Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C

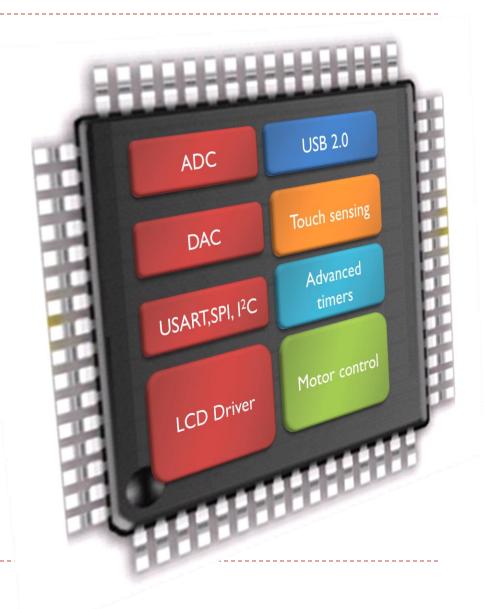
Chapter I Computer and Assembly Language

Zonghua Gu

Fall 2025

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

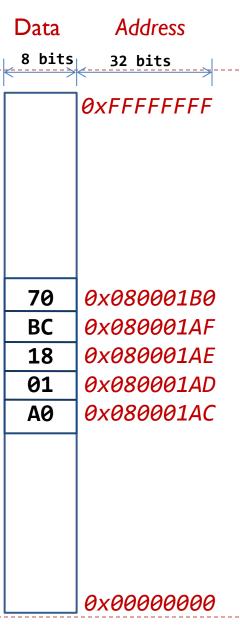


Embedded Systems



Memory

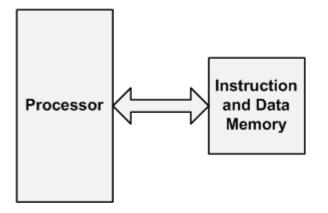
- Memory is arranged as a series of "locations"
 - ► Each location has a unique "address"
 - ► Each location holds a byte (byte-addressable)
 - e.g. the memory location at address 0x080001B0 contains the byte value 0x70, i.e., 112
- The number of locations in memory is limited
 - e.g. 4 GB of RAM
 - ▶ 1 Gigabyte (GB) = 2^{30} bytes
 - ▶ 2³² 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

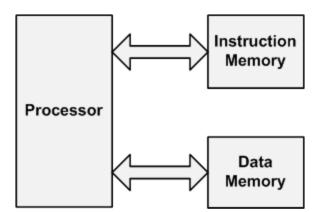
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

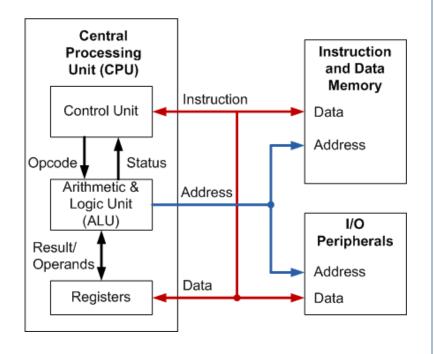
Data and instructions are stored into separate memories.



Computer Architecture

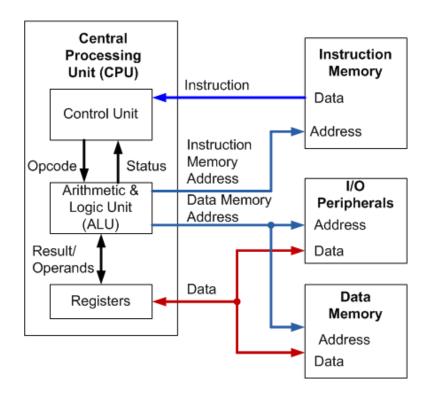
Von-Neumann

Instructions and data are stored in the same memory.



Harvard

Data and instructions are stored into separate memories.



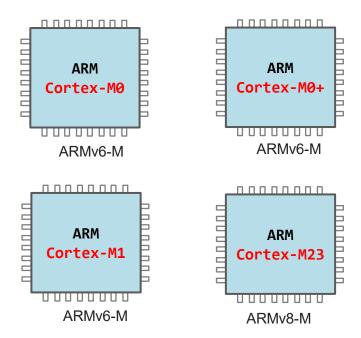
von Neumann vs. Harvard

- 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

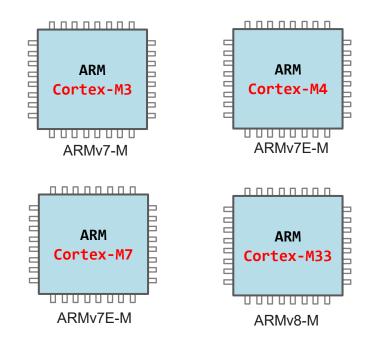
Von-Neumann

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
           check
      ADD r1, r1, r0
loop
      ADDS r0, r0, #1
check CMP
           r0, #10
           loop
self
           self
```

Assemble

Machine Program

▶ 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
```

