

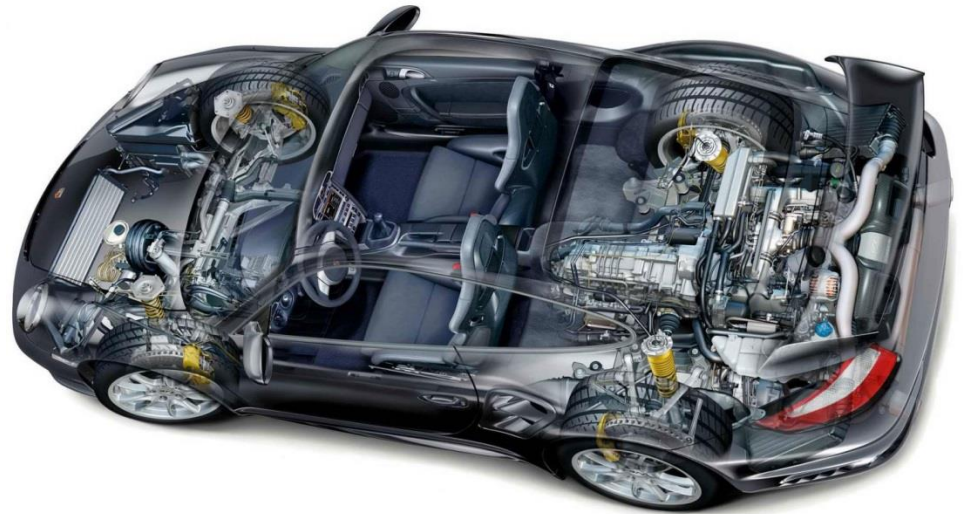
Chapter I

Computer and Assembly Language

Dr. Yifeng Zhu
Electrical and Computer Engineering
University of Maine

Spring 2018

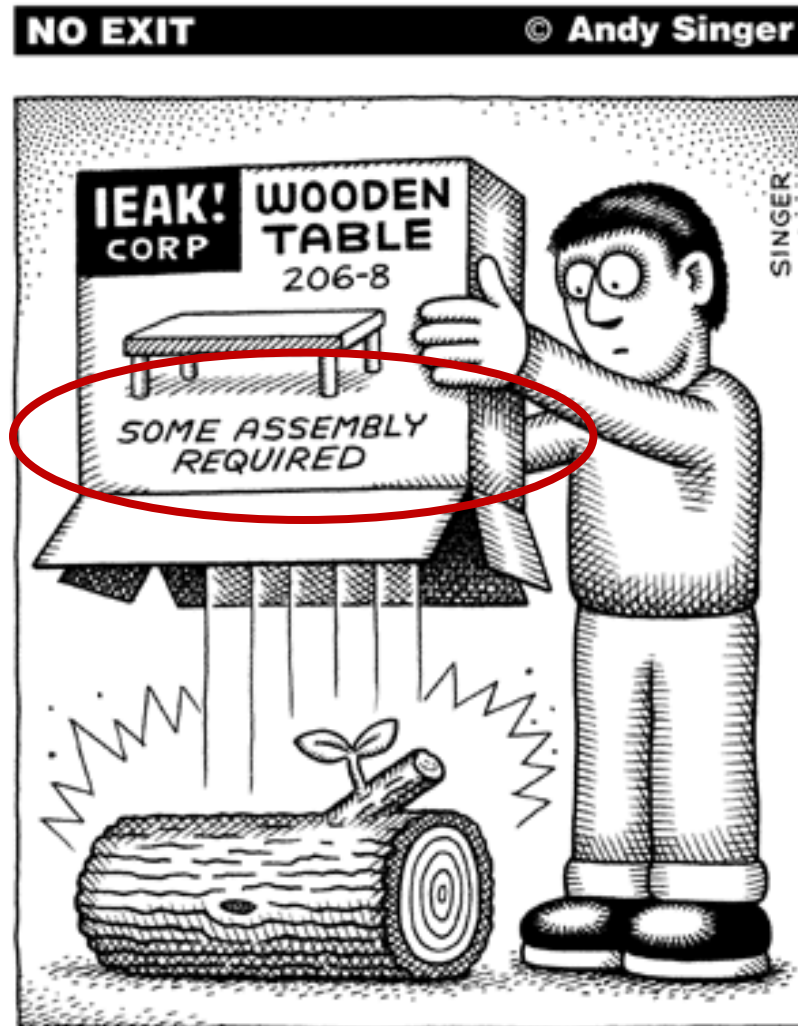
Embedded Systems



Amazon Warehouse



Assembly Programs



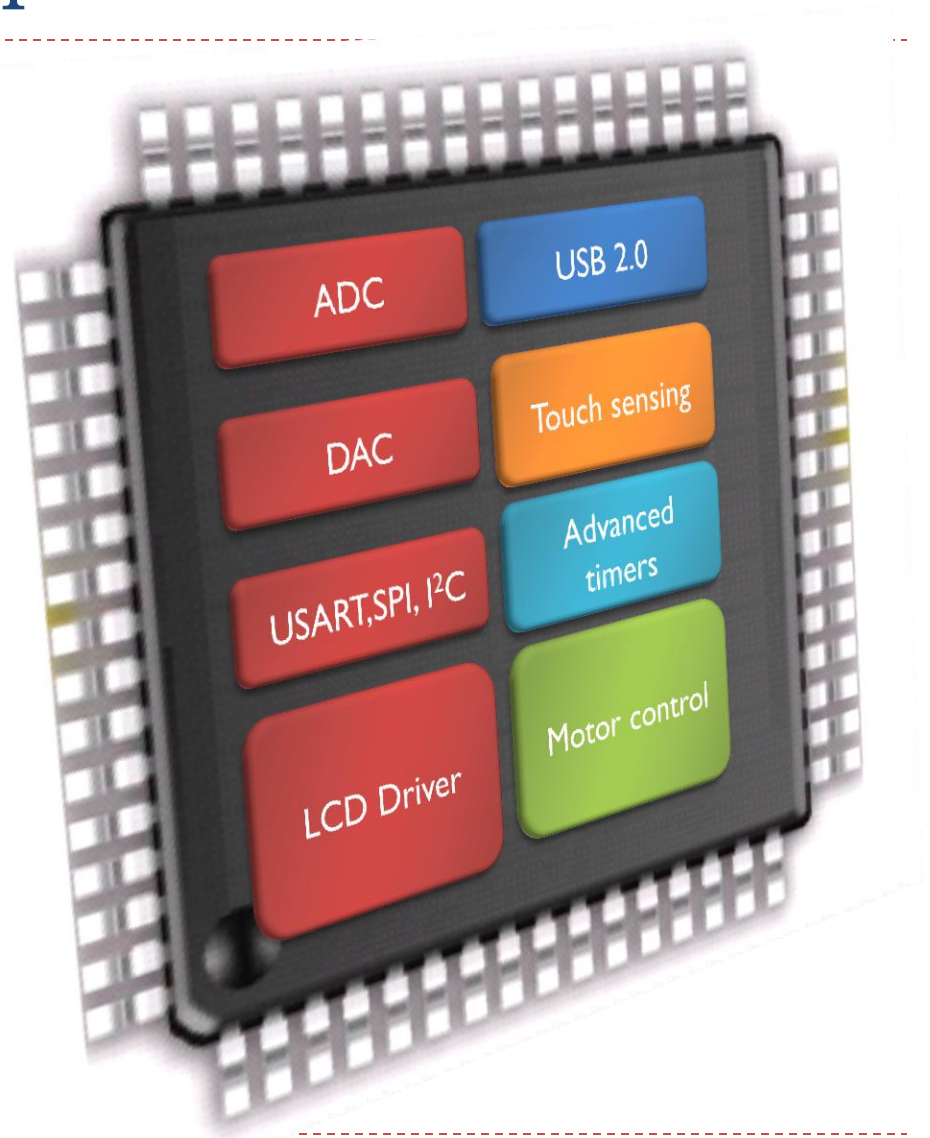
<http://www.andysinger.com/>

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
- ▶ In 2010 alone, **6.1 billion** ARM-based processor, representing **95%** of smartphones, **35%** of digital televisions and set-top boxes and **10%** of mobile computers
- ▶ As of 2014, over 50 billion ARM processors have been produced



iPhone 5 Teardown



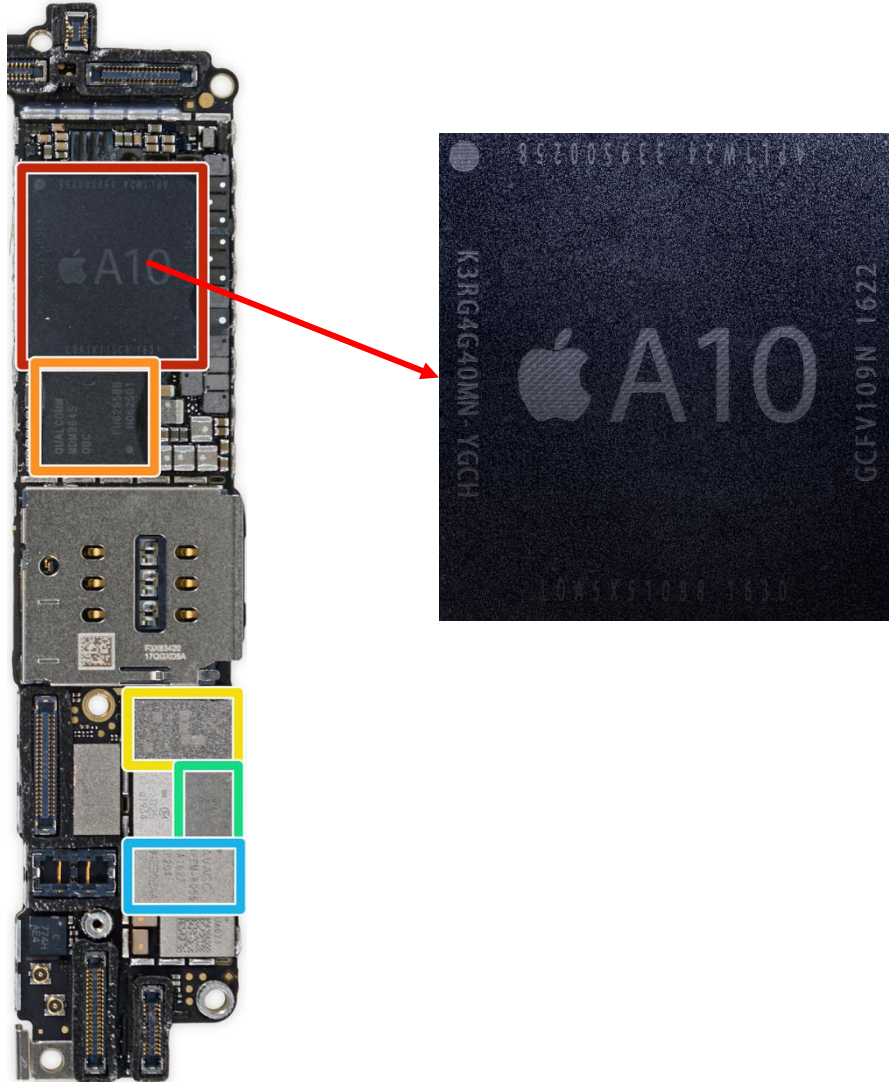
The A6 processor is the first Apple System-on-Chip (SoC) to use a custom design, based off the **ARMv7** instruction set.

iPhone 6 Teardown



The A8 processor is the first 64-bit ARM based SoC. It supports **ARM A64, A32, and T32** instruction set.

iPhone 7 Teardown



A10 processor:

