Chapter 1
Computer and Assembly Language

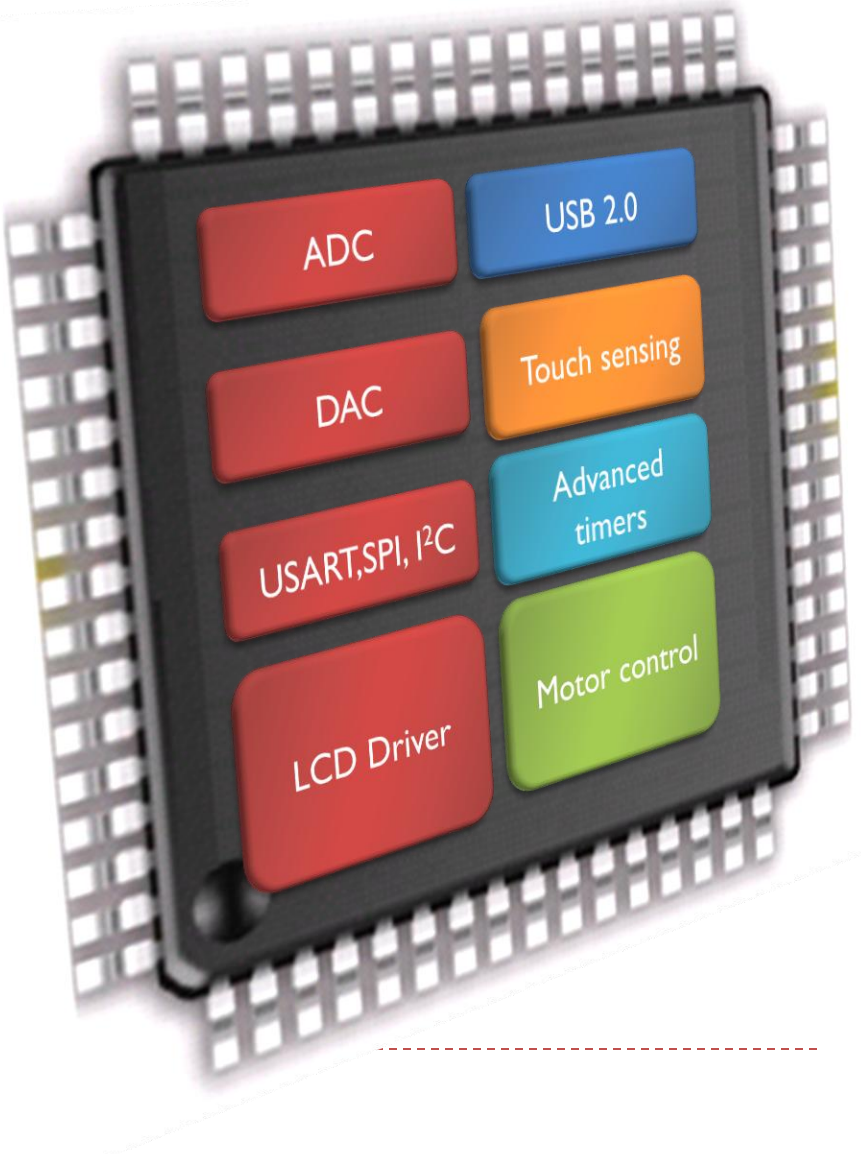
Zonghua Gu

Fall 2025

Acknowledgement: Lecture slides based on Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, University of Maine https://web.eece.maine.edu/~zhu/book/

Why ARM processor

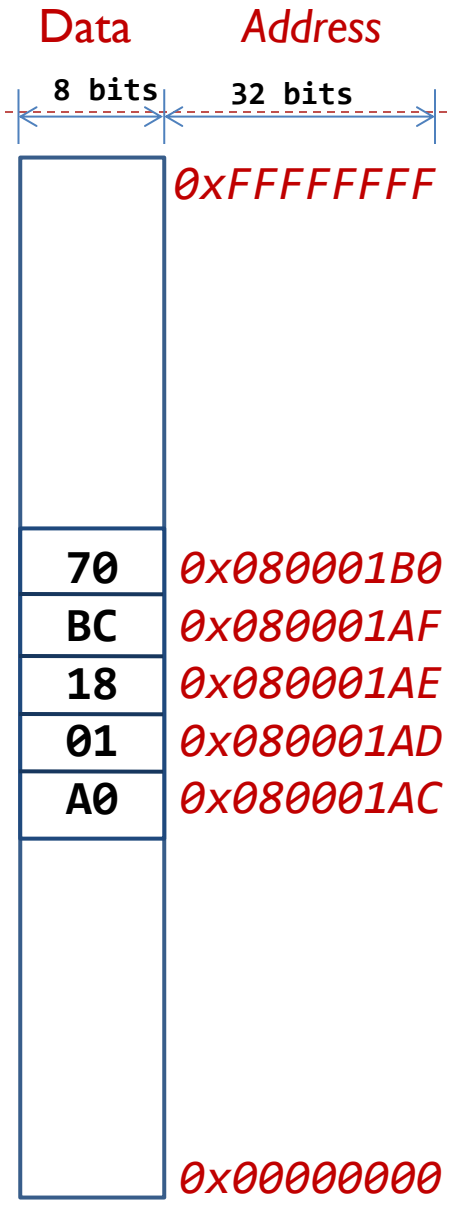
- ARM: Acron RISC Machine, founded in 1990
- Public company, Headquarter at Cambridge, England, UK, 2023 Revenue: US$2.68 billion
- Arm processors are used as the main CPU for most mobile phones and handhelds.
- The world's second fastest supercomputer (previously fastest) in 2022, the Japanese Fugaku is based on Arm AArch64 architecture



Embedded Systems

Memory

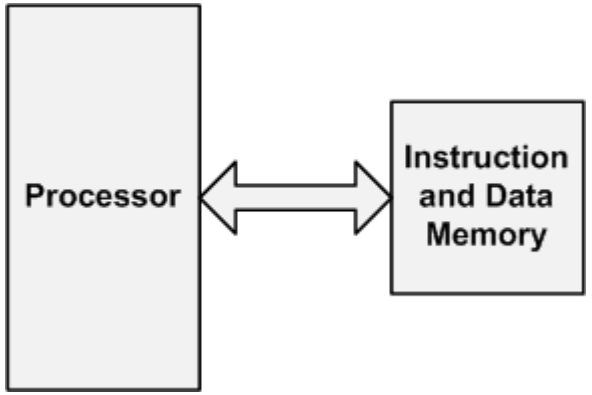
- Memory is arranged as a series of "locations"
  - Each location has a unique "address"
  - Each location holds a byte (***byte-addressable***)
  - e.g. the memory location at address `0x080001B0` contains the byte value `0x70`, i.e., 112
- The number of locations in memory is limited
  - e.g. 4 GB 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



Computer Architecture

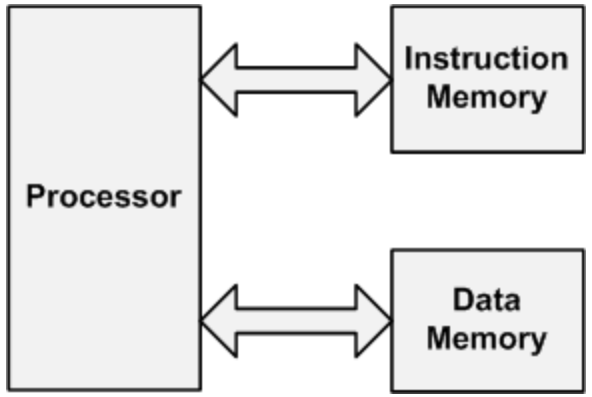
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

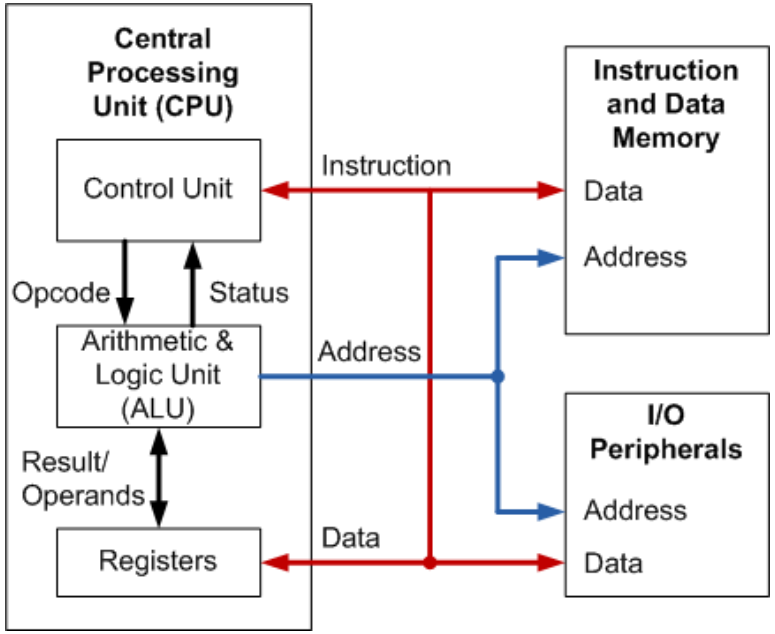
Data and instructions are stored into separate memories.



Computer Architecture

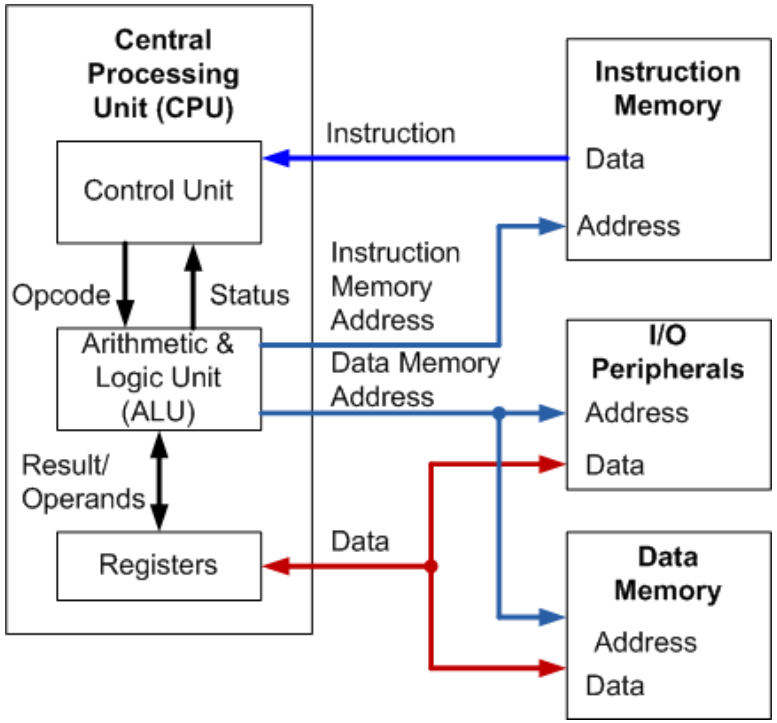
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

Data and instructions are stored into separate memories.