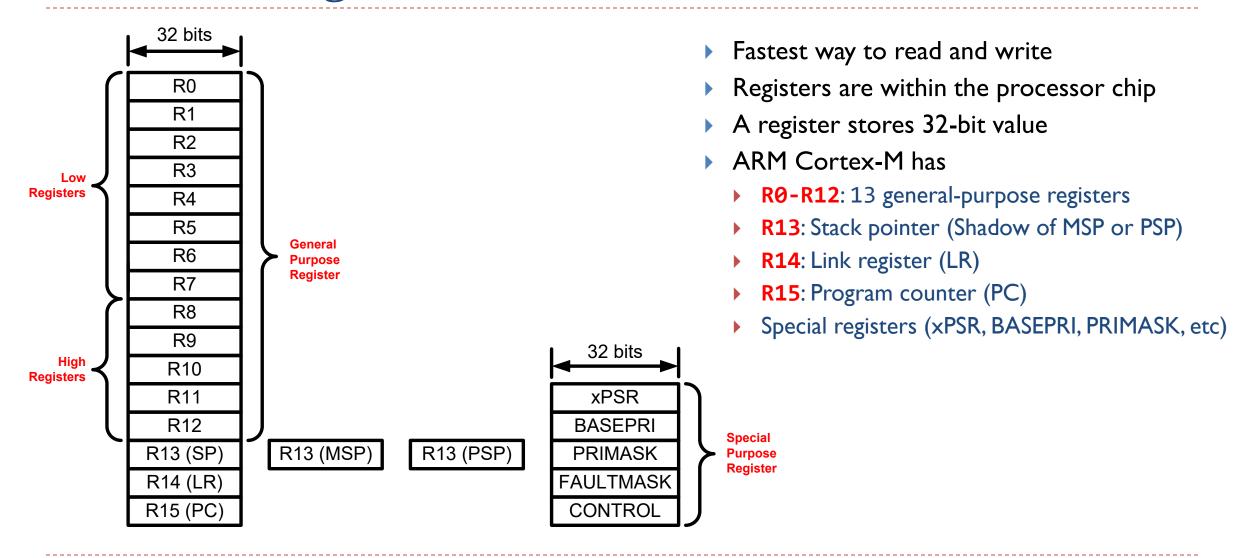
Machine Code

```
In Binary
```

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

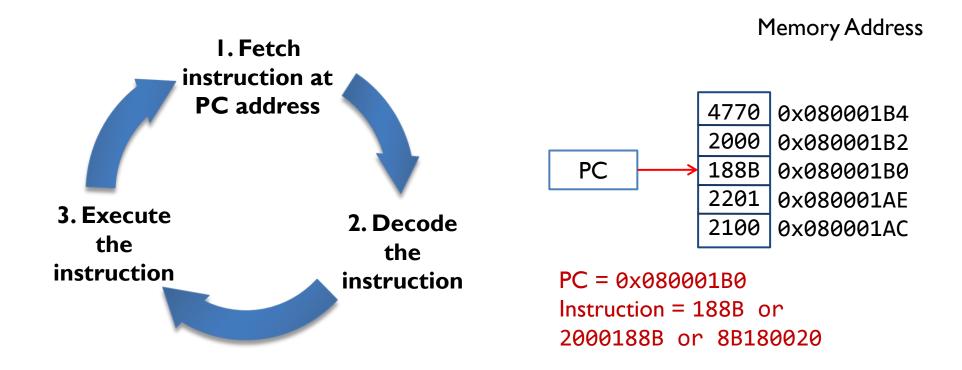
```
; MOVS r1, #0x00
; MOVS r2, #0x01
; ADDS r3, r1, r2
; MOVS r0, #0x00
; BX lr
```