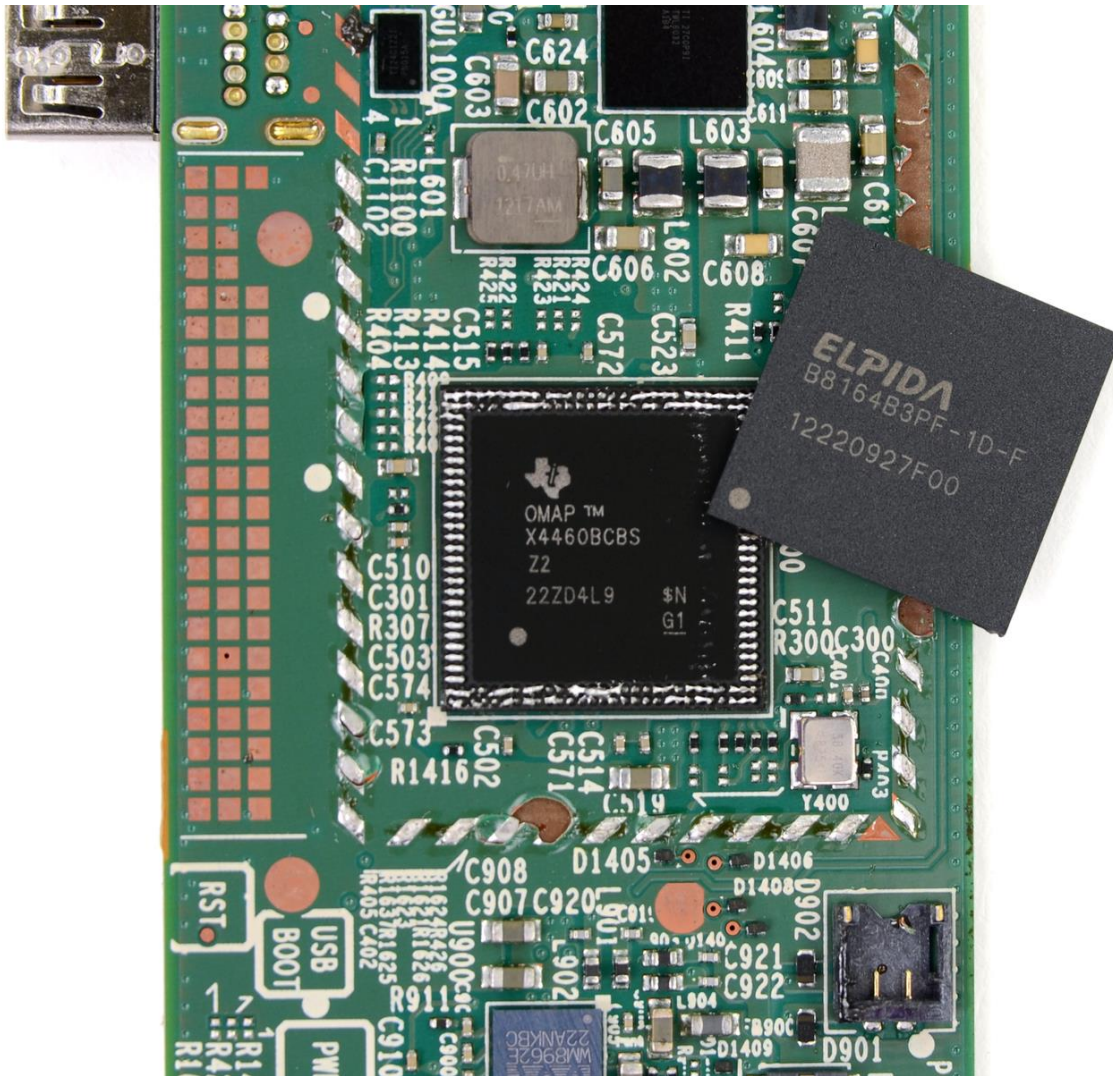
- 64-bit system on chip (SoC)
- **ARM**v8-A core

Apple Watch



- ▶ Apple S1 Processor
 - ▶ **32-bit ARMv7-A** compatible
 - ▶ # of Cores: **1**
 - ▶ CMOS Technology: 28 nm
 - ▶ L1 cache 32 KB data
 - ▶ L2 cache 256 KB
 - ▶ GPU PowerVR SGX543

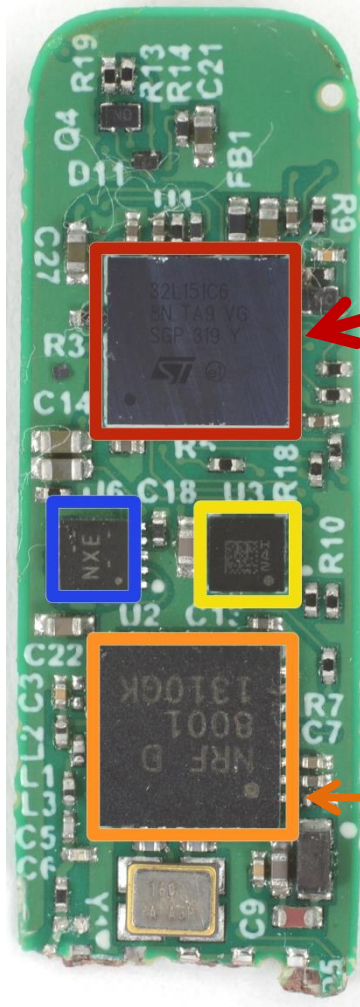
Kindle HD Fire



Texas Instruments
OMAP 4460 dual-
core processor



Fitbit Flex Teardown



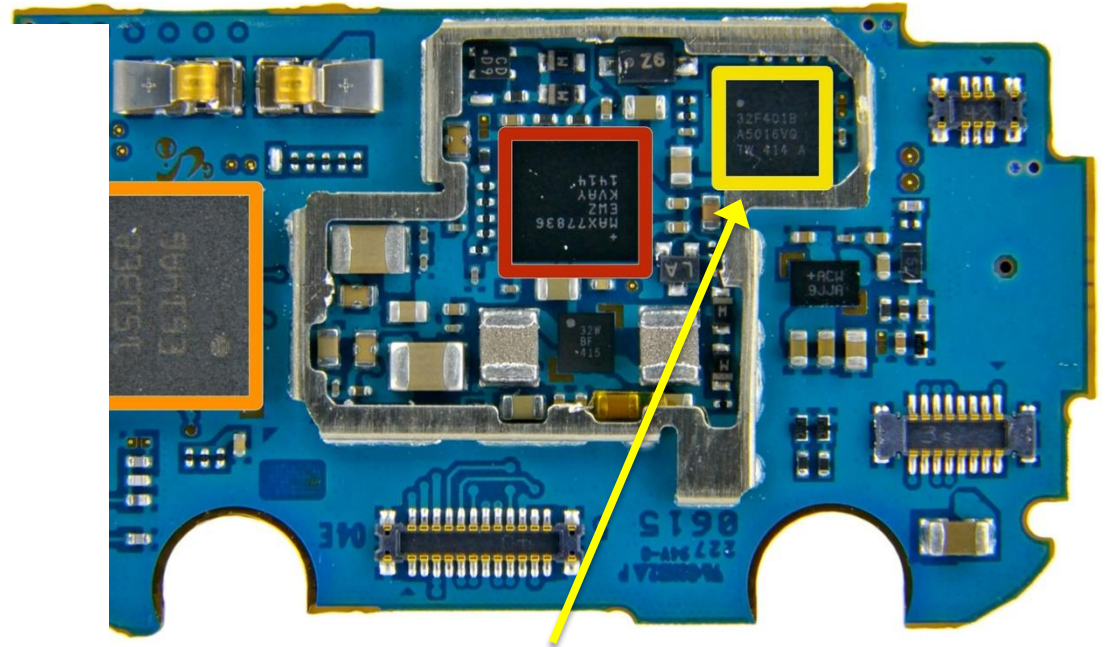
STMicroelectronics
32L15IC6 Ultra Low
Power **ARM Cortex M3**
Microcontroller

Nordic Semiconductor
nRF8001 Bluetooth Low
Energy Connectivity IC

Samsung Galaxy Gear

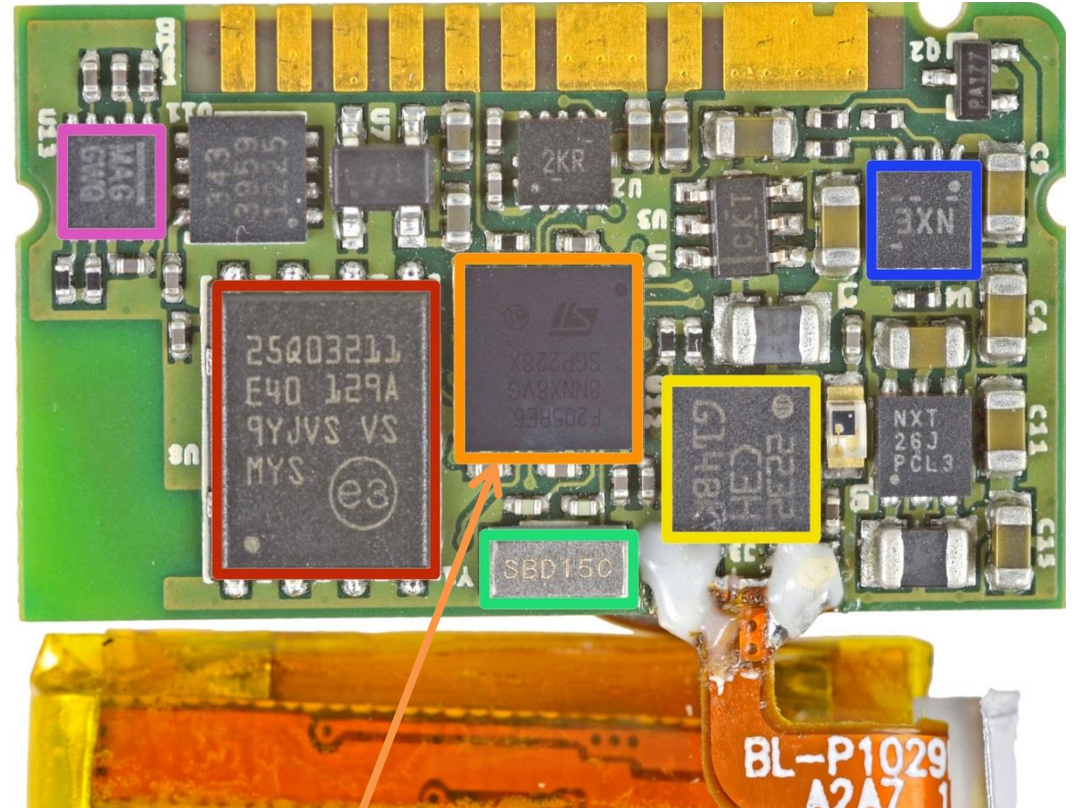


source: ifixit.com



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

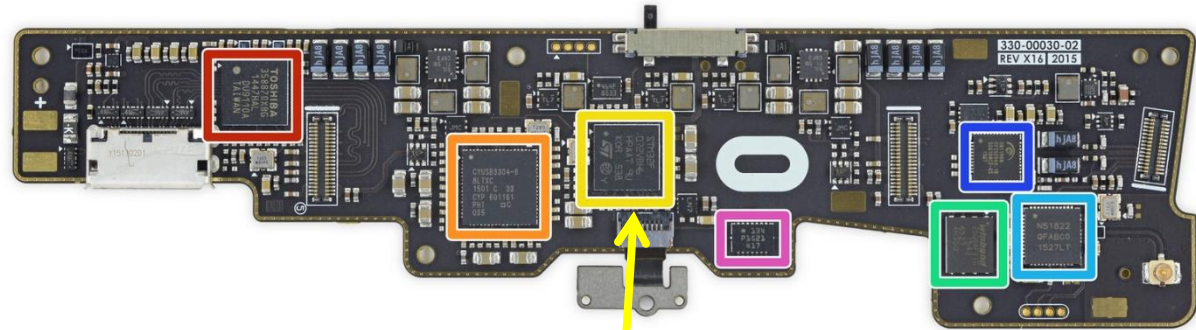
Pebble Smartwatch



source: ifixit.com

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

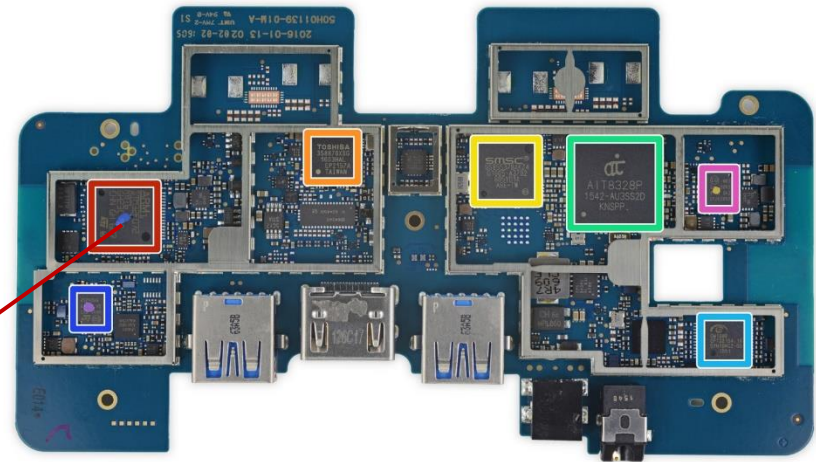
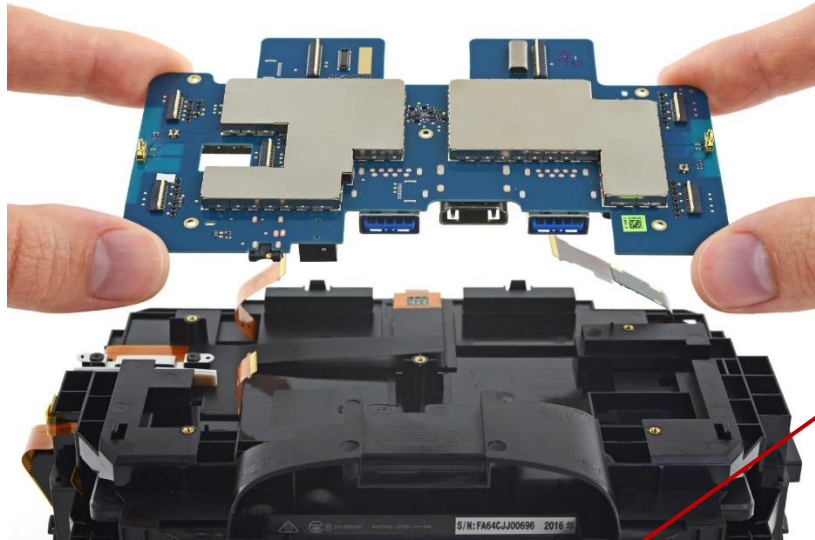
Oculus VR



