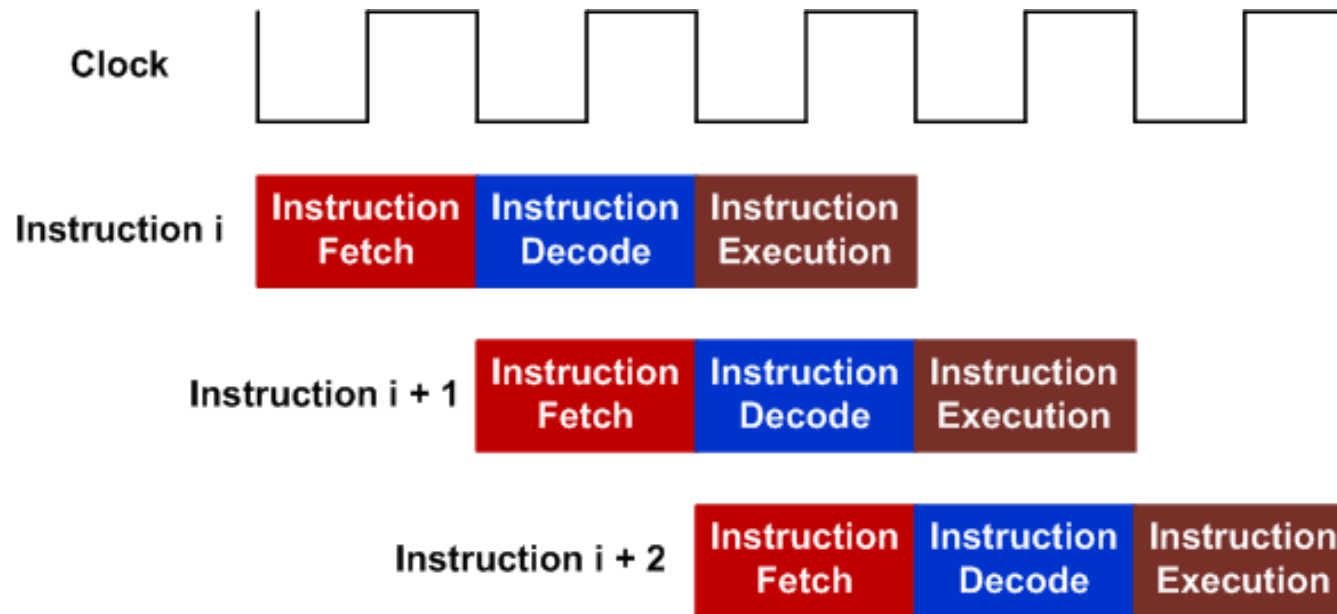
Program Execution

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



Three-state pipeline: Fetch, Decode, Execution

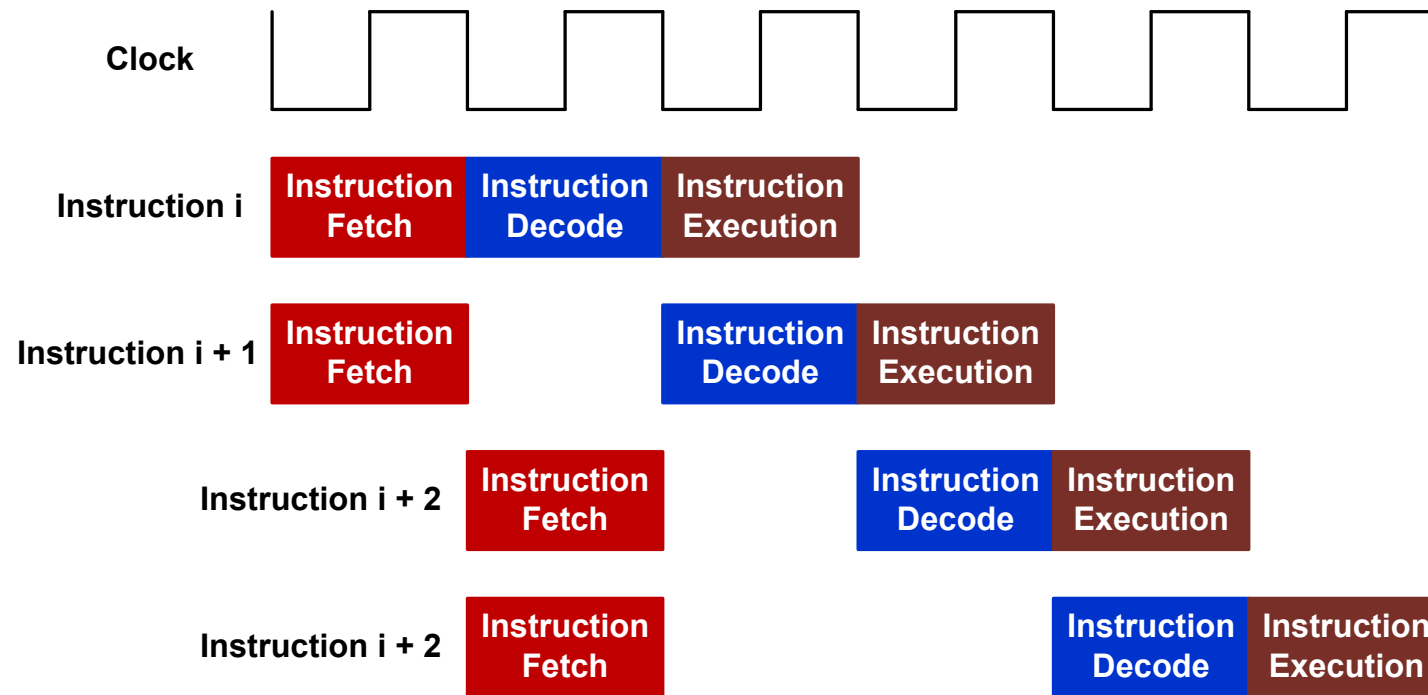
- ▶ **Pipelining** allows hardware resources to be fully utilized
- ▶ One 32-bit instruction or two 16-bit instructions can be fetched.



Pipeline of **32-bit** instructions

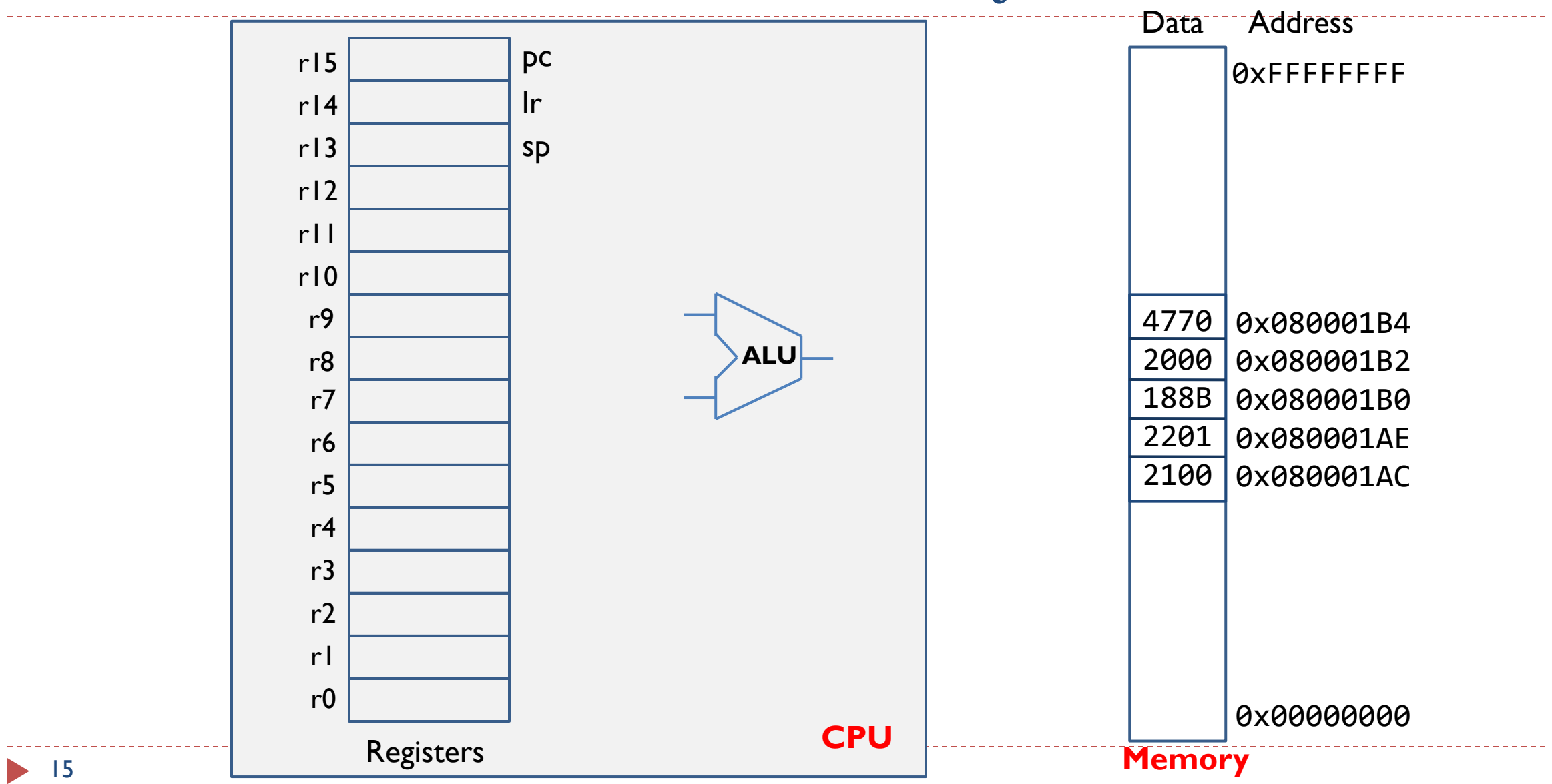
Three-state pipeline: Fetch, Decode, Execution