Processor Registers



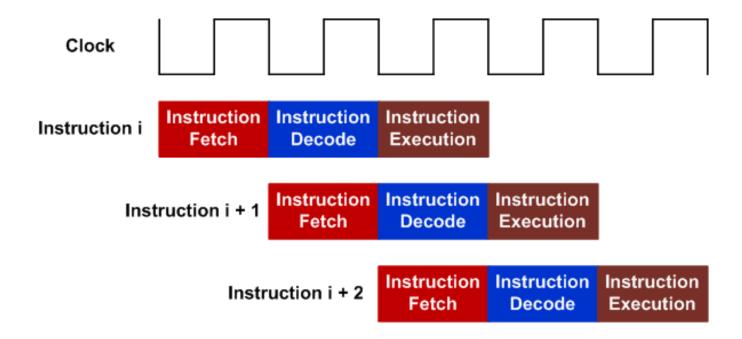
Program Execution

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

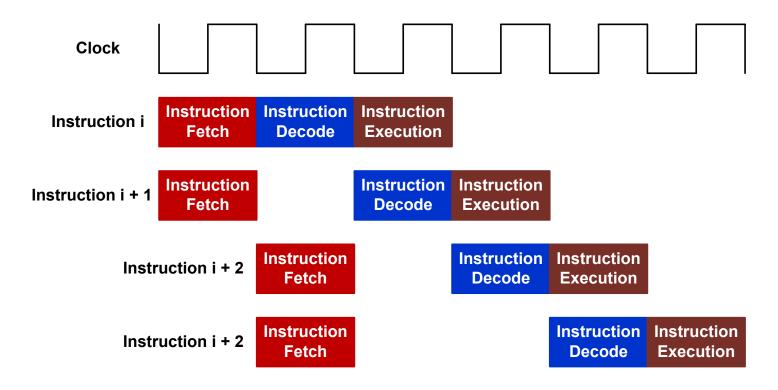
- Pipelining allows hardware resources to be fully utilized
- One 32-bit instruction or two 16-bit instructions can be fetched.



Pipeline of 32-bit instructions

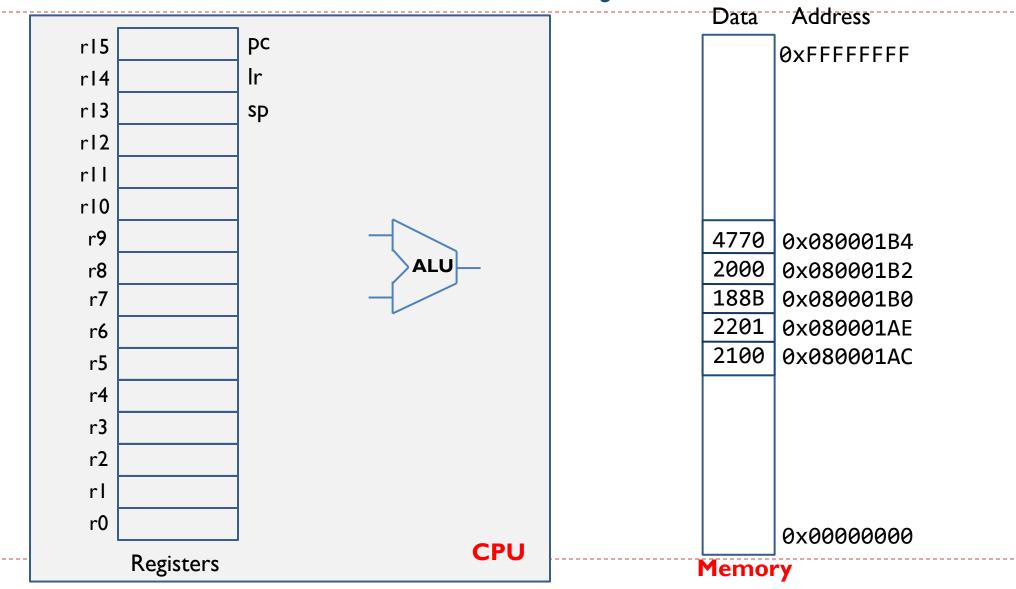
Three-state pipeline: Fetch, Decode, Execution