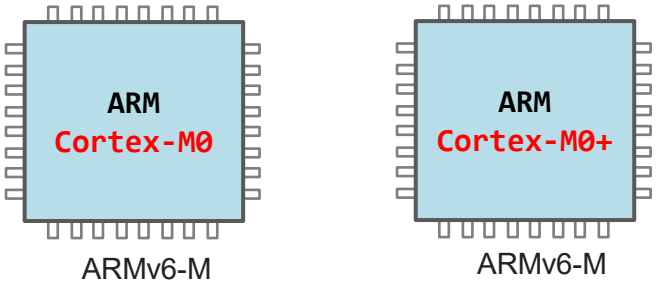
von Neumann vs. Harvard

- Harvard allows two simultaneous memory fetches.
- Most DSPs use Harvard architecture for streaming data:
  - greater memory bandwidth
  - more predictable bandwidth
- von Neumann allows more flexible placement of instructions and data, hence more efficient memory space utilization

ARM Cortex-M Series Family

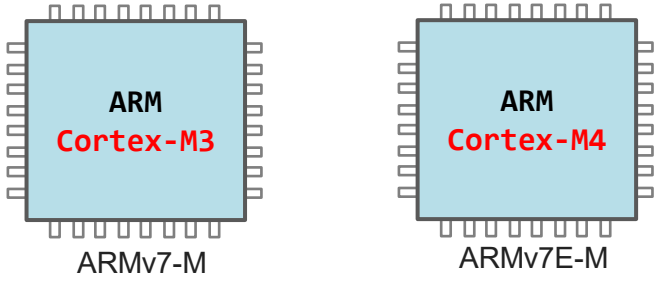
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

Data and instructions are stored into separate memories.