- ▶ **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 (each instruction fetch brings in 32 bits, two 16-bit instructions)

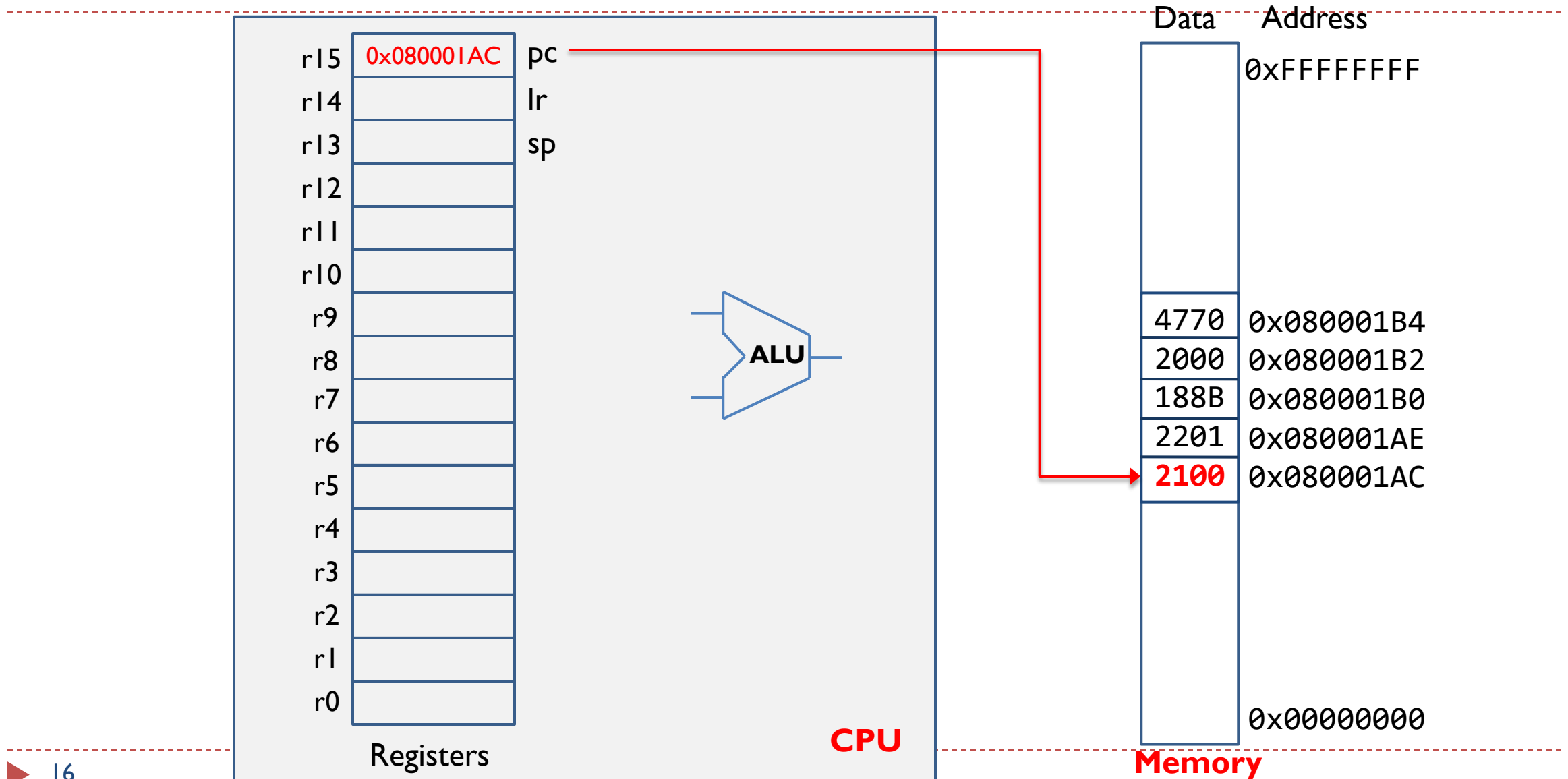
Machine codes are stored in memory



Fetch Instruction: pc = 0x08001AC

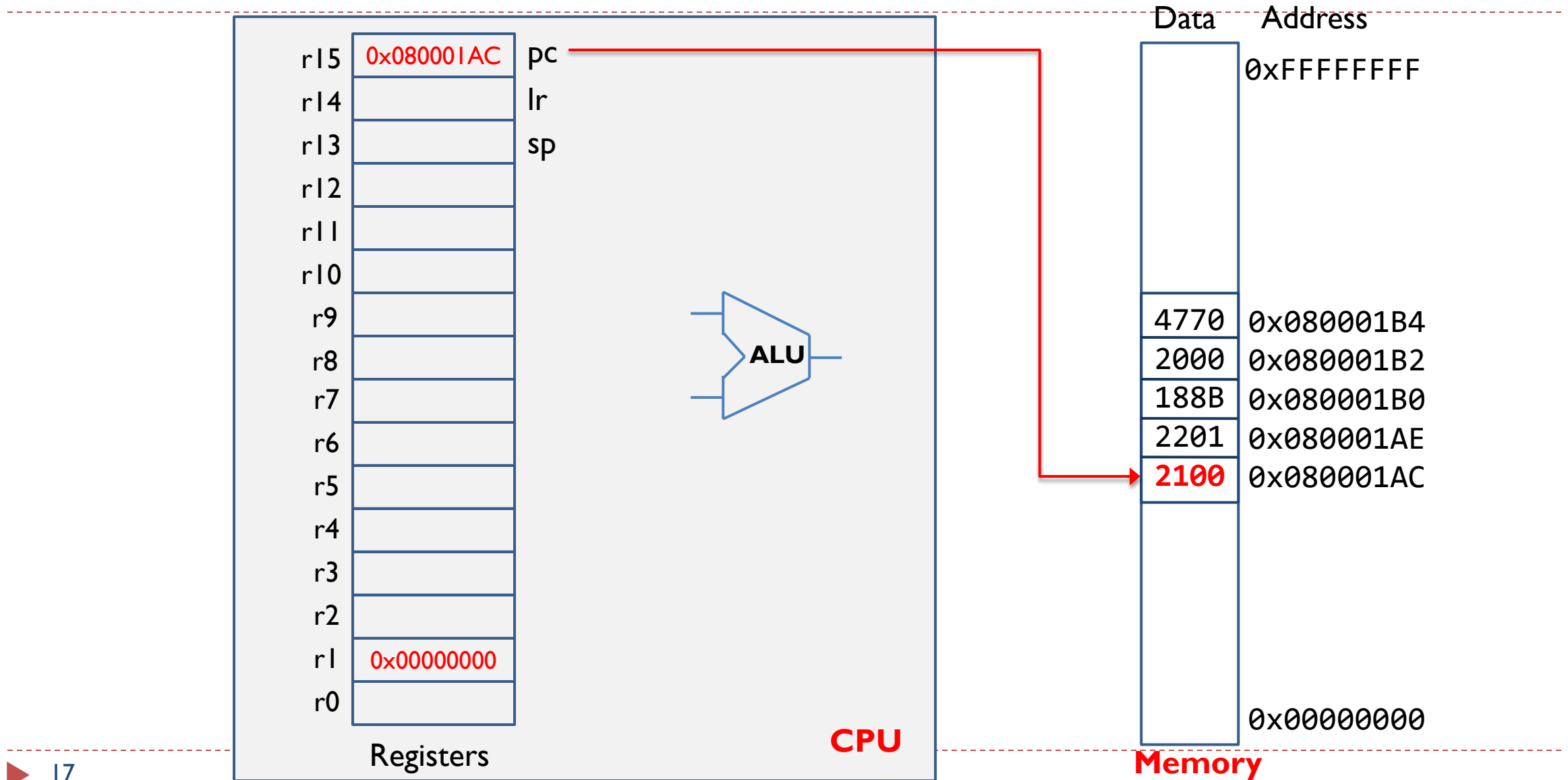
Decode Instruction: 2100 = **MOVS r1, #0x00**

