Chapter 1
Computer and Assembly Language

Dr. Yifeng Zhu
Electrical and Computer Engineering
University of Maine

Spring 2015

Embedded Systems

Amazon Warehouse

Assembly Programs

http://www.andysinger.com/

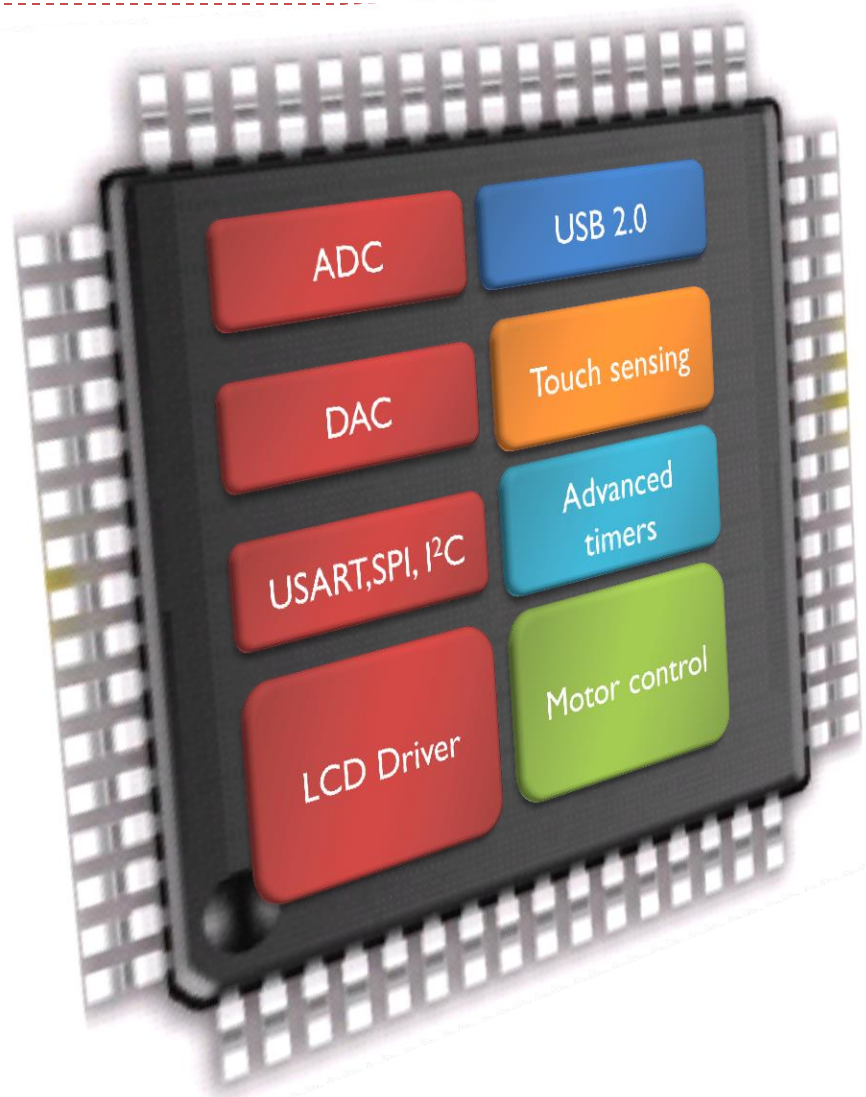
Why do we learn Assembly?

- Assembly isn't "just another language".
  - Help you understand how does the processor work
- Assembly program runs faster than high-level language. Performance critical codes must be written in assembly.
  - Use the profiling tools to find the performance bottle and rewrite that code section in assembly
  - Latency-sensitive applications, such as aircraft controller
  - Standard C compilers do not use some operations available on ARM processors, such ROR (Rotate Right) and RRX (Rotate Right Extended).
- Hardware/processor specific code,
  - Processor booting code
  - Device drivers
  - A test-and-set atomic assembly instruction can be used to implement locks and semaphores.
- Cost-sensitive applications
  - Embedded devices, where the size of code is limited, wash machine controller, automobile controllers
- The best applications are written by those who've mastered assembly language or fully understand the low-level implementation of the high-level language statements they're choosing.

Why ARM processor

- As of 2005, **98%** of the more than one billion mobile phones sold each year used ARM processors

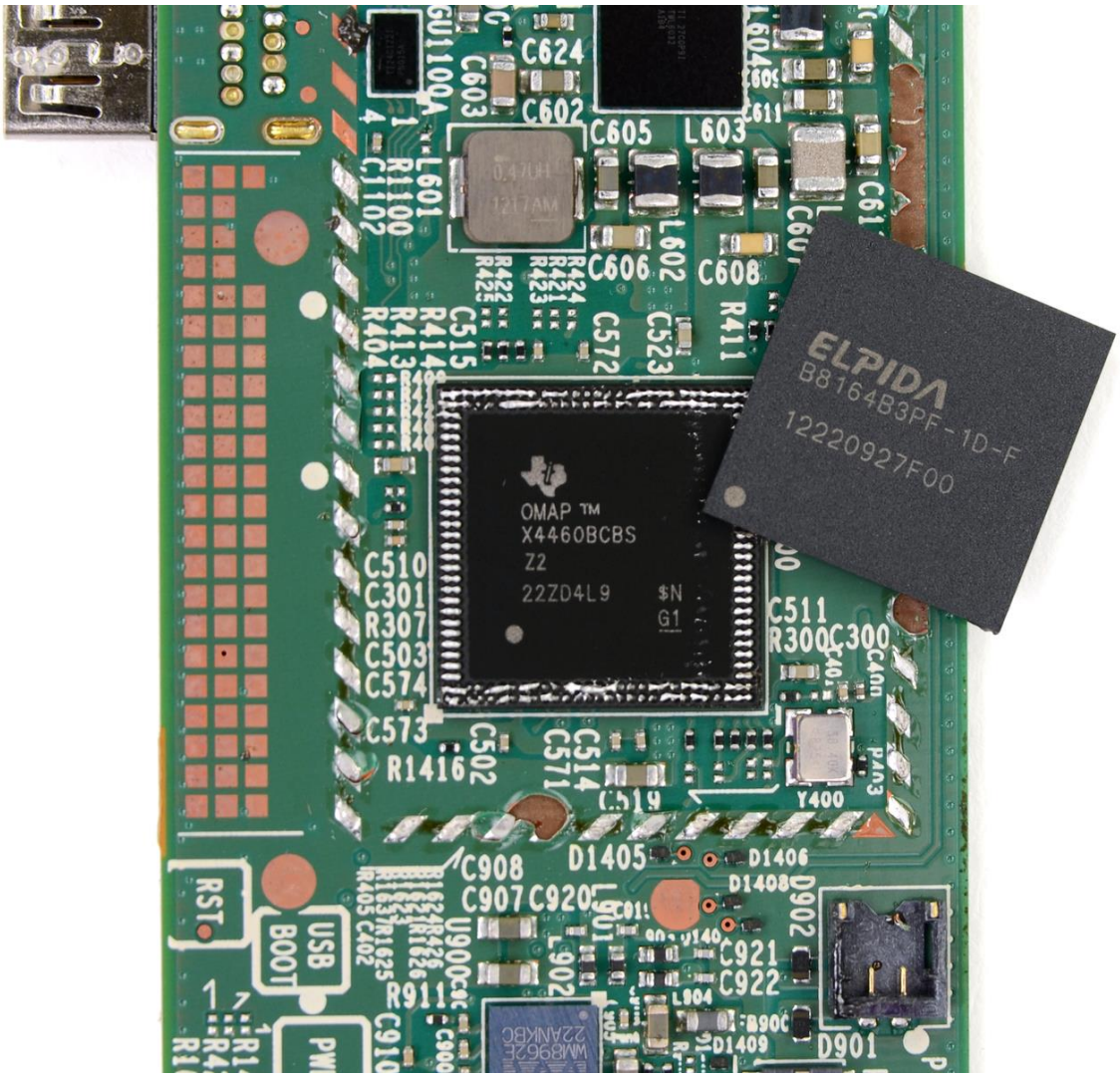
- As of 2009, ARM processors accounted for approximately **90%** of all embedded 32-bit RISC processors



iPhone 5 Teardown

The A6 processor is the first Apple System-on-Chip (SoC) to use a custom design, based off the **ARMv7** instruction set.

