EE 186: Introduction to Embedded Systems

Lecture 2

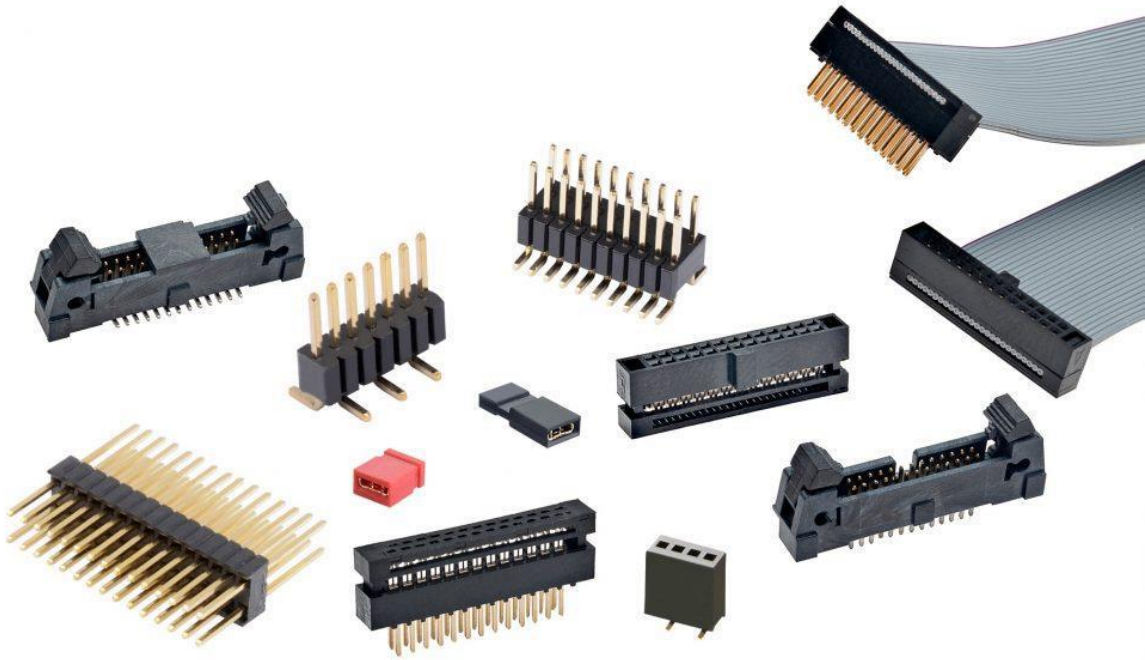
Zerina Kapetanovic

Announcements

- HW 1 released today
- We are handing out lab kits today and tomorrow
- Midterm moved from Tuesday to Thursday

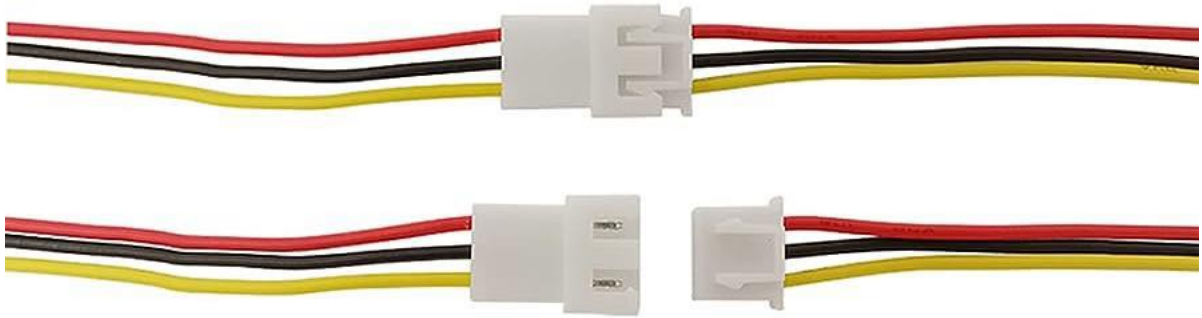
Component of the day...

Connectors and header pins!



Many different types

- Shroud
- Keyed
- Locking
- Jumper
- Size/Spacing
- Right angle
- Through hole
- Surface mount



Last class…

- What is an embedded system?
- Course overview
- High-level overview of ARM System Architecture
- Refresher: Number Systems
- GPIO and Registers Primer

What are we learning about today?

- Instruction Set Architecture (ISA)
- Memory Organization
- Processor Registers and the Application Binary Interface (ABI)
- Loading Code and Data into Memory
- Instruction (Assembly)
  - Arithmetic and Logic Instructions
  - Bitwise Logic
  - Move, Loads, and Stores
  - Flow Control and Branching
- Indexing Problem

Our goal

To understand how the Instruction Set Architecture (ISA), memory, and registers work together

How does the microcontroller interact with the outside world via Memory Mapped Input/Output (MMIO)

Quick Recap

Many manufacturers ship ARM products ...