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



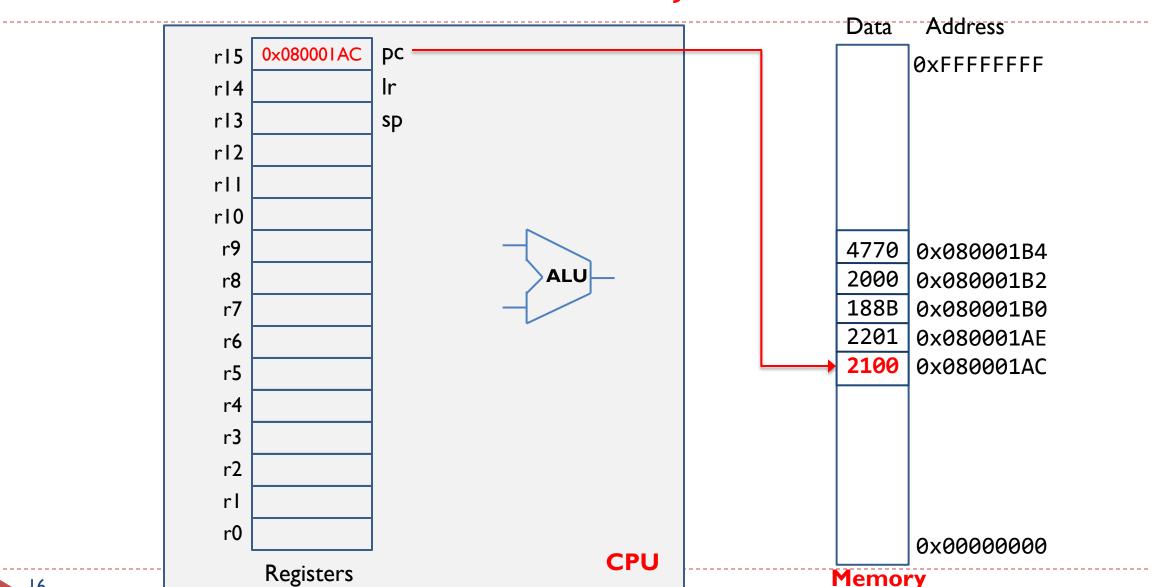
Pipeline of 16-bit instructions

Machine codes are stored in memory



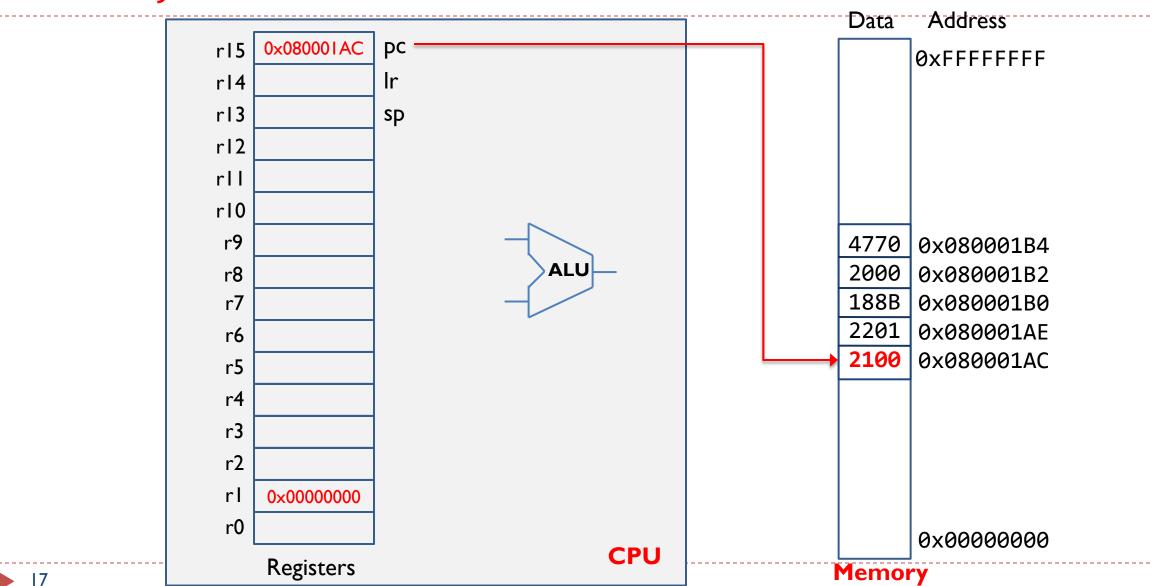
Fetch Instruction: pc = 0x08001ACDecode Instruction: 2100 = MOVS r1, #0x00

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



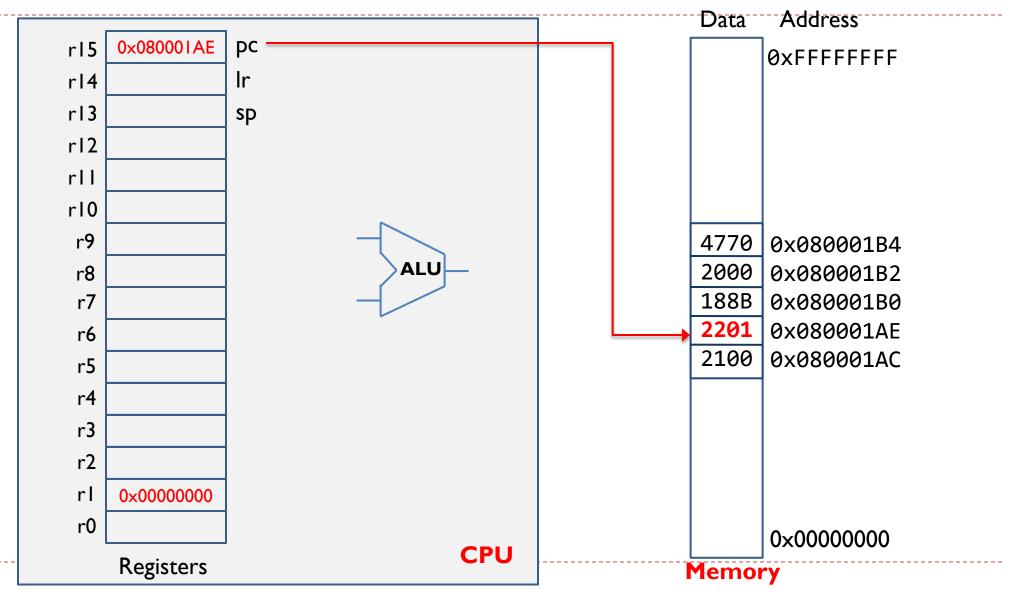
Execute Instruction:

MOVS r1, #0x00

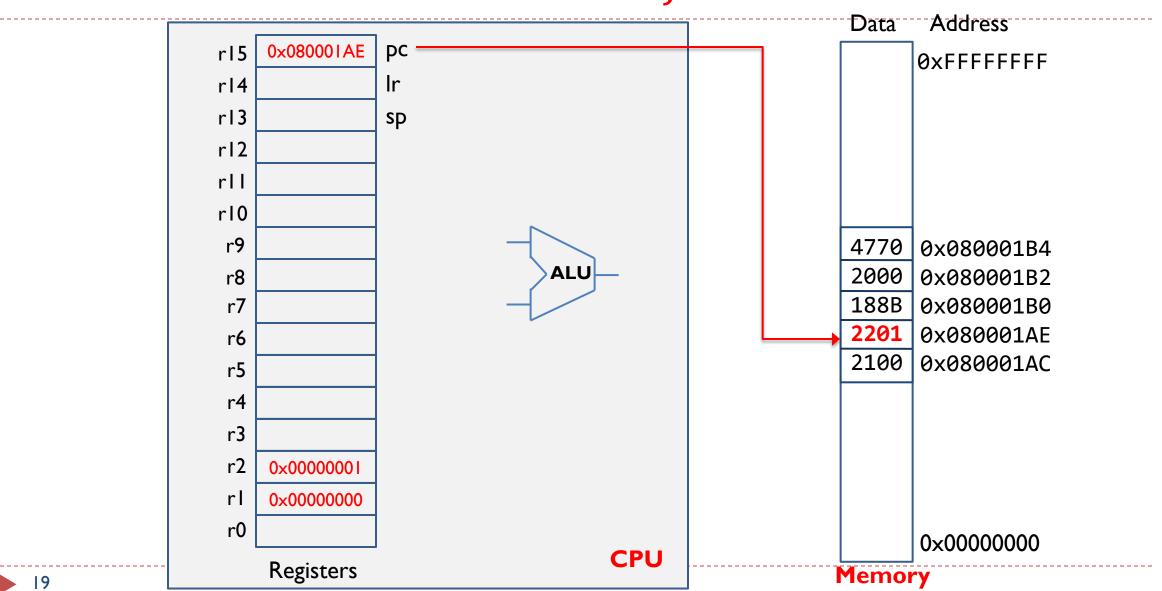


Fetch Next Instruction: pc = pc + 2

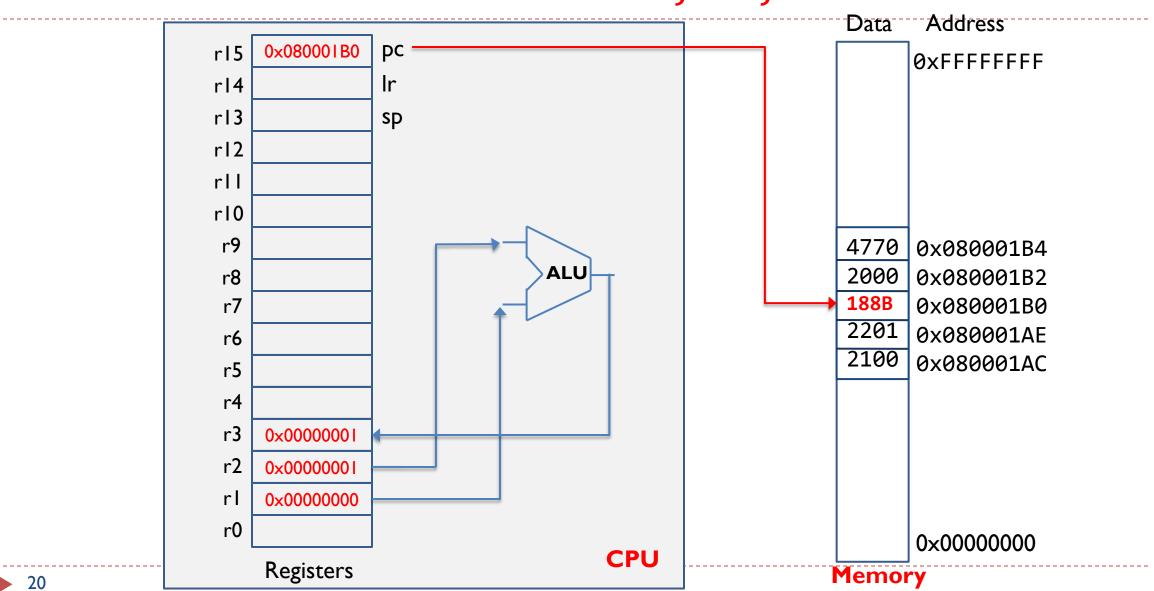
- Thumb-2 consists of a mix of 16- & 32-bit instructions
- In reality, we always fetch 4 bytes from the instruction memory (either one 32-bit instruction or two 16-bit instructions)
- To simplify the demo, we assume we only fetch 2 bytes from the instruction memory in this example.