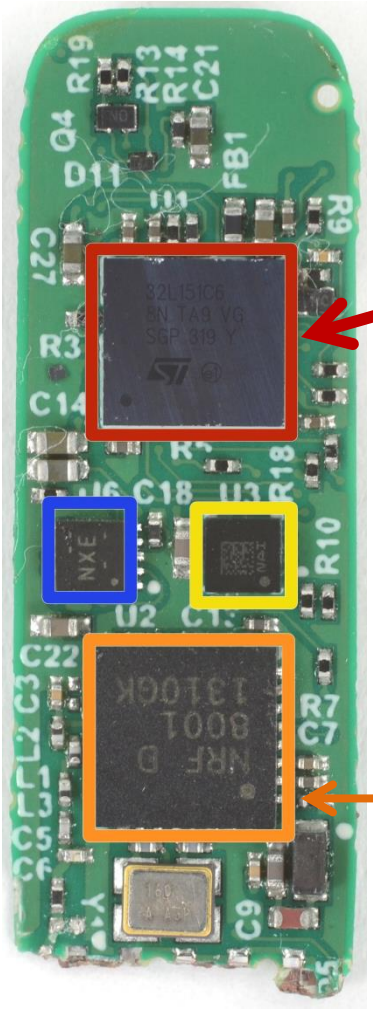
Kindle HD Fire



Texas Instruments **OMAP 4460** dual-core processor

http://www.ifixit.com

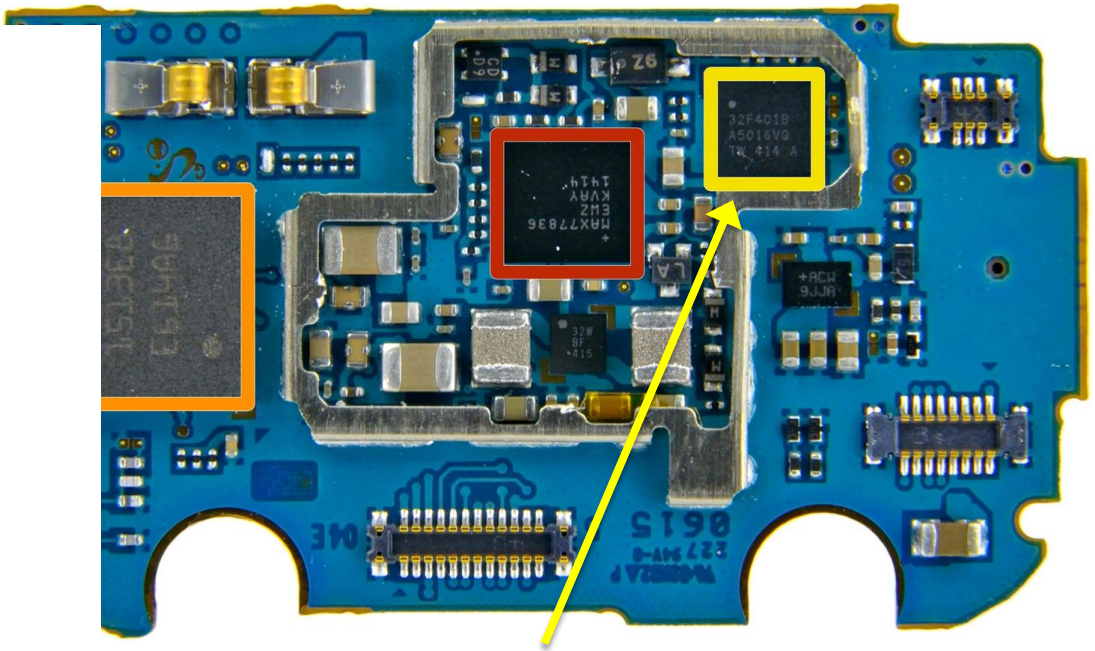
Fitbit Flex Teardown



STMicroelectronics 32L151C6 Ultra Low Power ARM Cortex M3 Microcontroller

Nordic Semiconductor nRF8001 Bluetooth Low Energy Connectivity IC

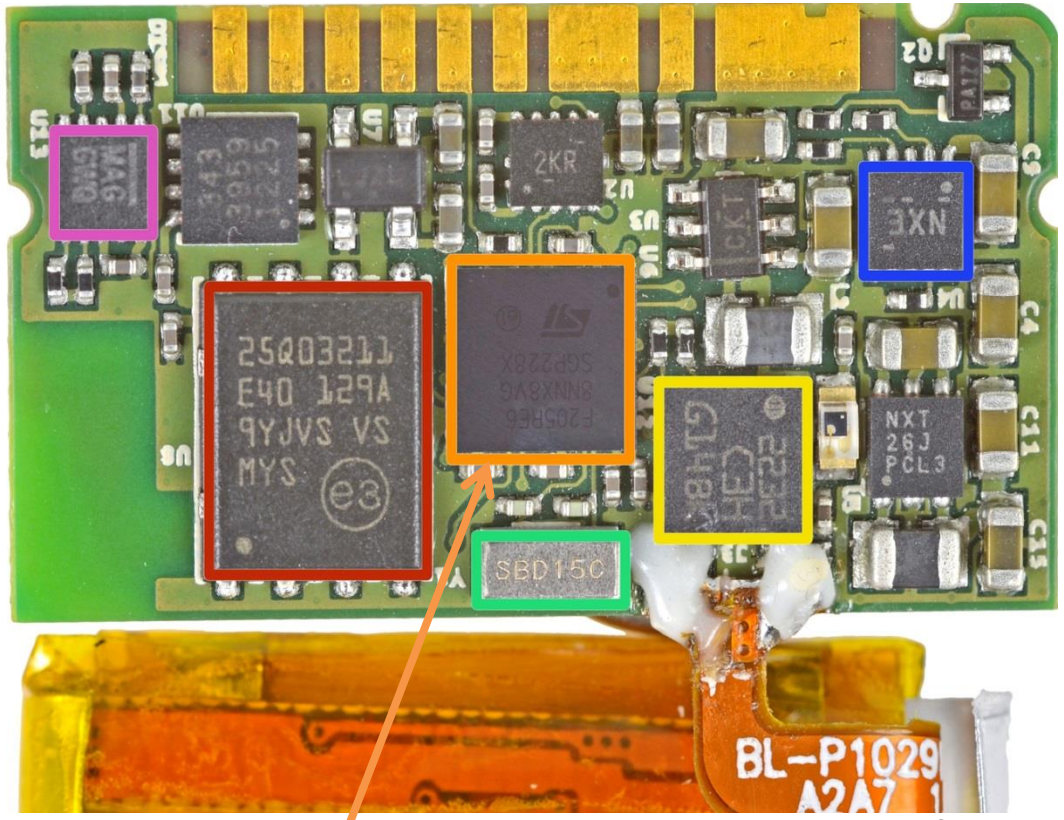
Samsung Galaxy Gear



- STMicroelectronics STM32F401B **ARM-Cortex M4** MCU with 128KB Flash

source: ifixit.com

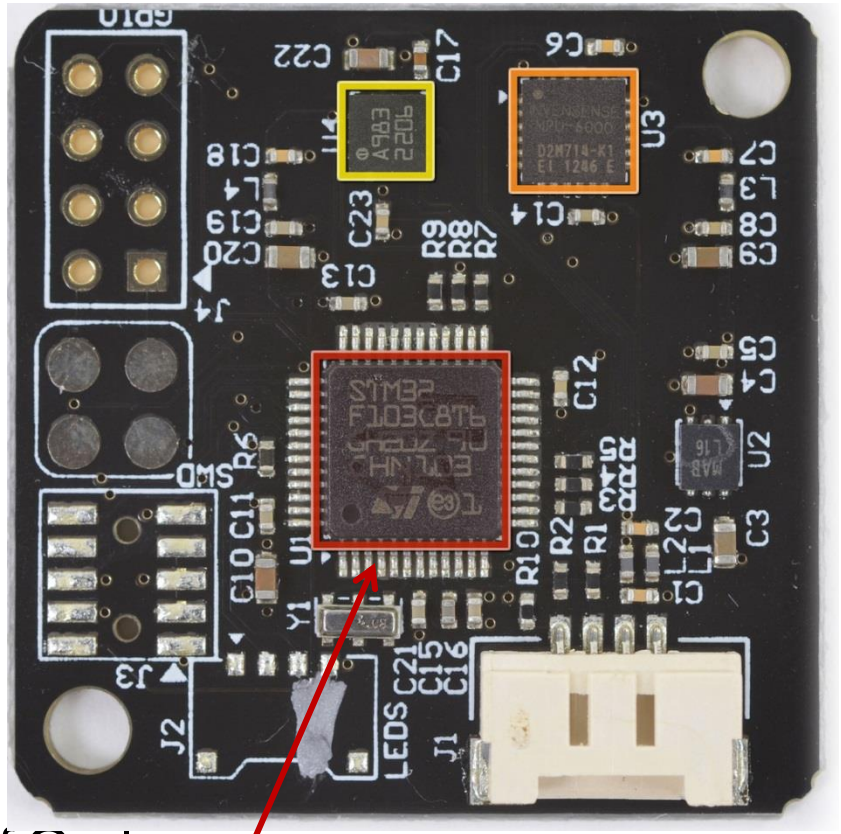
Pebble Smartwatch



source: ifixit.com

- STMicroelectronics STM32F205RE **ARM Cortex-M3** MCU, with a maximum speed of 120 MHz

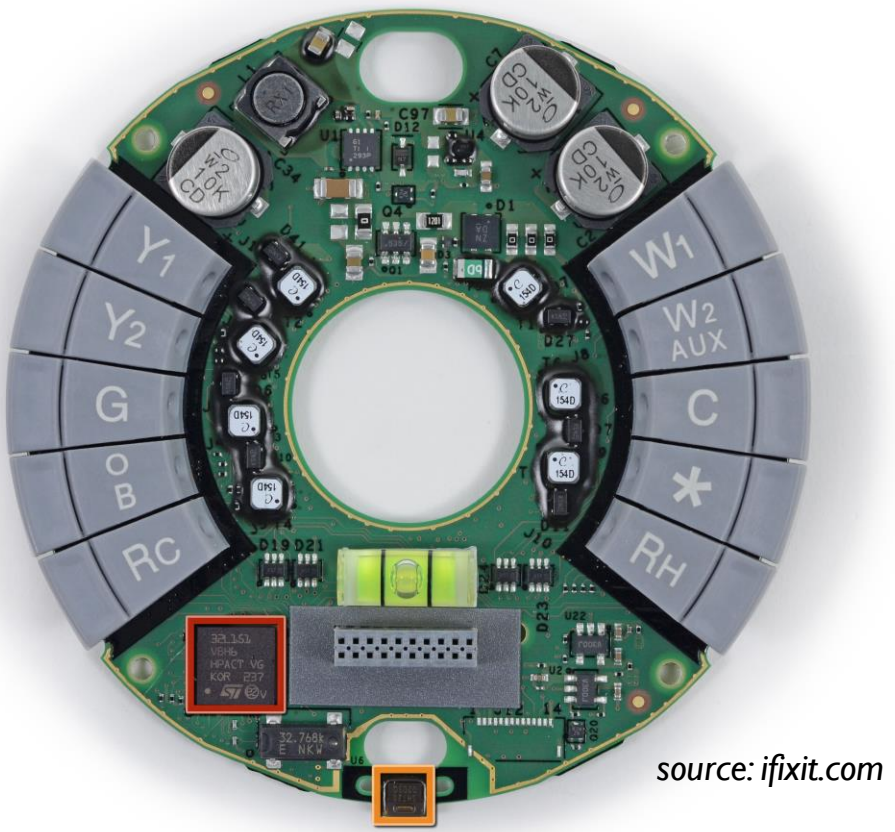
Oculus VR



source: *ifixit.com*

- Facebook's $2 Billion Acquisition Of Oculus
- STMicroelectronics **STM32L**100RB Ultra-low-power 32-bit Value Line **ARM Cortex-M3** MCU with 128 Kbytes Flash, 32 MHz CPU, LCD, USB

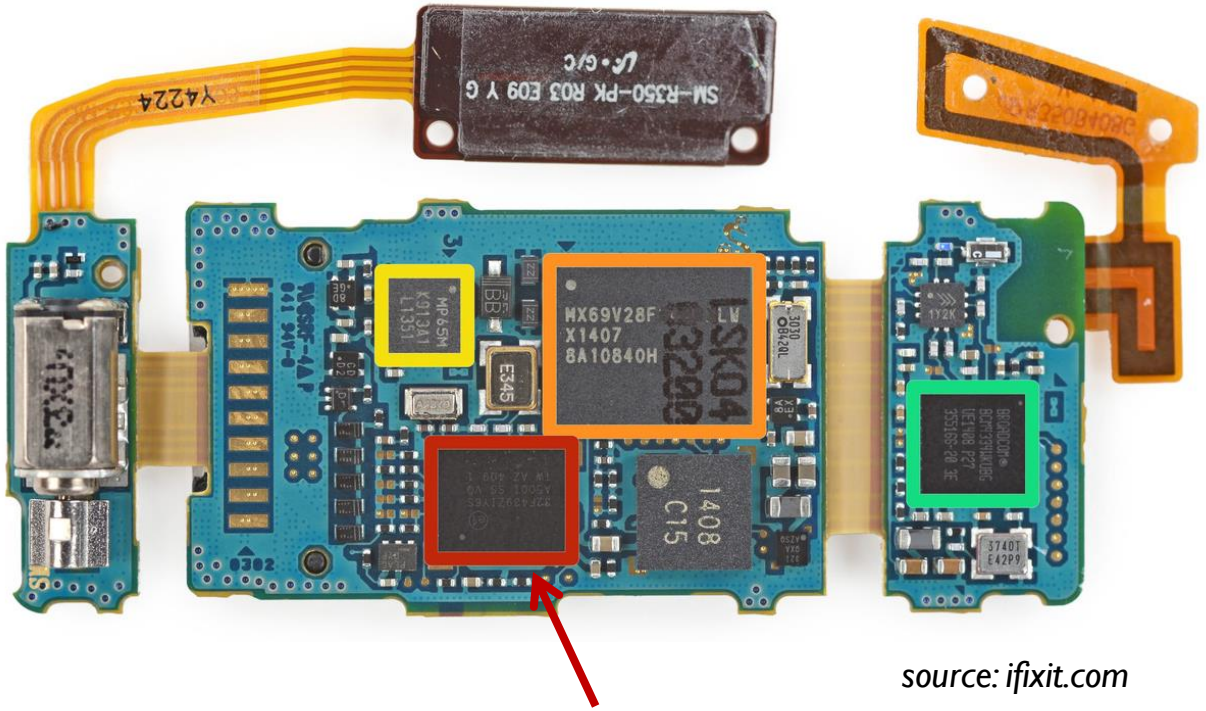
Nest Learning Thermostat



source: *ifixit.com*

- ST Microelectronics **STM32L**151VB ultra-low-power 32 MHz ARM **Cortex-M3** MCU