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
```

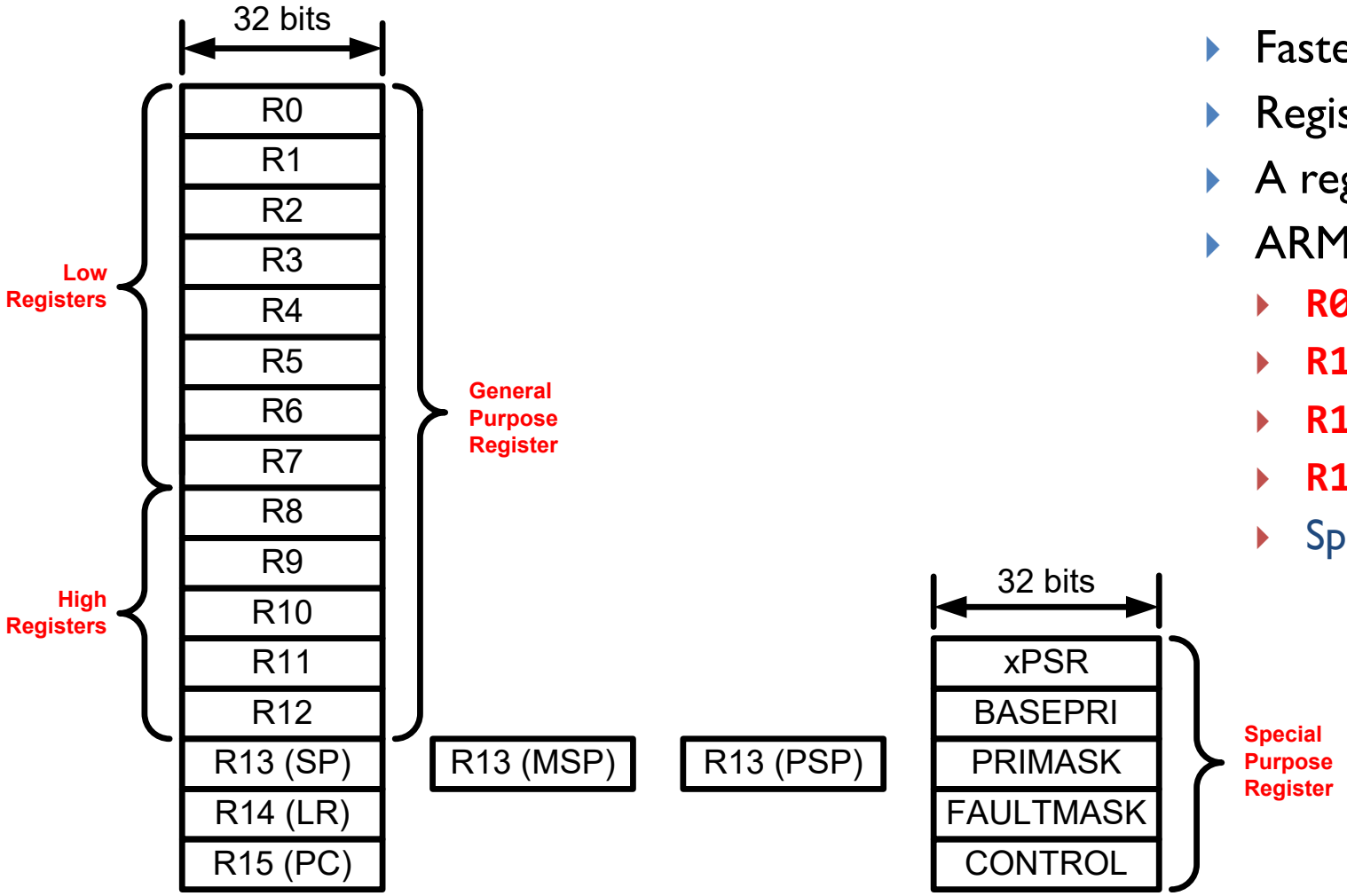
In Binary

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

In Hex

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

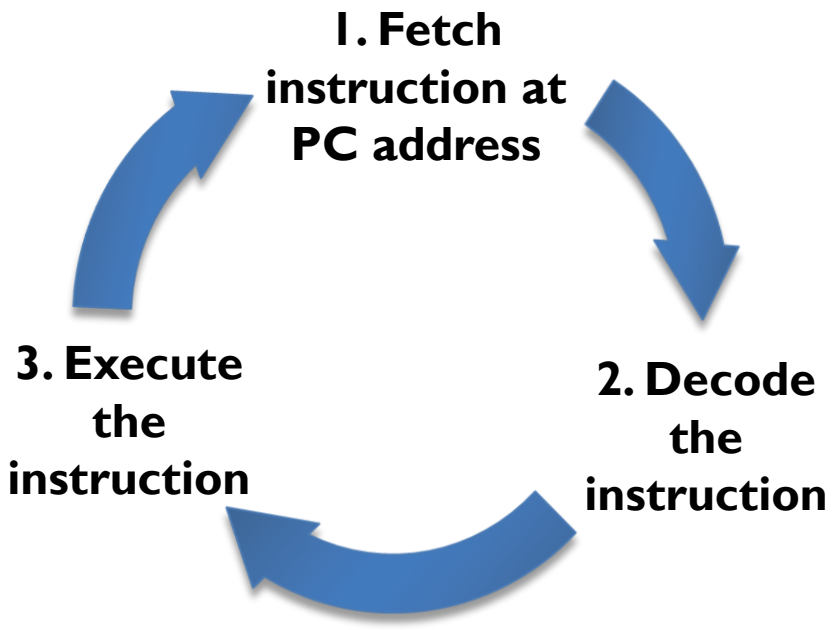
Processor Registers

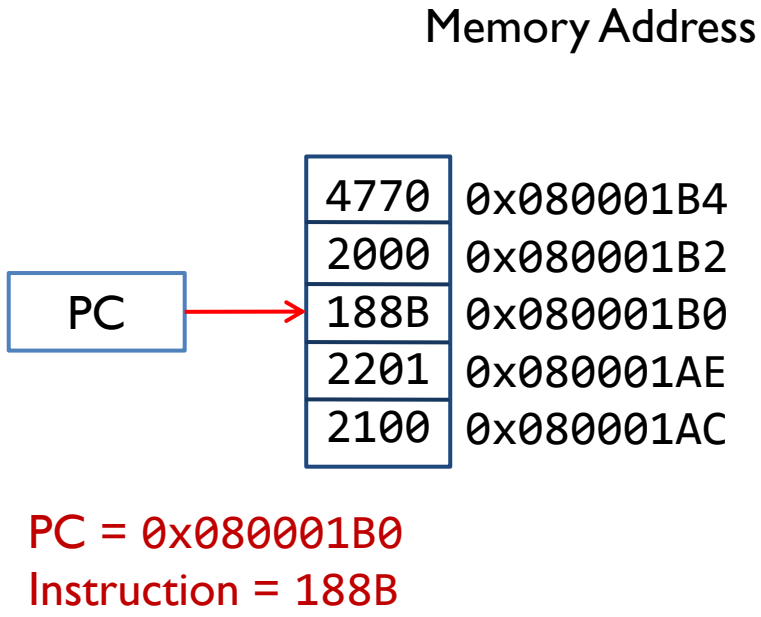


- Fastest way to read and write
- Registers are within the processor chip
- A register stores 32-bit value
- ARM Cortex-M has
  - **R0-R12**: 13 general-purpose registers
  - **R13**: Stack pointer (Shadow of MSP or PSP)
  - **R14**: Link register (LR)
  - **R15**: Program counter (PC)
  - Special registers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

Program Execution

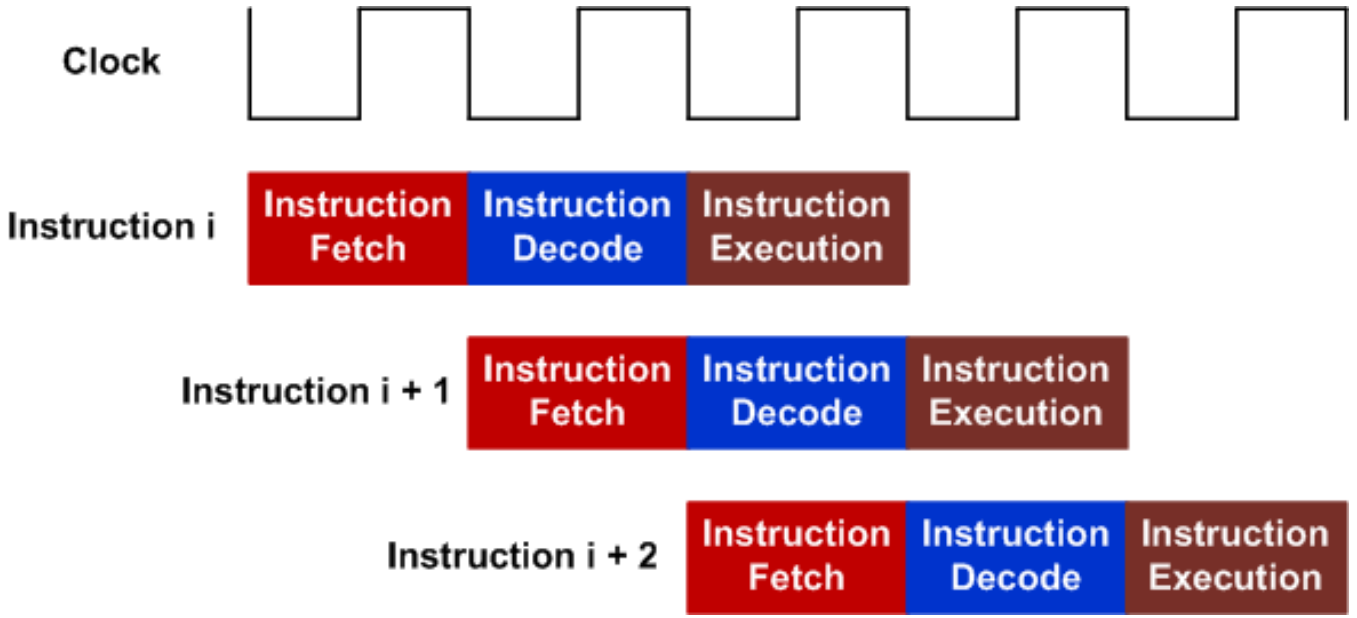