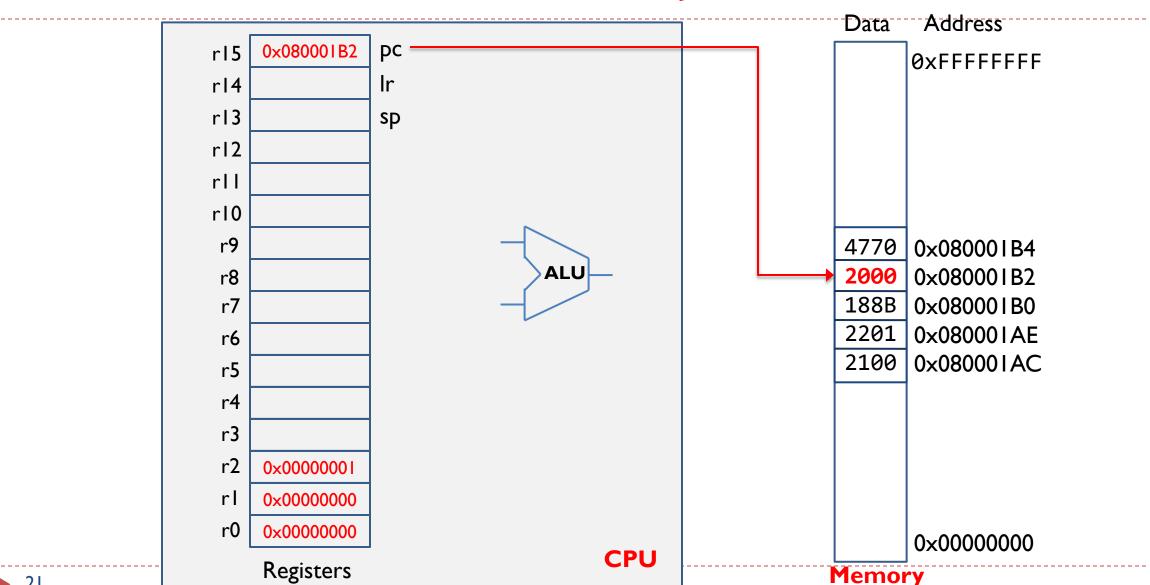
Fetch Next Instruction: pc = pc + 2 Decode & Execute: 2201 = MOVS r2, #0x01