Samsung Gear Fit Fitness Tracker

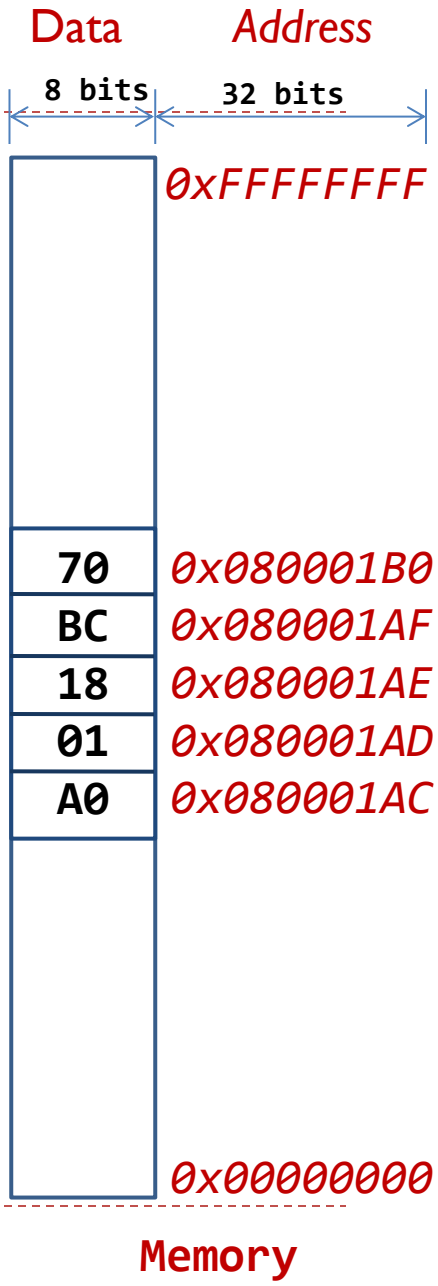


source: *ifixit.com*

- STMicroelectronics **STM32F4**39ZI 180 MHz, 32 bit **ARM Cortex-M4** CPU

Memory

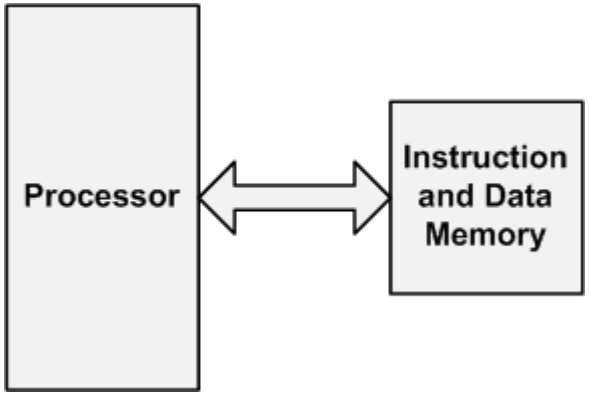
- Memory is arranged as a series of "locations"
  - Each location has a unique "address"
  - Each location holds a byte (***byte-addressable***)
  - e.g. the memory location at address `0x080001B0` contains the byte value `0x70`, i.e., 112
- The number of locations in memory is limited
  - e.g. 4 GB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

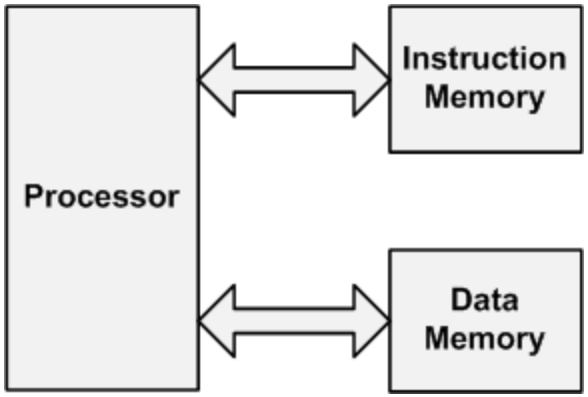
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

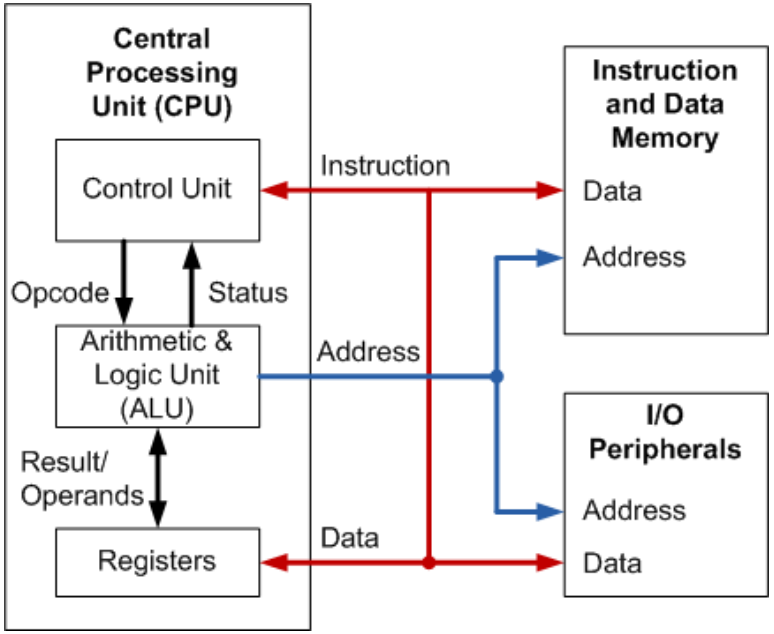
Data and instructions are stored into separate memories.



Computer Architecture

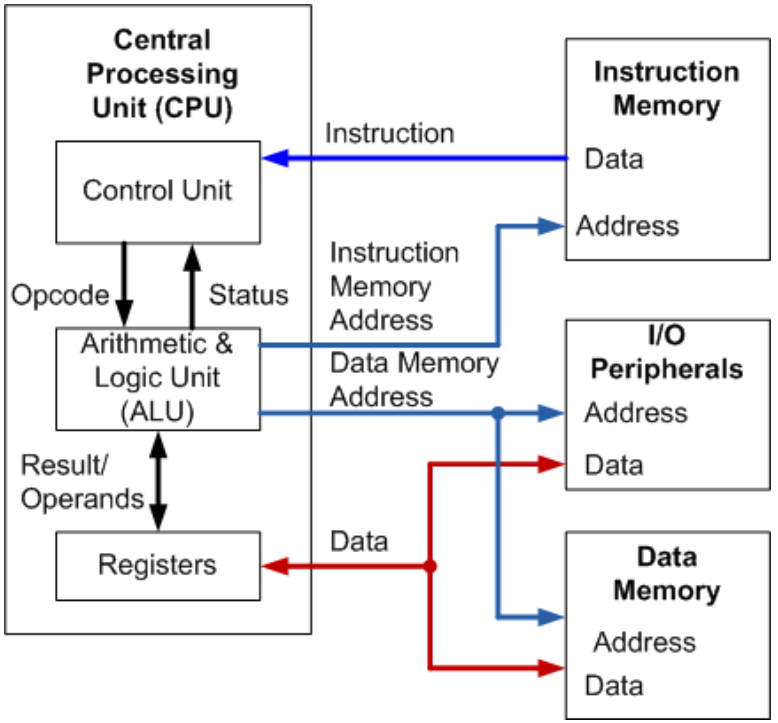
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

Data and instructions are stored into separate memories.



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
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

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

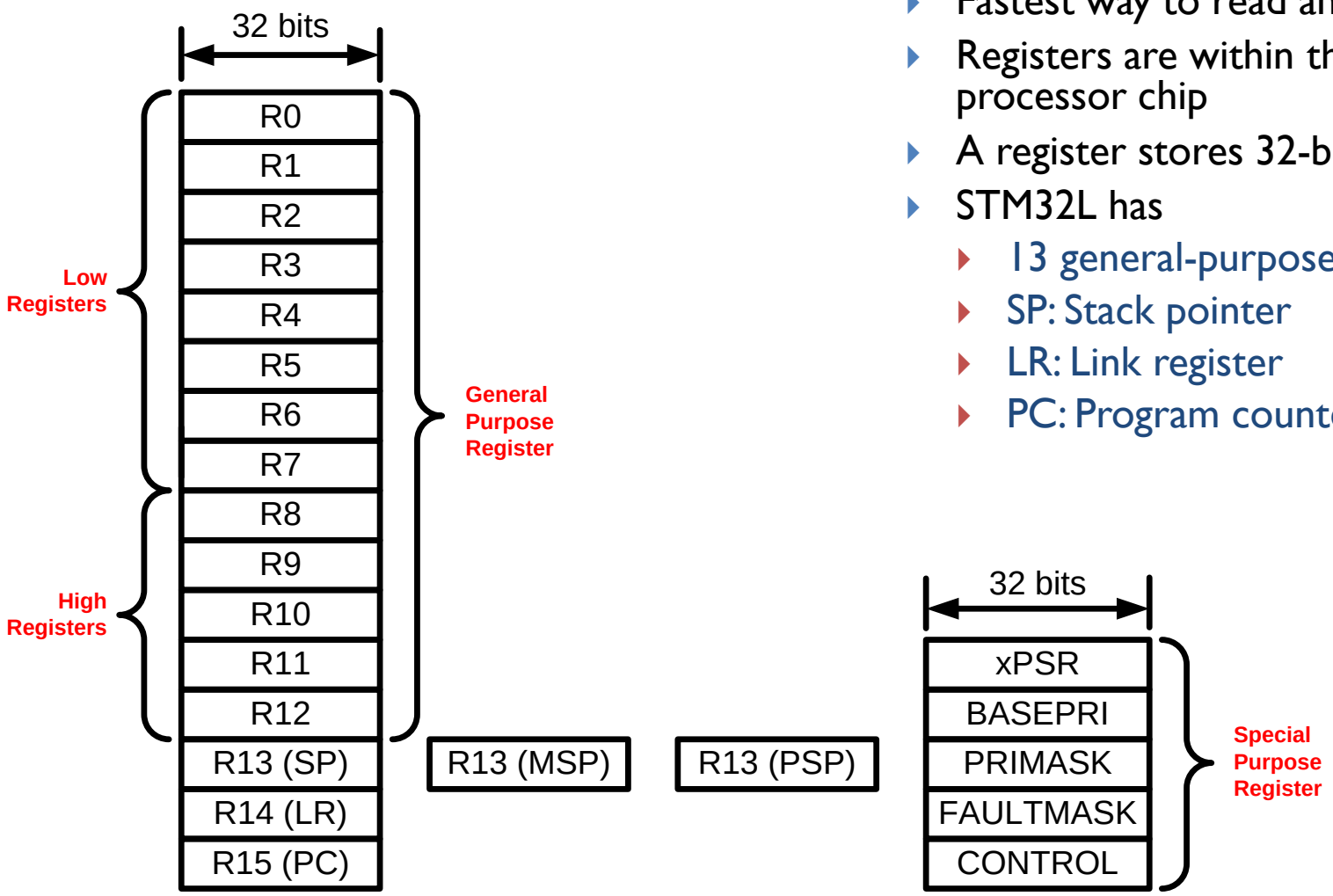
```
MOVS r1, #0x00    ; int a = 0
MOVS r2, #0x01    ; int b = 1
ADDS r3, r1, r2   ; c = a + b
MOVS r0, 0x00     ; set return value
BX lr             ; return
```

assembler

Machine Code

In Binary	In Hex	
0010000100000000	2100	; MOVS r1, #0x00
0010001000000001	2201	; MOVS r2, #0x01
0001100010001011	188B	;ADDS r3, r1, r2
0010000000000000	2000	; MOVS r0, #0x00
0100011101110000	4770	; BX lr