- ▶ Facebook's \$2 Billion Acquisition Of Oculus in 2014
- ▶ ST Microelectronics STM32F072VB **ARM Cortex-M0** 32-bit RISC Core Microcontroller

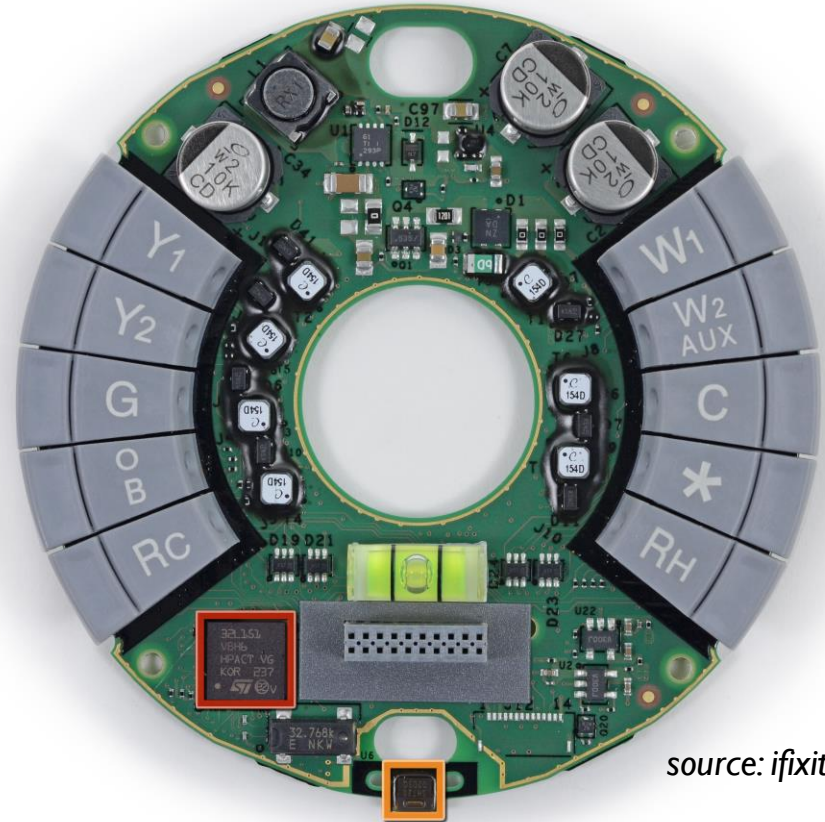
source: ifixit.com

HTC Vive



STMicroelectronics 32F072R8
ARM Cortex-M0
Microcontroller

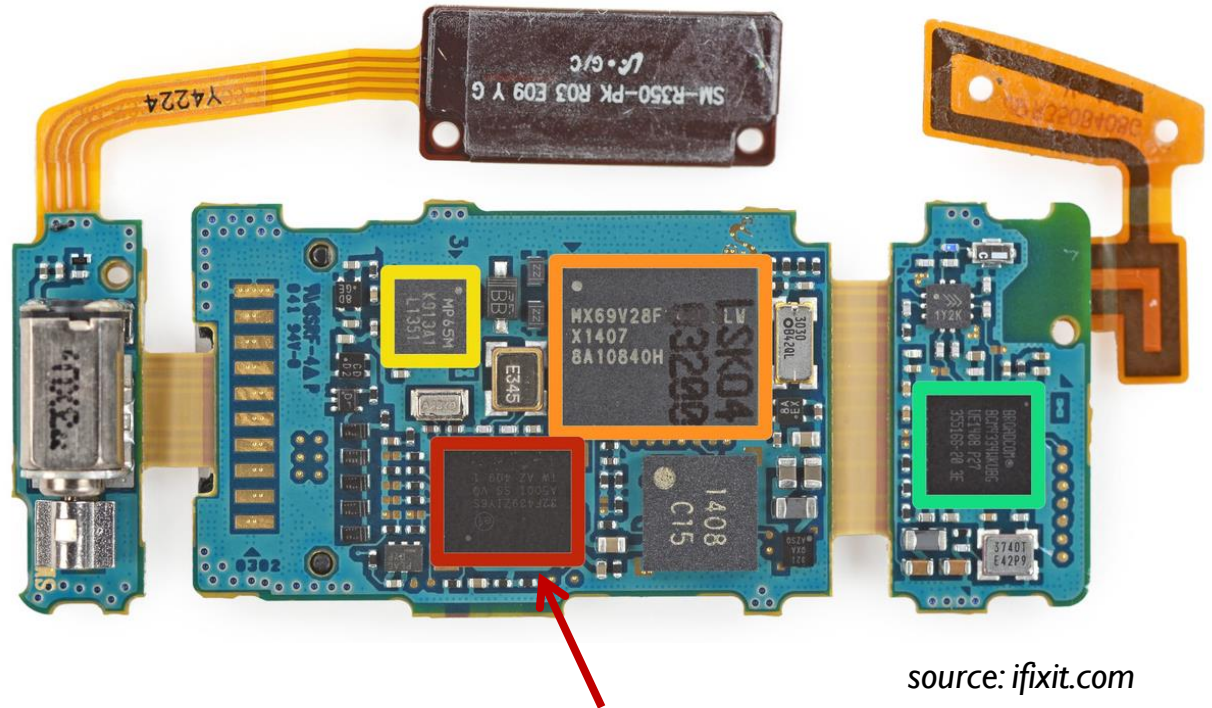
Nest Learning Thermostat



source: ifixit.com

- ▶ ST Microelectronics **STM32L151VB** ultra-low-power 32 MHz ARM **Cortex-M3** MCU

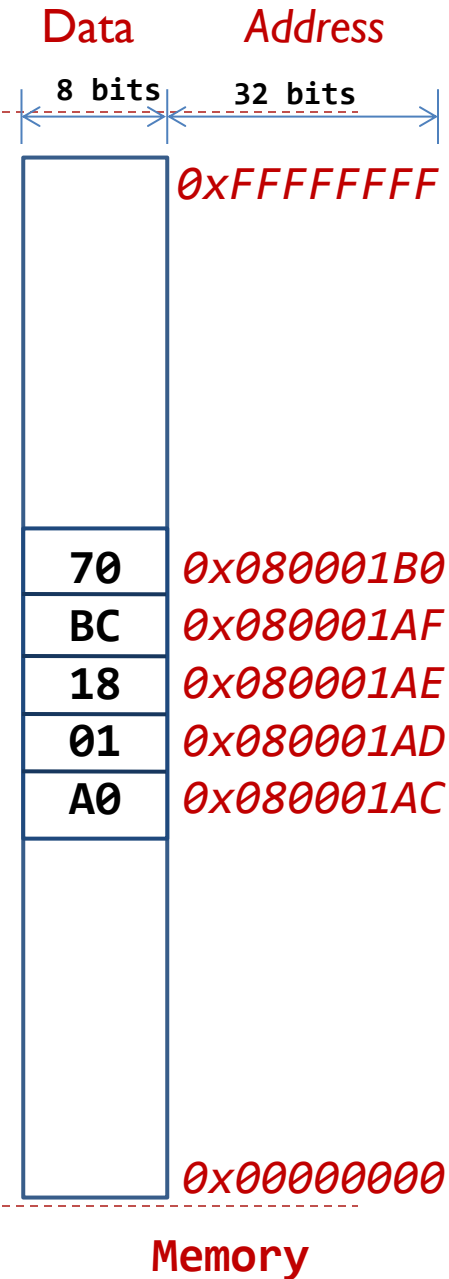
Samsung Gear Fit Fitness Tracker



- ▶ STMicroelectronics **STM32F439ZI** 180 MHz, 32 bit **ARM Cortex-M4** CPU

Memory

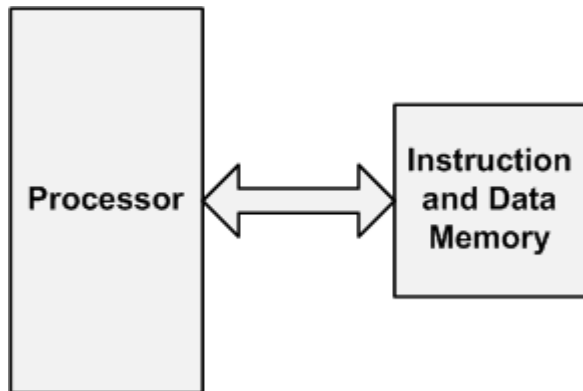
- ▶ Memory is arranged as a series of “locations”
 - ▶ Each location has a unique “address”
 - ▶ Each location holds a byte (**byte-addressable**)
 - ▶ e.g. the memory location at address `0x080001B0` contains the byte value `0x70`, i.e., 112
- ▶ The number of locations in memory is limited
 - ▶ e.g.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

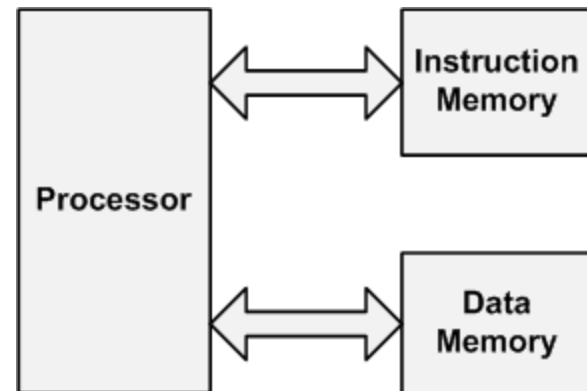
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

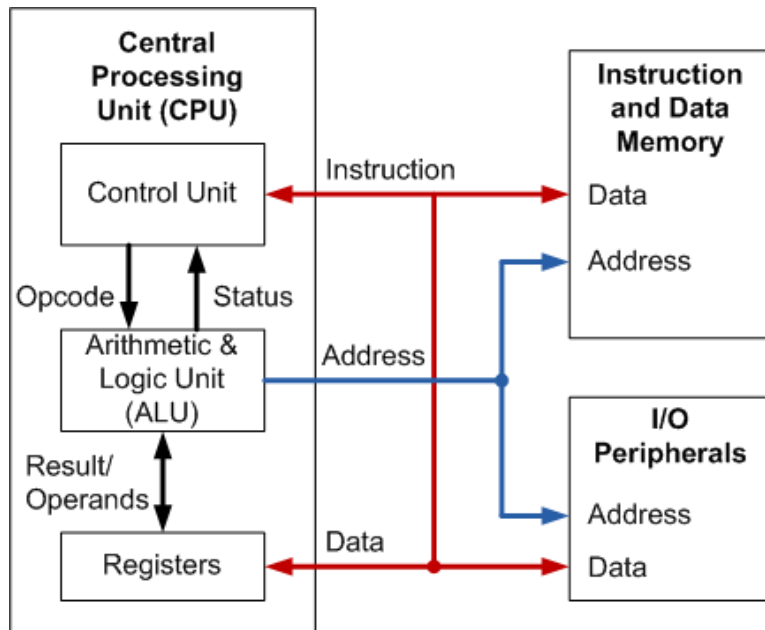
Data and instructions are stored into separate memories.



Computer Architecture

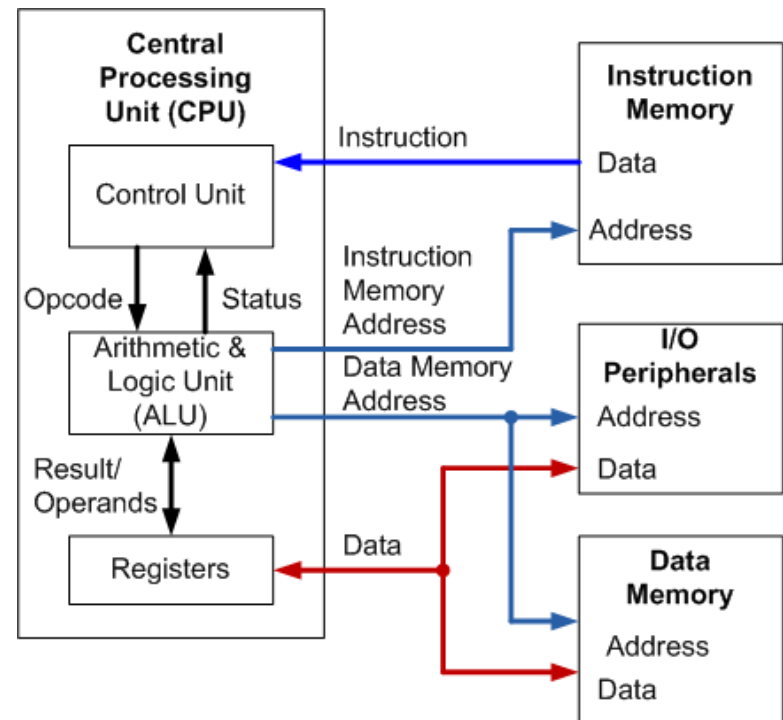
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

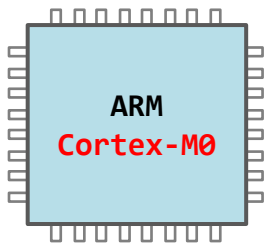
Data and instructions are stored into separate memories.