arm

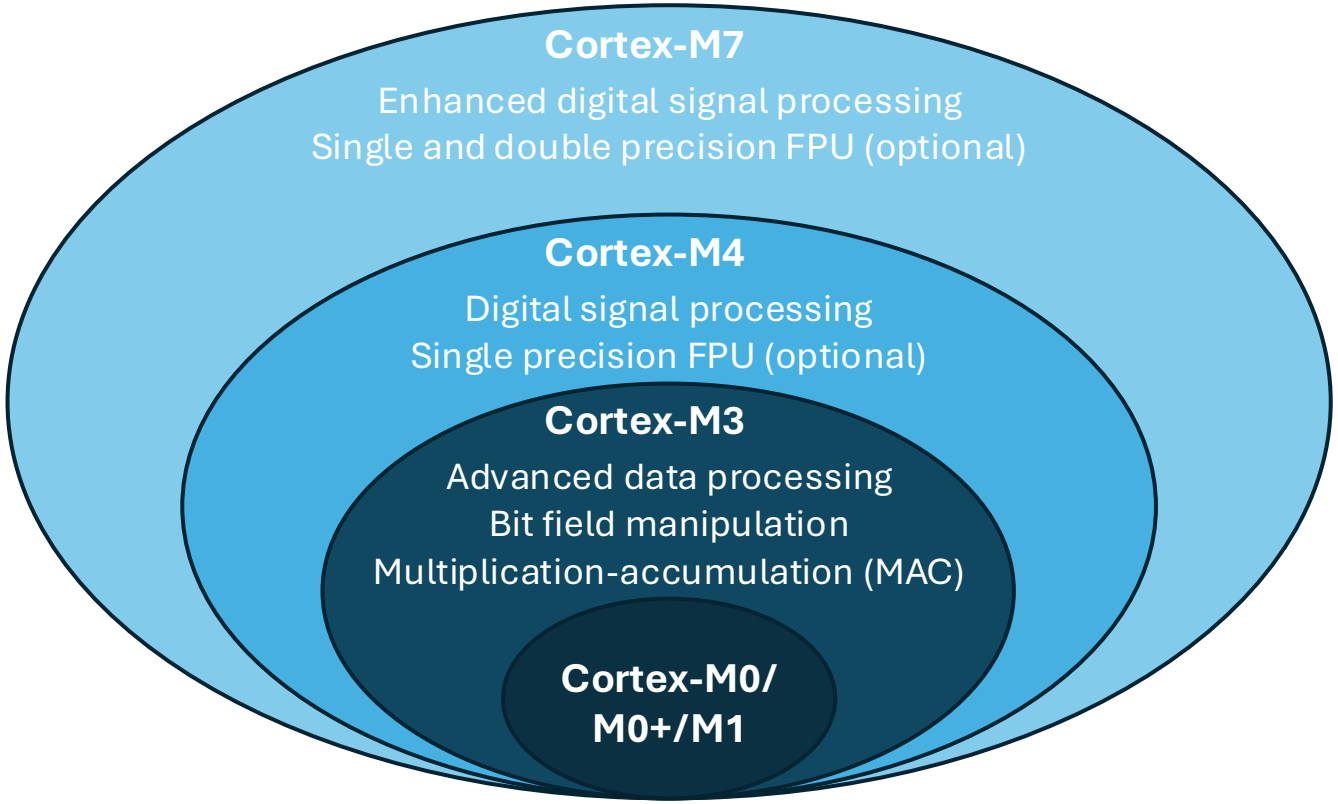
ARMCortex Processors

ARM Cortex-A: application processor cores for performance-intensive systems (e.g., smartphones)

ARM Cortex-R: high-performance cores for real-time applications (e.g., cars)

ARM Cortex-M: microcontroller cores for embedded applications (e.g., IoT devices)

ARM Cortex-M Family

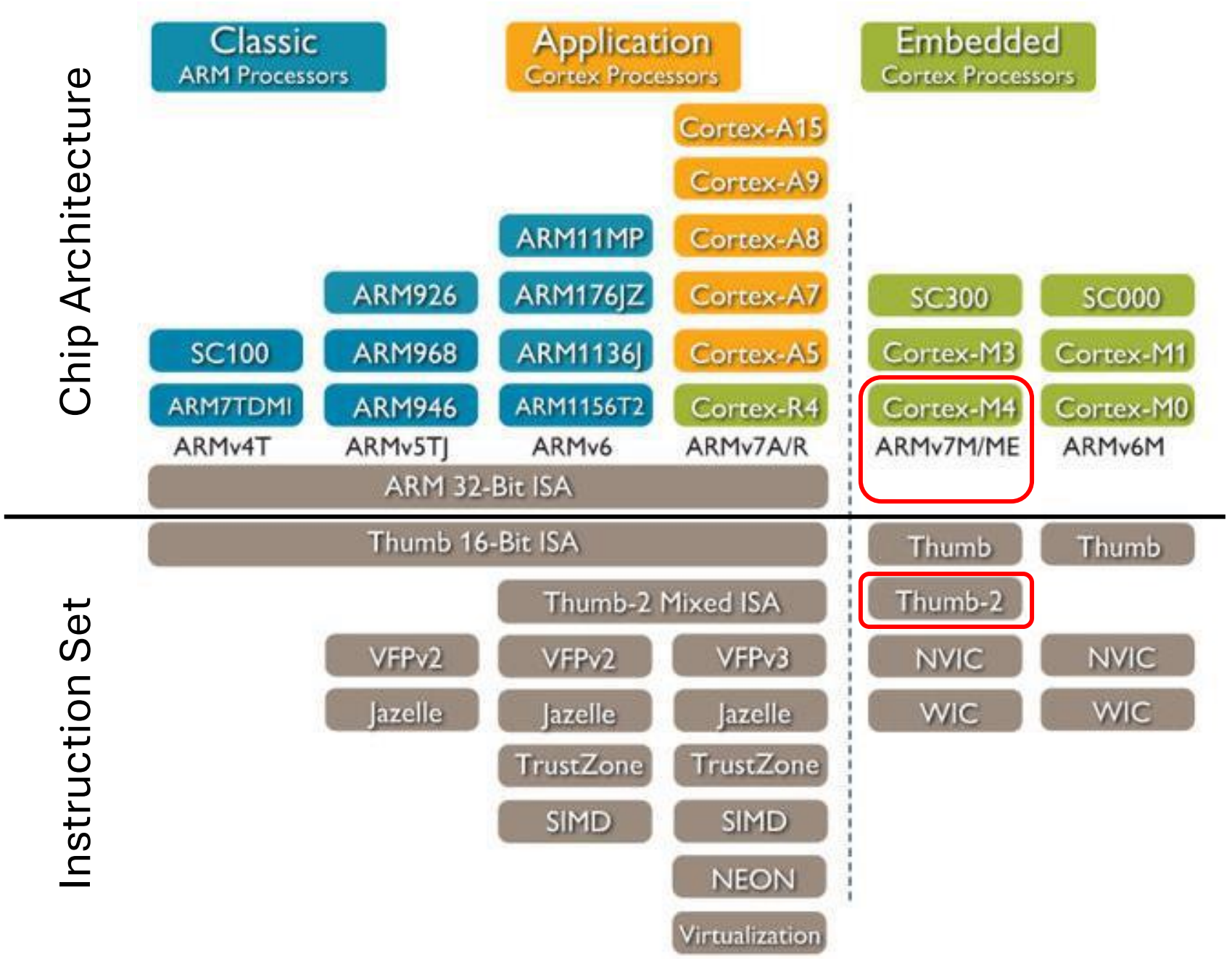


We are focusing on the **ARMv7-M architecture** and working with the **Cortex-M4**