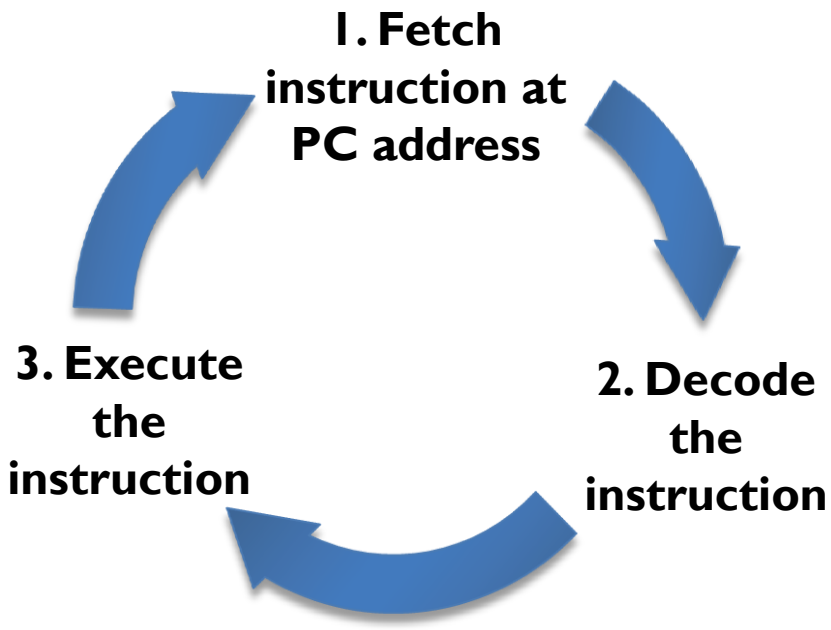
Processor Registers

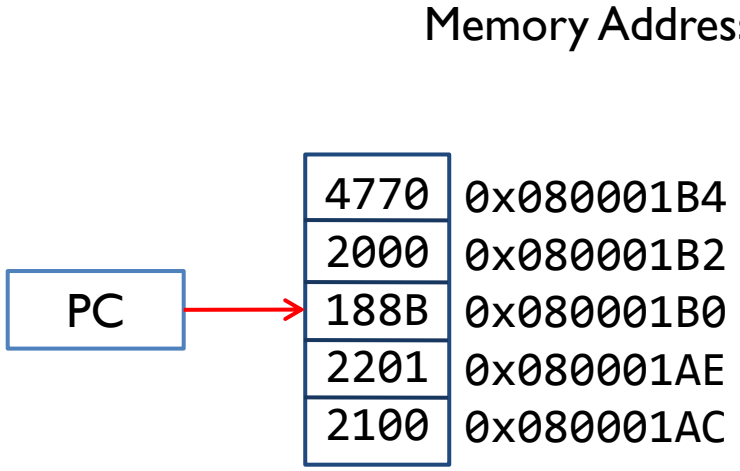


- Fastest way to read and write
- Registers are within the processor chip
- A register stores 32-bit value
- STM32L has
  - 13 general-purpose registers
  - SP: Stack pointer
  - LR: Link register
  - PC: Program counter

Program Execution

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

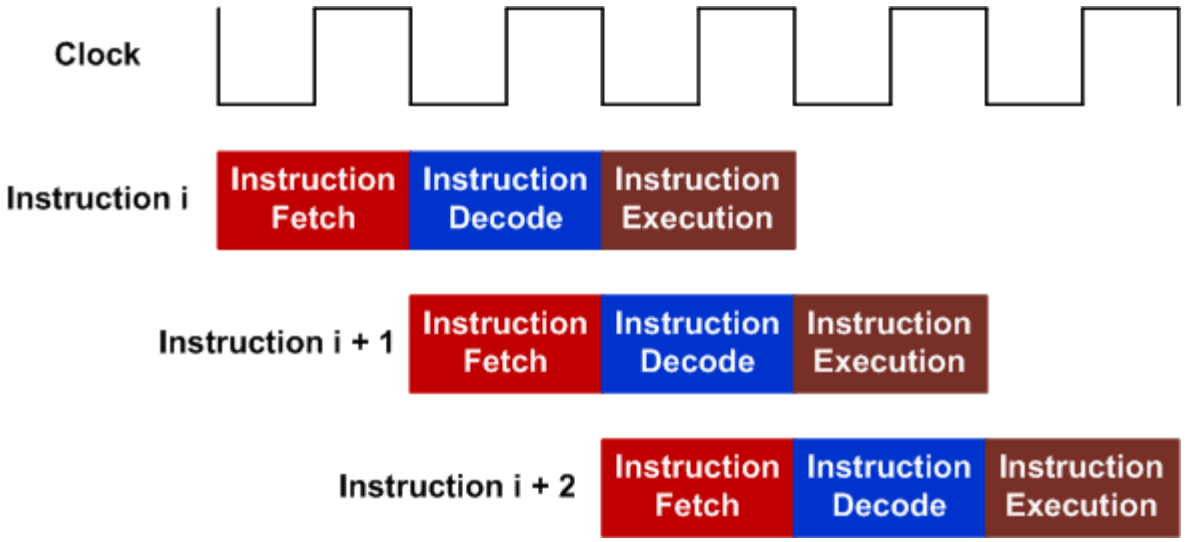




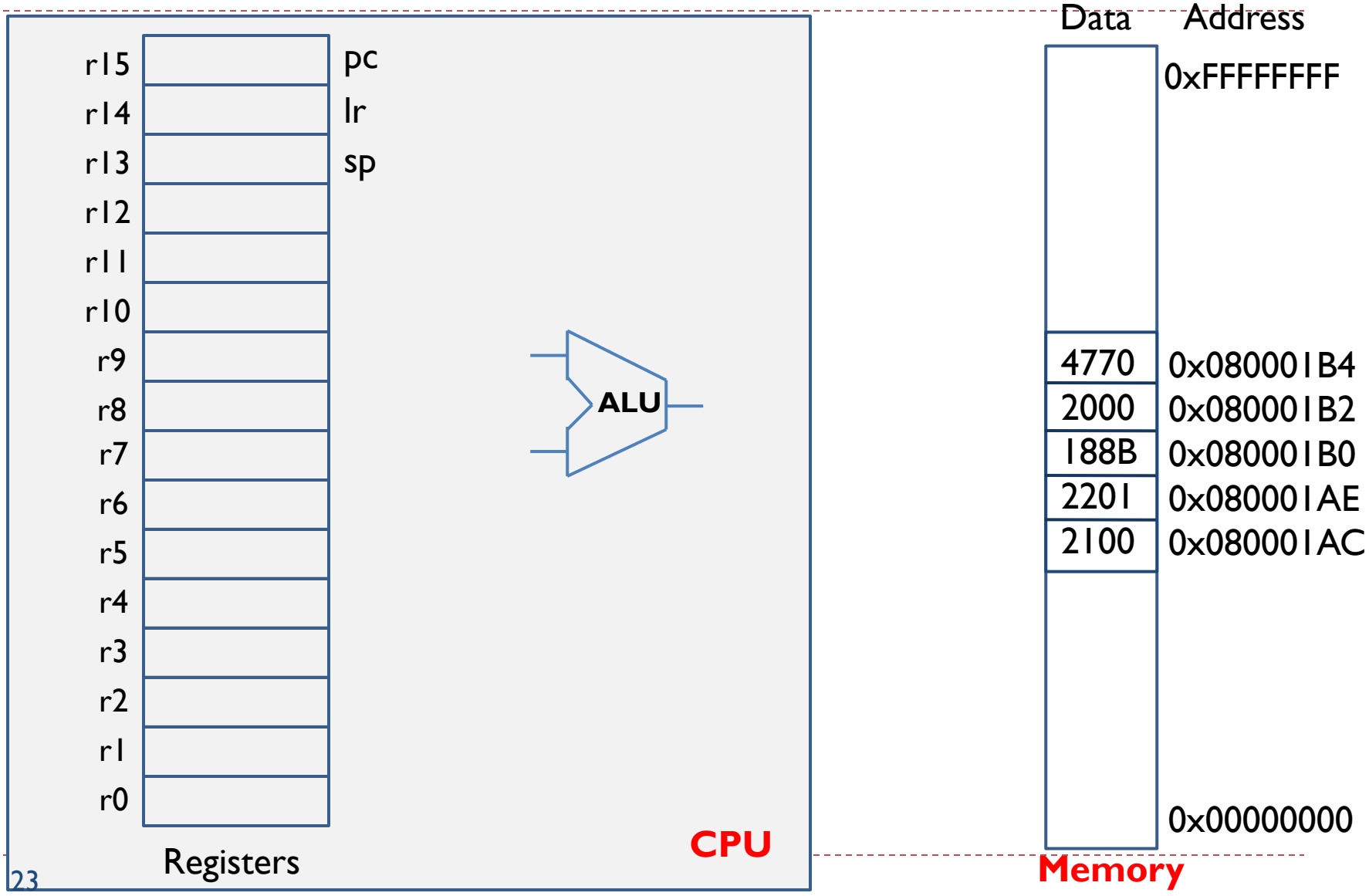
PC = 0x080001B0
Instruction = 188B or
2000188B or 8B180020

Three-state pipeline: Fetch, Decode, Execution

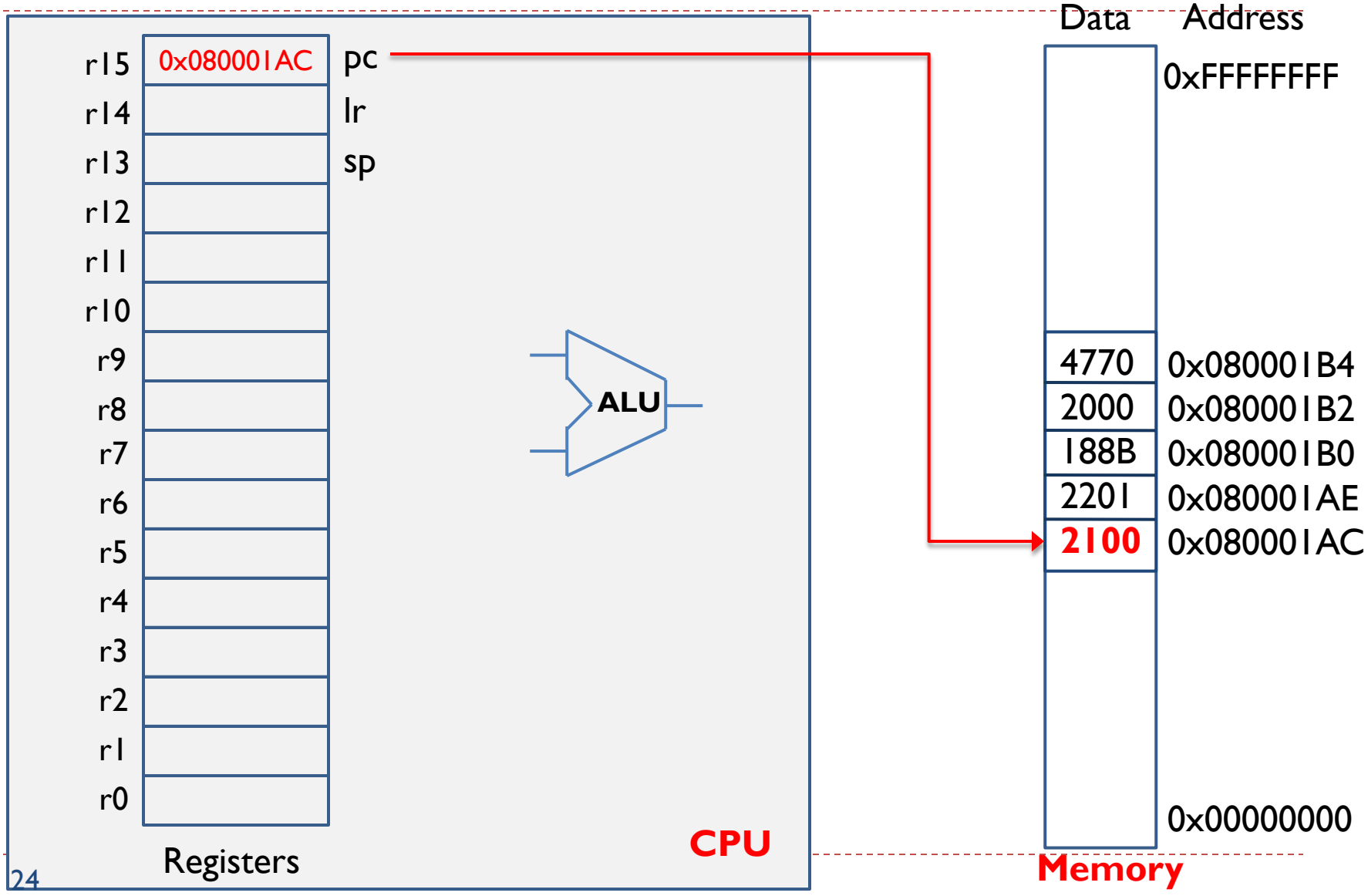
- **Pipelining** allows hardware resources to be fully utilized.