- **Program Counter (PC)** is a register that holds the memory address of the next instruction to be fetched from the memory.





Three-state pipeline: Fetch, Decode, Execution

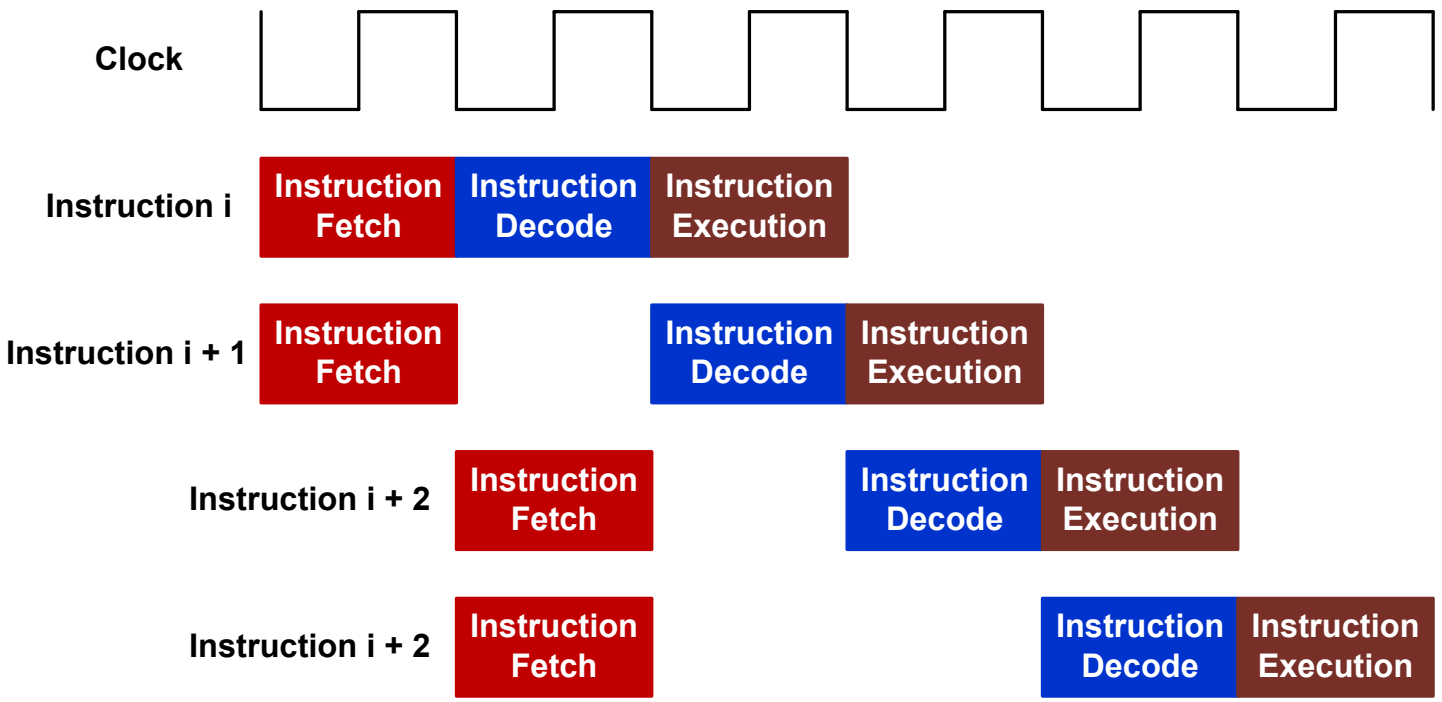
- **Pipelining** allows hardware resources to be fully utilized
- One 32-bit instruction or two 16-bit instructions can be fetched.



Pipeline of 32-bit instructions

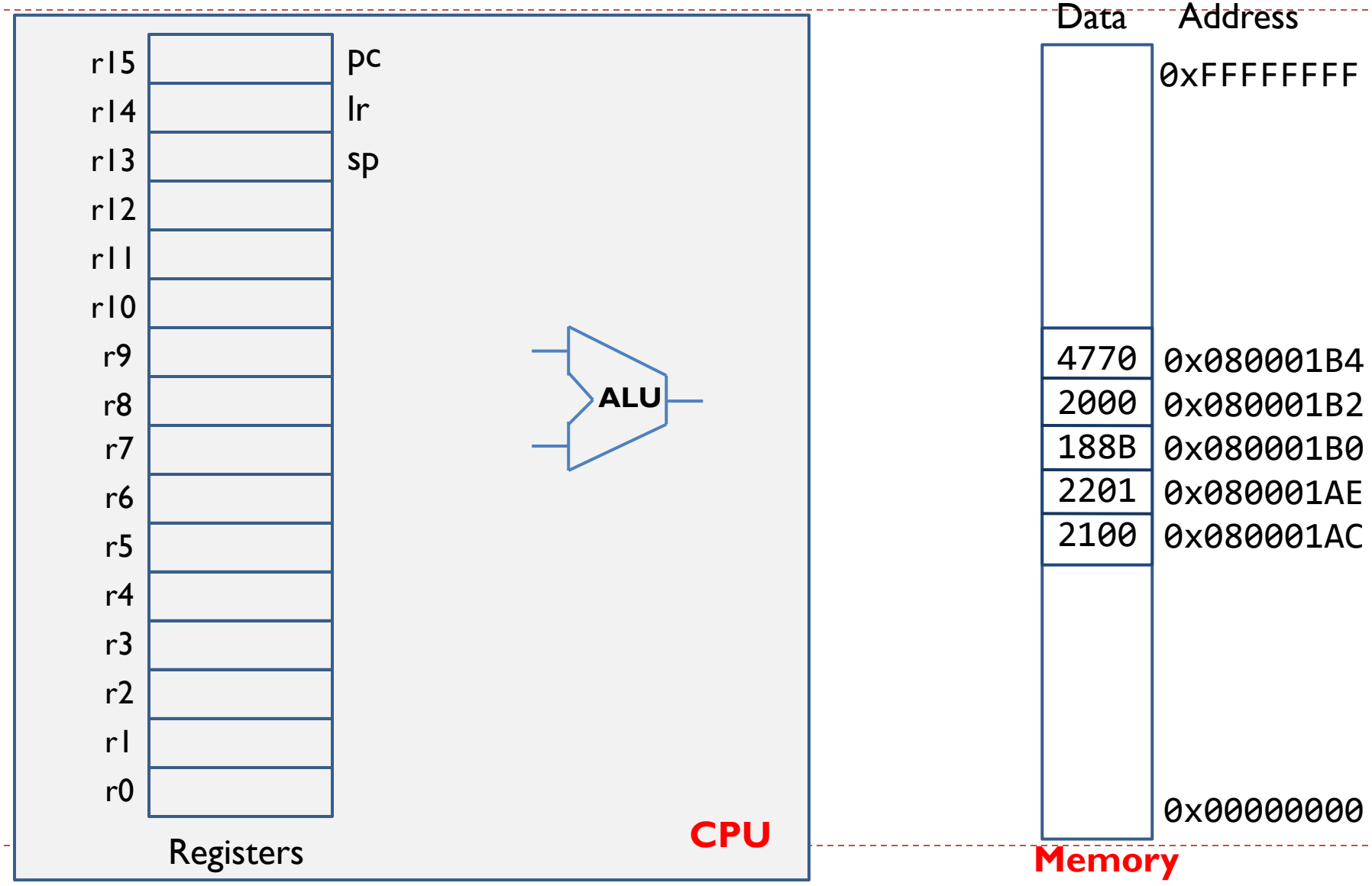
Three-state pipeline: Fetch, Decode, Execution

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



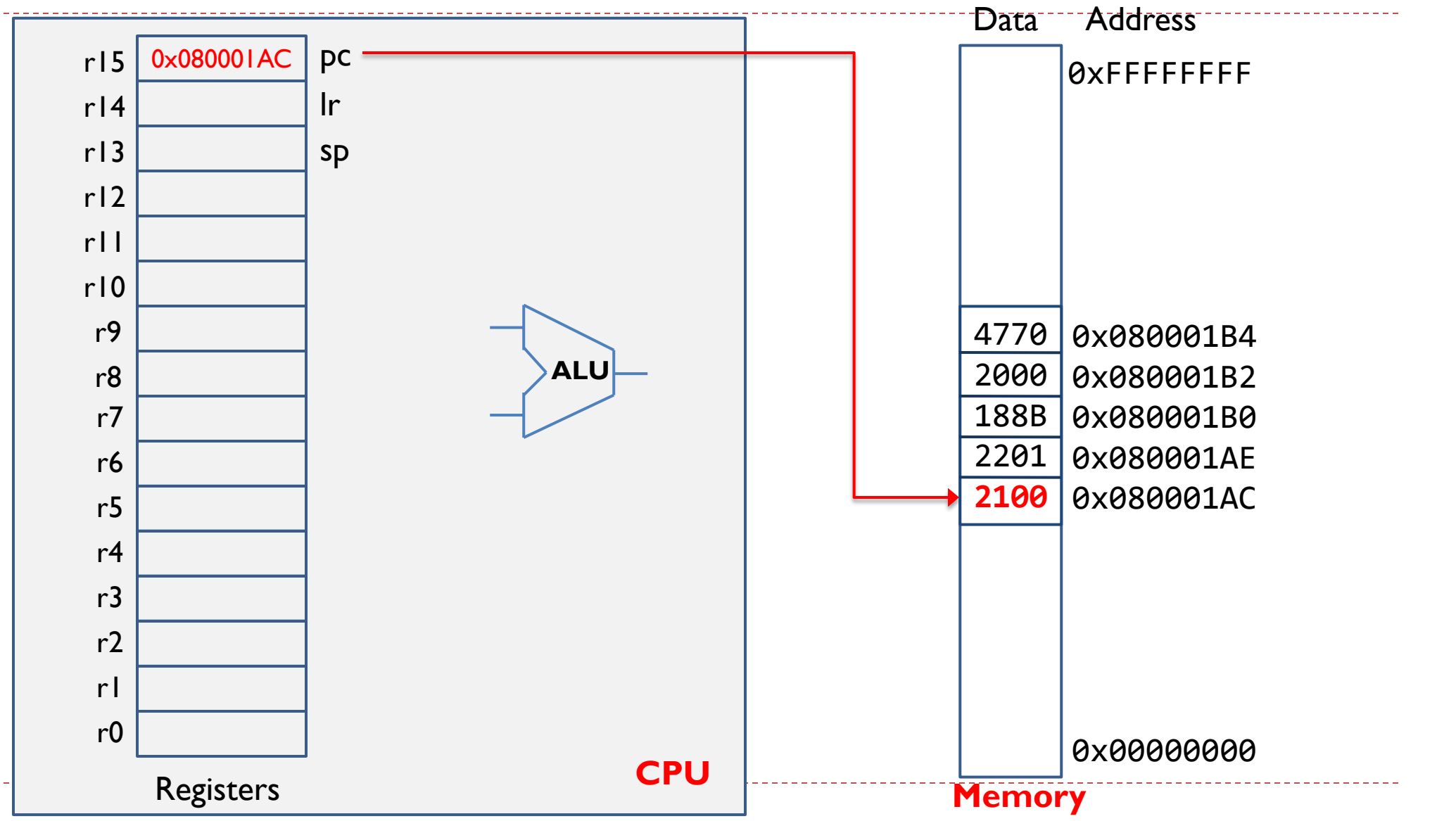
Pipeline of 16-bit instructions (each instruction fetch brings in 32 bits, two 16-bit instructions)

Machine codes are stored in memory