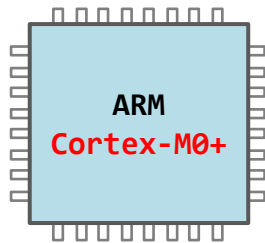
ARM Cortex-M Series Family

Von-Neumann

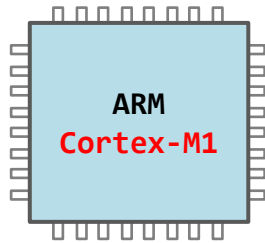
Instructions and data are stored in the same memory.



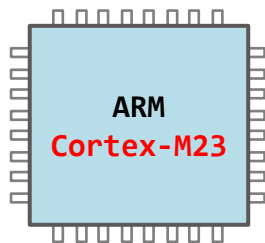
ARMv6-M



ARMv6-M



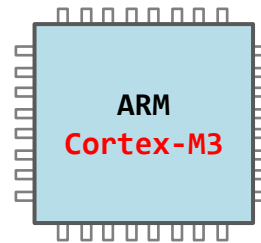
ARMv6-M



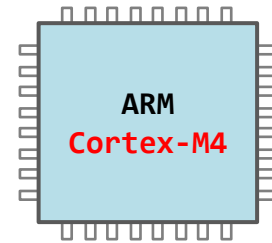
ARMv8-M

Harvard

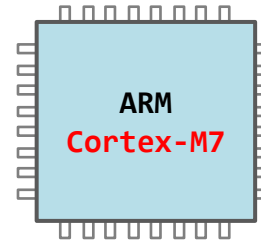
Data and instructions are stored into separate memories.



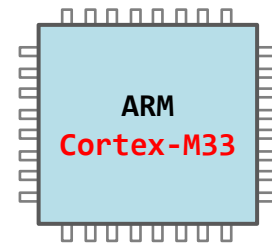
ARMv7-M



ARMv7E-M



ARMv7E-M



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  
loop    B      check  
        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

```
0010000100000000  
0010001000000001  
0001100010001011  
0010000000000000  
0100011101110000
```

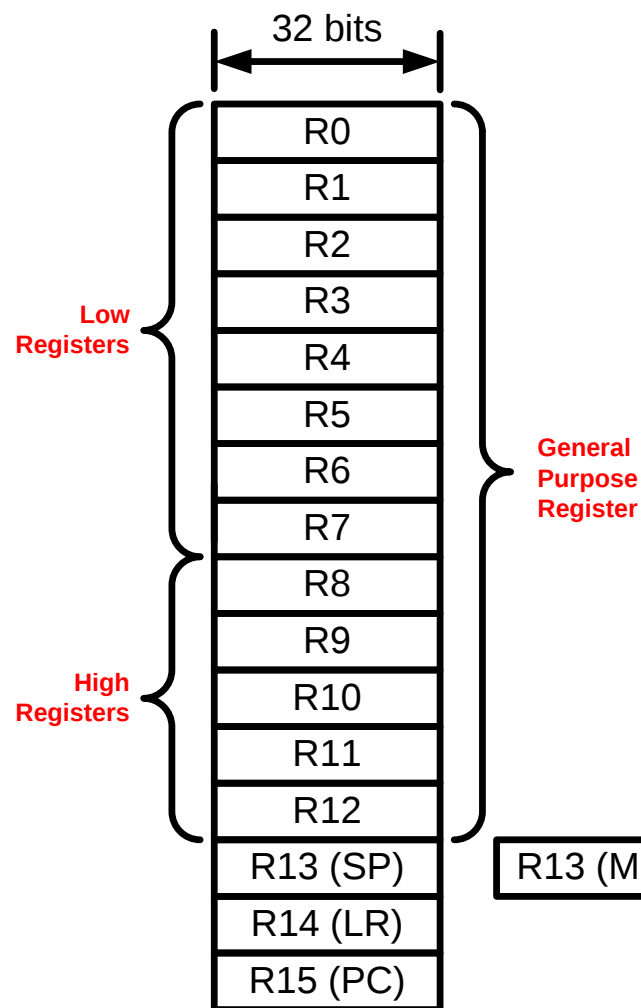
In Binary

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

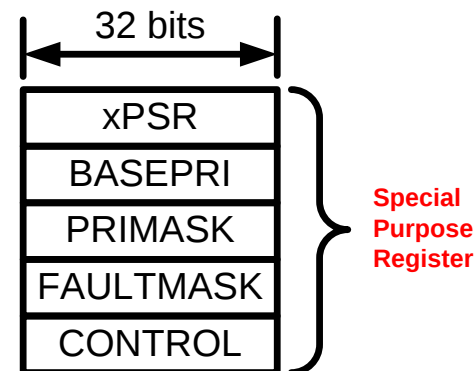
In Hex

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

Processor Registers

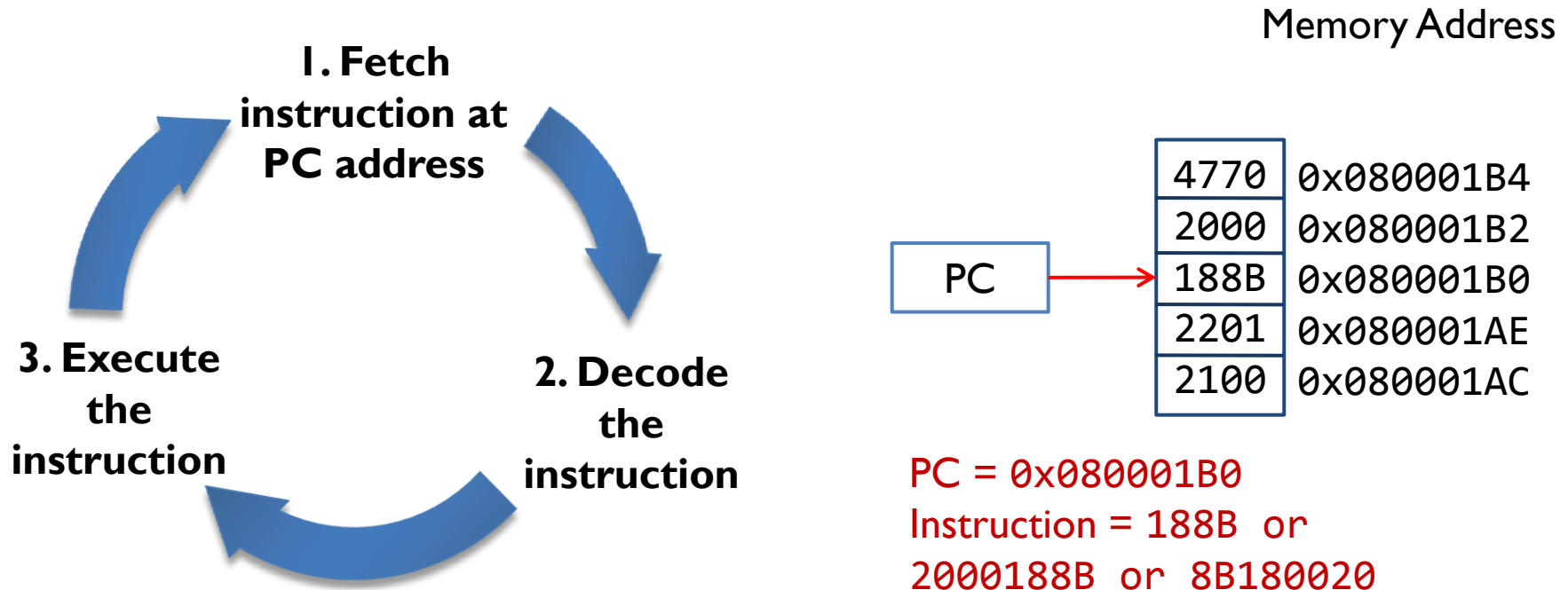


- ▶ Fastest way to read and write
- ▶ Registers are within the processor chip
- ▶ A register stores 32-bit value
- ▶ STM32L has
 - ▶ **R0-R12**: 13 general-purpose registers
 - ▶ **R13**: Stack pointer (Shadow of MSP or PSP)
 - ▶ **R14**: Link register (LR)
 - ▶ **R15**: Program counter (PC)
 - ▶ Special registers (xPSR, BASEPRI, PRIMASK, etc)



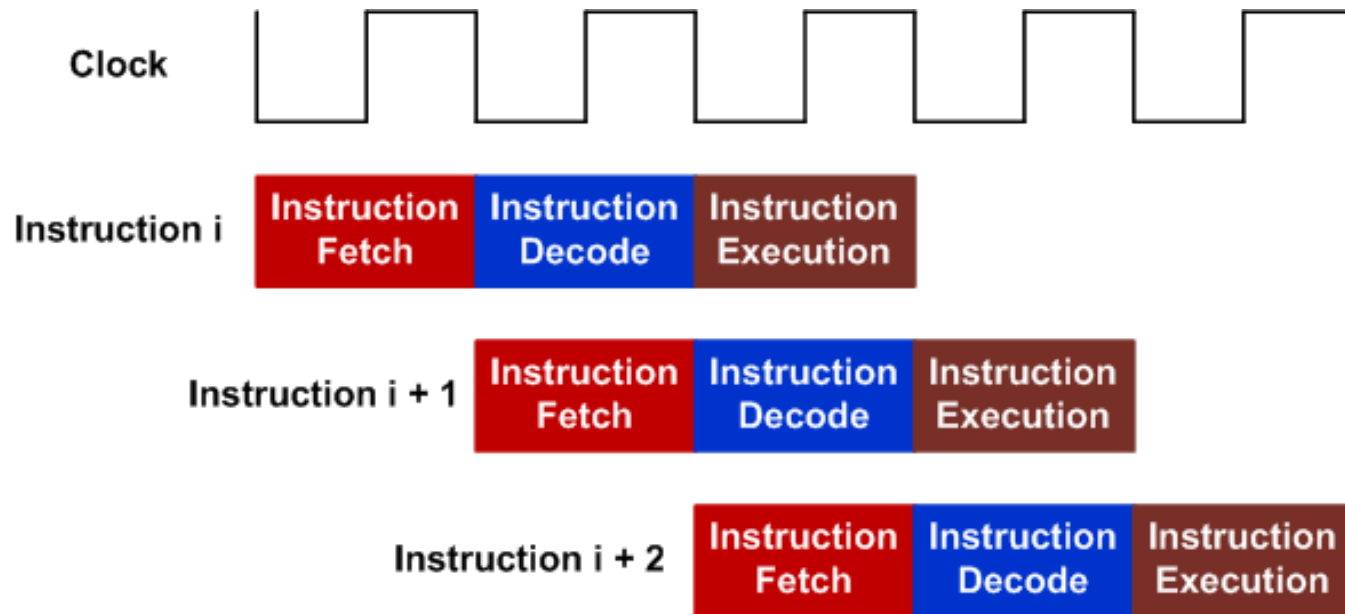
Program Execution

- ▶ **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

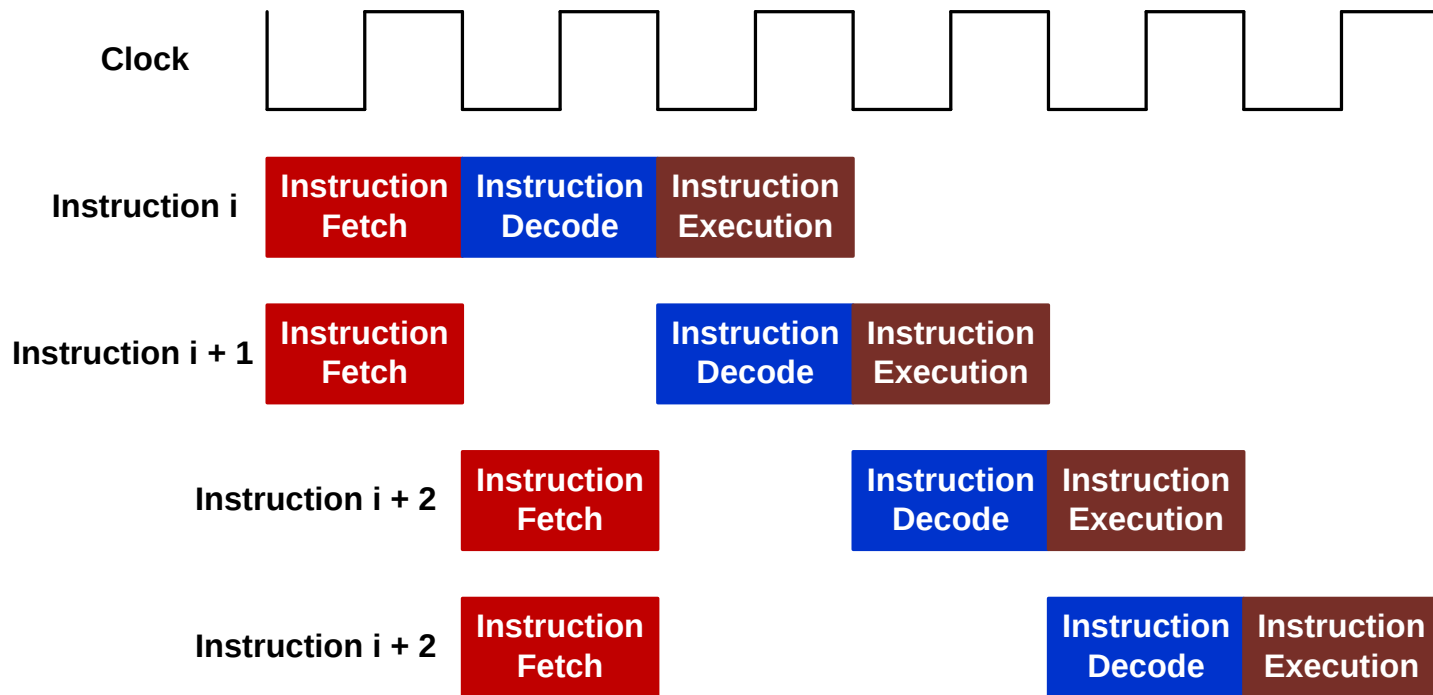
- ▶ **Pipelining** allows hardware resources to be fully utilized
- ▶ One 32-bit instruction or two 16-bit instructions can be fetched.