Instruction Set Architecture (ISA)

“ The ISA acts as an interface between the hardware and the software, specifying both what the processor is capable of doing as well as how it gets done” – ARM

- It defines how software controls the CPU
- It is an agreement between the programmer and the hardware designer
- The ISA defines…
  - **Instruction:** add, subtract, move, load
  - **Data Types:** words, halfwords, signed, unsigned
  - **Data Storage:** registers, memory
  - **Addressing schema**
  - **Data flow control**



ARM processors mainly support four different assembly instruction sets:

- Thumb
- Thumb-2
- ARM32
- ARM64

ARM Processor Instructions

Thumb 16-bit ISA

- Cost – some reduction in performance
- Reduced…
  - Number of operands
  - Register range
  - Memory range
  - Range of constants (immediates)

ARM 32-bit ISA

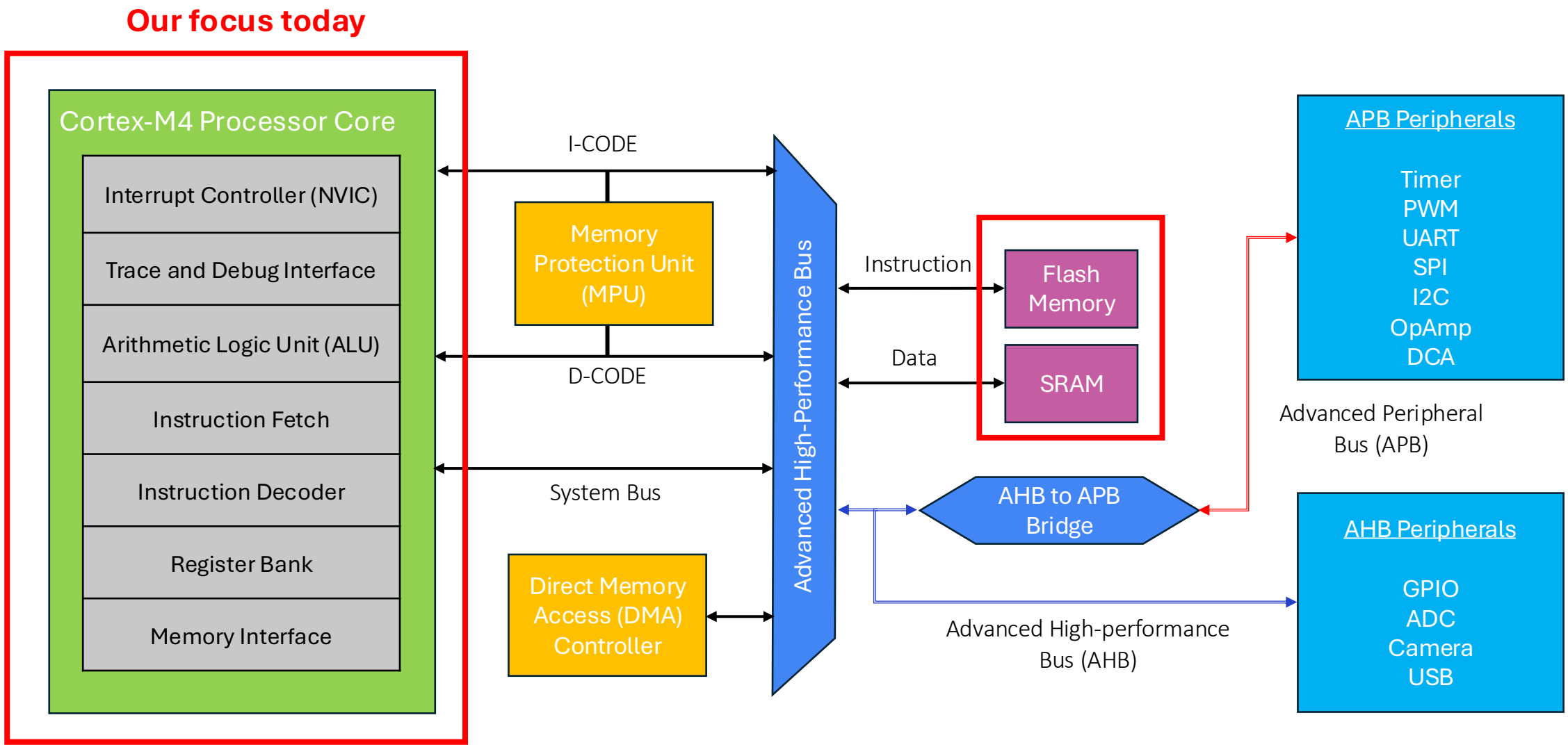
- More operands and registers
- Flexible memory range
- Runs faster because each instruction does more
  - Multiple operands in one instead of multiple instructions

Thumb-2 ISA

- Mix of both 16-bit and 32-bit instructions
- Consists of Thumb 16-bit ISA + reduced set of ARM 32-bit instructions
- Optimized tradeoff between code density and speed

? Why does reducing the size of instruction memory benefit many embedded systems?