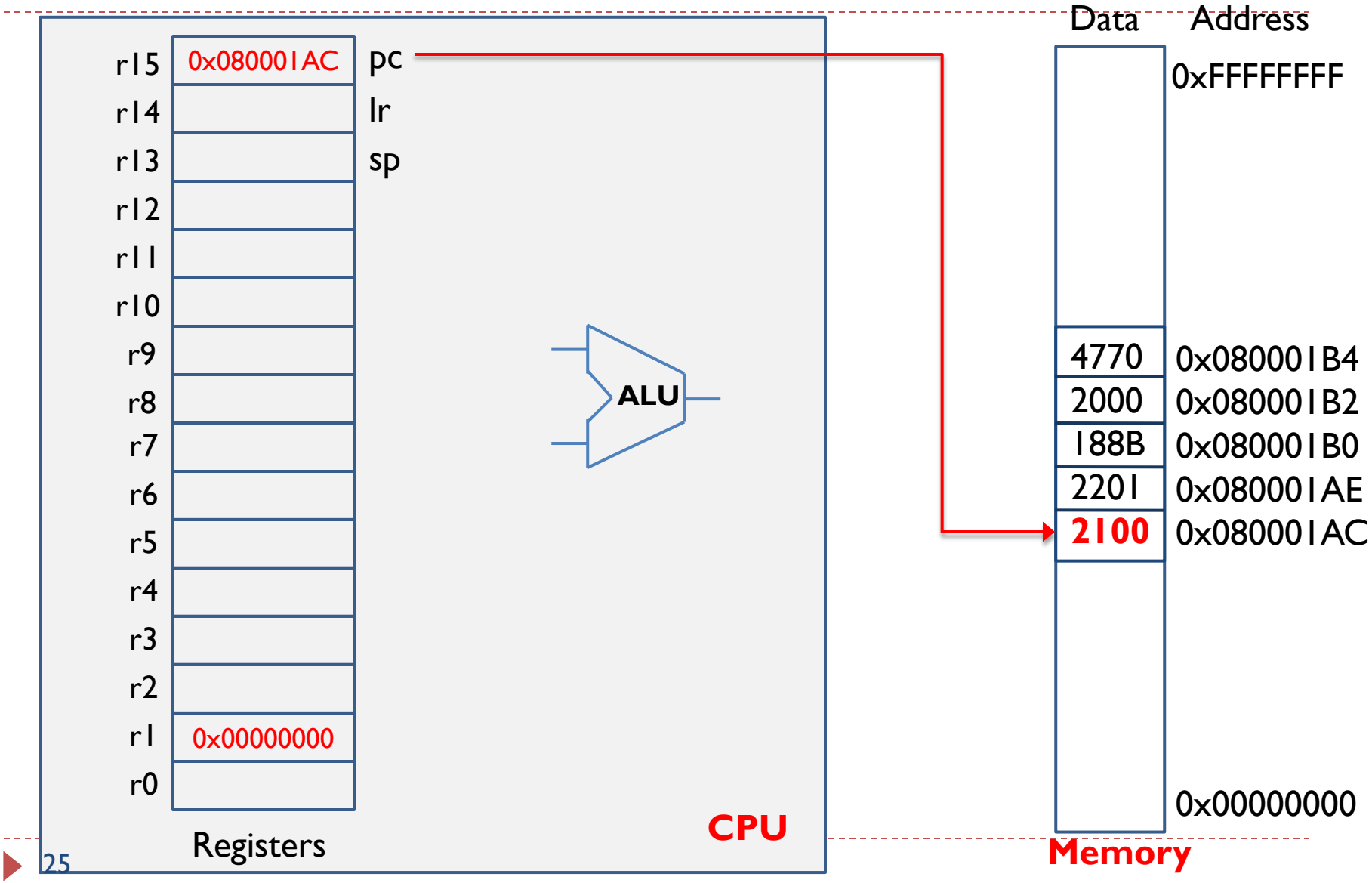
Machine codes are stored in memory



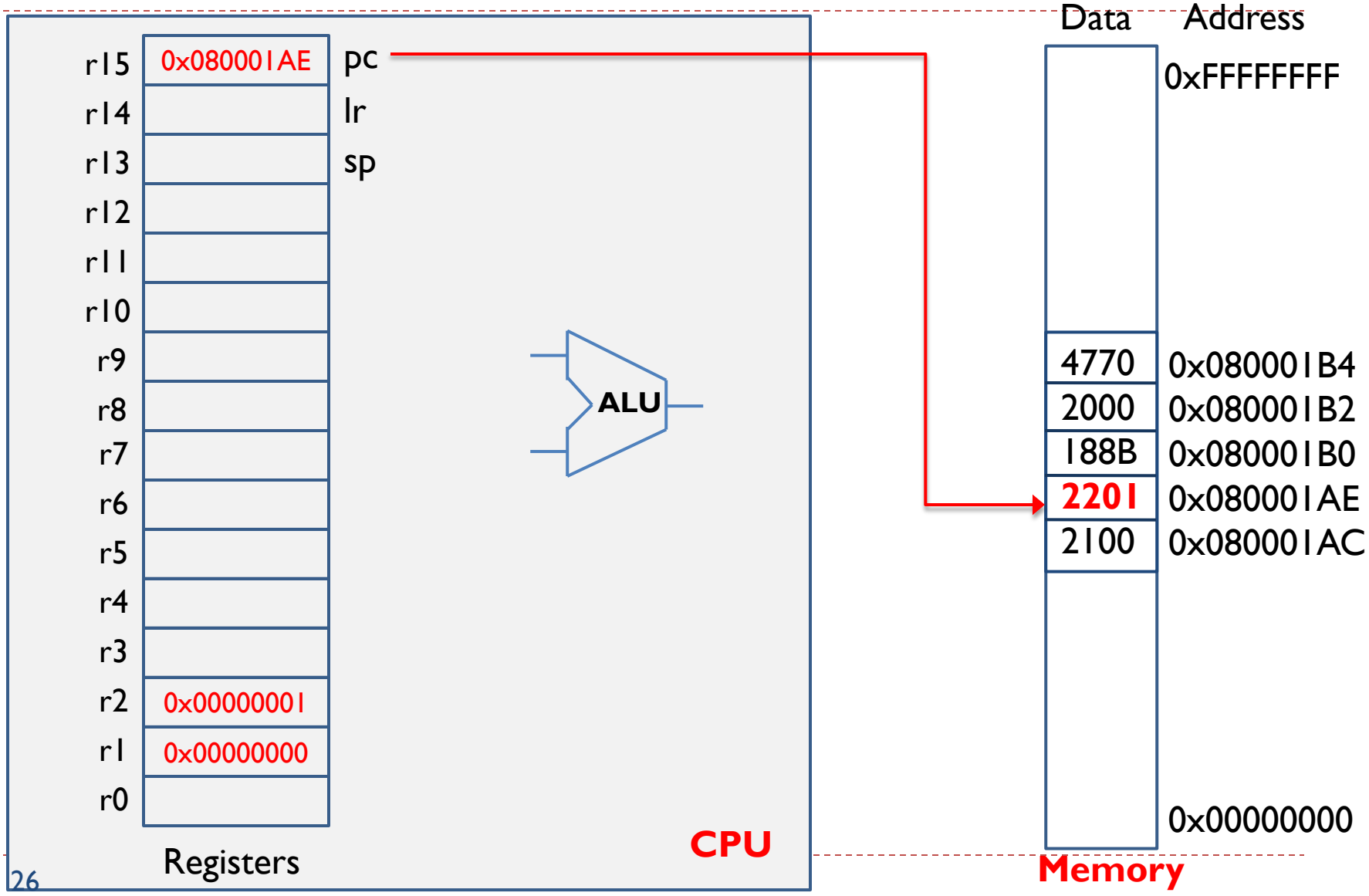
Fetch Instruction: pc = 0x08001AC
Decode Instruction: 2100 = MOVS r1, #0x00