2100 encodes the whole instruction
MOVS r1, #0x00 (details omitted)



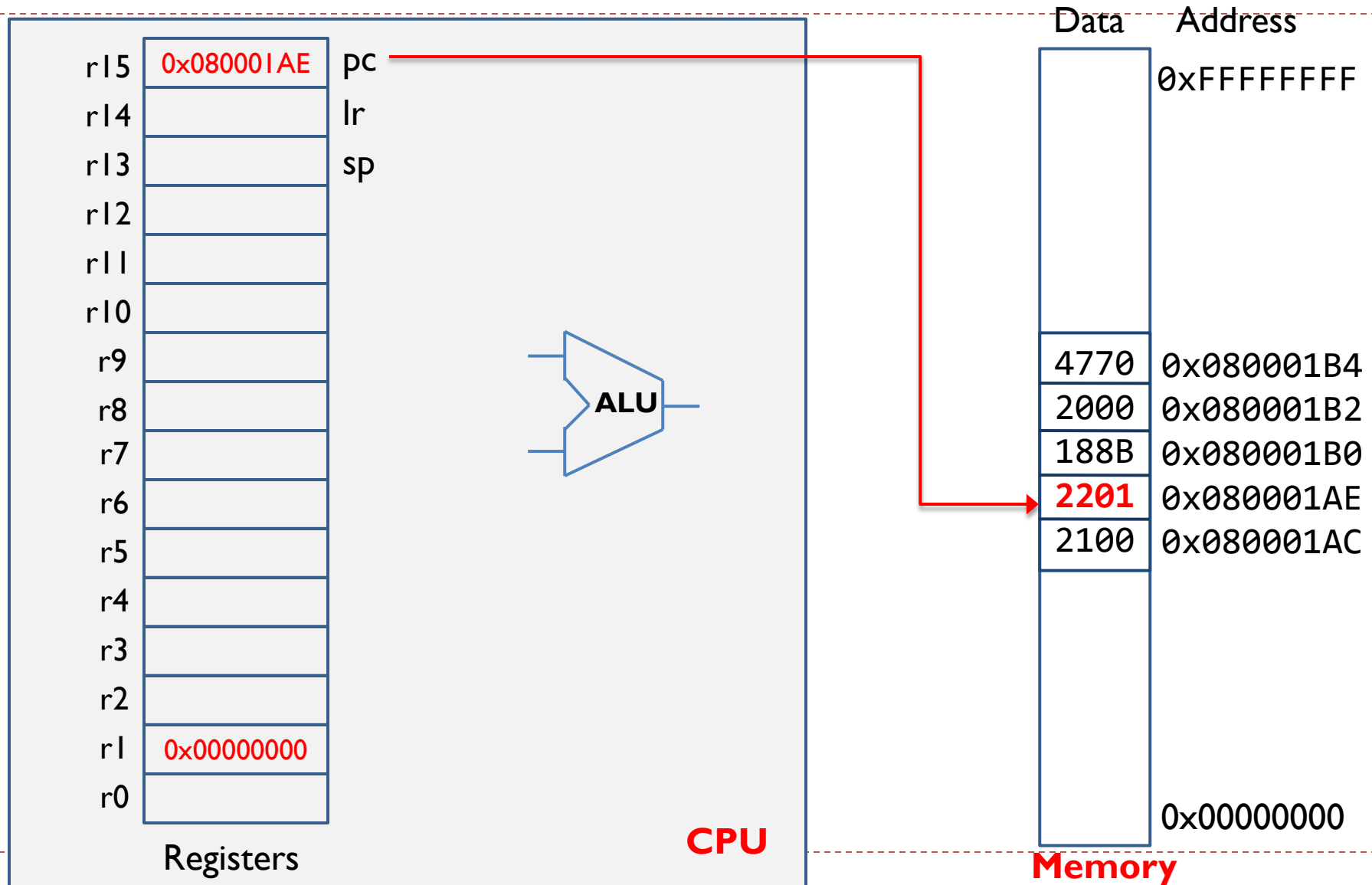
Execute Instruction:

MOVS r1, #0x00



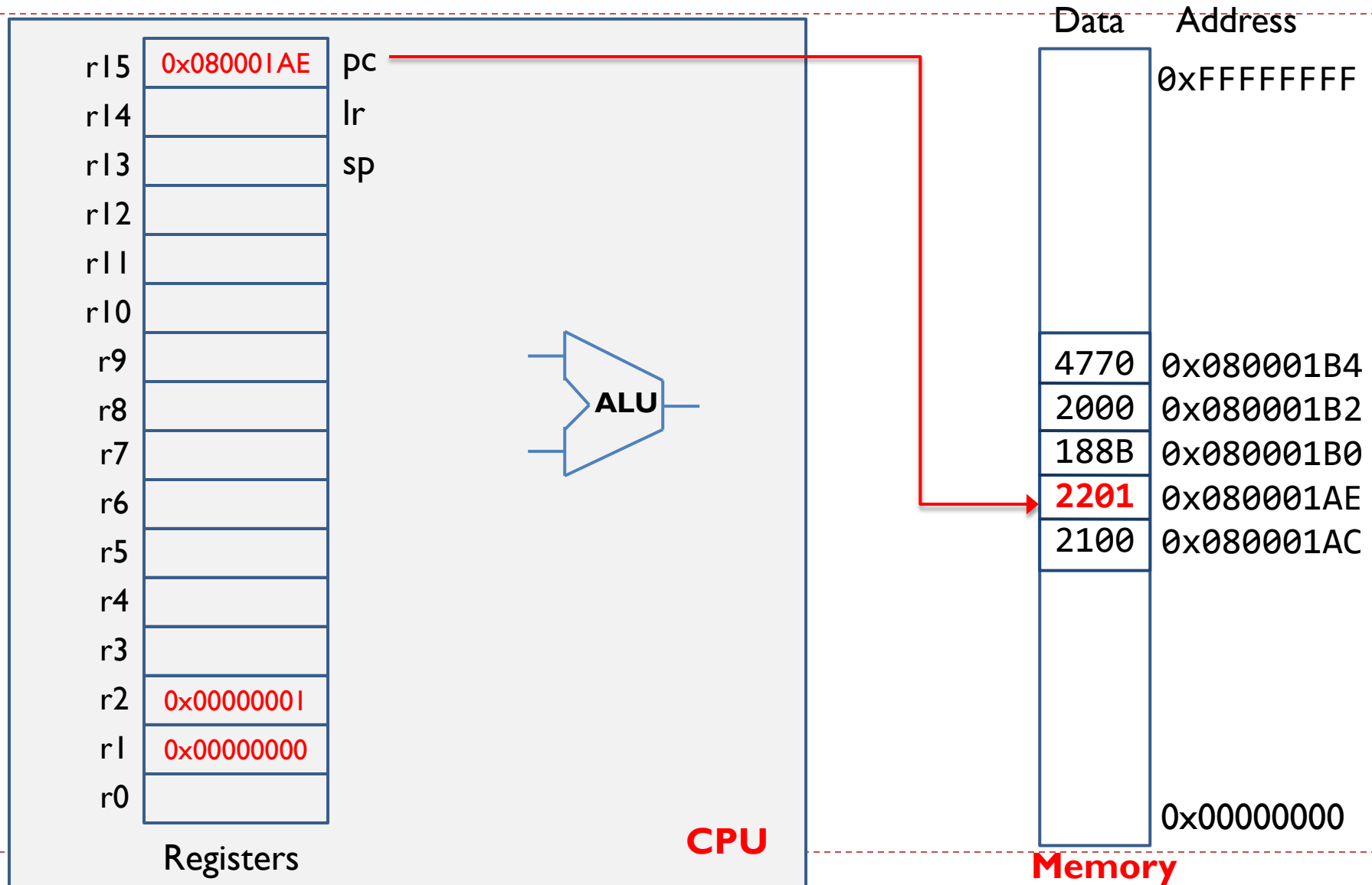
Fetch Next Instruction: $pc = pc + 2$

- 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.