(Simplified) ARM Cortex M4 Microcontroller

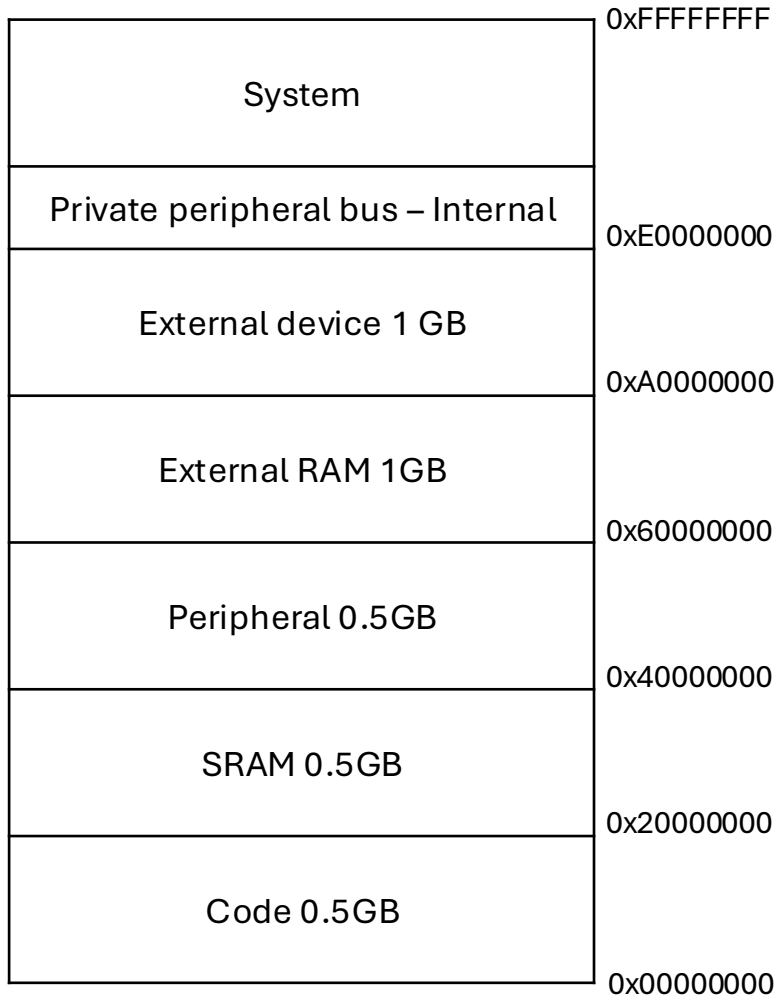


What are we learning about today?

- Instruction Set Architecture (ISA)
- **Memory Organization**
- Processor Registers and the Application Binary Interface (ABI)
- Loading Code and Data into Memory
- Instruction (Assembly)
  - Arithmetic and Logic Instructions
  - Move, Loads, and Stores
  - Flow Control and Branching
- Indexing Problem

Memory Organization

- Memory locations are limited
  - 4GB of address space
  - 1 Gigabyte (GB) = 2^{30} bytes
  - 2^{32} locations → 4,294,967,296 locations!
- Unified memory structure with designated ranges. Independent of memory type or function
- Memory regions are predefined for code interoperability.



Memory Organization

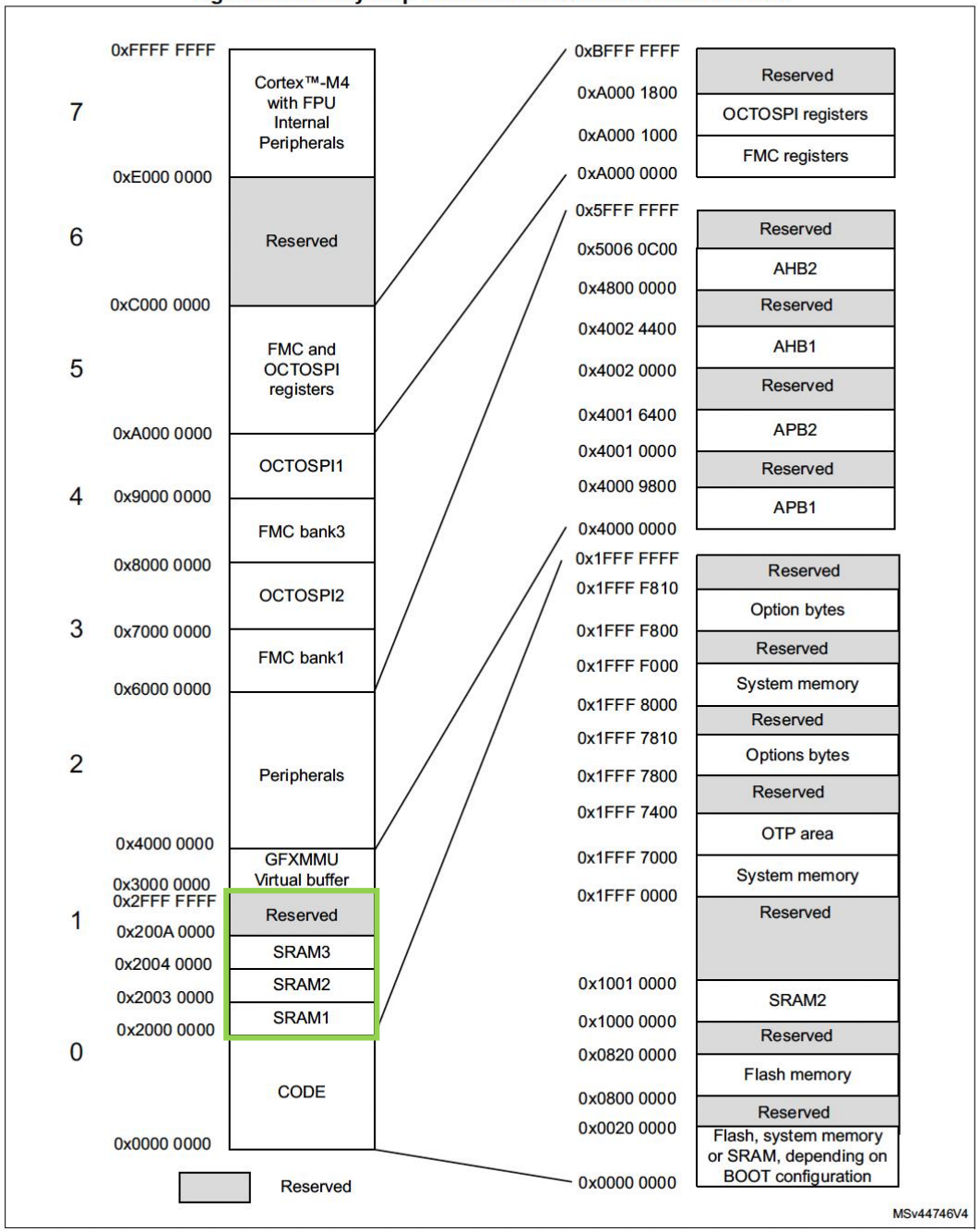
Memory map for the STM32L4R5ZI

- 4GB is the max
- ”Reserved” = **not implemented on chip**
- **Example:**
  - ARMv7-M architecture allocated 500MB of SRAM
  - The STM only instantiates 640KB SRAM on the STM32L4R5ZI