Pipeline of 32-bit instructions

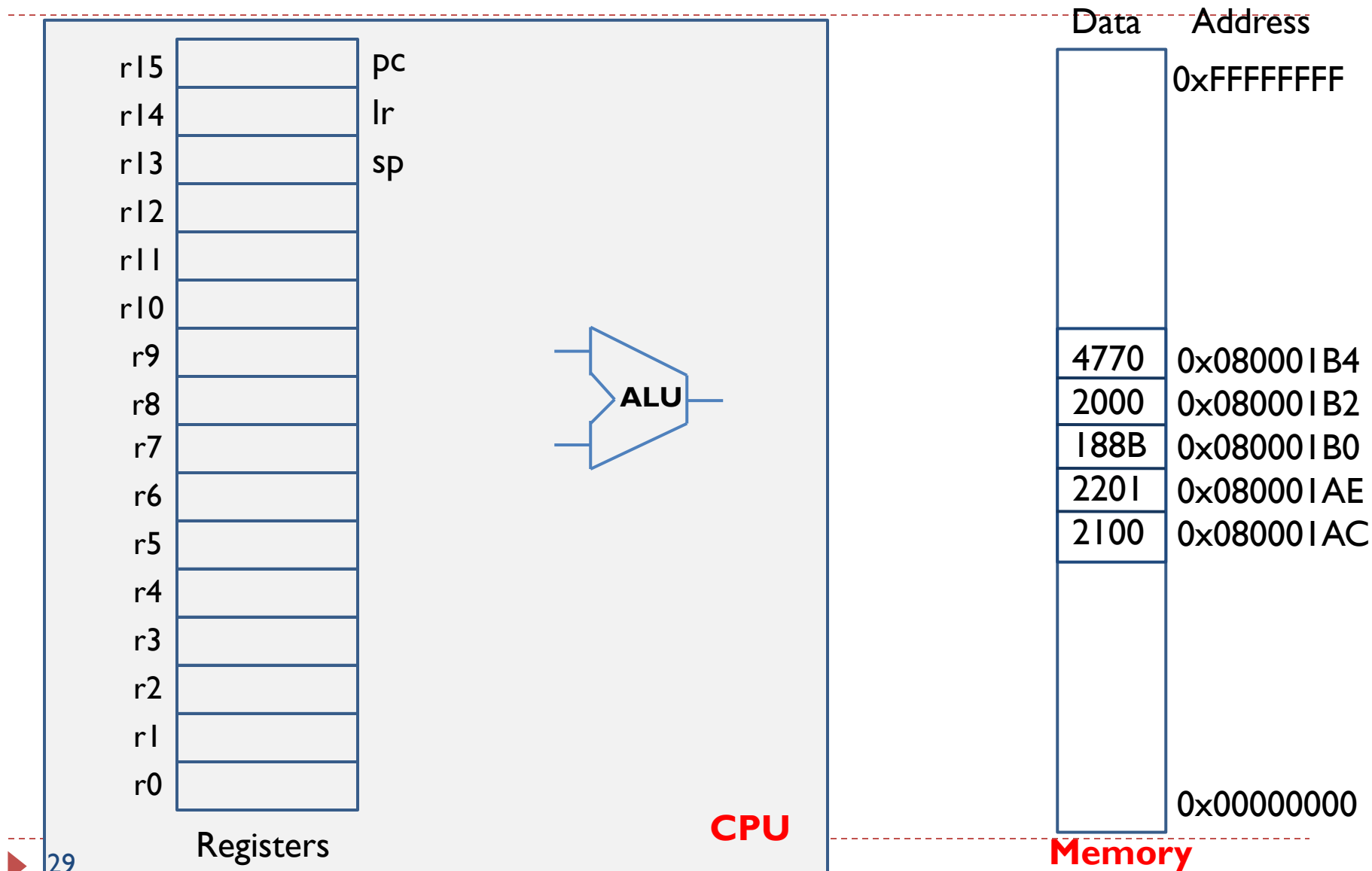
Three-state pipeline: Fetch, Decode, Execution