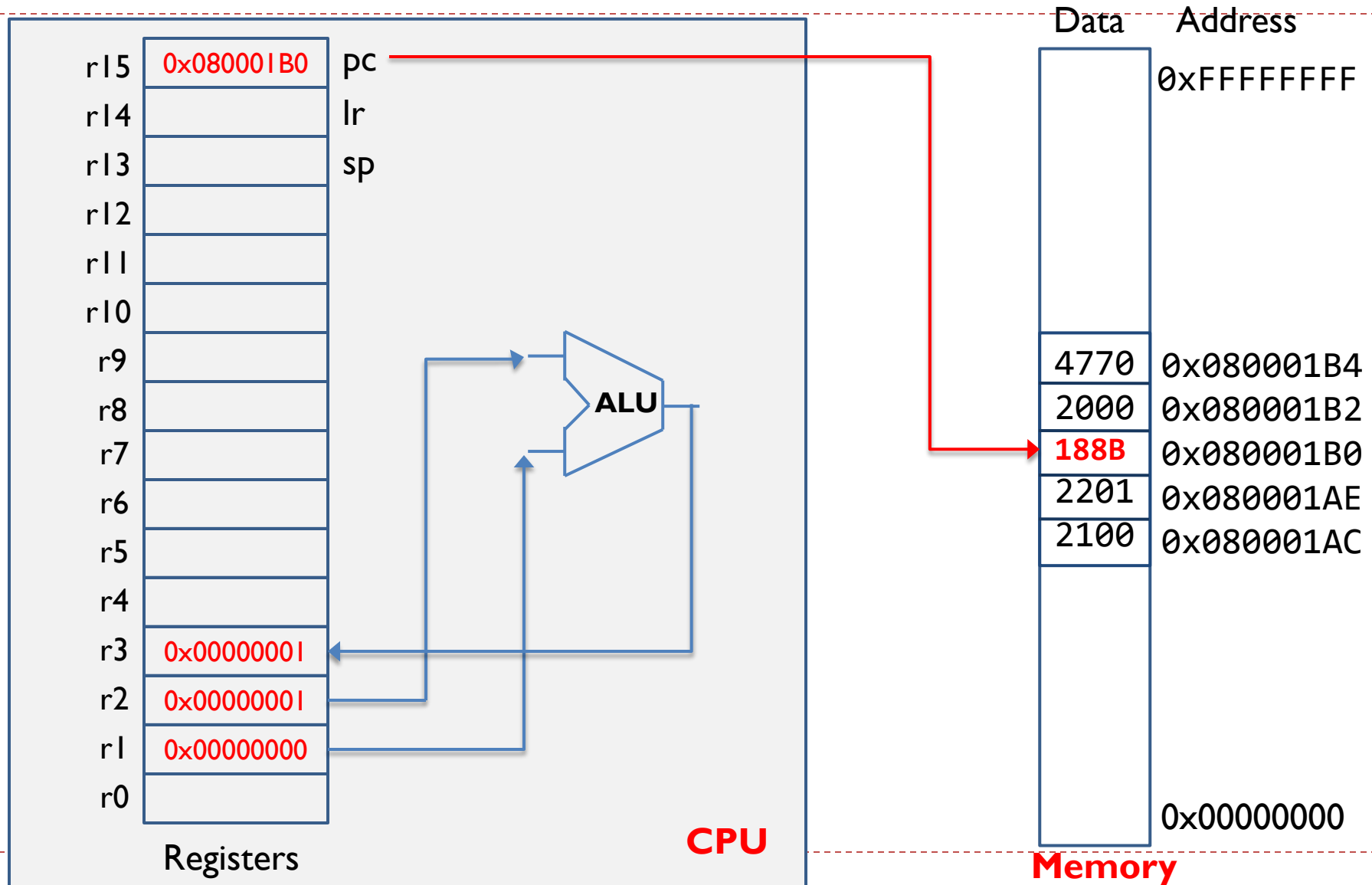
Fetch Next Instruction: $pc = pc + 2$

Decode & Execute: $2201 = \text{MOVS } r2, \#0x01$



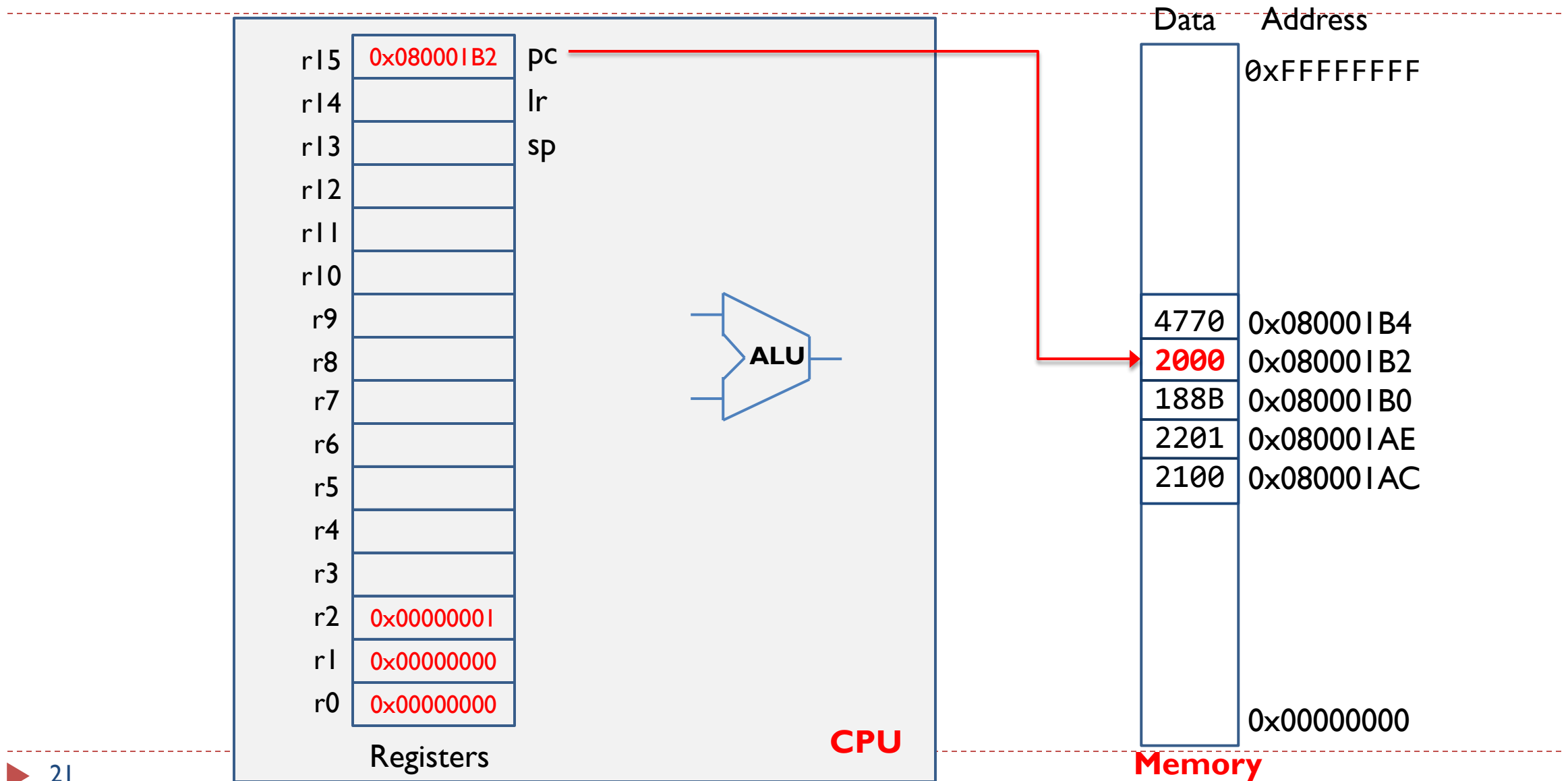
Fetch Next Instruction: $pc = pc + 2$

Decode & Execute: $188B = \text{ADDS } r3, r1, r2$



Fetch Next Instruction: $pc = pc + 2$

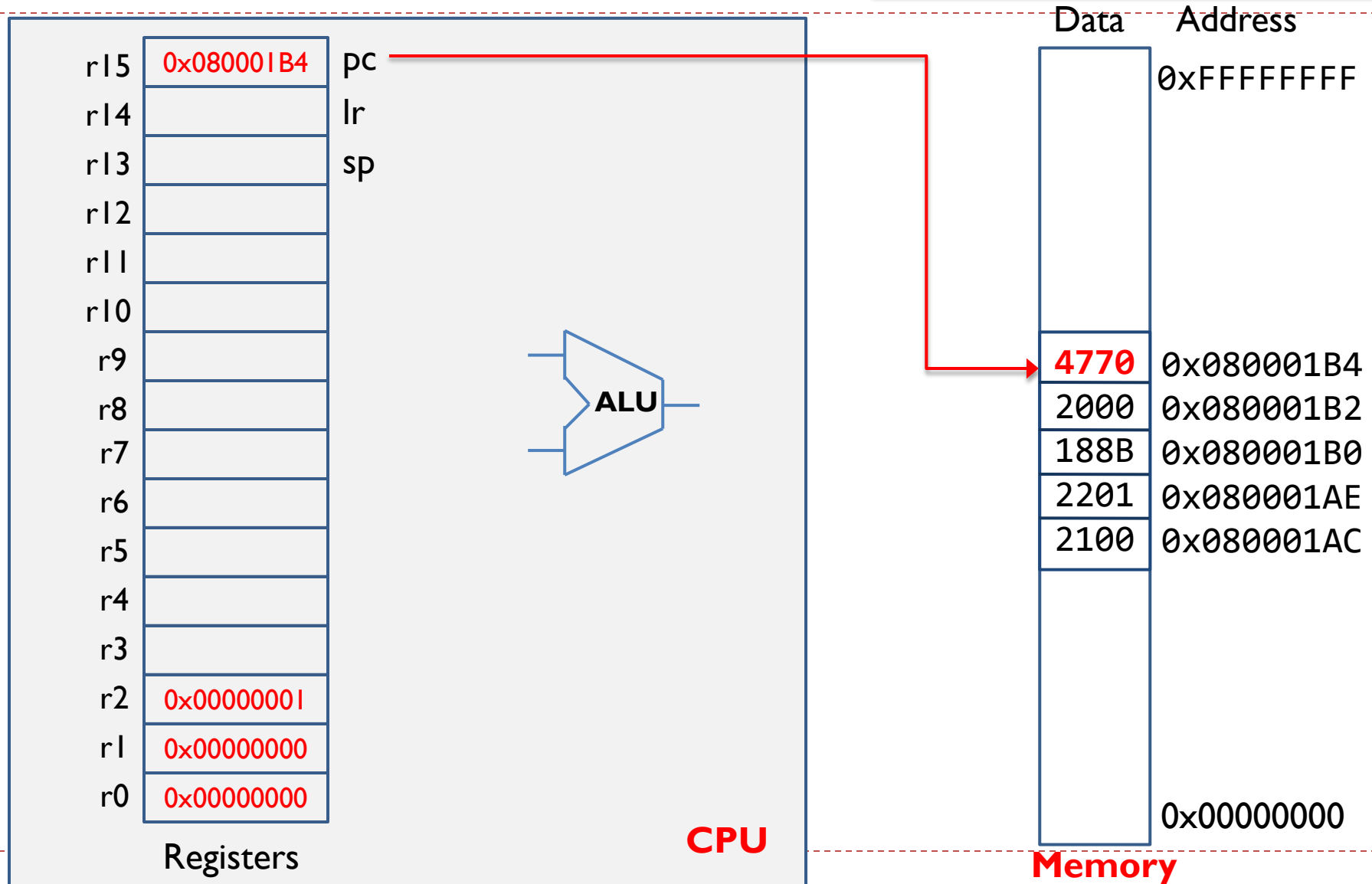
Decode & Execute: $2000 = \text{MOVS } r0, \#0x00$



Fetch Next Instruction: $pc = pc + 2$

Decode & Decode: $4770 = \text{BX } lr$

BX lr is “branch-and-exchange” return instruction: it branches to the address held in the link register (lr) and sets execution state



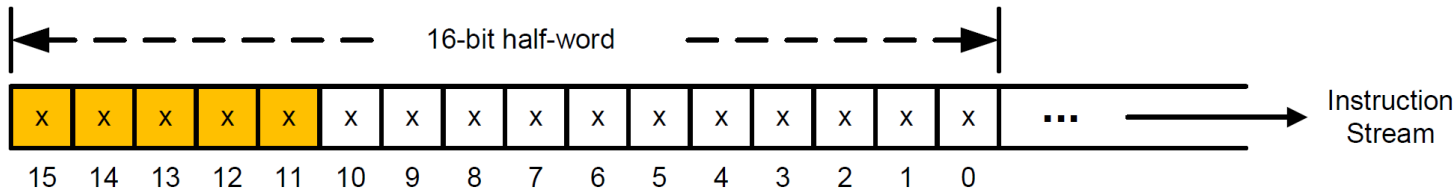
Realities

- ▶ 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:

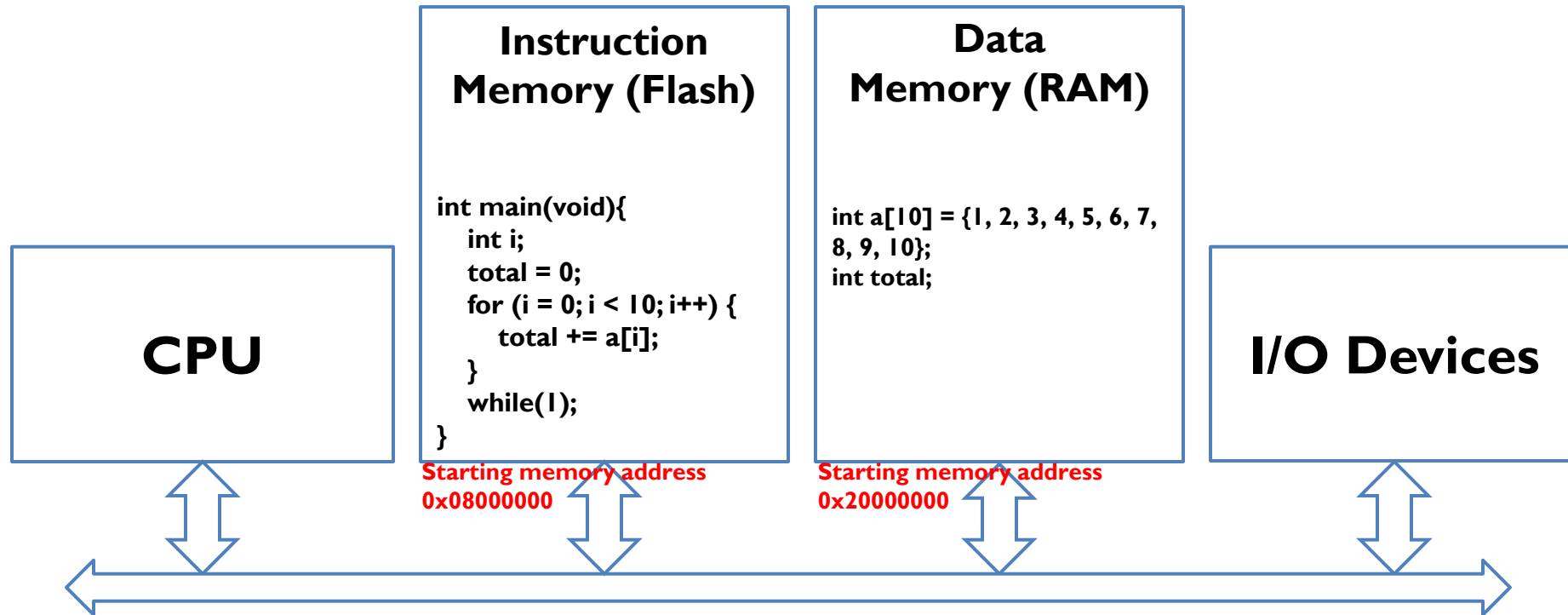
Calculate the Sum of an Array

```
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



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

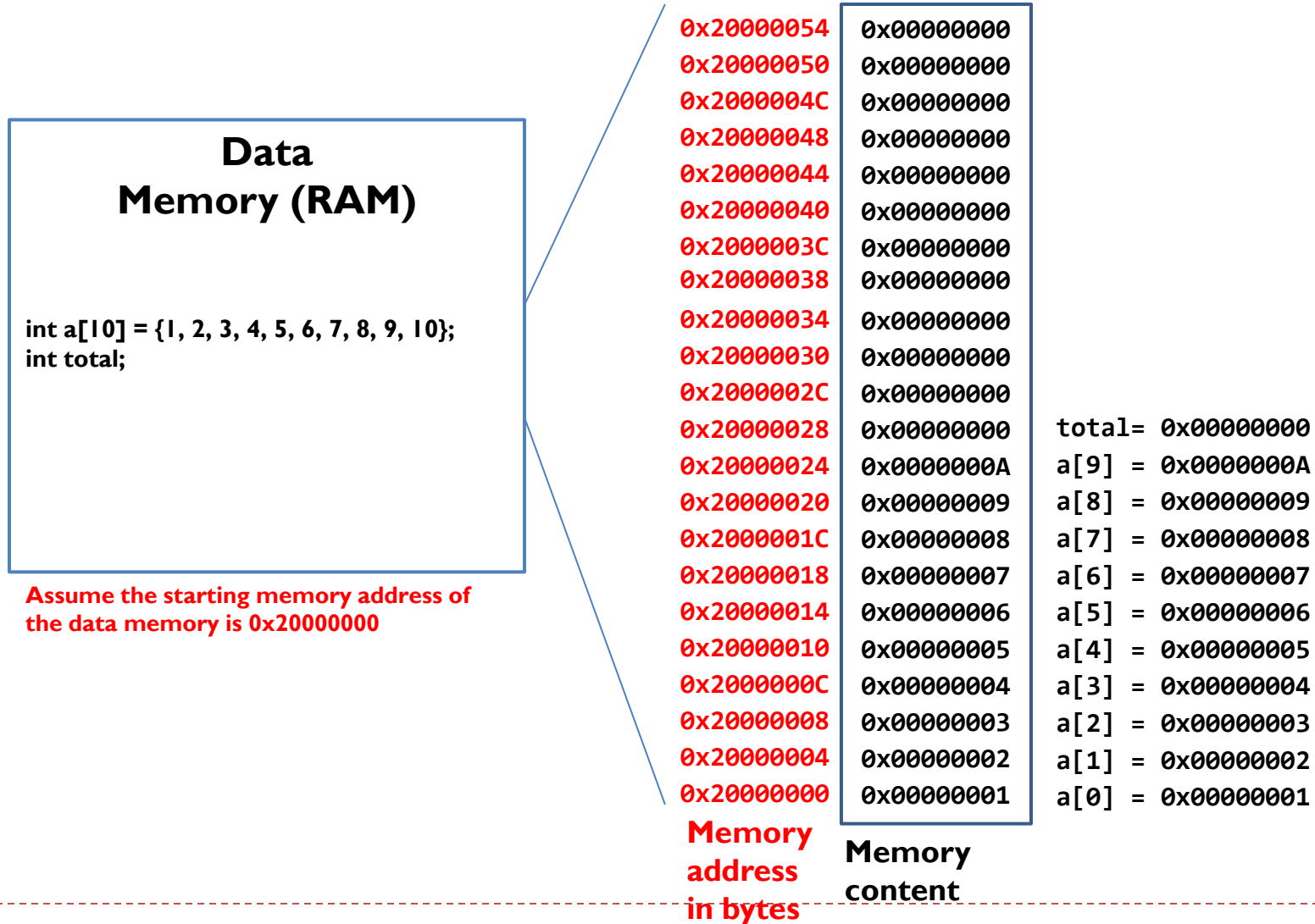
```
0010 0001 0000 0000  
0100 1010 0000 1000  
0110 0000 0001 0001  
0010 0000 0000 0000  
1110 0000 0000 1000  
0100 1001 0000 0111  
1111 1000 0101 0001  
0001 0000 0010 0000  
0100 1010 0000 0100  
0110 1000 0001 0010  
0100 0100 0001 0001  
0100 1010 0000 0011  
0110 0000 0001 0001  
0001 1100 0100 0000  
0010 1000 0000 1010  
1101 1011 1111 0100  
1011 1111 0000 0000  
1110 0111 1111 1110
```



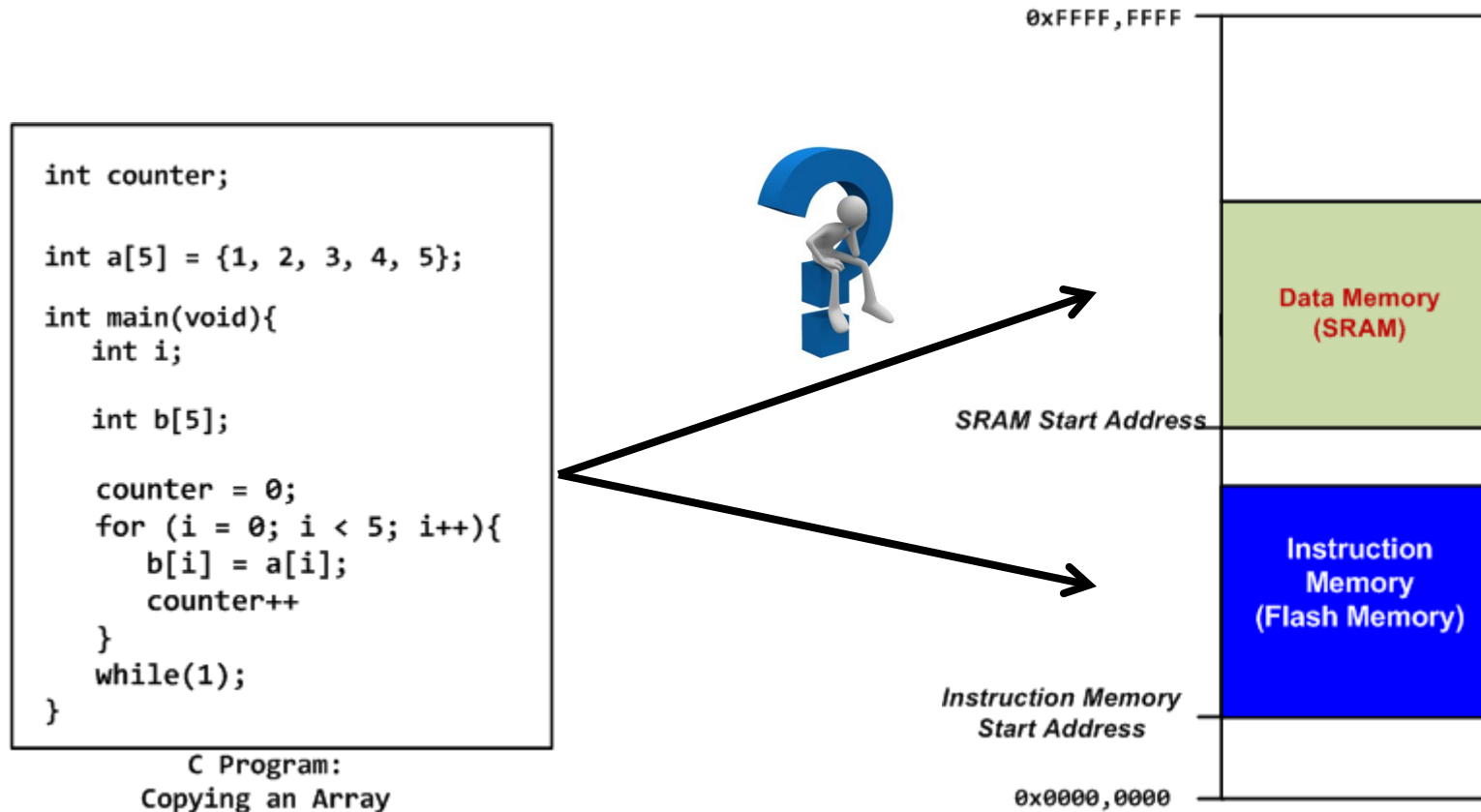
```
MOVS r1, #0x00  
LDR r2, = total_addr  
STR r1, [r2, #0x00]  
MOVS r0, #0x00  
B Check  
Loop: LDR r1, = a_addr  
LDR r1, [r1, r0, LSL #2]  
LDR r2, = total_addr  
LDR r2, [r2, #0x00]  
ADD r1, r1, r2  
LDR r2, = total_addr  
STR r1, [r2, #0x00]  
ADDS r0, r0, #1  
Check: CMP r0, #0x0A  
BLT Loop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