From the STM32 Reference Manual

2.2.2 **Memory map and register boundary addresses**

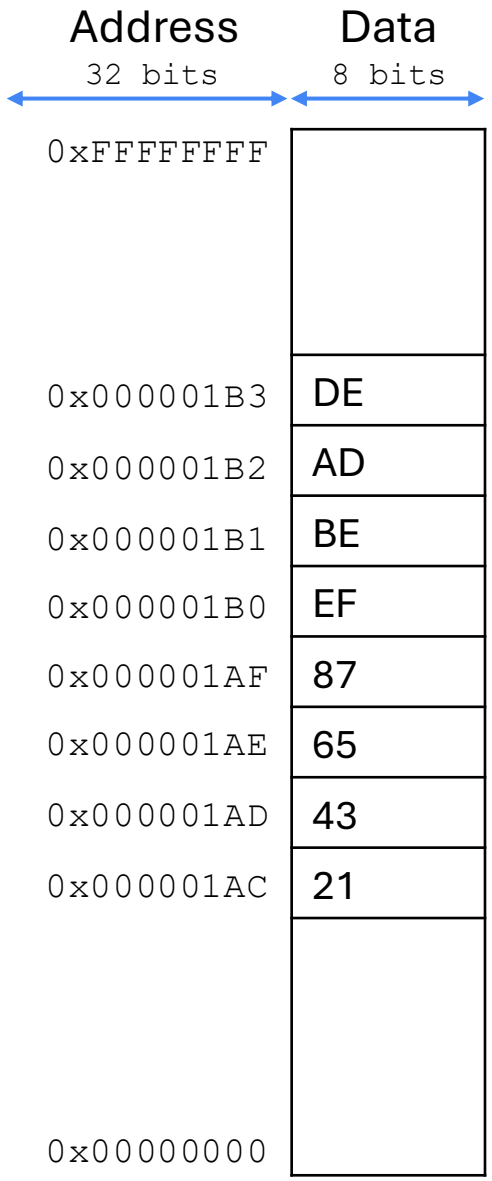
Figure 3. Memory map for STM32L4Rxxx and STM32L4Sxxx



Memory Addressing

Each memory location has a **32-bit address**

Each memory location holds a **byte** (8-bit value)