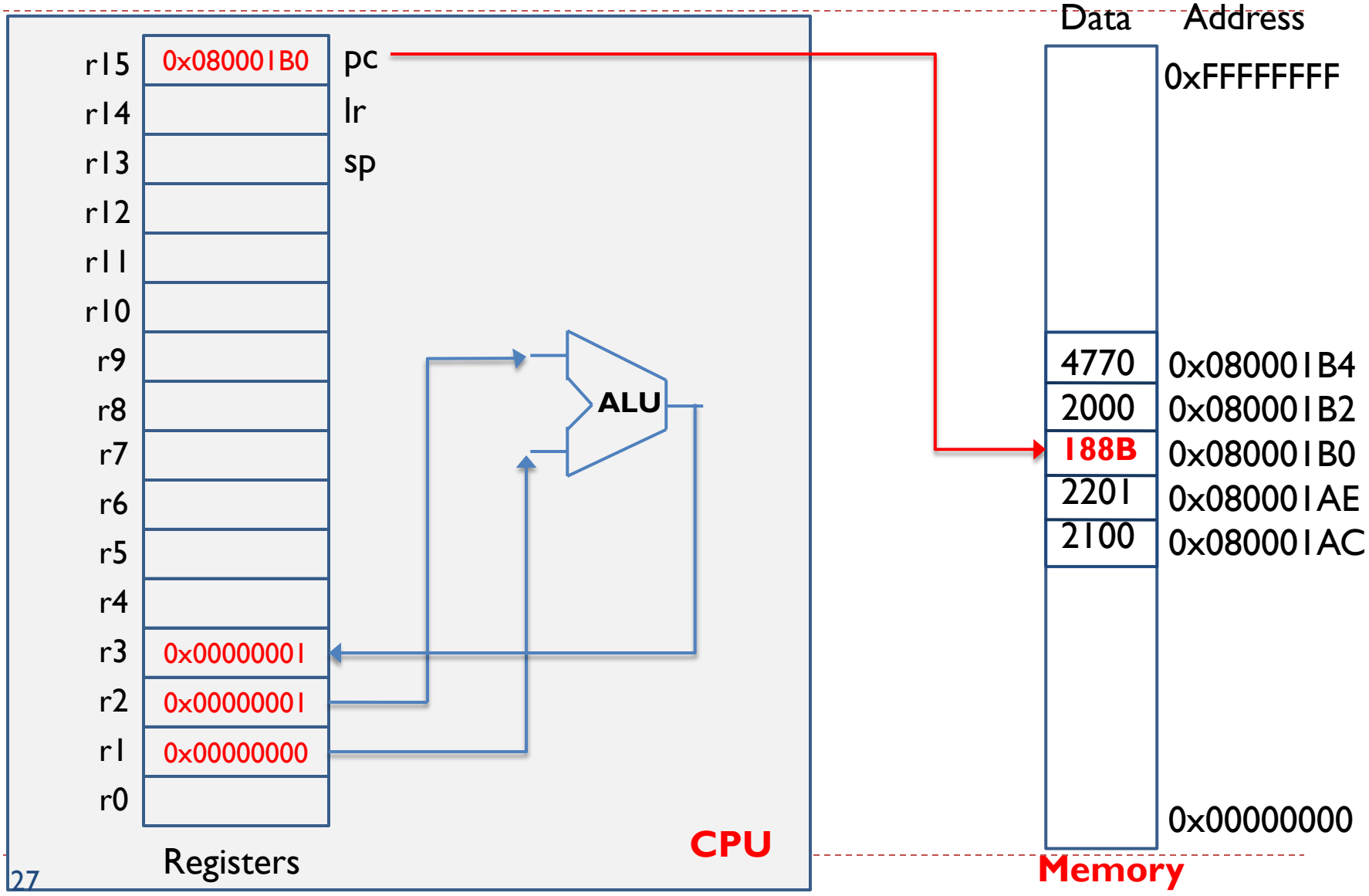
Execute Instruction:
MOVS r1, #0x00



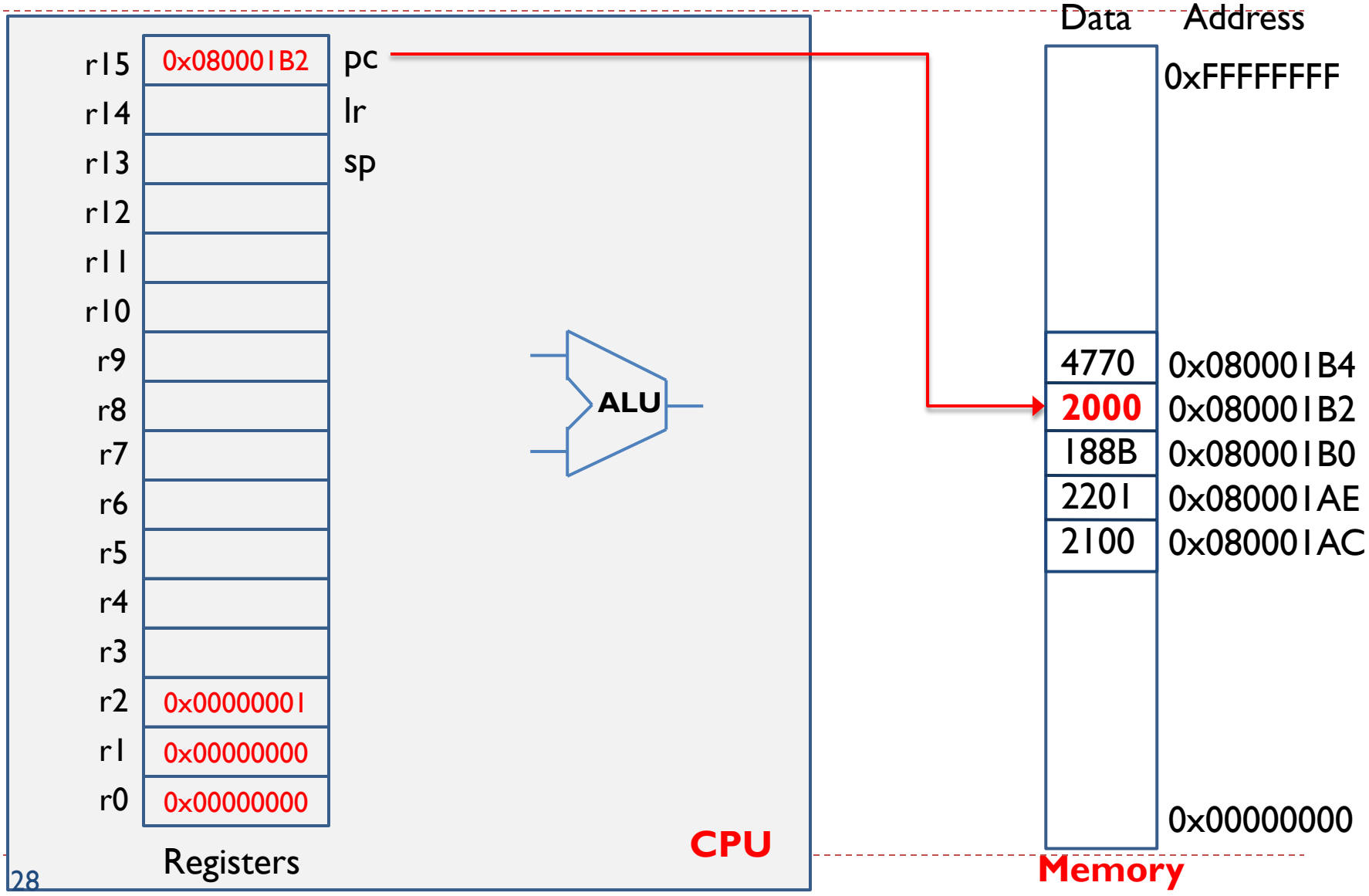
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2201 = MOVS r2, #0x01



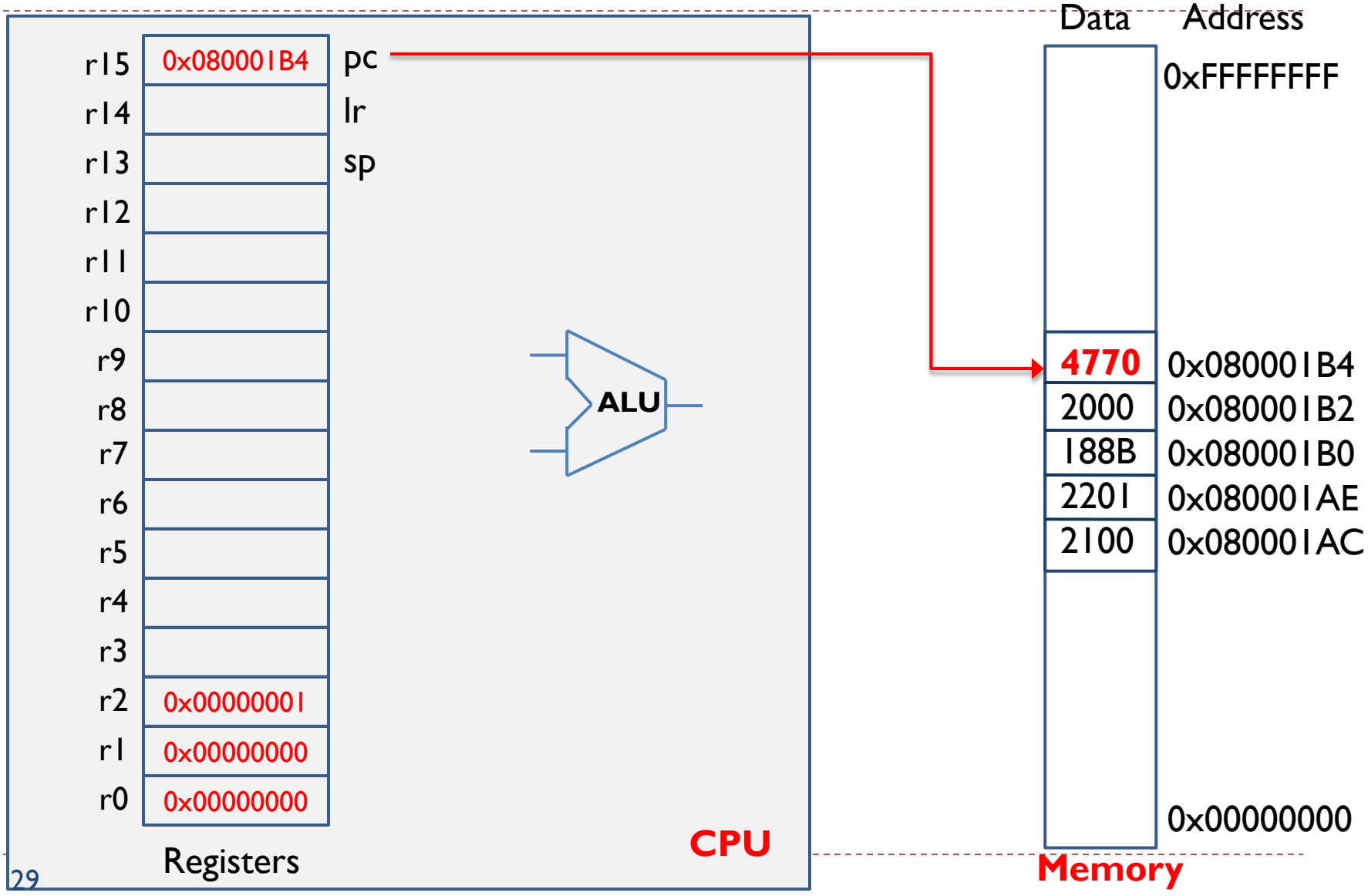
Fetch Next Instruction: pc = pc + 2
Decode & Execute: 188B = ADDS r3, r1, r2



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



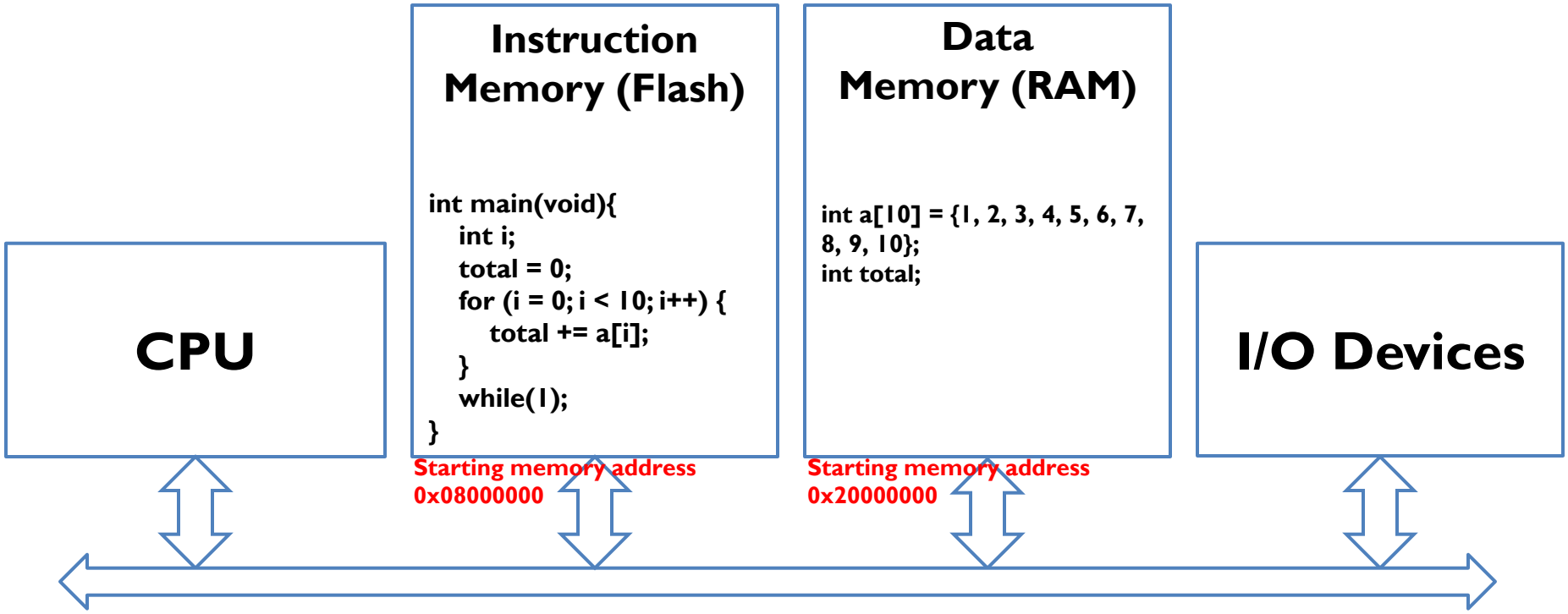
Fetch Next Instruction: pc = pc + 2
Decode & Decode: 4770 = BX lr