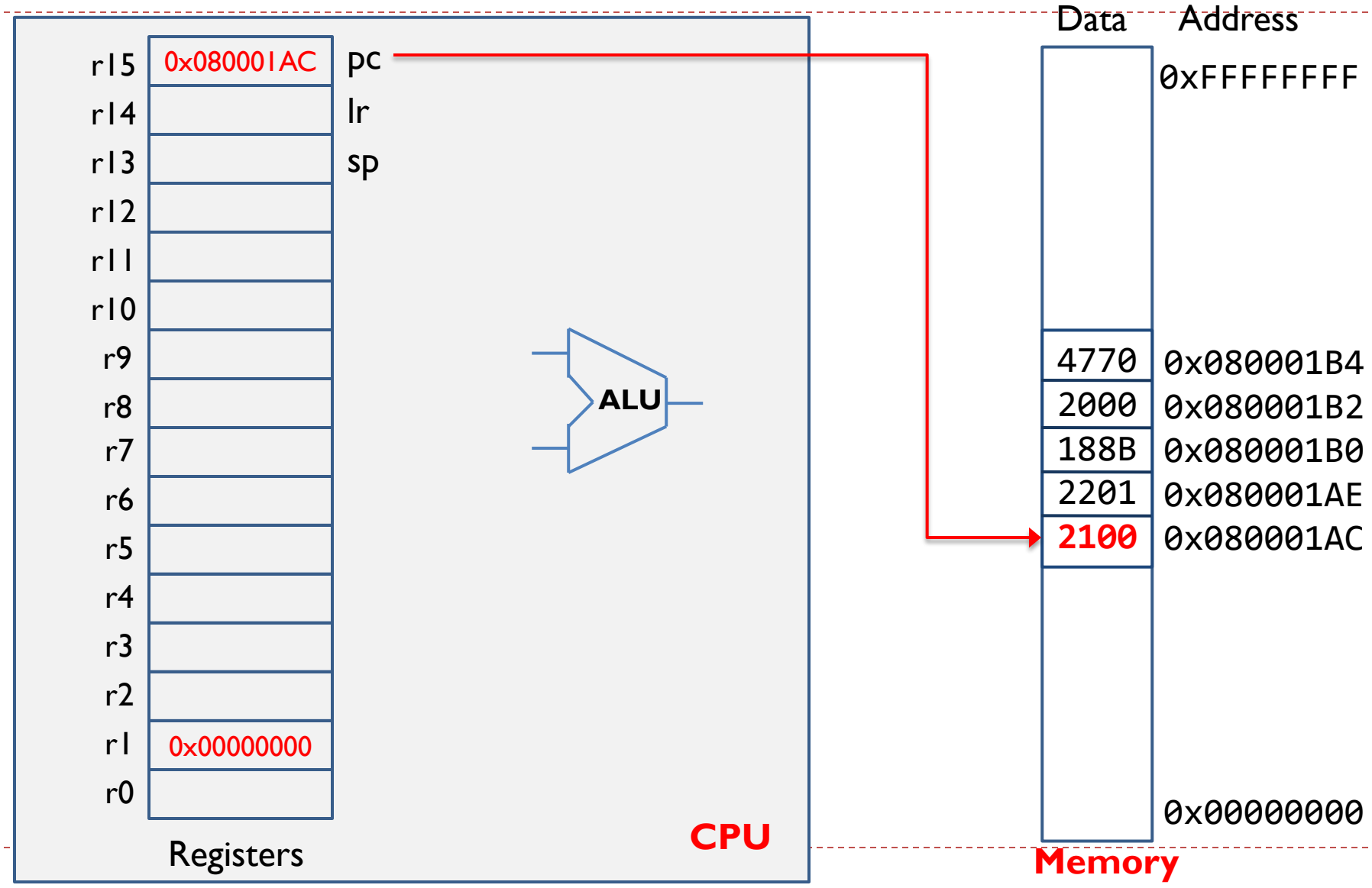
Fetch Instruction: pc = 0x08001AC
Decode Instruction: 2100 = MOVS r1, #0x00

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



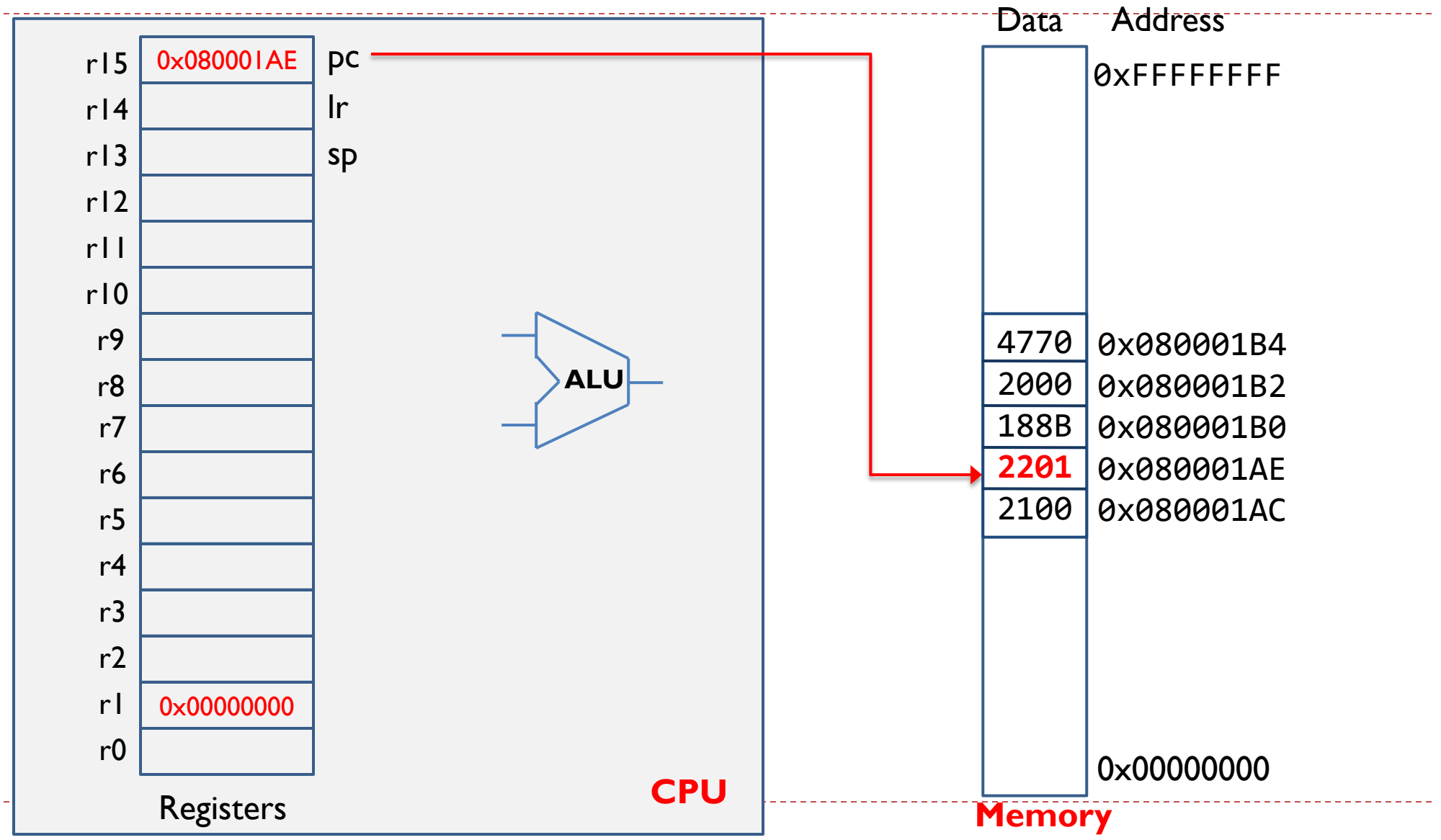
Execute Instruction:

MOVS r1, #0x00

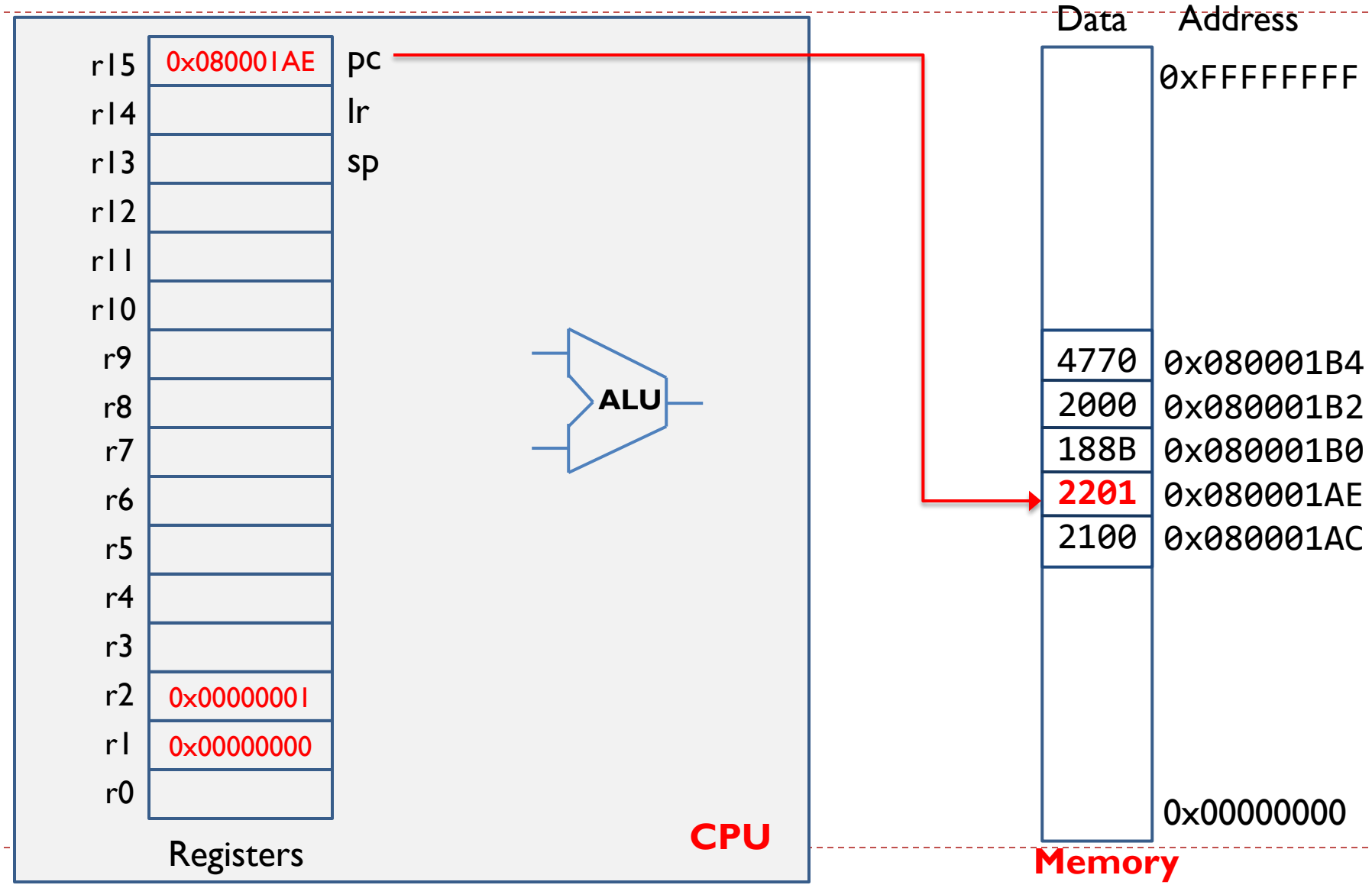


Fetch Next Instruction: pc = pc + 2

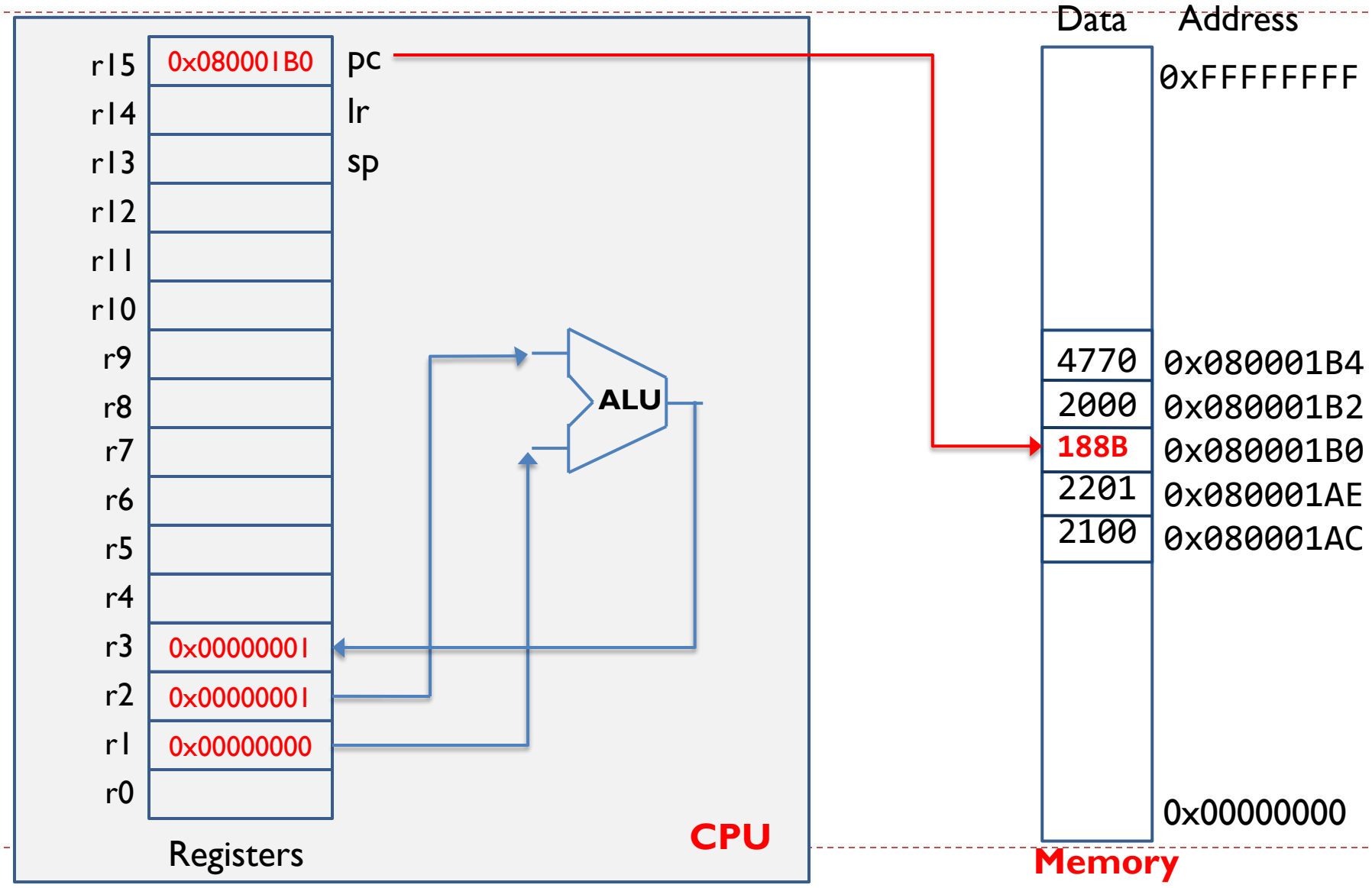
- Thumb-2 consists of a mix of 16- & 32-bit instructions
- In reality, we always fetch 4 bytes from the instruction memory (either one 32-bit instruction or two 16-bit instructions)
- To simplify the demo, we assume we only fetch 2 bytes from the instruction memory in this example.



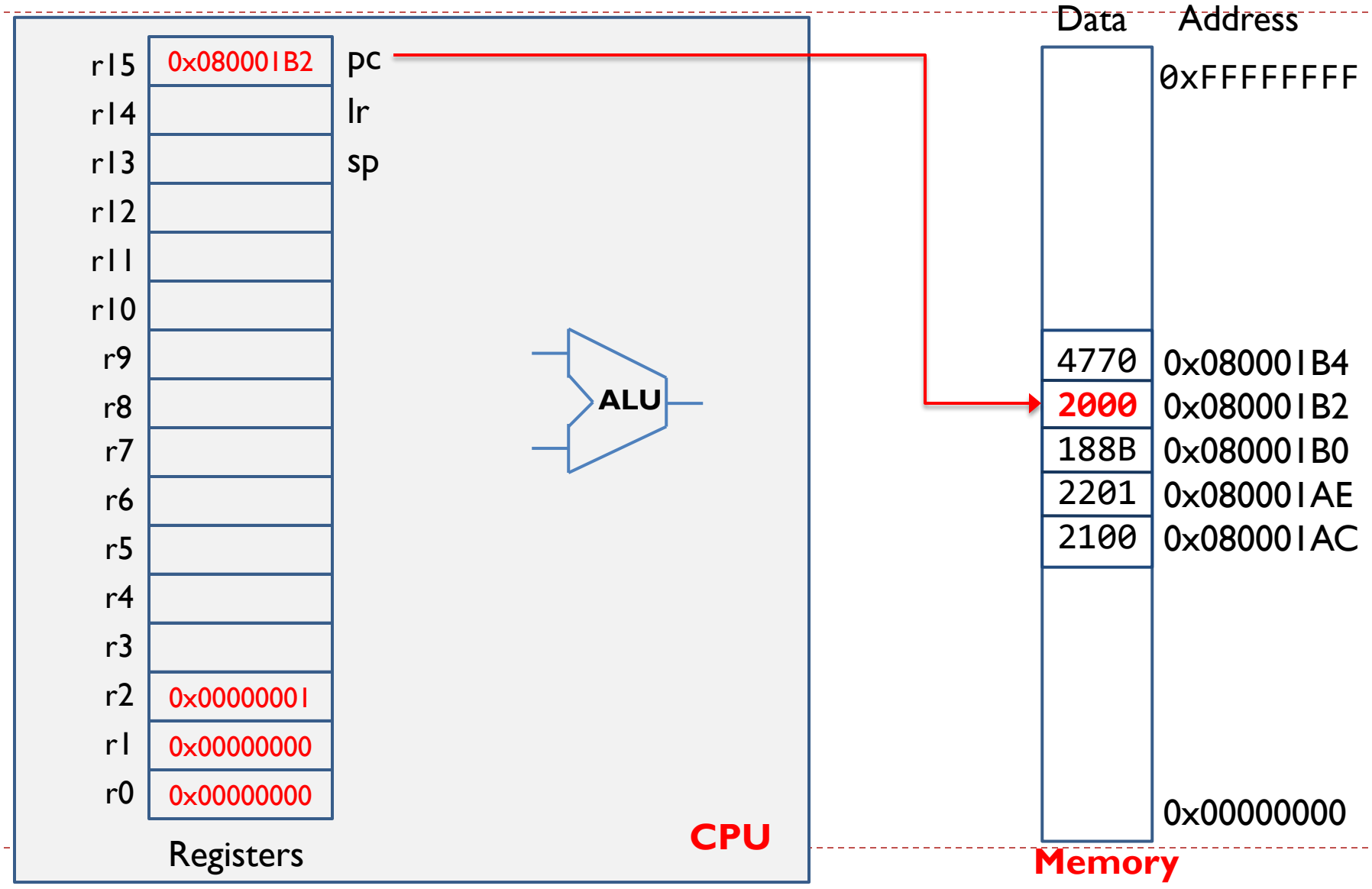
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2201 = MOVS r2, #0x01