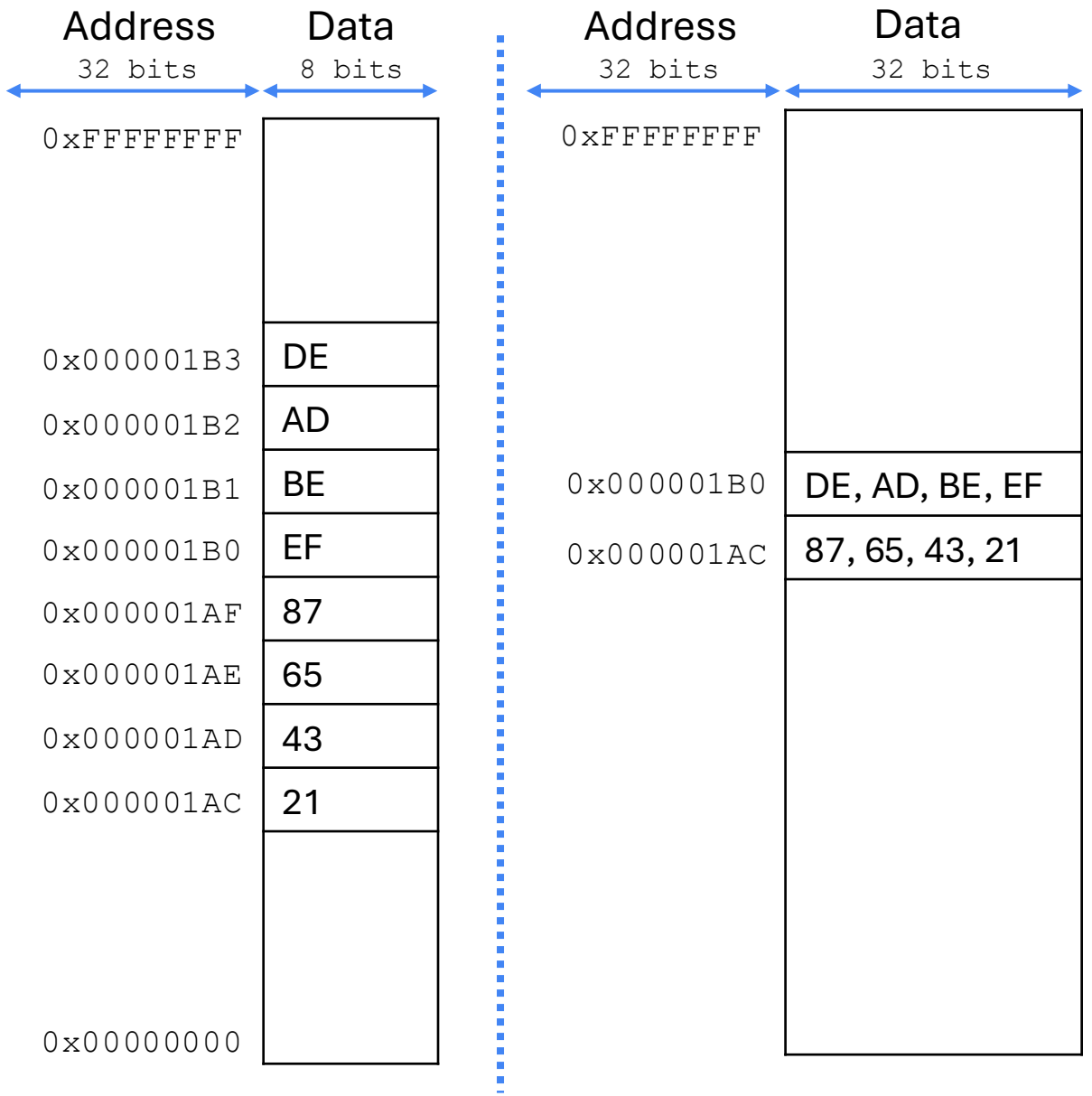


Memory Addressing

This is a 32-bit architecture, so many operations work on word size data

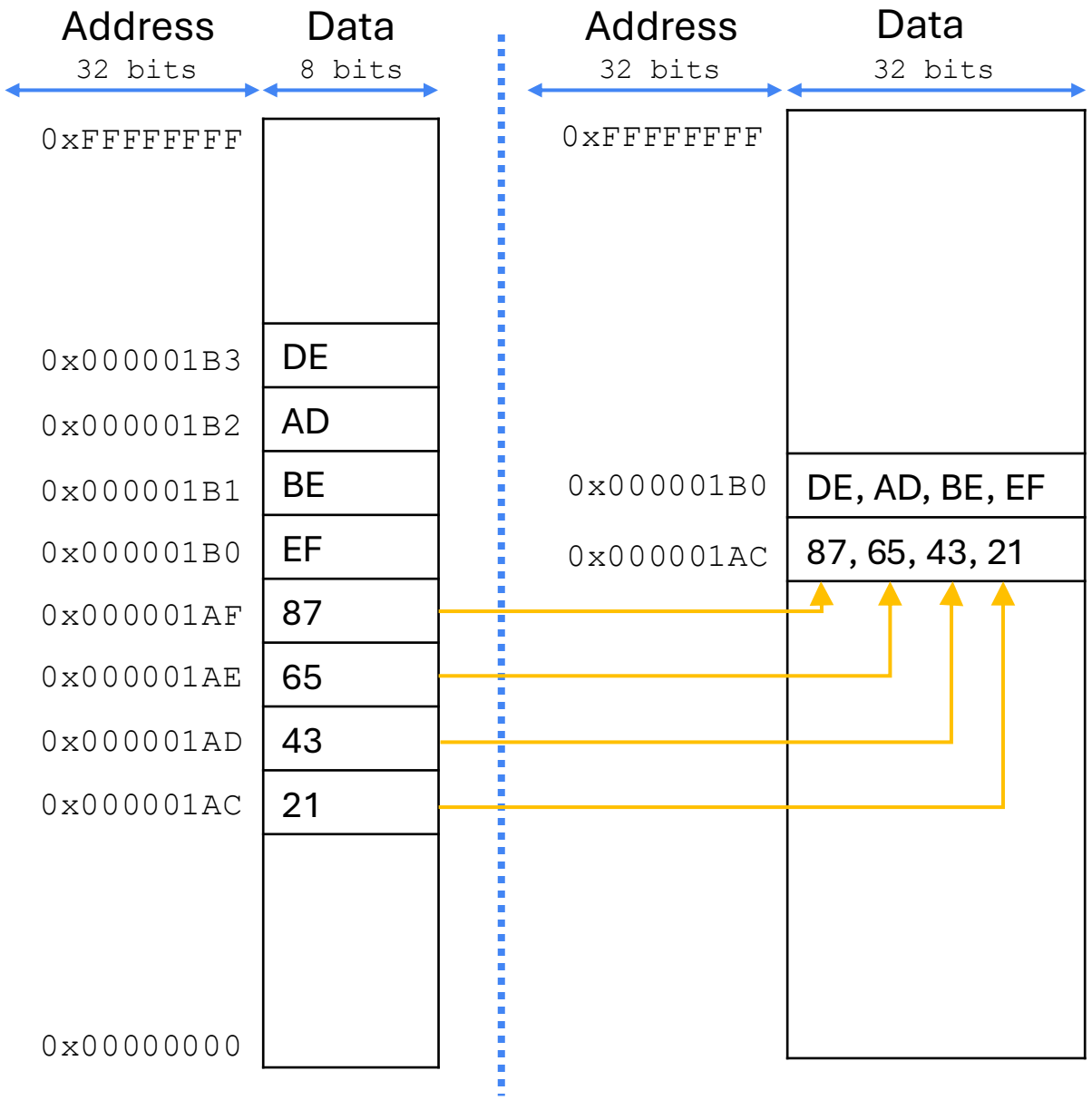
You can think of memory as being **32-bits wide** that is **byte-addressable**

Address increments by 4



“Endianness”

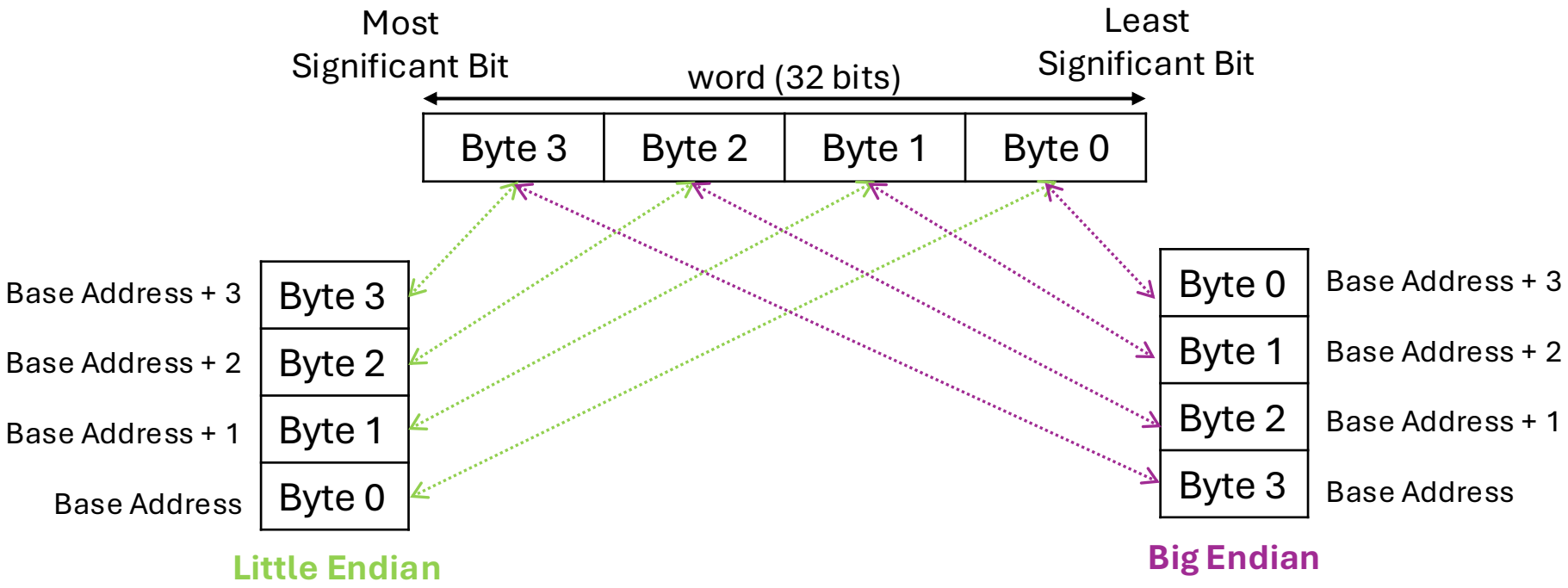
Notice the order of the data!