Example:

Calculate the Sum of an Array

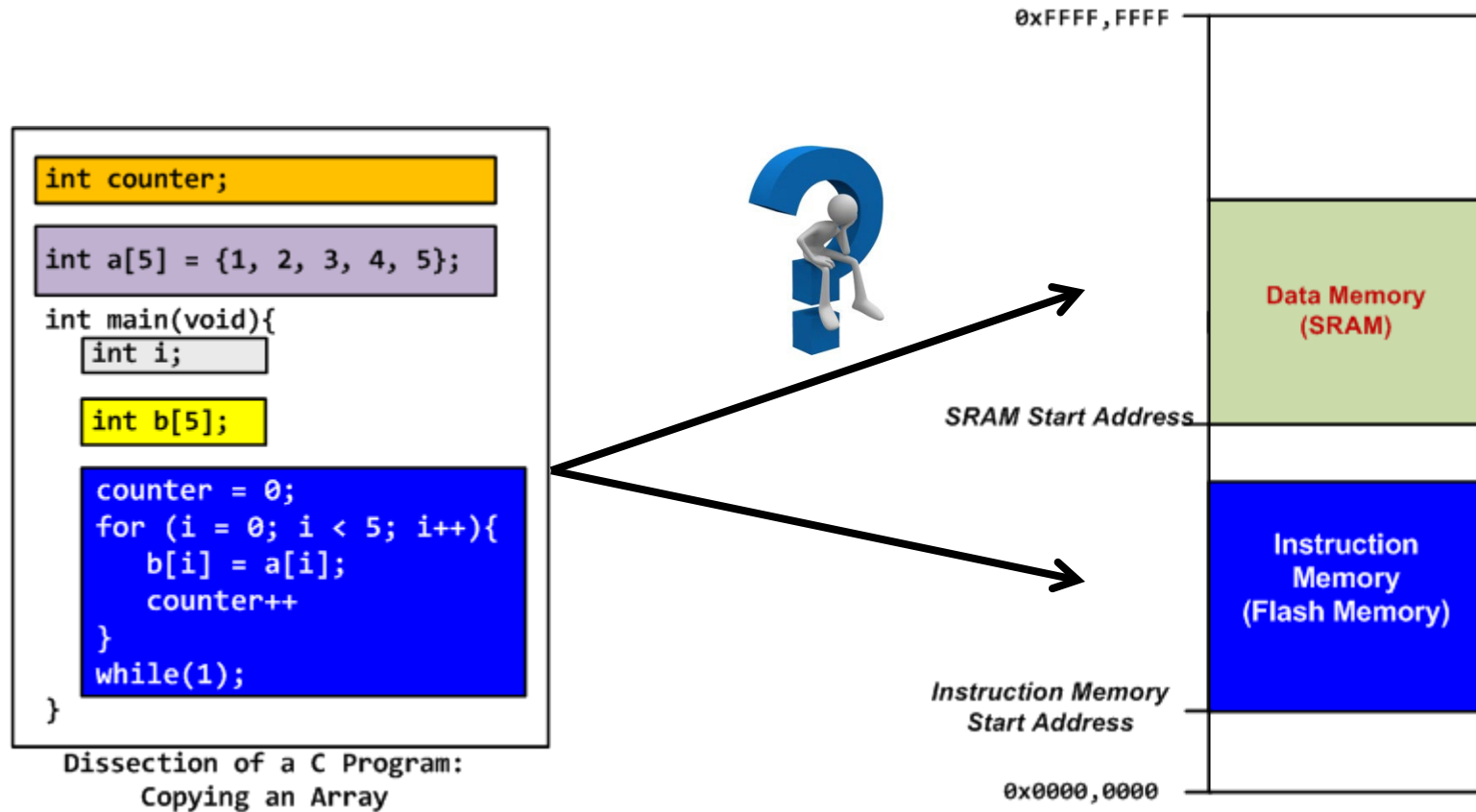


Loading Code and Data into Memory

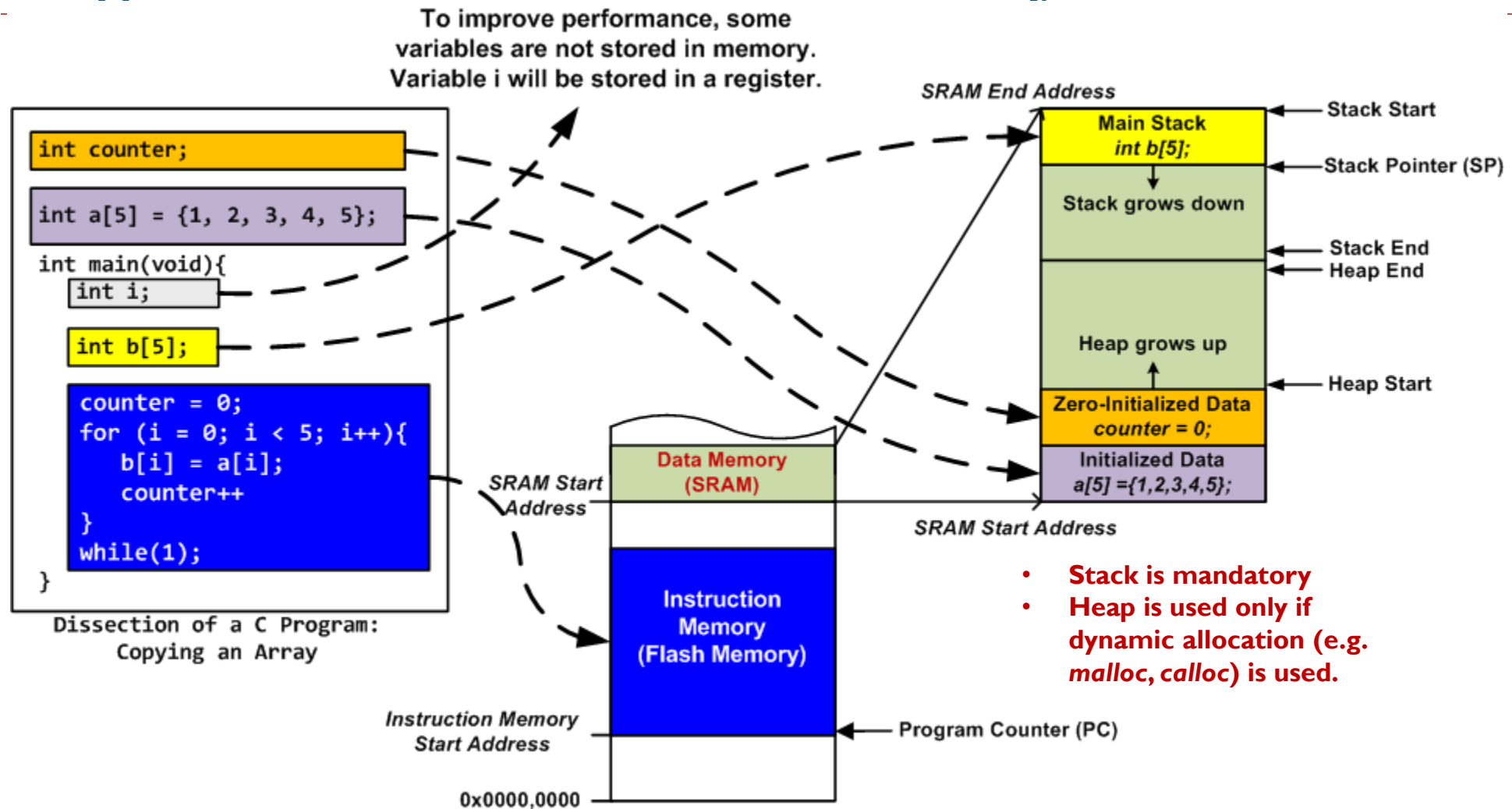


ARM Cortex-M uses a modified Harvard architecture: a unified address space with separate instruction and data paths.