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
```

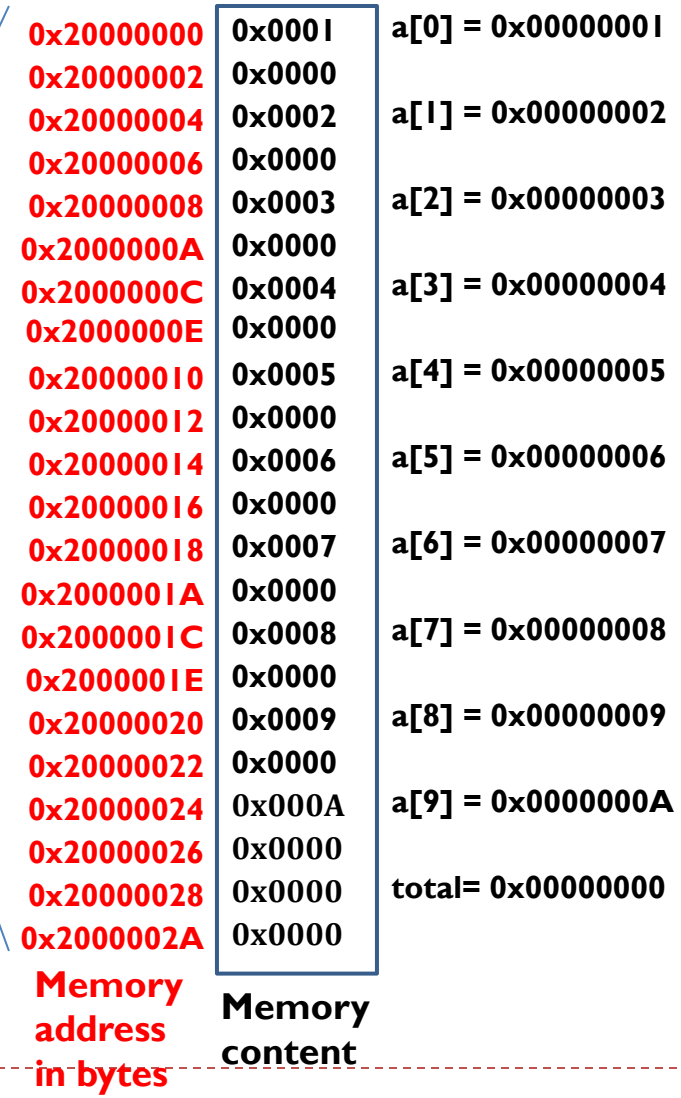
```
        MOVS  r1,#0x00
        LDR   r2,= total_addr
        STR   r1,[r2,#0x00]
        MOVS  r0,#0x00
        B     Check
Loop:   LDR   r1,= a_addr
        LDR   r1,[r1,r0,LSL #2]
        LDR   r2,= total_addr
        LDR   r2,[r2,#0x00]
        ADD   r1,r1,r2
        LDR   r2,= total_addr
        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

Data Memory (RAM)

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

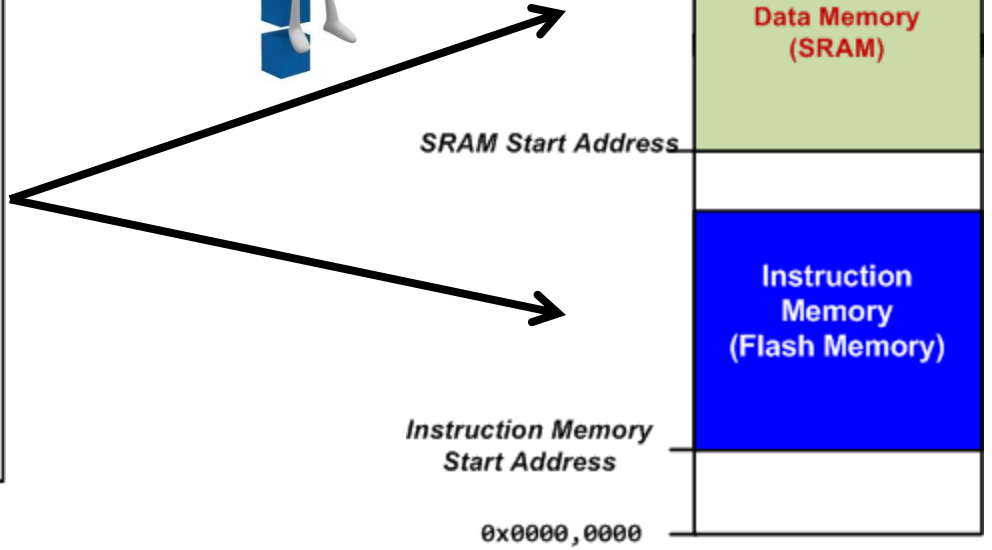
Assume the starting memory address of the data memory is 0x20000000



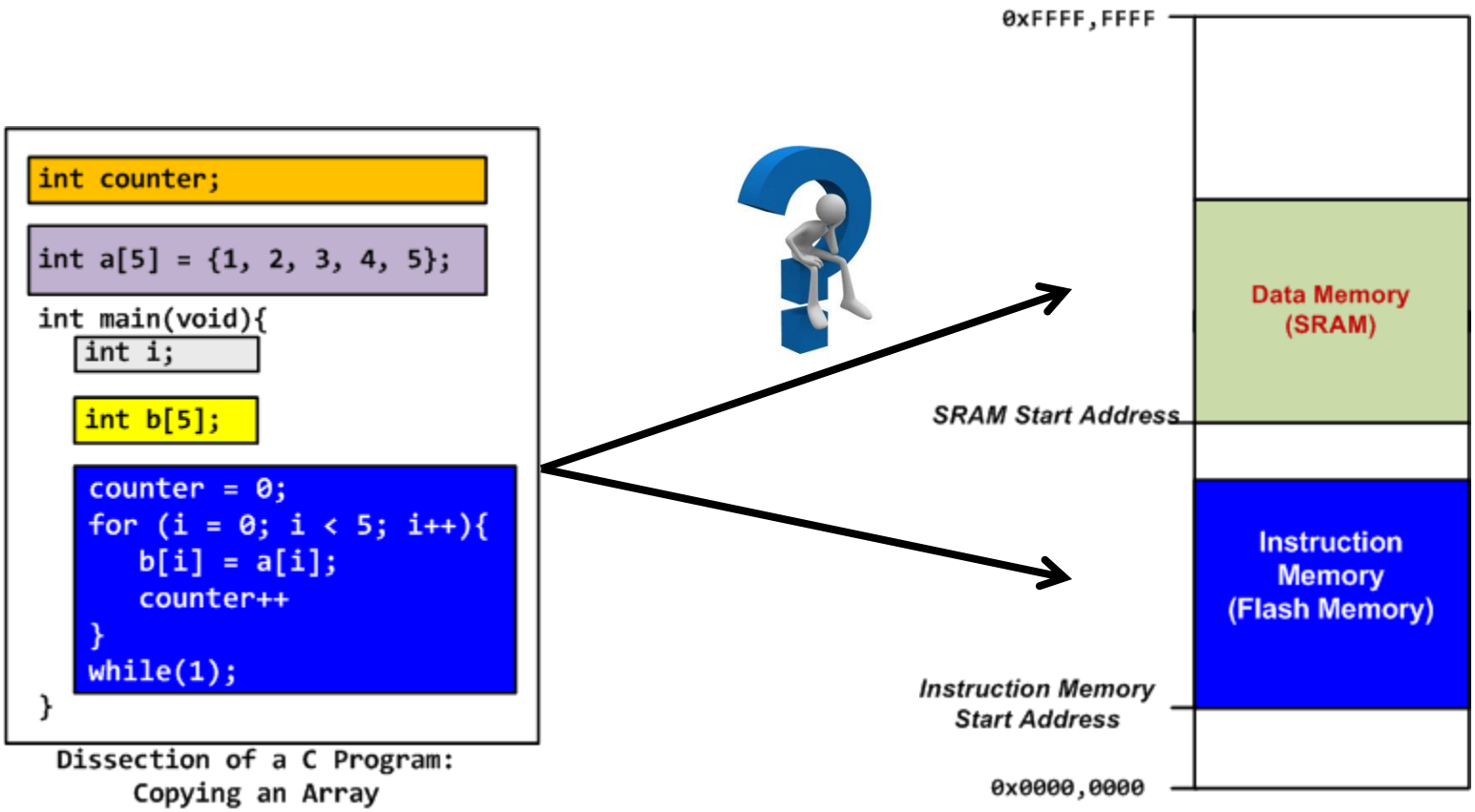
Loading Code and Data into Memory

```
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);
}
```