Big Endian vs Little Endian

Little Endian: Increasing numeric significance with increasing memory addresses
Big Endian: The opposite, most significant byte first

Thumb-2 ISA supports **both**. The **default is Little Endian**

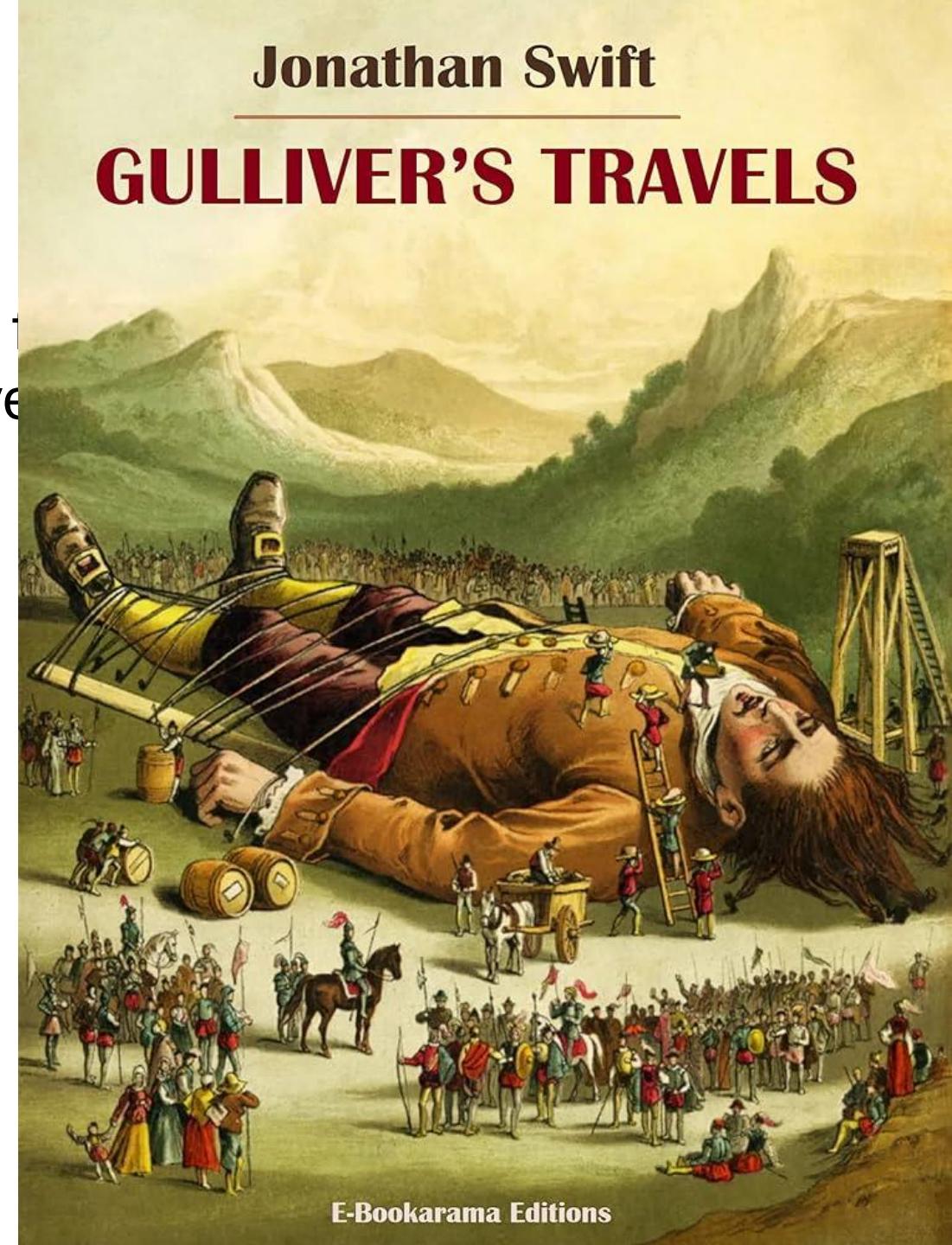


Fun Fact!