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

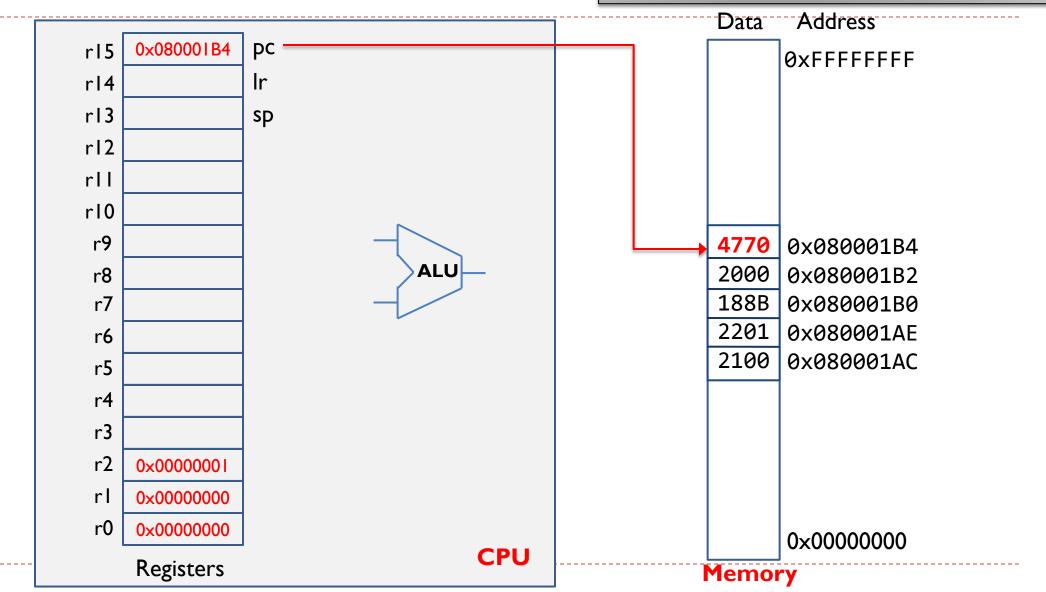


Fetch Next Instruction: pc = pc + 2Decode & Execute: 2000 = MOVS r0, #0x00



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

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



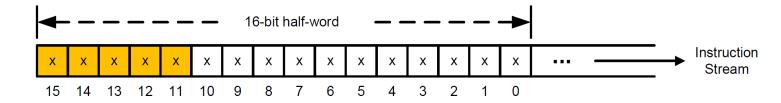
Realities

- 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



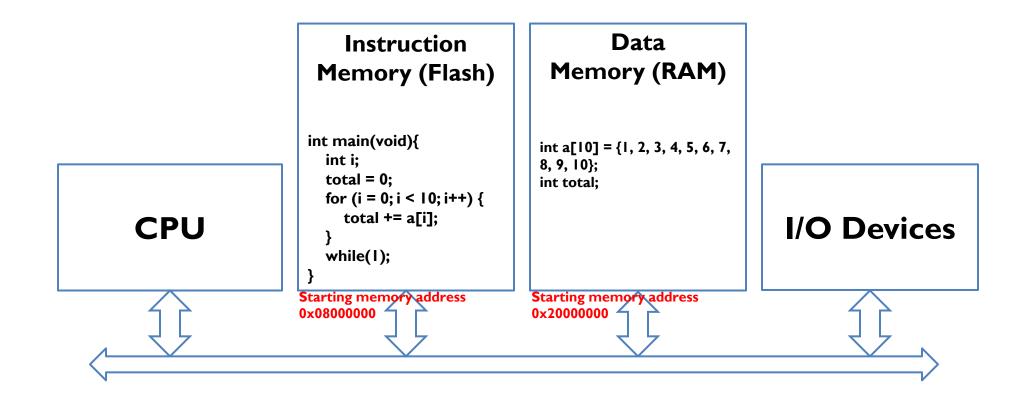
If bit [15-11] = 11101, 11110, or 11111, then, it is the first half-word of a 32-bit instruction. Otherwise, it is a 16-bit instruction.

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);
}</pre>
```

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);
}</pre>
```

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 rI,#0x00
     LDR r2, = total addr
     STR
           rl,[r2,#0x00]
     MOVS r0,#0x00
           Check
Loop: LDR rl, = a_addr
           rl, [rl, r0, LSL #2]
     LDR
     LDR r2, = total addr
     LDR r2, [r2, #0x00]
     ADD
           rl, rl, r2
     LDR r2, = total addr
     STR
           rl, [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

0x20000054 0x00000000 0x20000050 0x00000000 0x2000004C 0x00000000 0x20000048 0x00000000 0x20000044 0x00000000 0x20000040 0x00000000 0x2000003C 0x00000000 0x20000038 0x00000000 0x20000034 0x00000000 0x20000030 0x00000000 0x2000002C 0x00000000 0x20000028 0x00000000 0x20000024 A0000000A 0x20000020 0x00000009 0x2000001C 0x00000008 0x20000018 0x00000007 0x20000014 0x00000006 0x20000010 0x00000005 0x2000000C 0x00000004 0x20000008 0x00000003 0x20000004 0x00000002 0x20000000 0x00000001 **Memory**