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



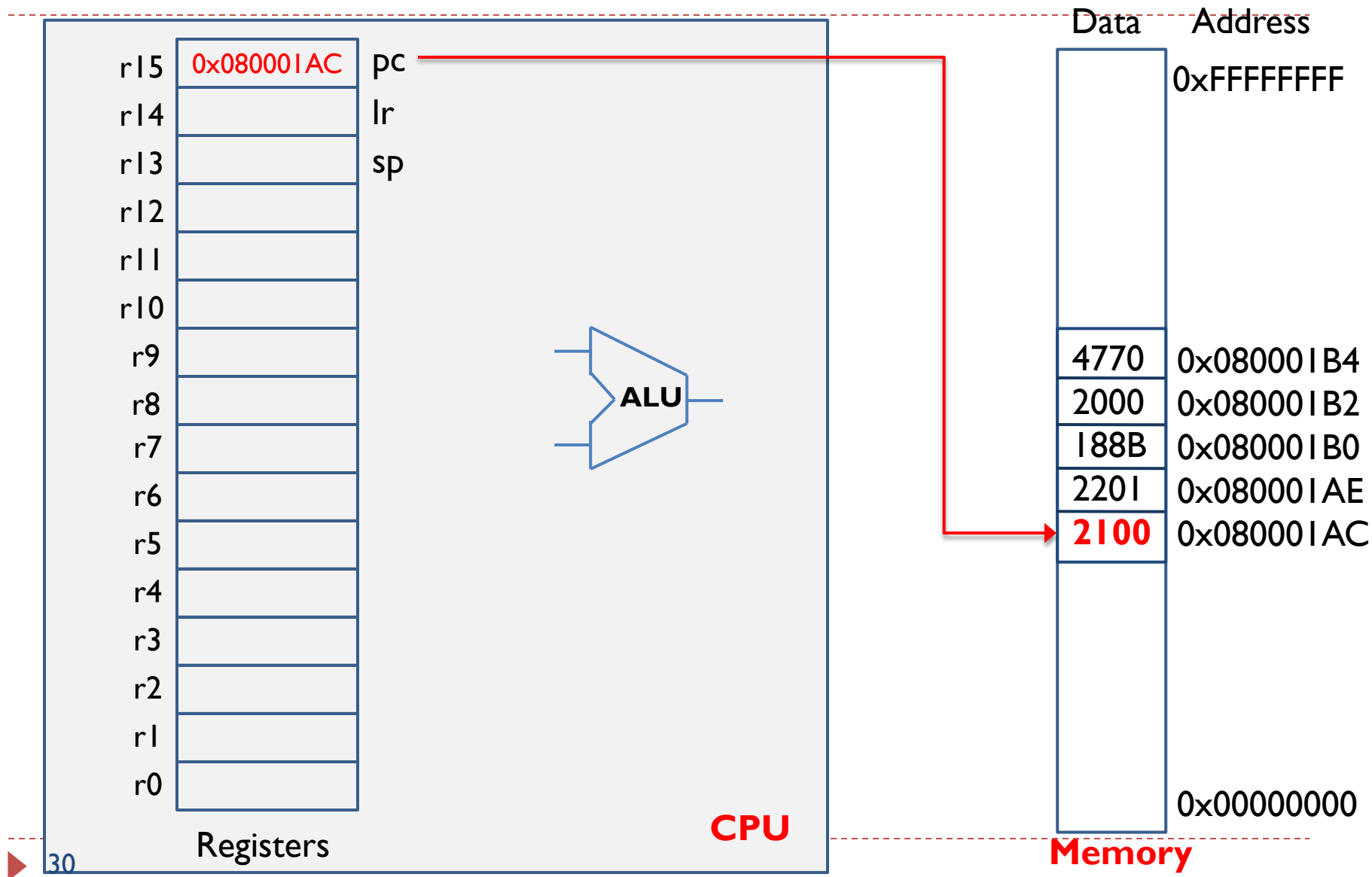
Pipeline of 16-bit instructions

Machine codes are stored in memory



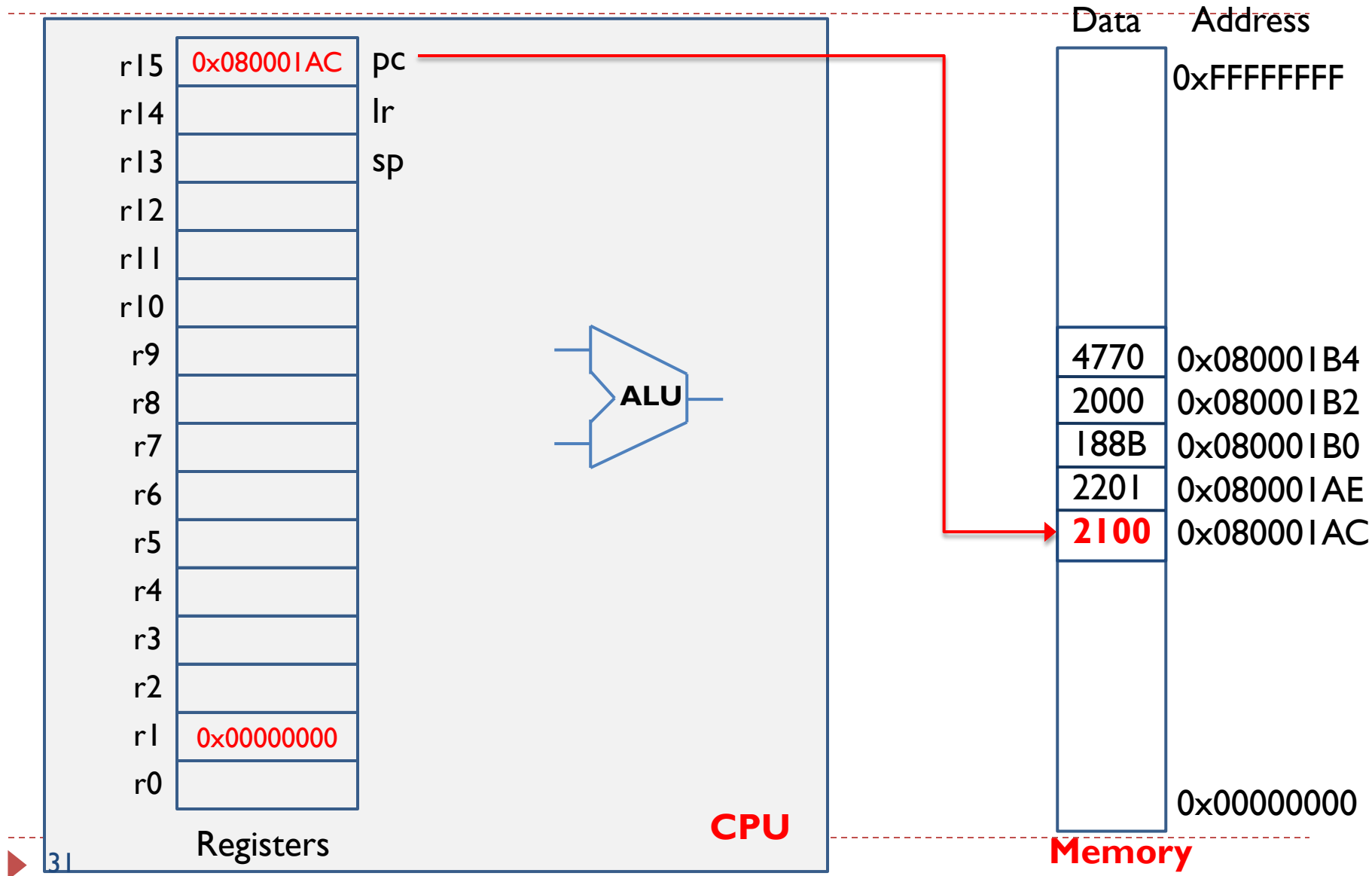
Fetch Instruction: pc = 0x08001AC

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



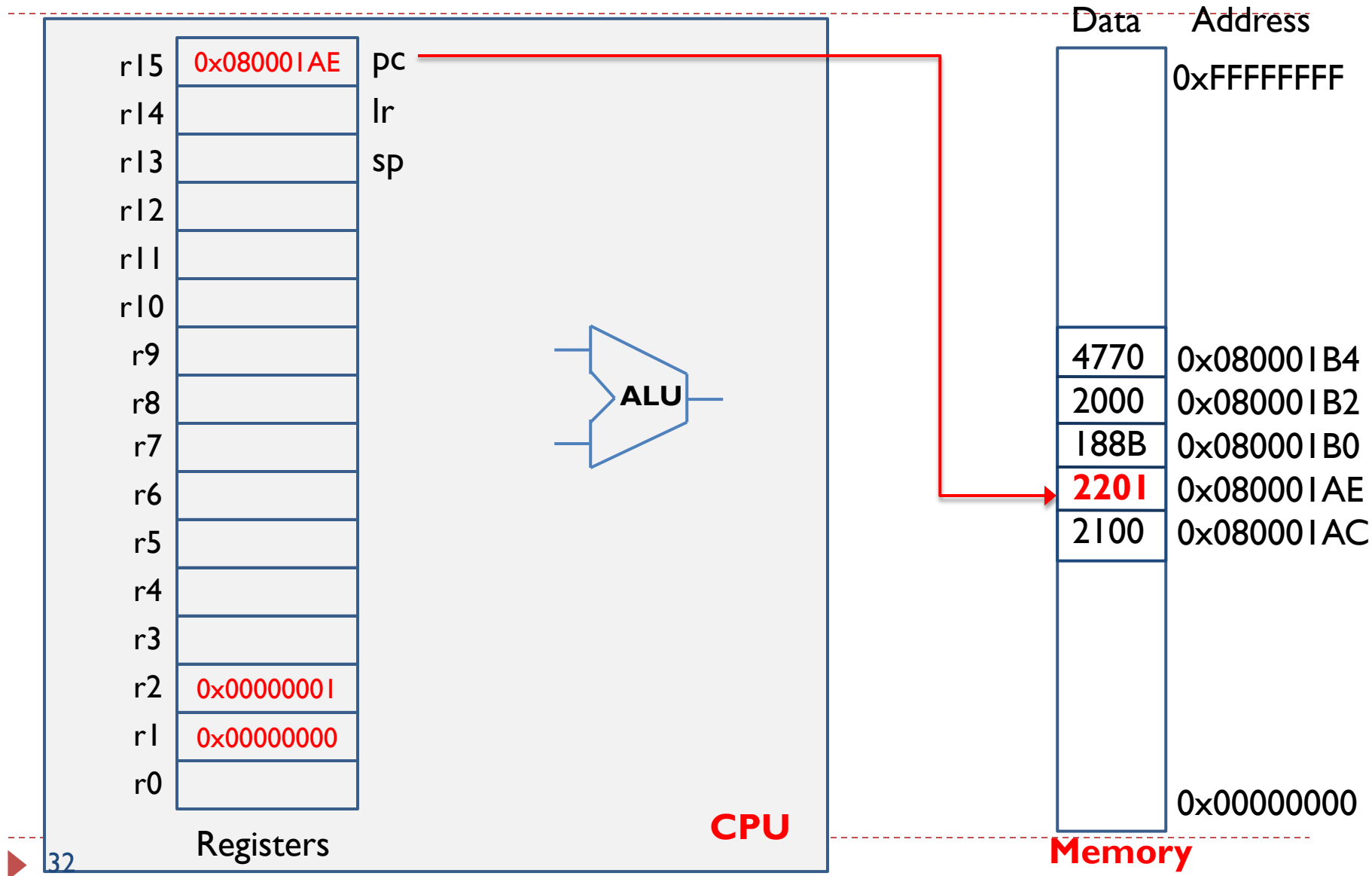
Execute Instruction:

MOVS r1, #0x00



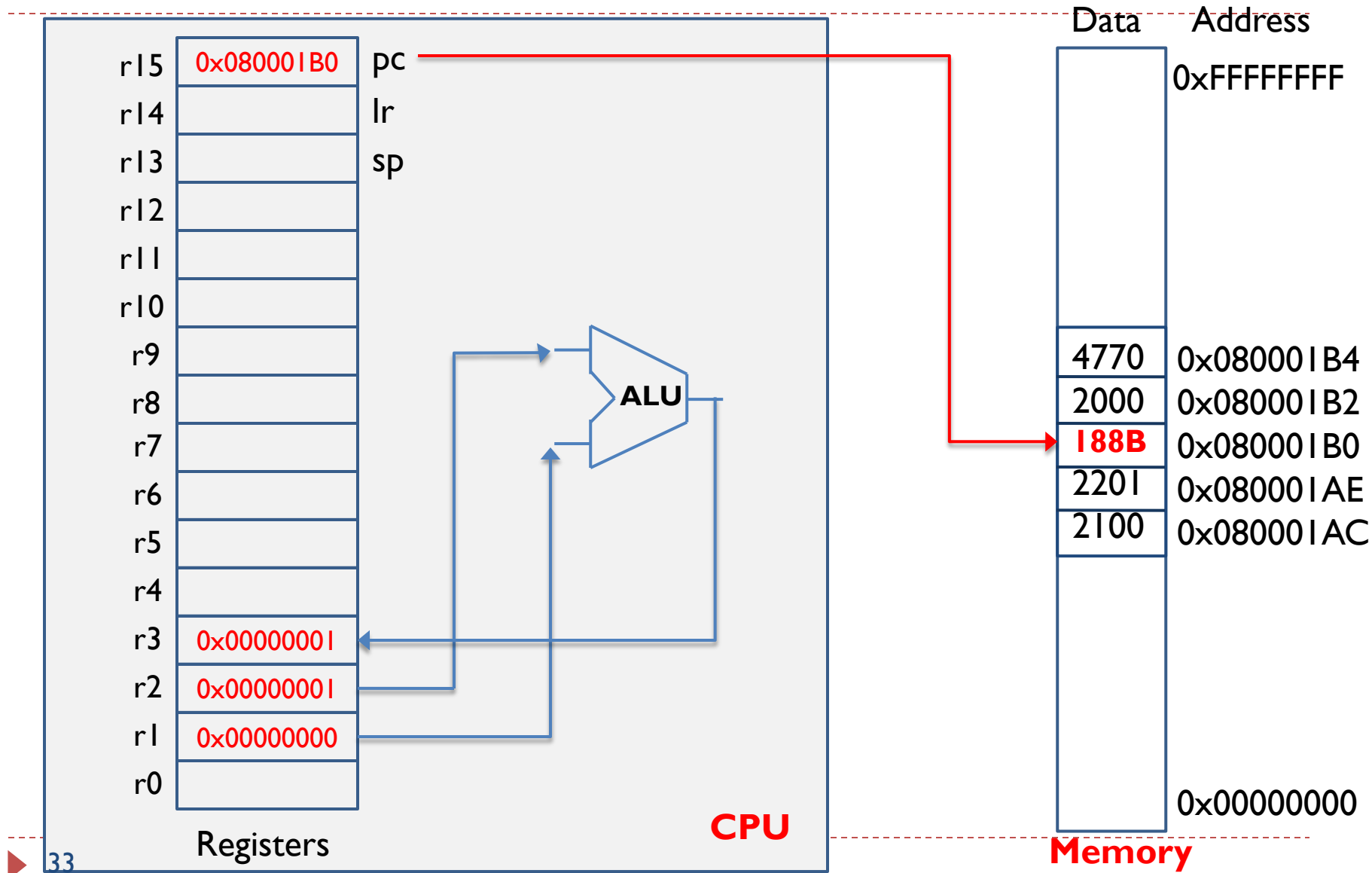
Fetch Next Instruction: $pc = pc + 2$

Decode & Execute: 2201 = **MOVS r2, #0x01**



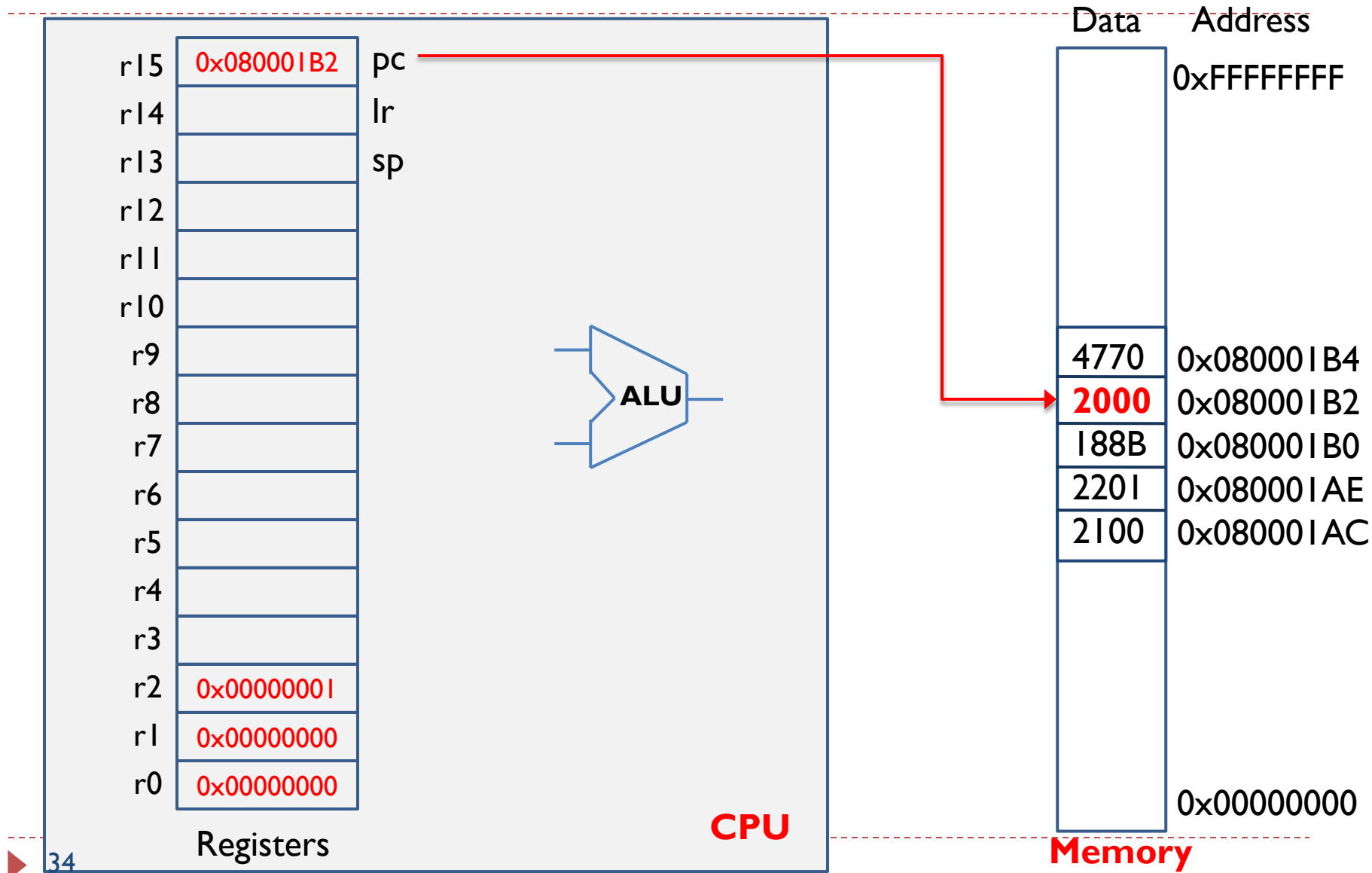
Fetch Next Instruction: $pc = pc + 2$

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



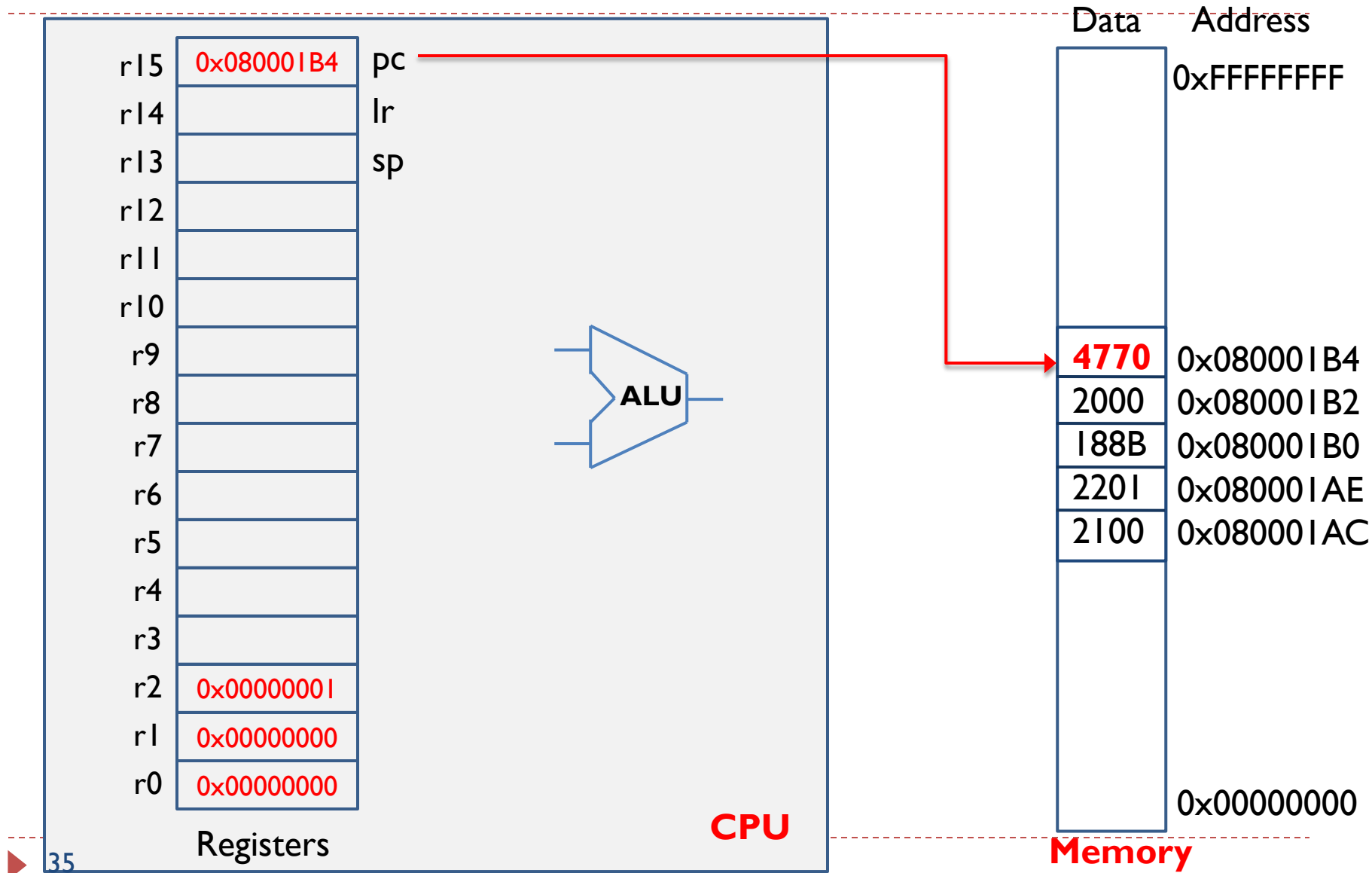
Fetch Next Instruction: $pc = pc + 2$

Decode & Execute: 2000 = **MOVS r0, #0x00**



Fetch Next Instruction: $pc = pc + 2$

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



Example:

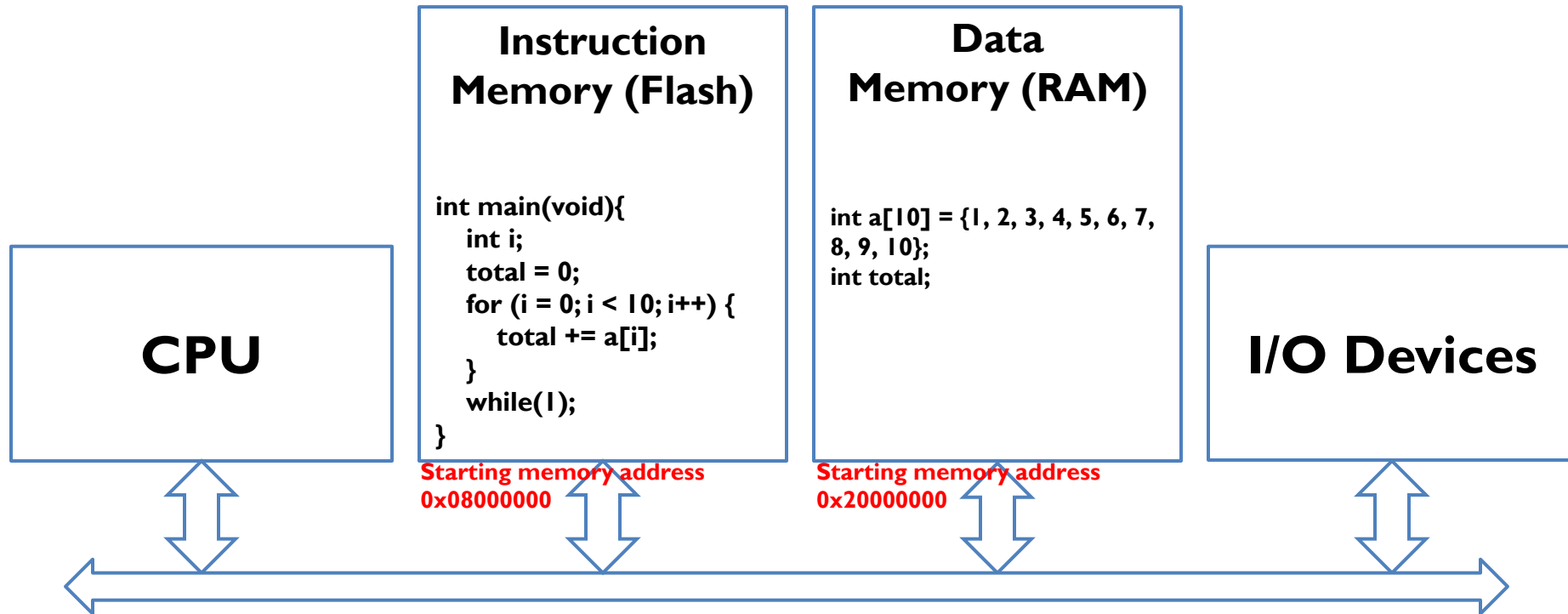
Calculate the Sum of an Array

```
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

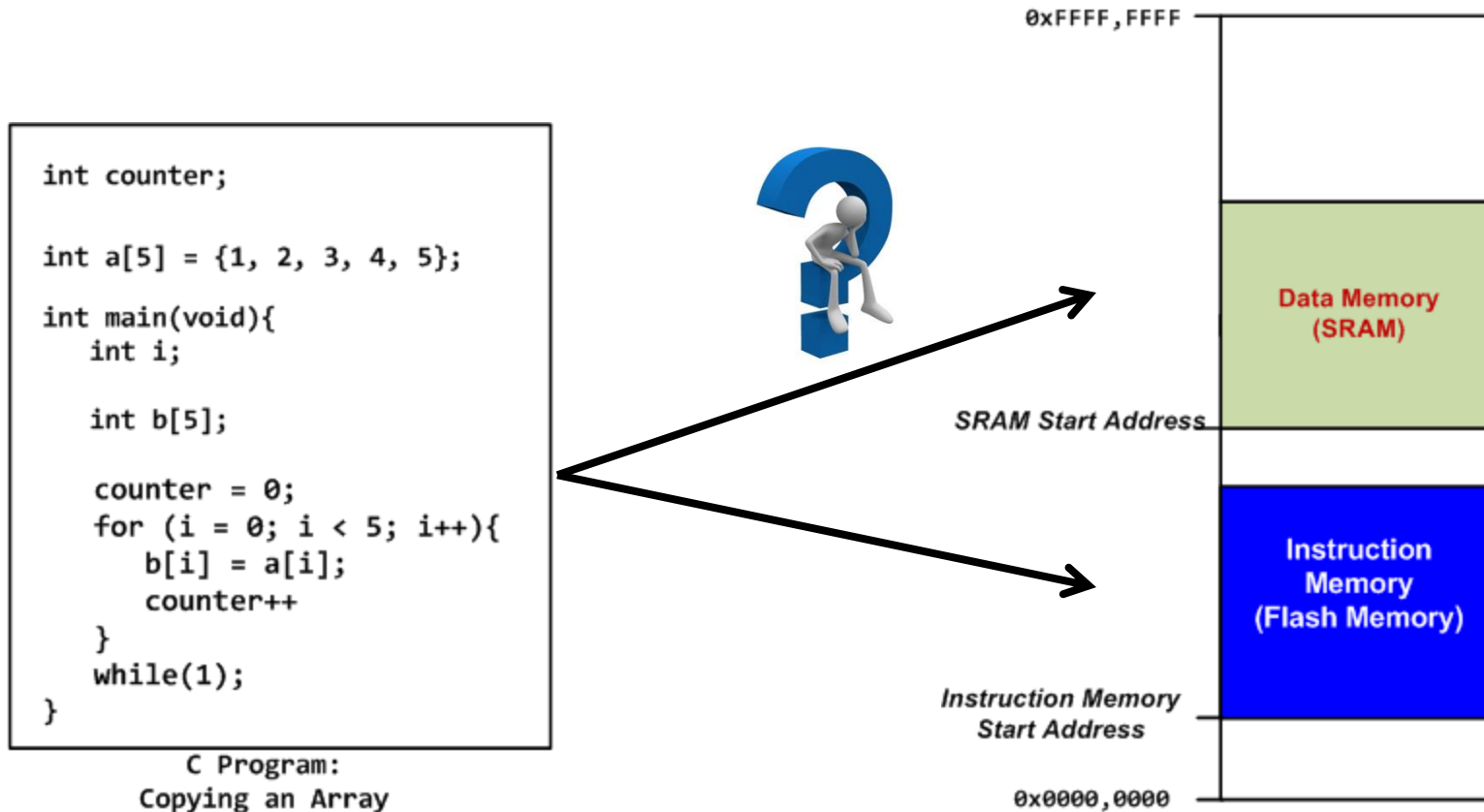
0x20000000	0x0001	a[0] = 0x00000001
0x20000002	0x0000	
0x20000004	0x0002	a[1] = 0x00000002
0x20000006	0x0000	
0x20000008	0x0003	a[2] = 0x00000003
0x2000000A	0x0000	
0x2000000C	0x0004	a[3] = 0x00000004
0x2000000E	0x0000	
0x20000010	0x0005	a[4] = 0x00000005
0x20000012	0x0000	
0x20000014	0x0006	a[5] = 0x00000006
0x20000016	0x0000	
0x20000018	0x0007	a[6] = 0x00000007
0x2000001A	0x0000	
0x2000001C	0x0008	a[7] = 0x00000008
0x2000001E	0x0000	
0x20000020	0x0009	a[8] = 0x00000009
0x20000022	0x0000	
0x20000024	0x000A	a[9] = 0x0000000A
0x20000026	0x0000	
0x20000028	0x0000	total= 0x00000000
0x2000002A	0x0000	

Memory
address
in bytes

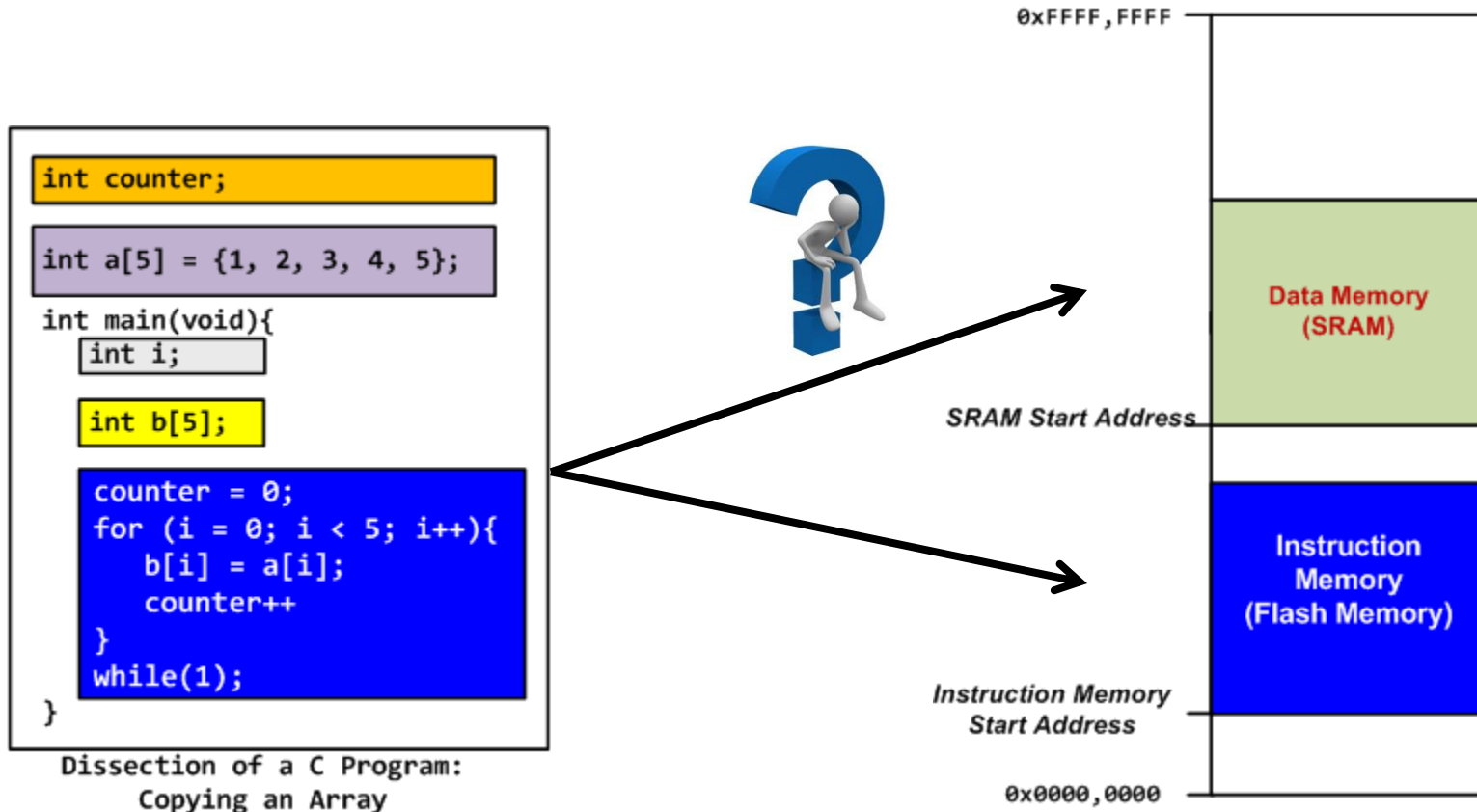
Memory
content



Loading Code and Data into Memory

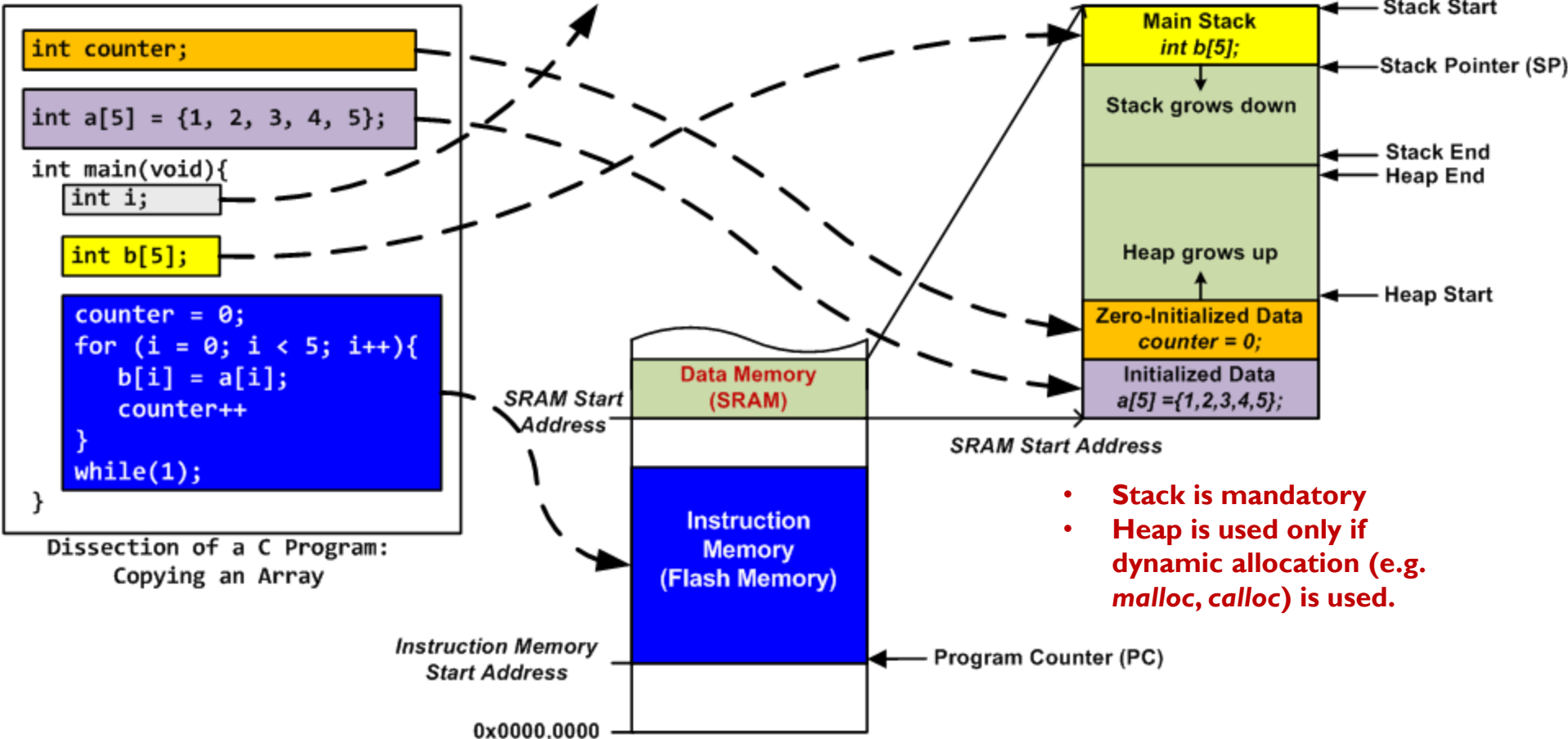


Loading Code and Data into Memory

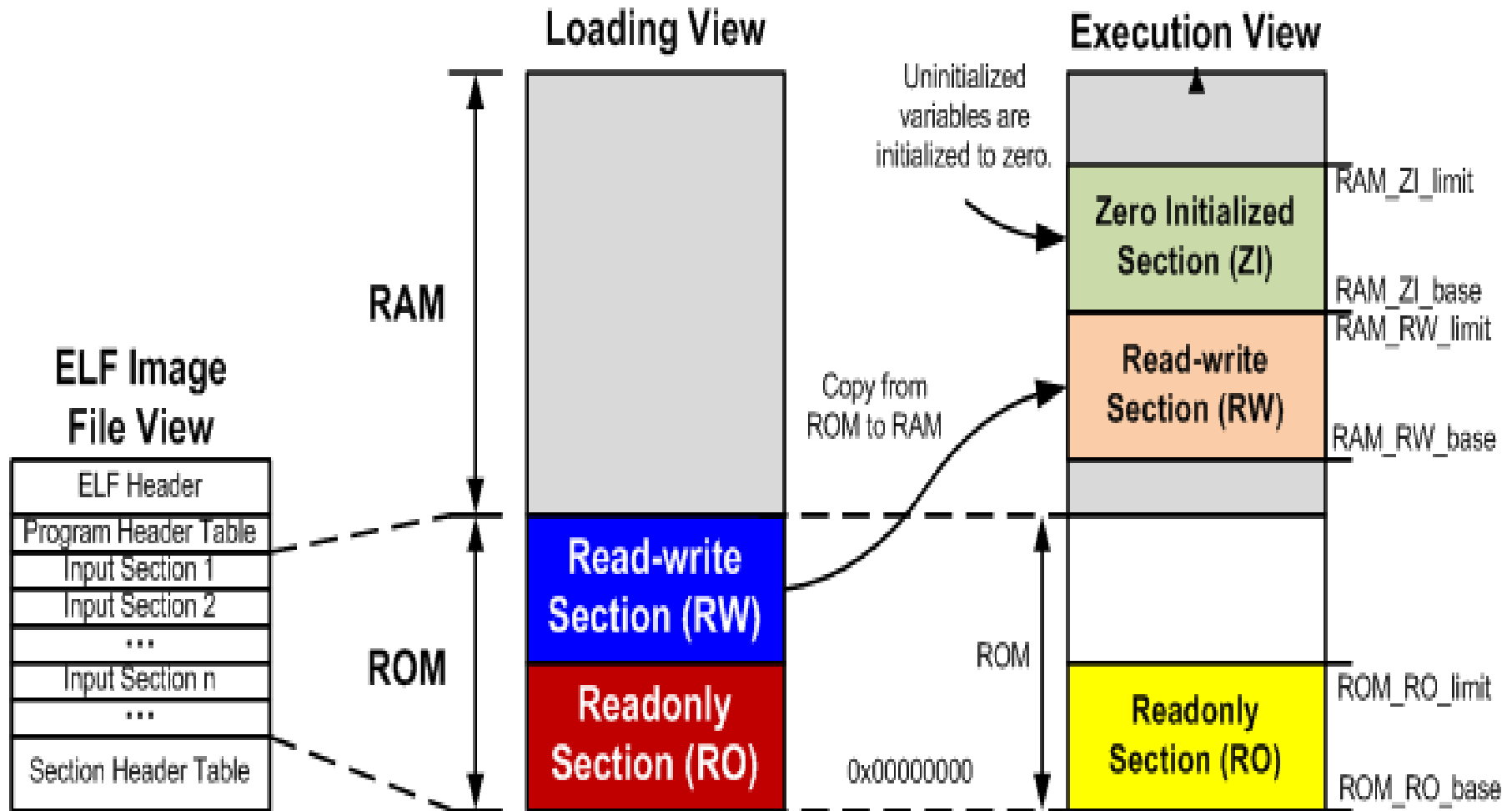


Loading Code and Data into Memory

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



View of a Binary Program



STM32L476 Discovery – HMI

Integrated ST-Link/V2-1 (for programming and debugging)

LCD 96 segments

Motion Mems (9-axis)

push buttons and joystick,
2 color LEDs

Quad SPI NOR Flash
16 MB

USB OTG connector

from st.com

STM32L476 Discovery - Audio and connector

APC connector (for Apple connector)

MFX to auto-measure power consumption

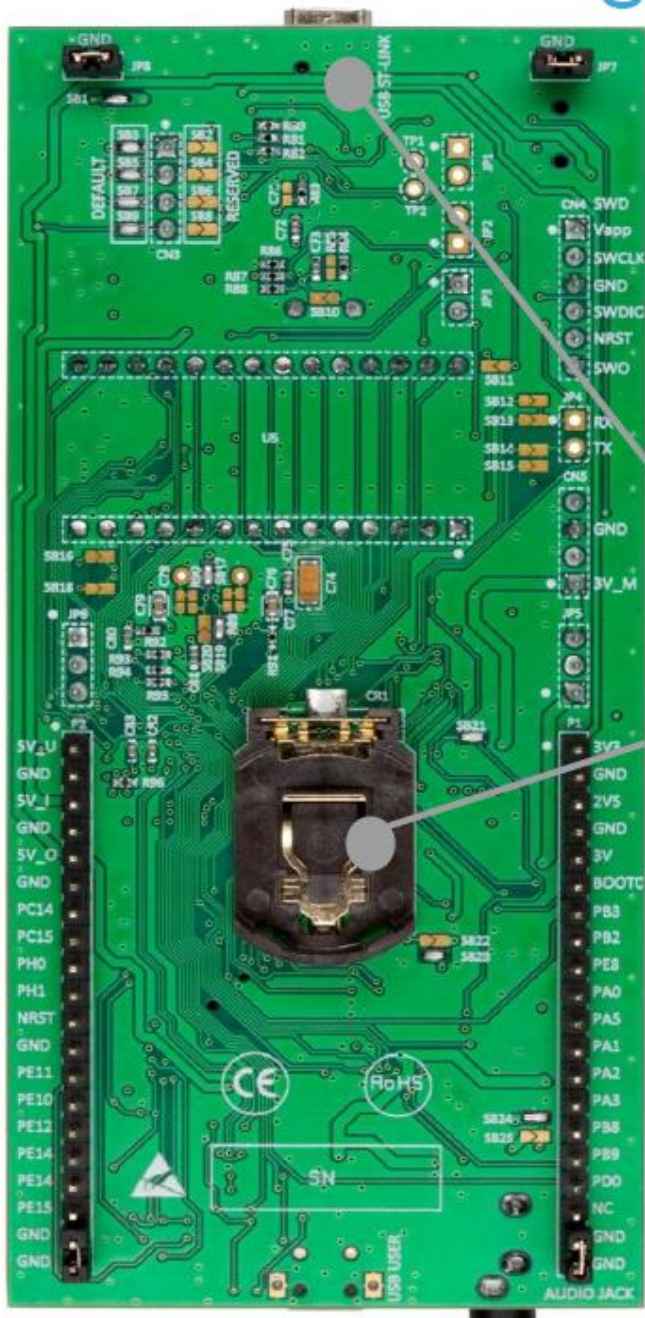
Direct access to all MCU I/Os

Audio Codec and 3.5 mm connector

Microphone Mems



STM32L476 Discovery - back side



Flexible board power supply
CR2032 battery or USB

from st.com

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

Memory Map