Fetch Next Instruction: pc = pc + 2
Decode & Execute: 188B = ADDS r3, r1, r2

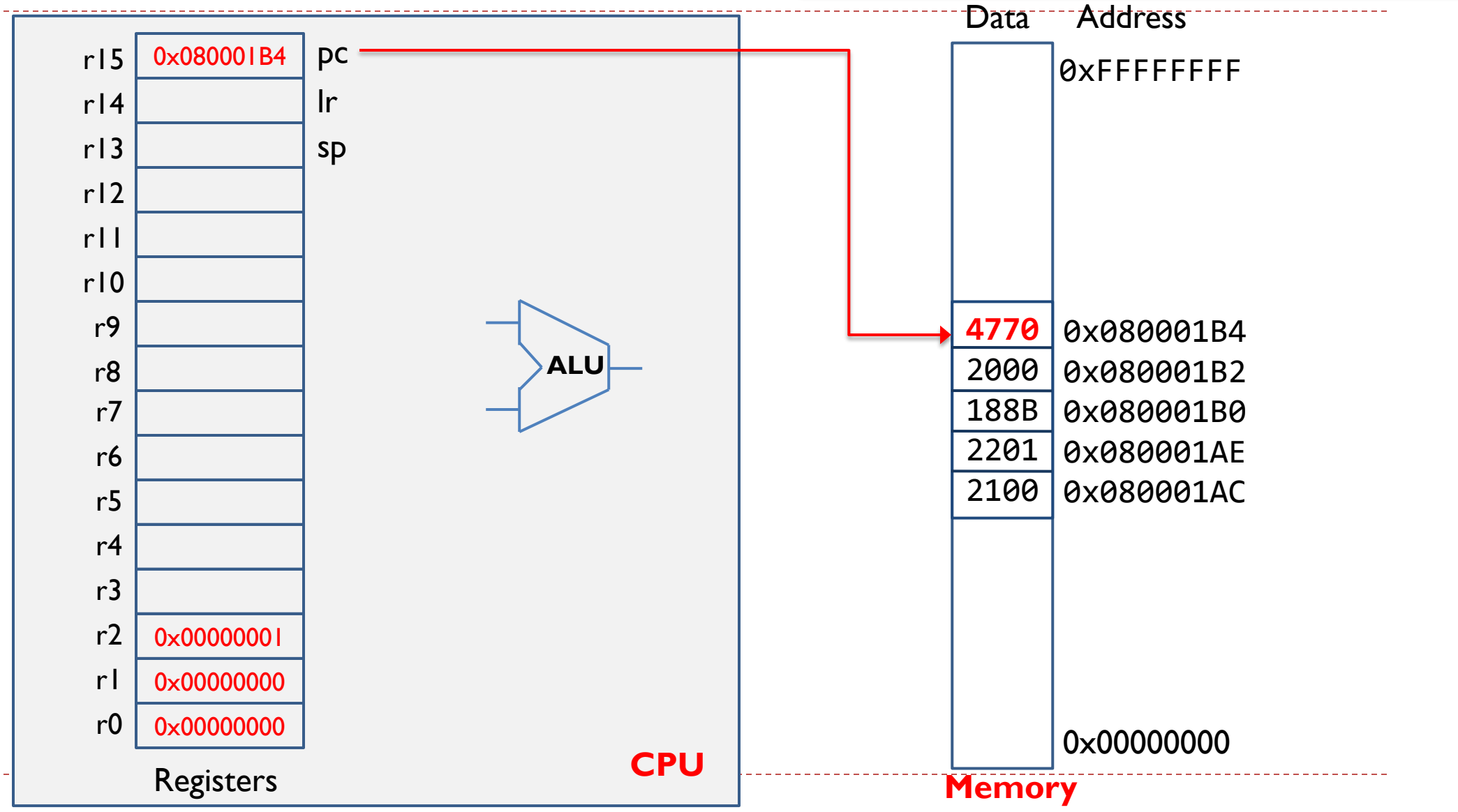


Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2000 = MOVS r0, #0x00



Fetch Next Instruction: pc = pc + 2
Decode & Decode: 4770 = BX lr

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



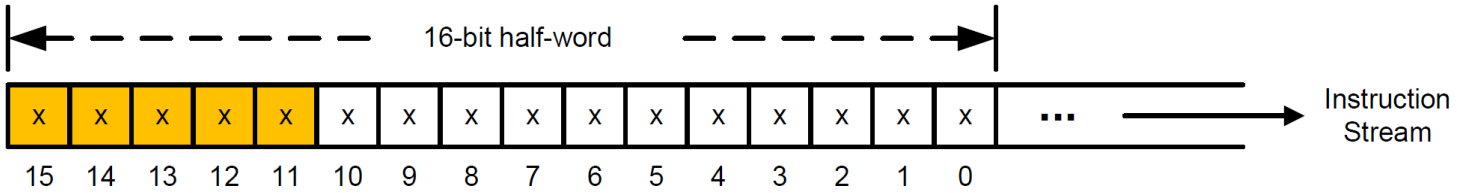
Realities

- 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



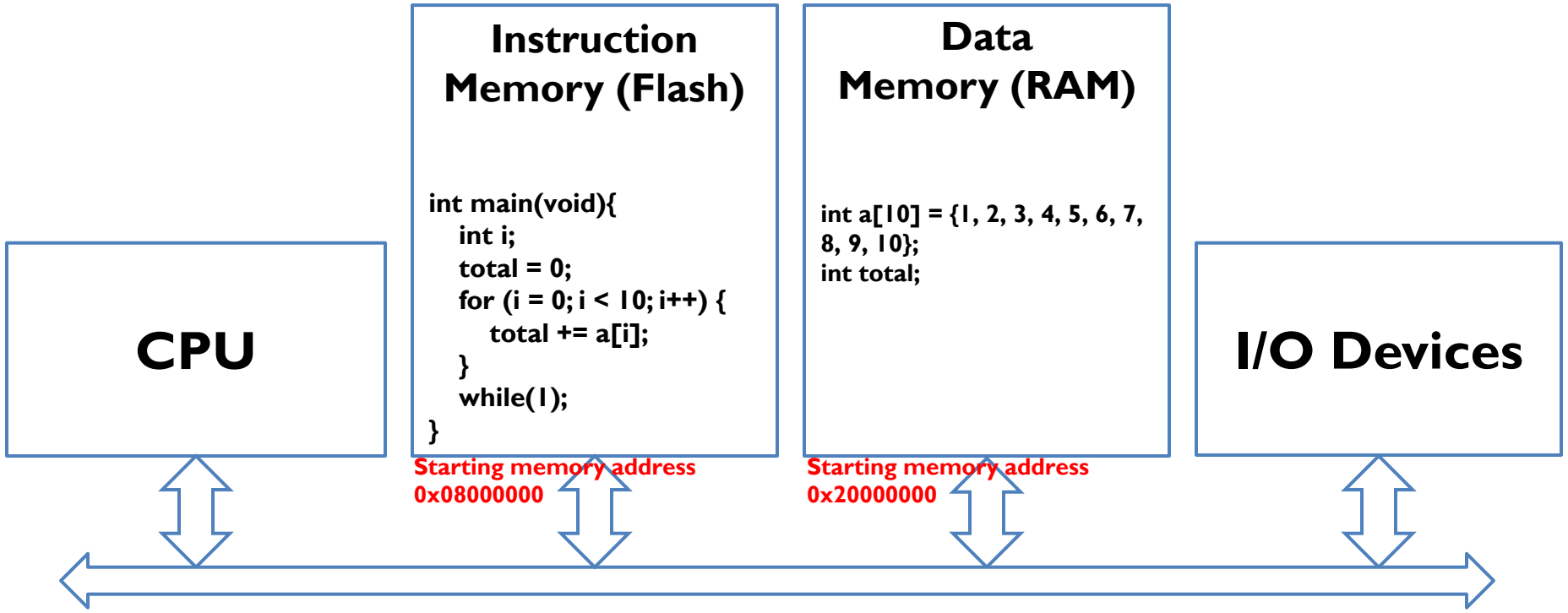
If bit [15-11] = `11101`, `11110`, or `11111`, then, it is the first half-word of a 32-bit instruction.

Otherwise, it is a 16-bit instruction.

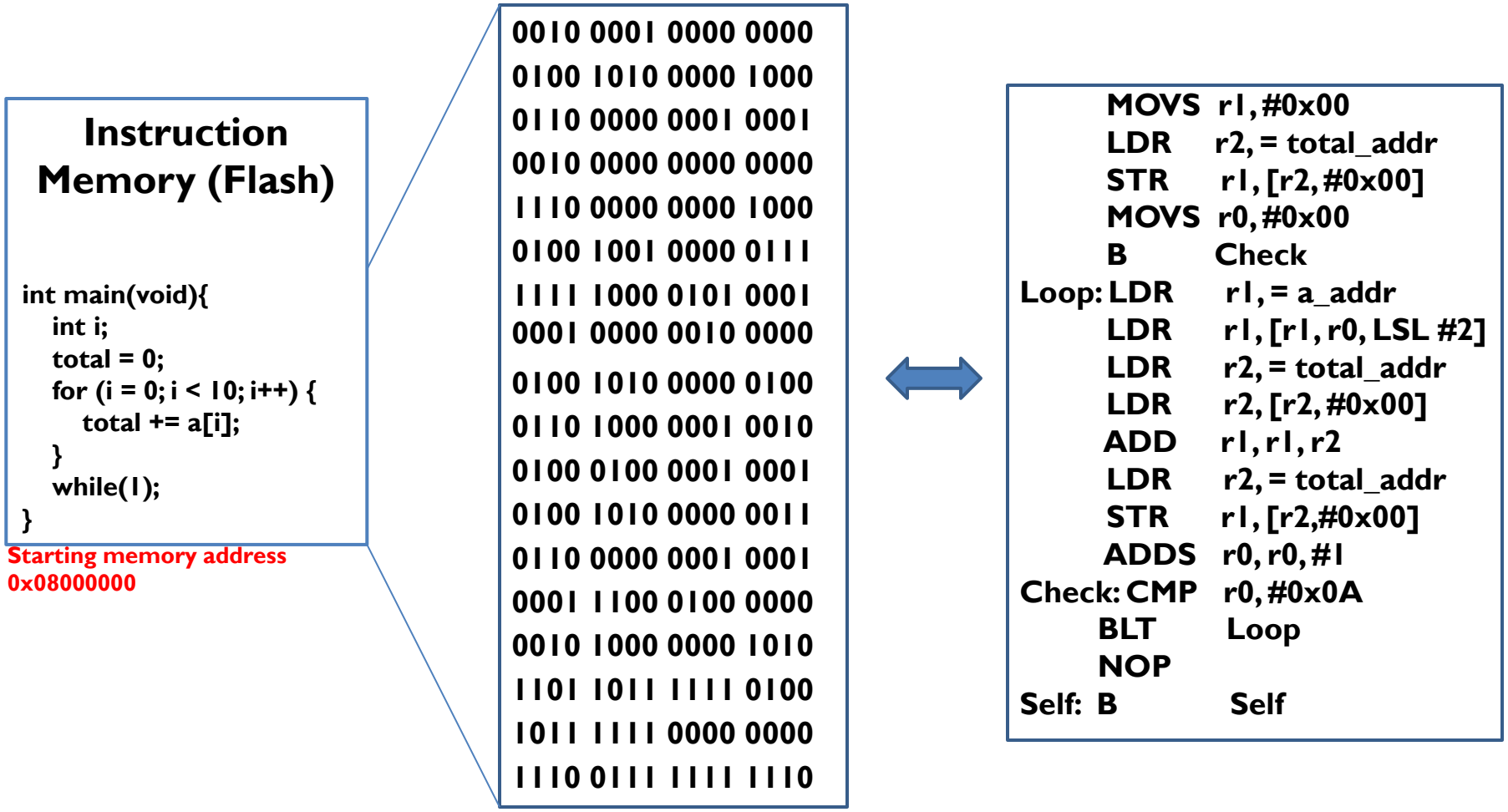
Example:
Calculate the Sum of an Array