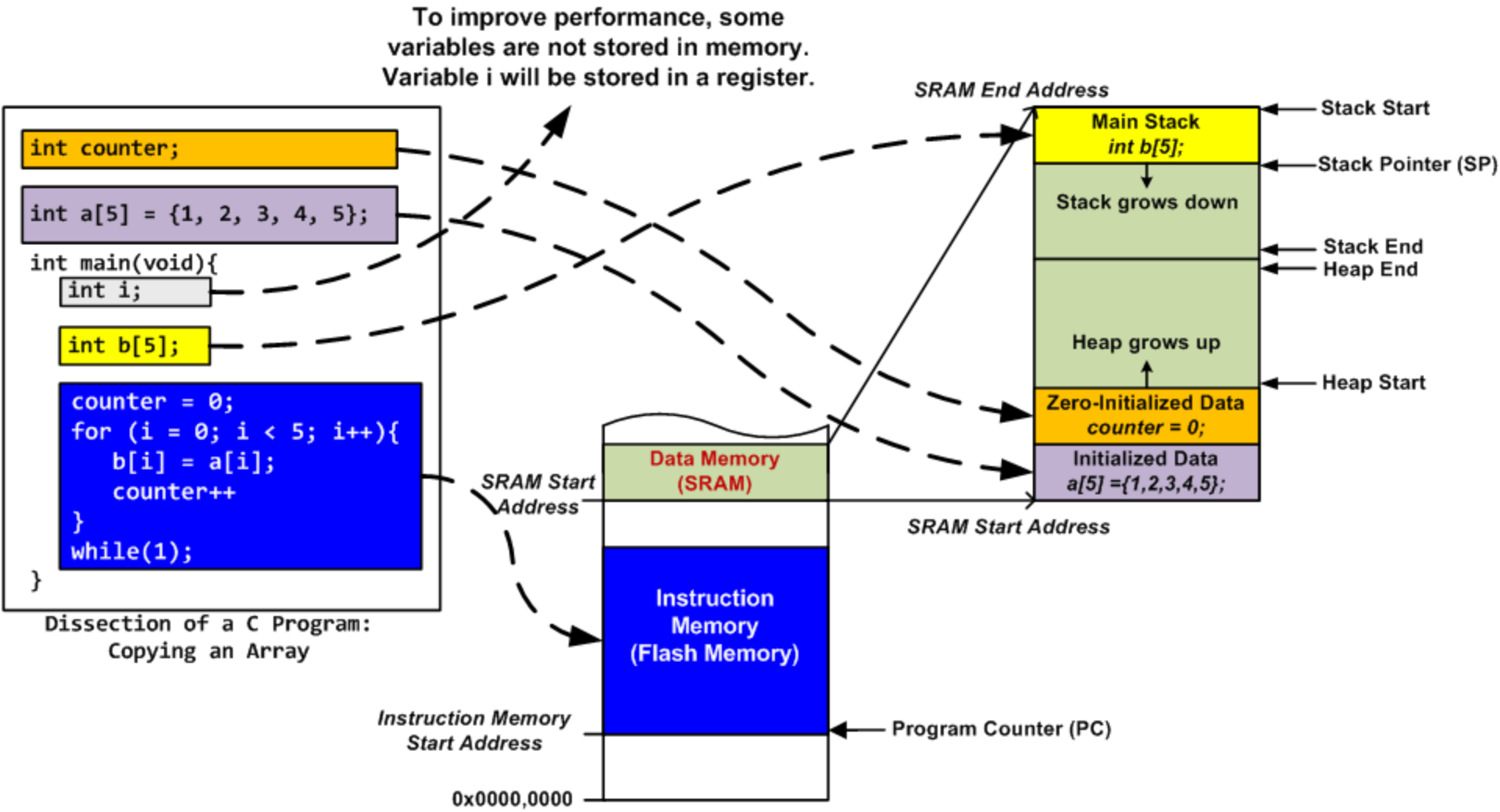
C Program:
Copying an Array



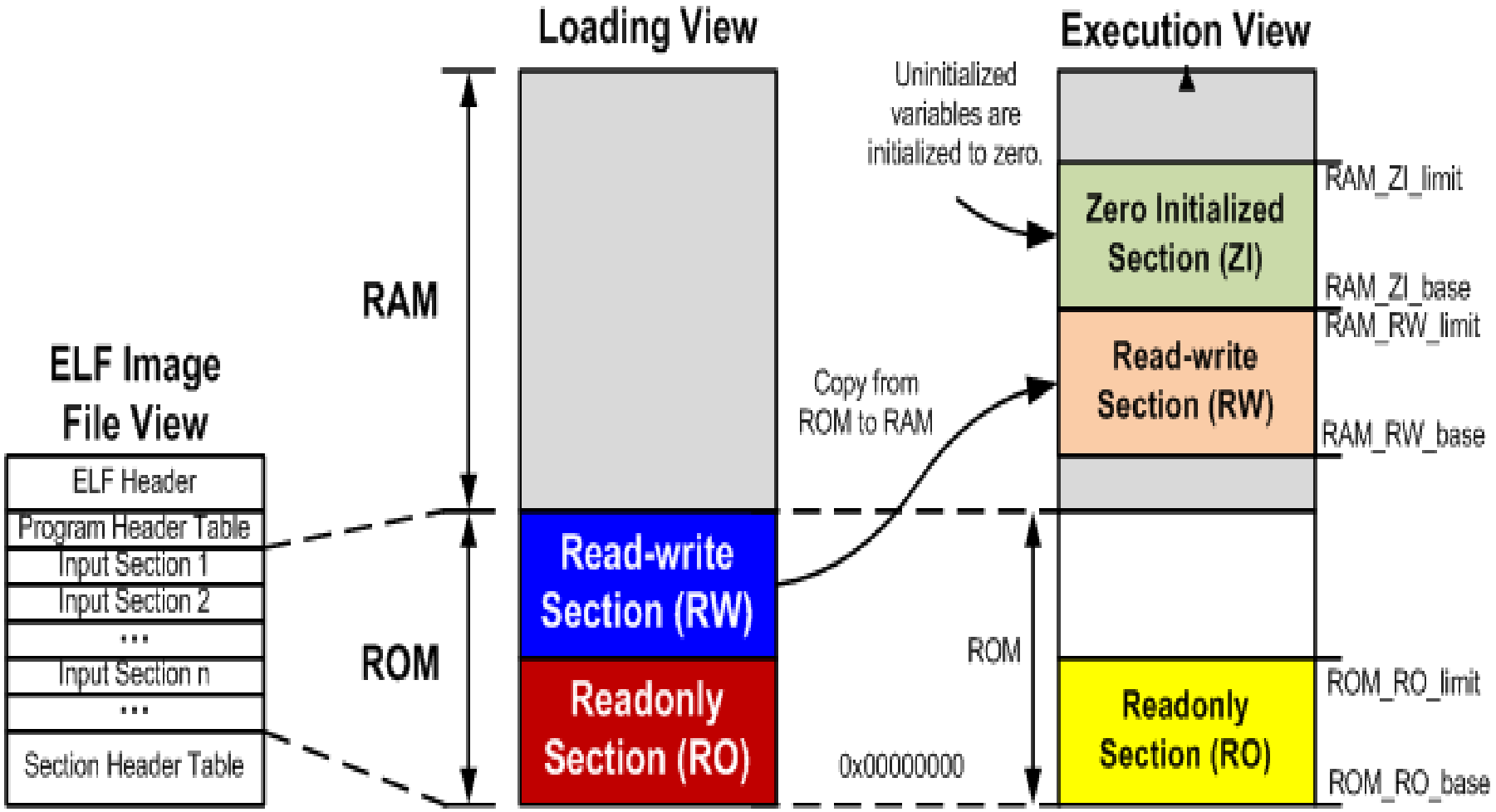
Loading Code and Data into Memory



Loading Code and Data into Memory



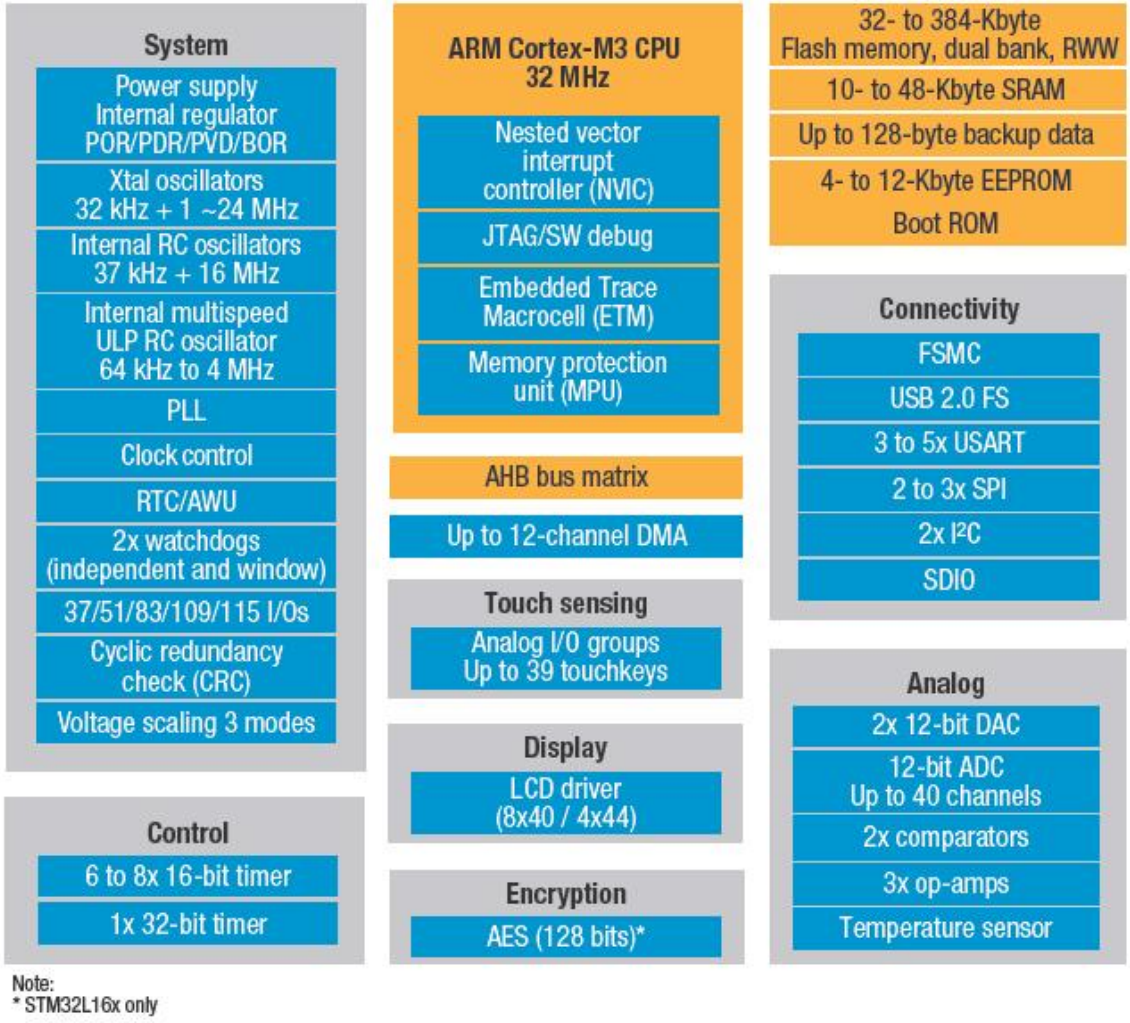
View of a Binary Program



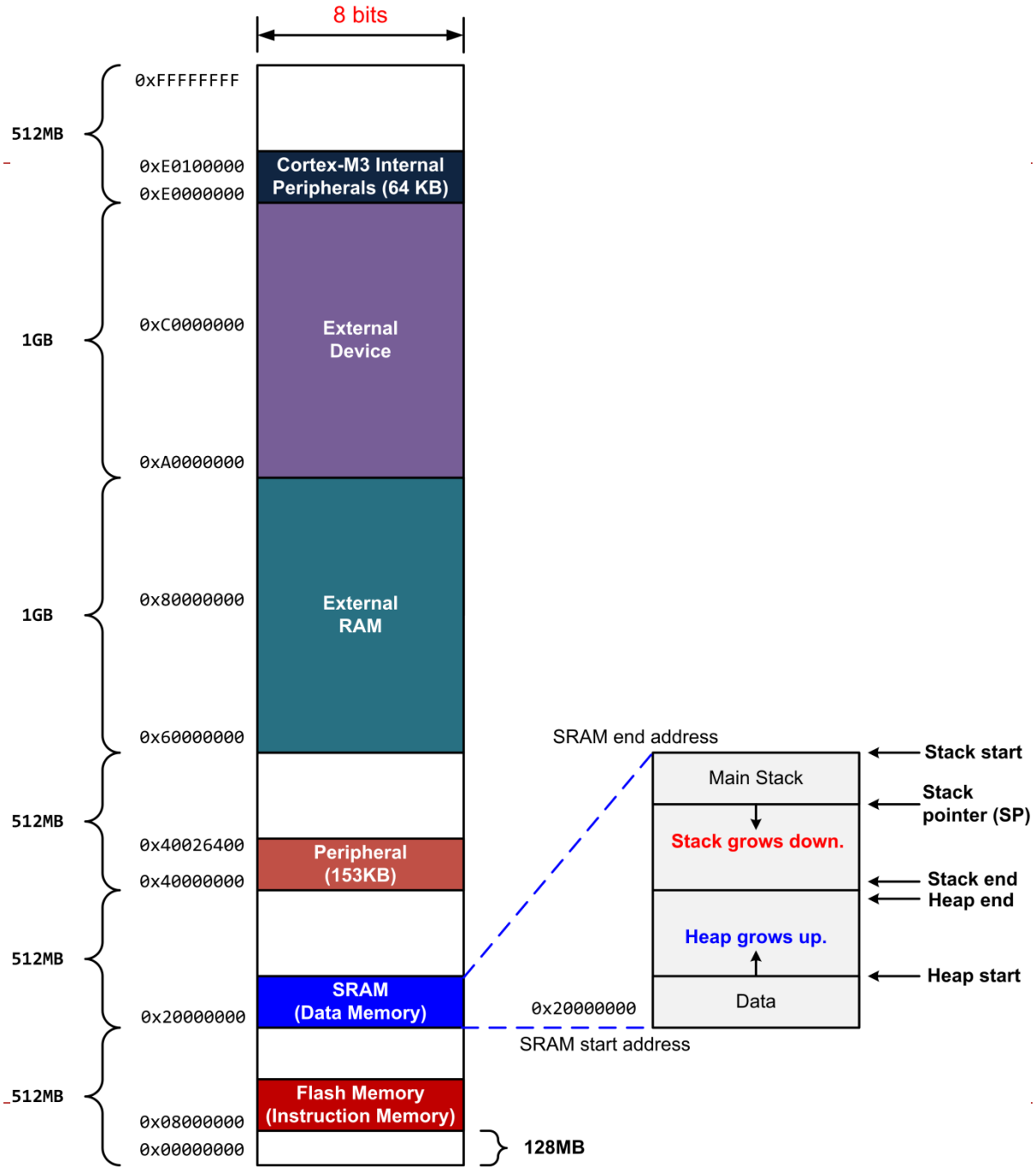
STM32L15x Block Diagram

- STM32 L1 platform
  - Up to 32 MHz Fcpu speed
  - 1.65 to 3.6 V
  - Single supply
  - Voltage scaling
  - ADC and DAC down to 1.8 V
  - 48 to 144 pins (BGA 5x5, QFN 7x7 available)
  - 8x40 LCD, USB2.0 FS
  - Touch Sense (RC, Multi Touch, Capacitive)

- New 384 Kbytes STM32 L1
  - Up to 144 pins
  - 384KB Flash (dual bank)
  - 12KB E²PROM
  - 48KB SRAM
  - Additional USART SPI, I²C
  - 1x32bit Timer
  - FSMC
  - SDIO
  - New RTC
  - Up to 3 Op-Amps



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