Loading Code and Data into Memory

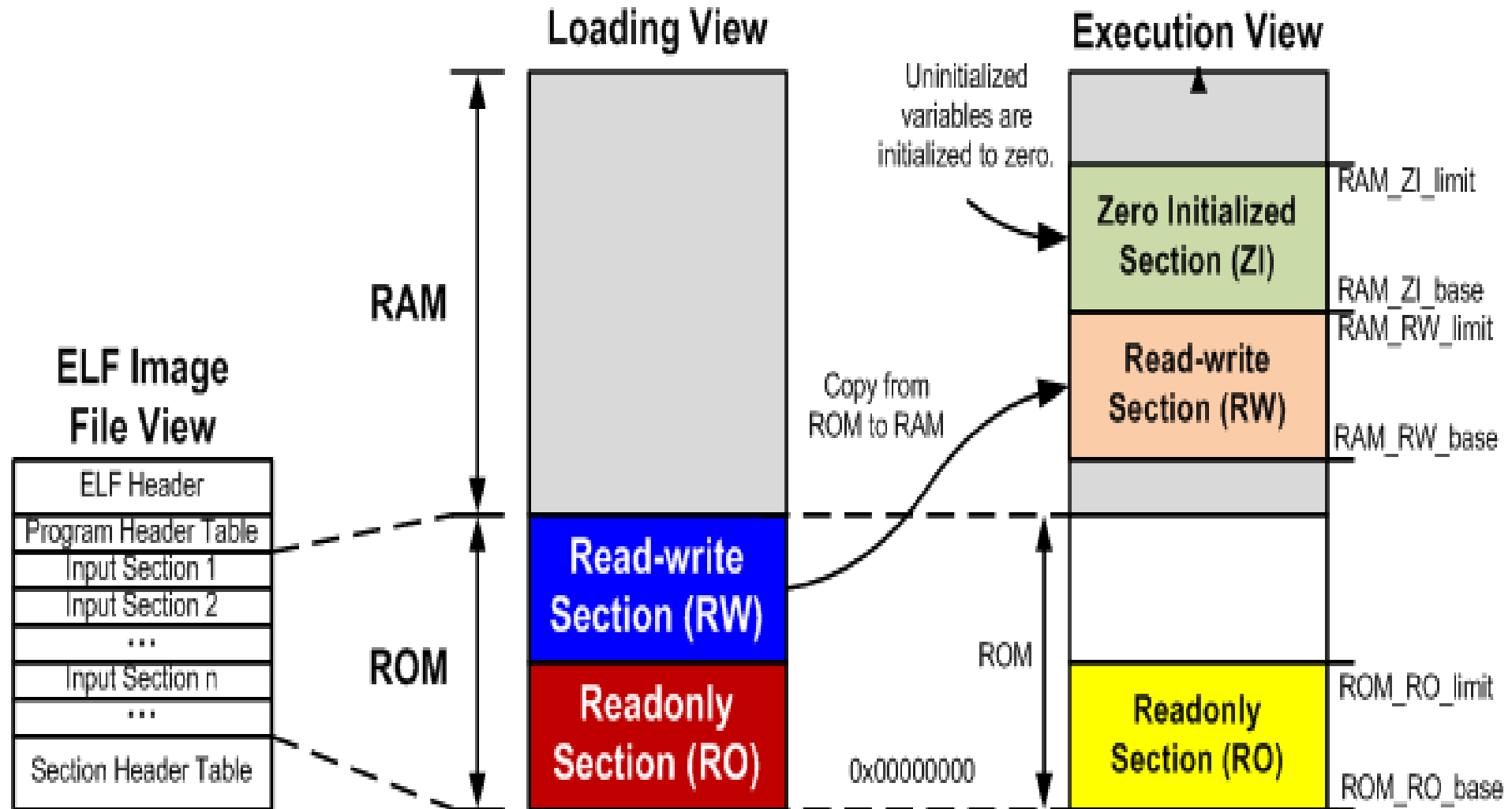


Loading Code and Data into Memory

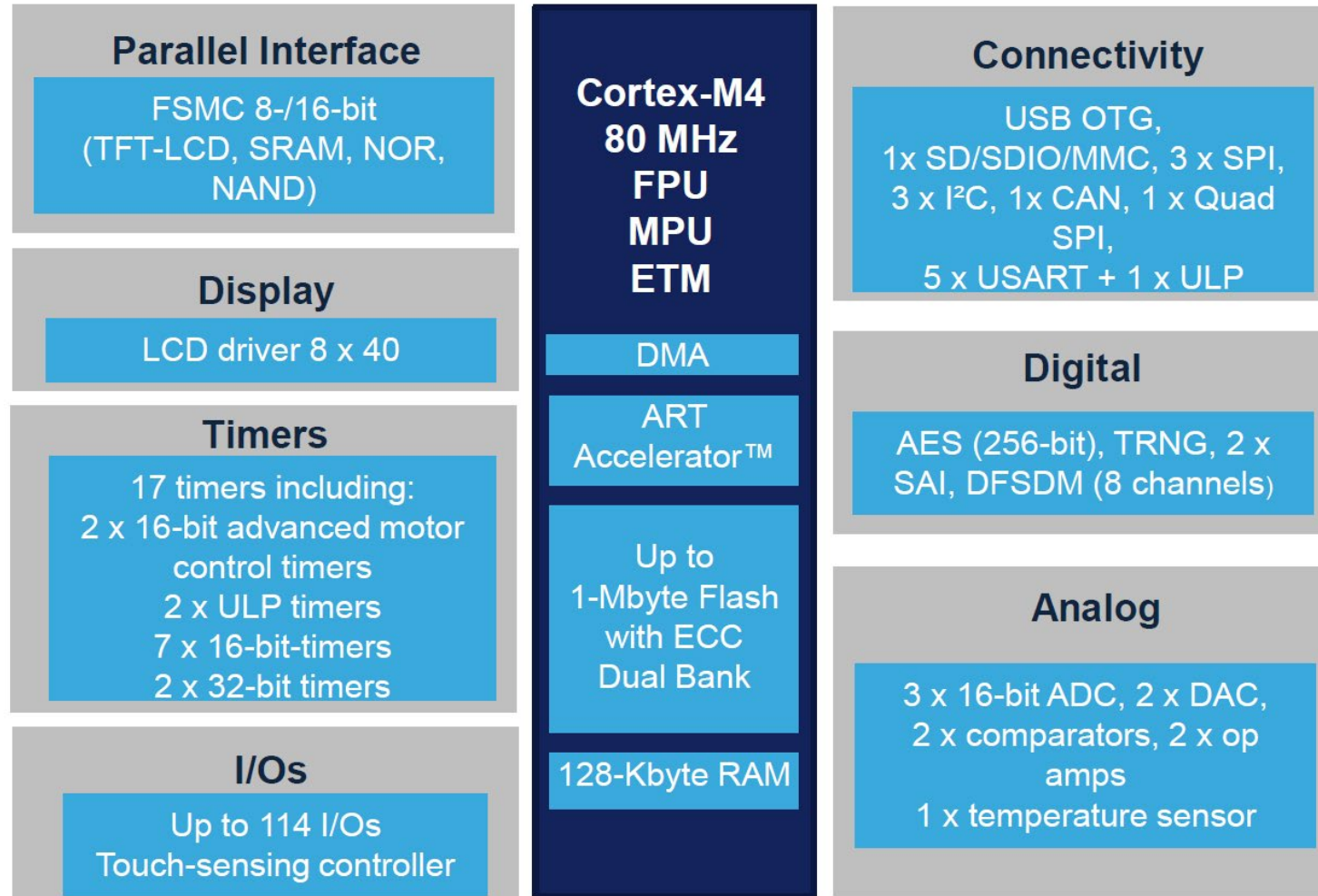


Instruction memory is programmed in Flash once at load time; only data sections are copied/initialized in SRAM at reset

View of a Binary Program



STM32L4



Memory Map