```
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



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;
```

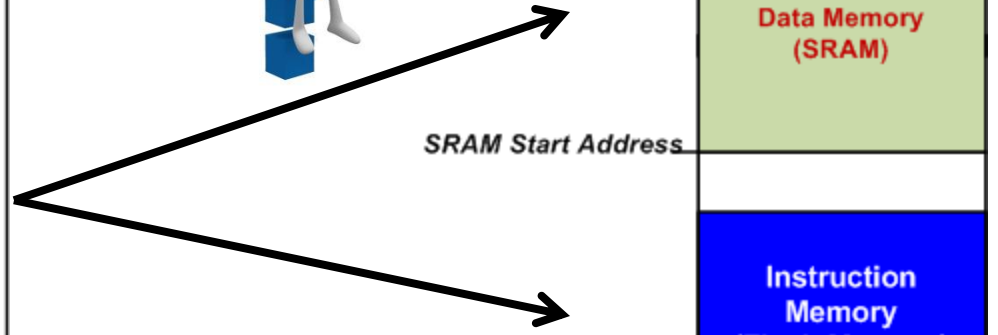
Assume the starting memory address of the data memory is 0x20000000

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

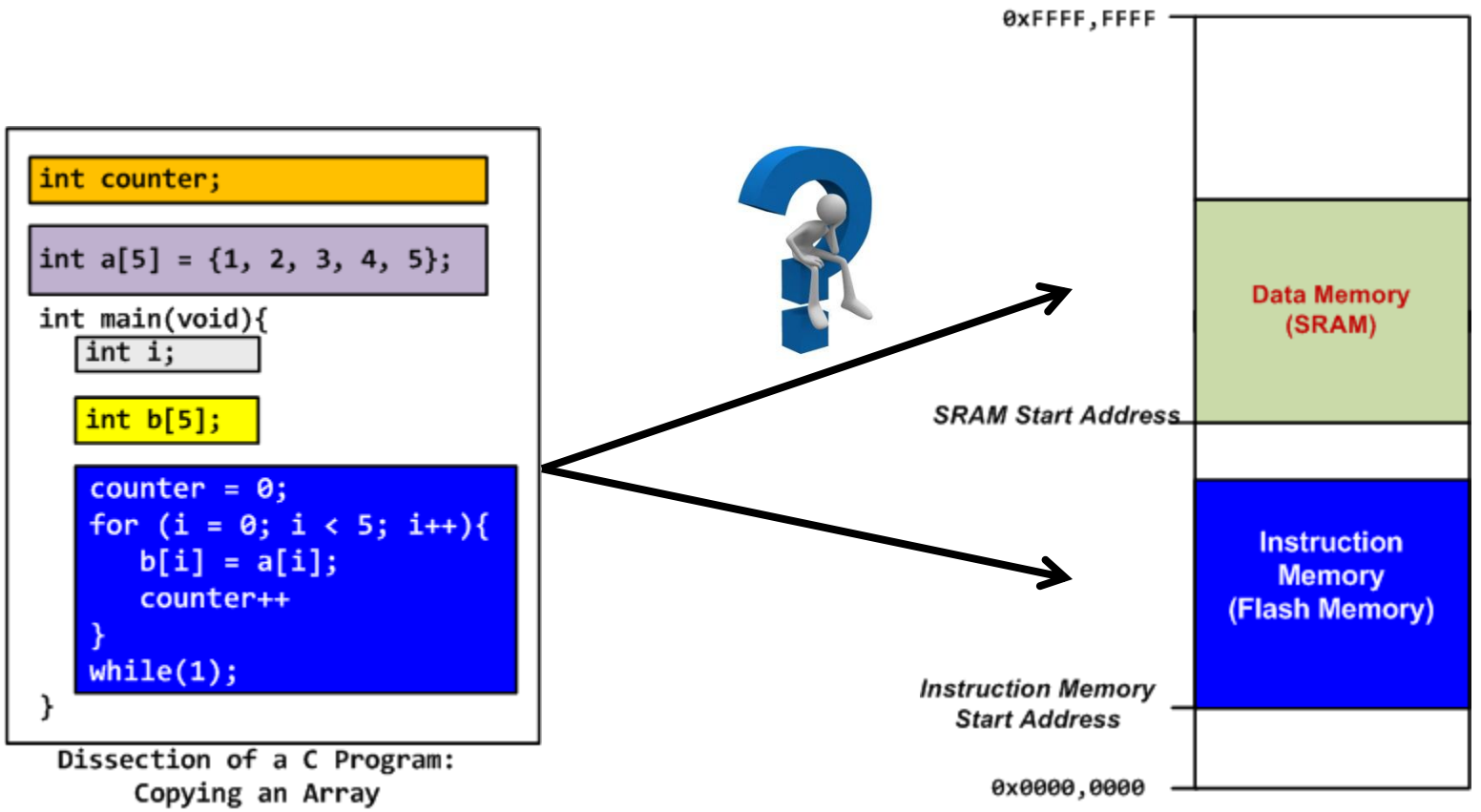
Loading Code and Data into Memory