a[2] = 0x000000003 a[1] = 0x000000002 a[0] = 0x00000001

total= 0x00000000

a[9] = 0x00000000A

a[8] = 0x00000009

a[7] = 0x00000008

a[6] = 0x00000007

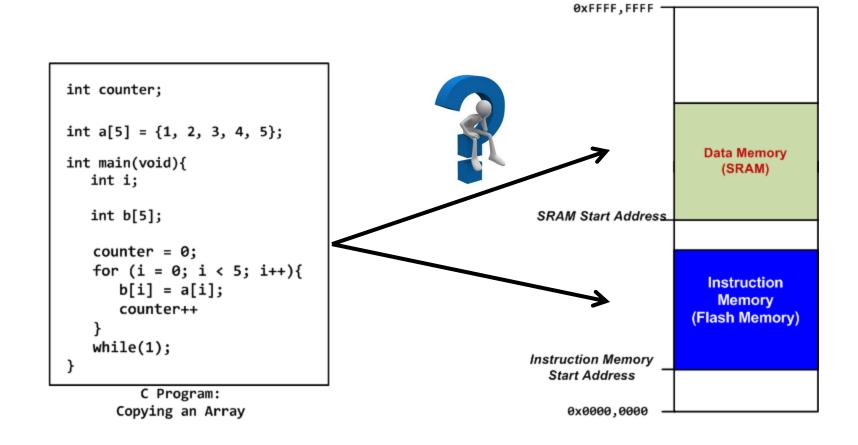
a[5] = 0x00000006

a[4] = 0x000000005

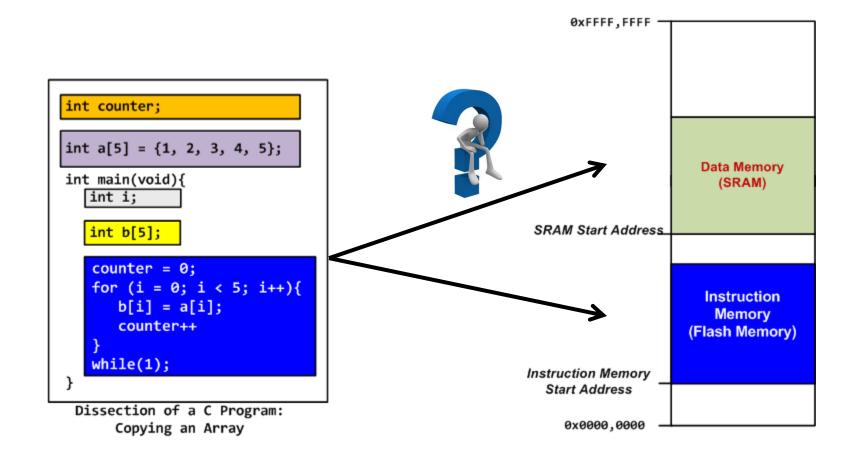
a[3] = 0x00000004

address Content in bytes

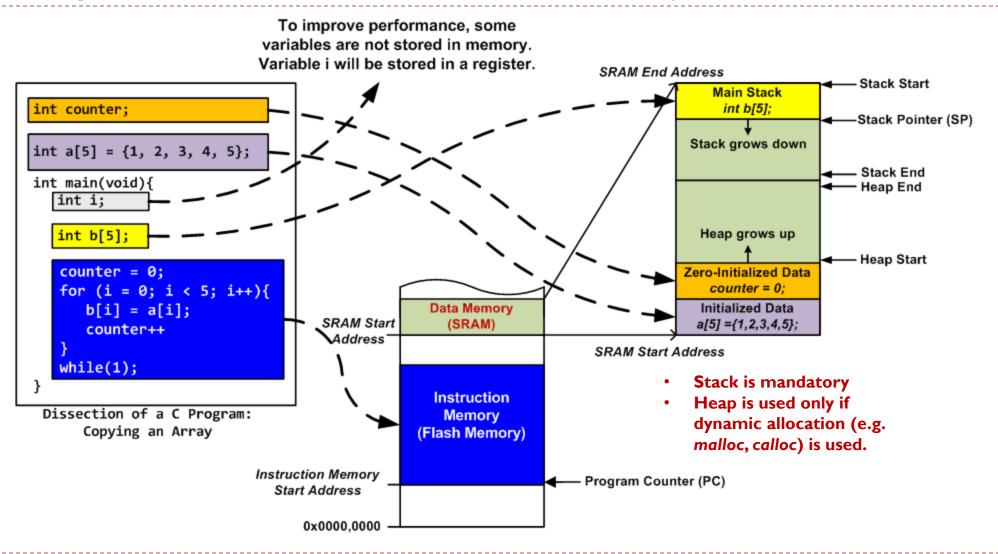
Loading Code and Data into Memory



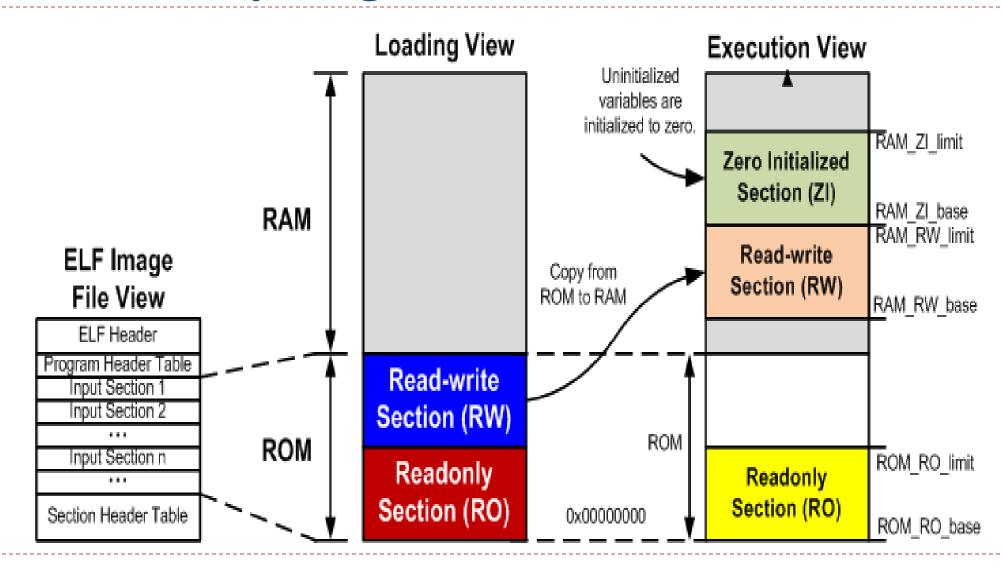
Loading Code and Data into Memory



Loading Code and Data into Memory



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