```
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);
}
```

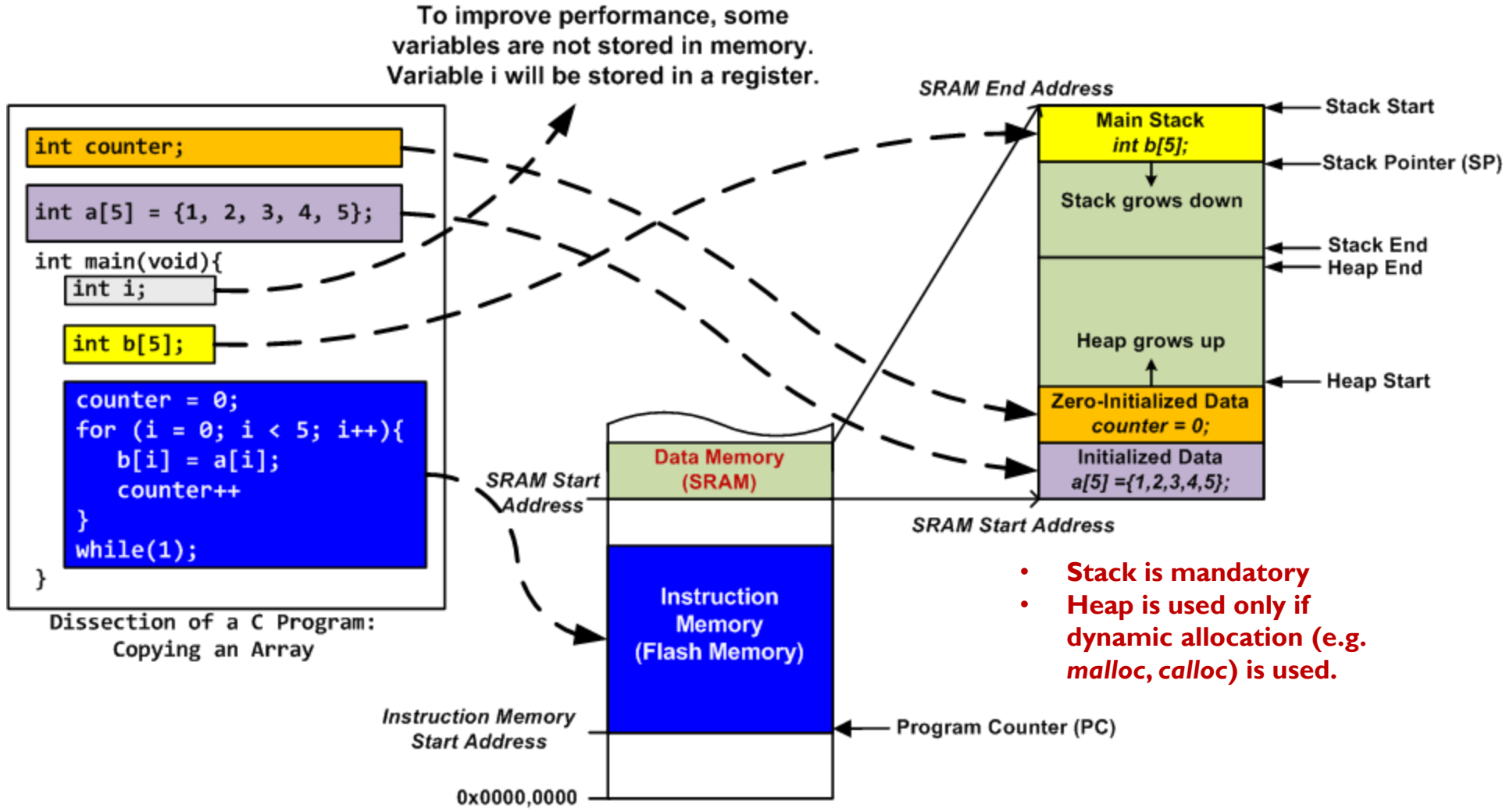
C Program:
Copying an Array



Loading Code and Data into Memory

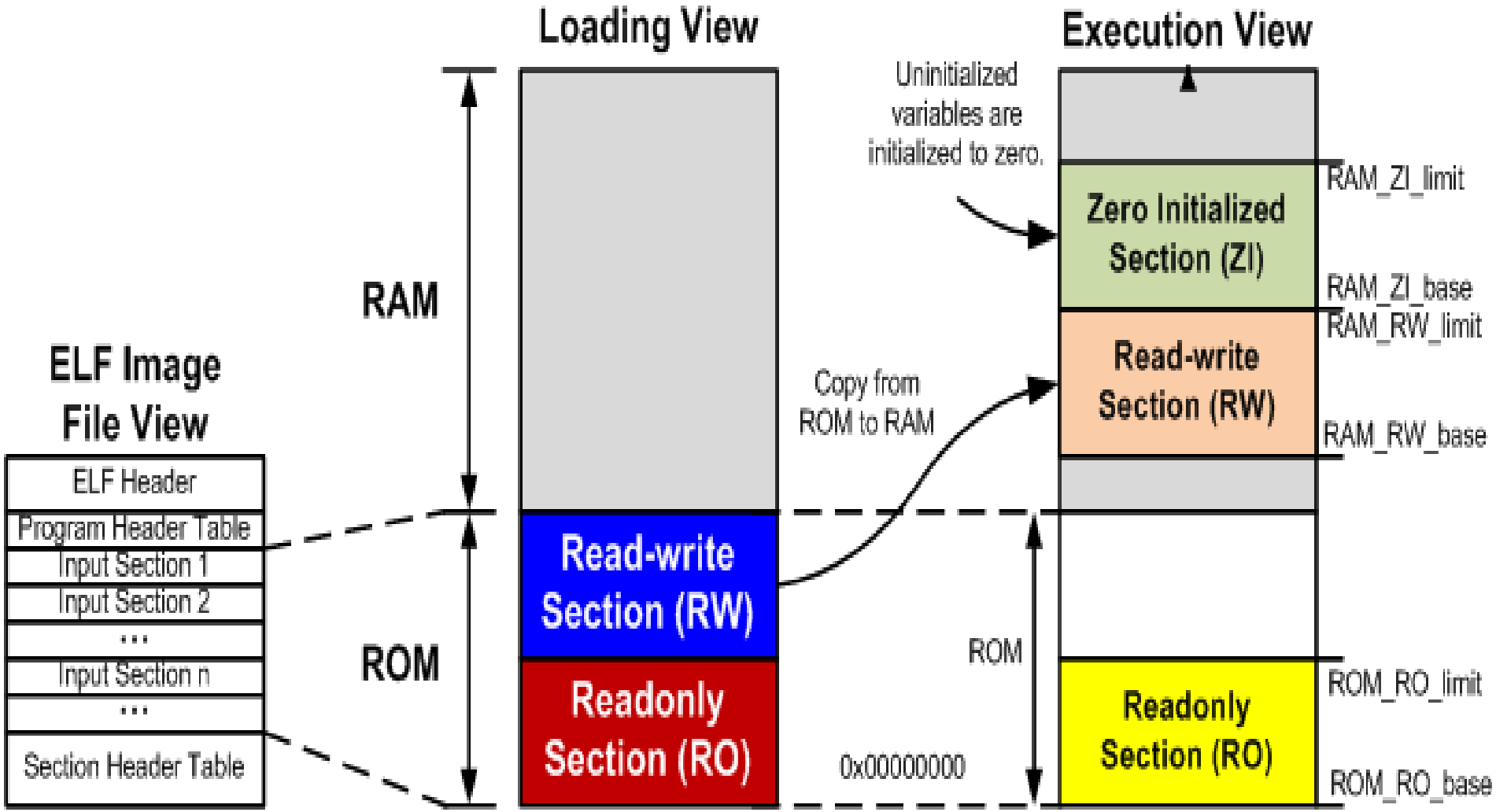


Loading Code and Data into Memory



- **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

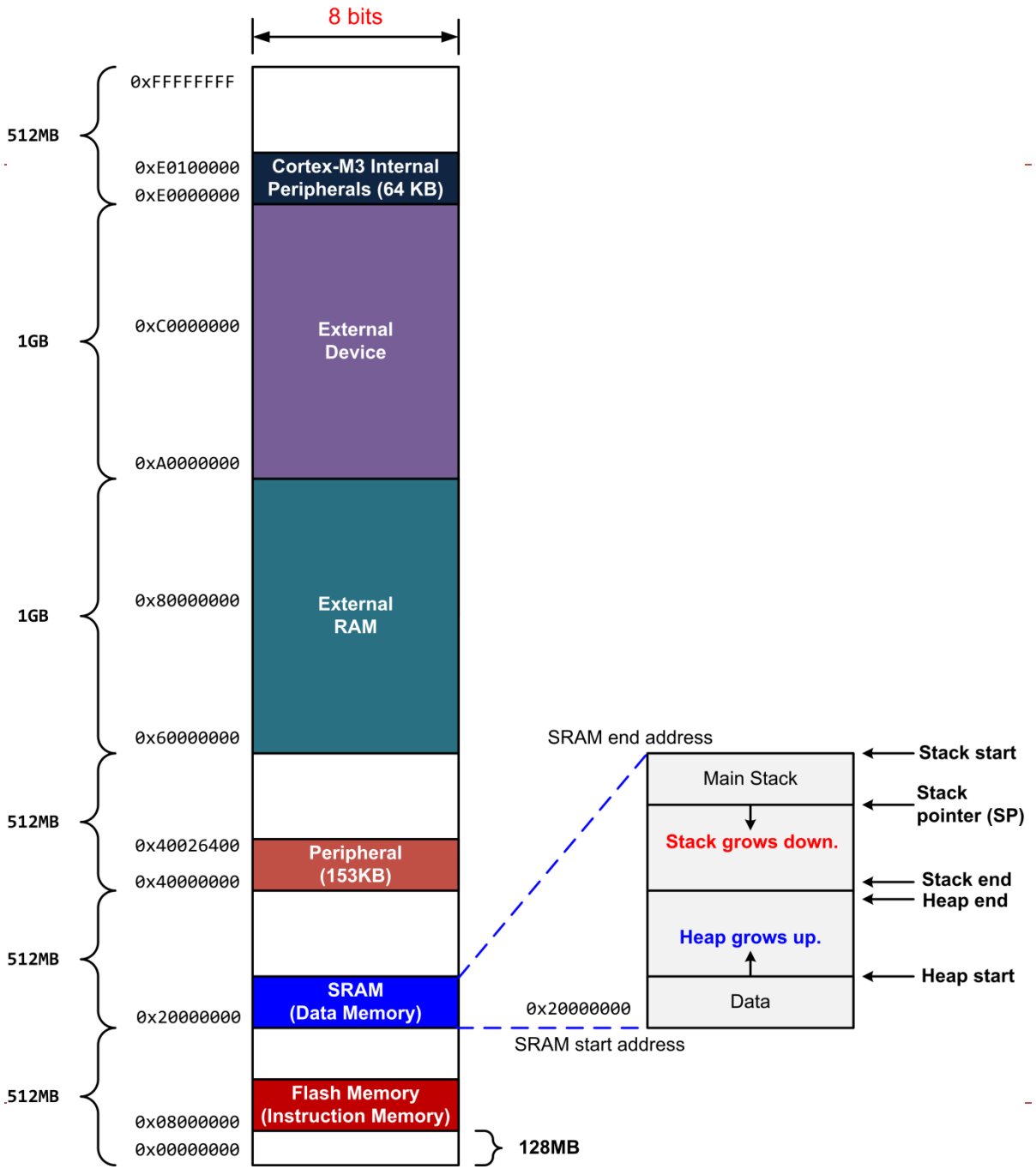
Connectivity
USB OTG,
1x SD/SDIO/MMC, 3 x SPI,
3 x I²C, 1x CAN, 1 x Quad SPI,
5 x USART + 1 x ULP

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
0xFFFFFFFF
512MB
0xE0100000
Cortex-M3 Internal Peripherals (64 KB)
0xE0000000
1GB
0xC0000000
External Device
0xA0000000
1GB
0x80000000
External RAM
0x60000000
512MB
0x40026400
Peripheral (153KB)
0x40000000
512MB
SRAM (Data Memory)
0x20000000
512MB
Flash Memory (Instruction Memory)
0x08000000
0x00000000
128MB
SRAM end address
Main Stack
Stack start
Stack pointer (SP)
Stack grows down.
Stack end
Heap end
Heap grows up.
Heap start
Data
0x20000000
SRAM start address