The term “big and little endian” comes f
Gulliver’s Trave

Little Endian

Big Endian

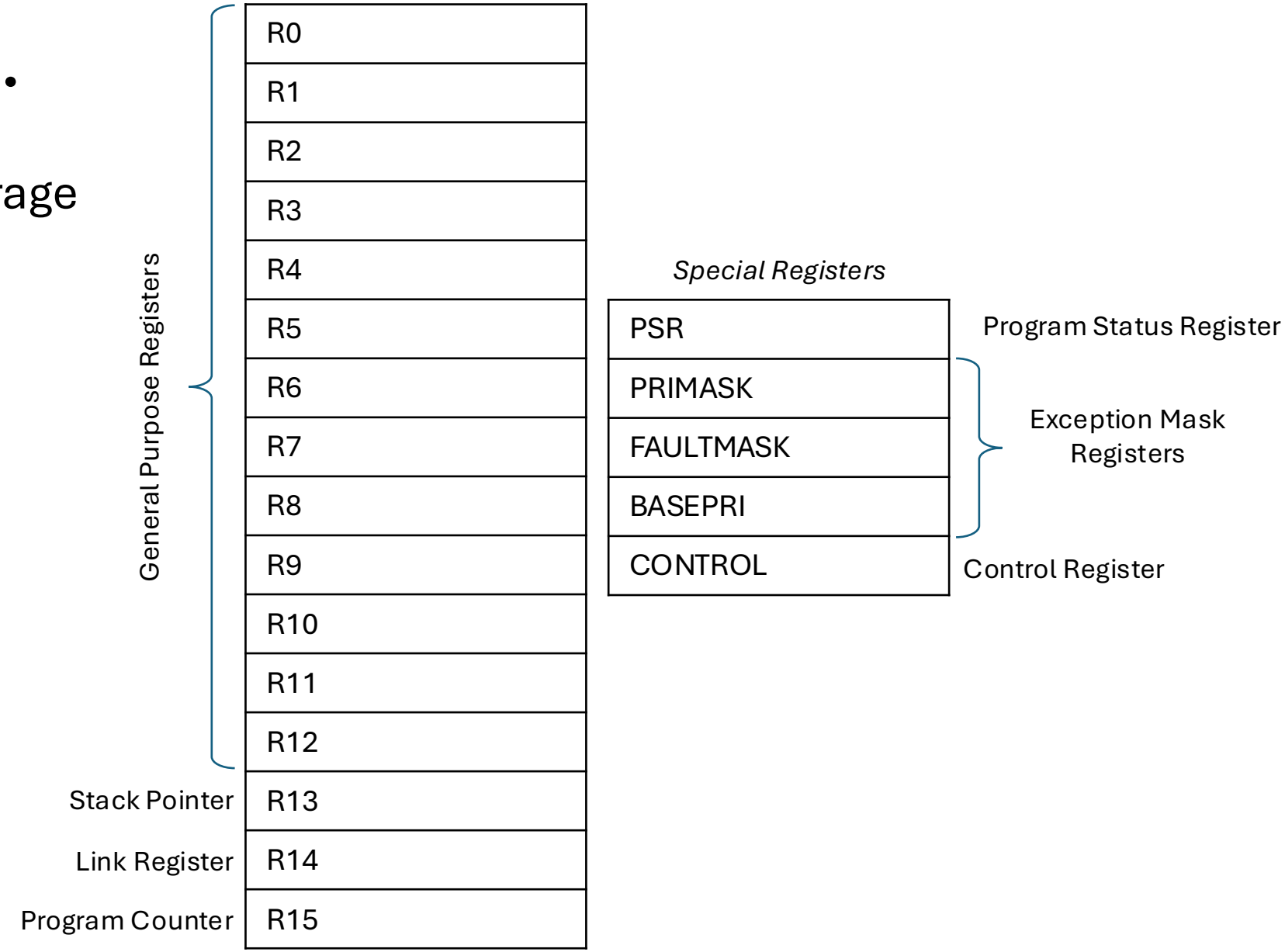


What are we learning about today?

- Instruction Set Architecture (ISA)
- Memory Organization
- **Processor Registers and the Application Binary Interface (ABI)**
- Loading Code and Data into Memory
- Instruction (Assembly)
  - Arithmetic and Logic Instructions
  - Bitwise Logic
  - Move, Loads, and Stores
  - Flow Control and Branching
- Indexing Problem

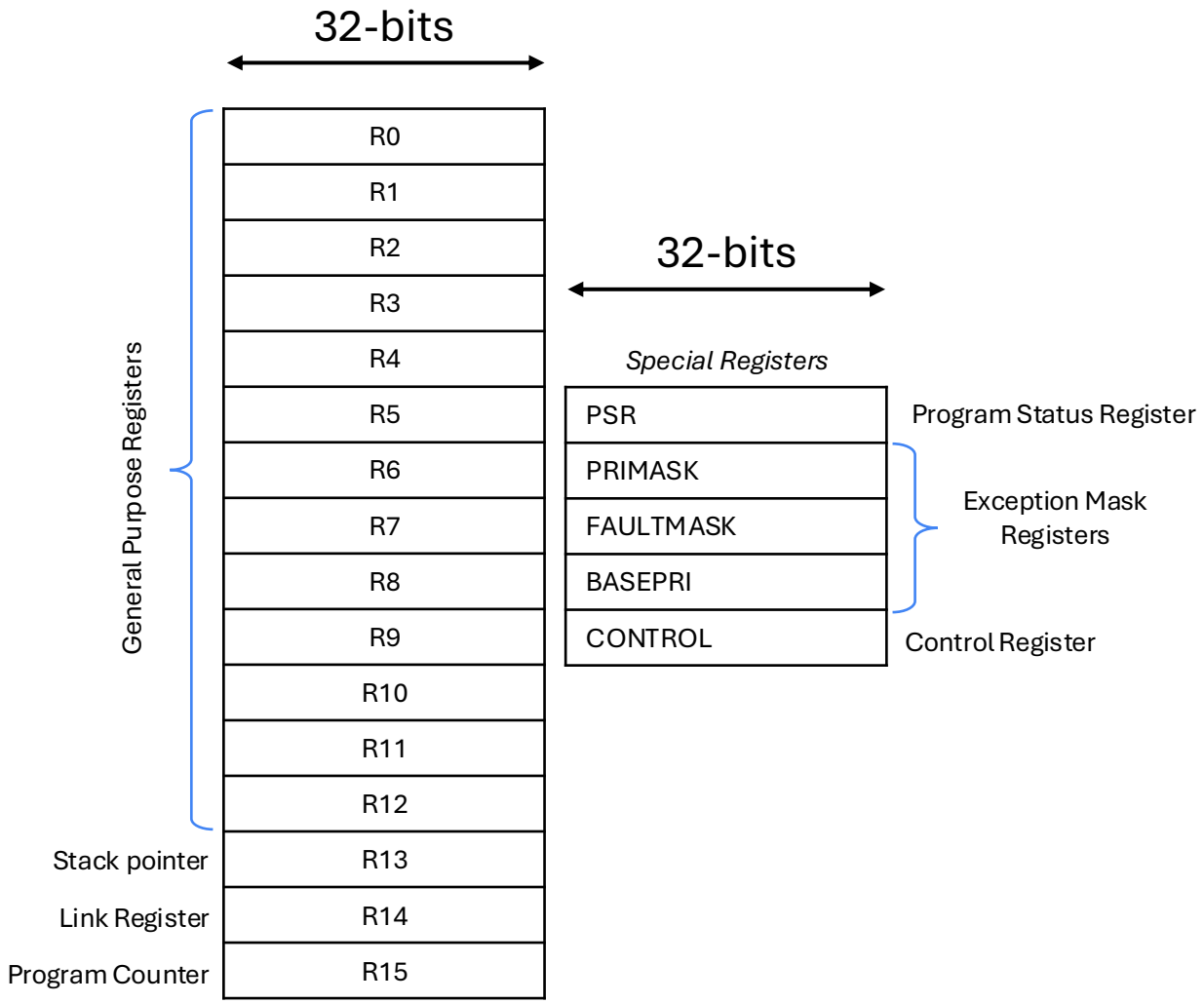
From last lecture…

Registers are high speed storage inside the processor



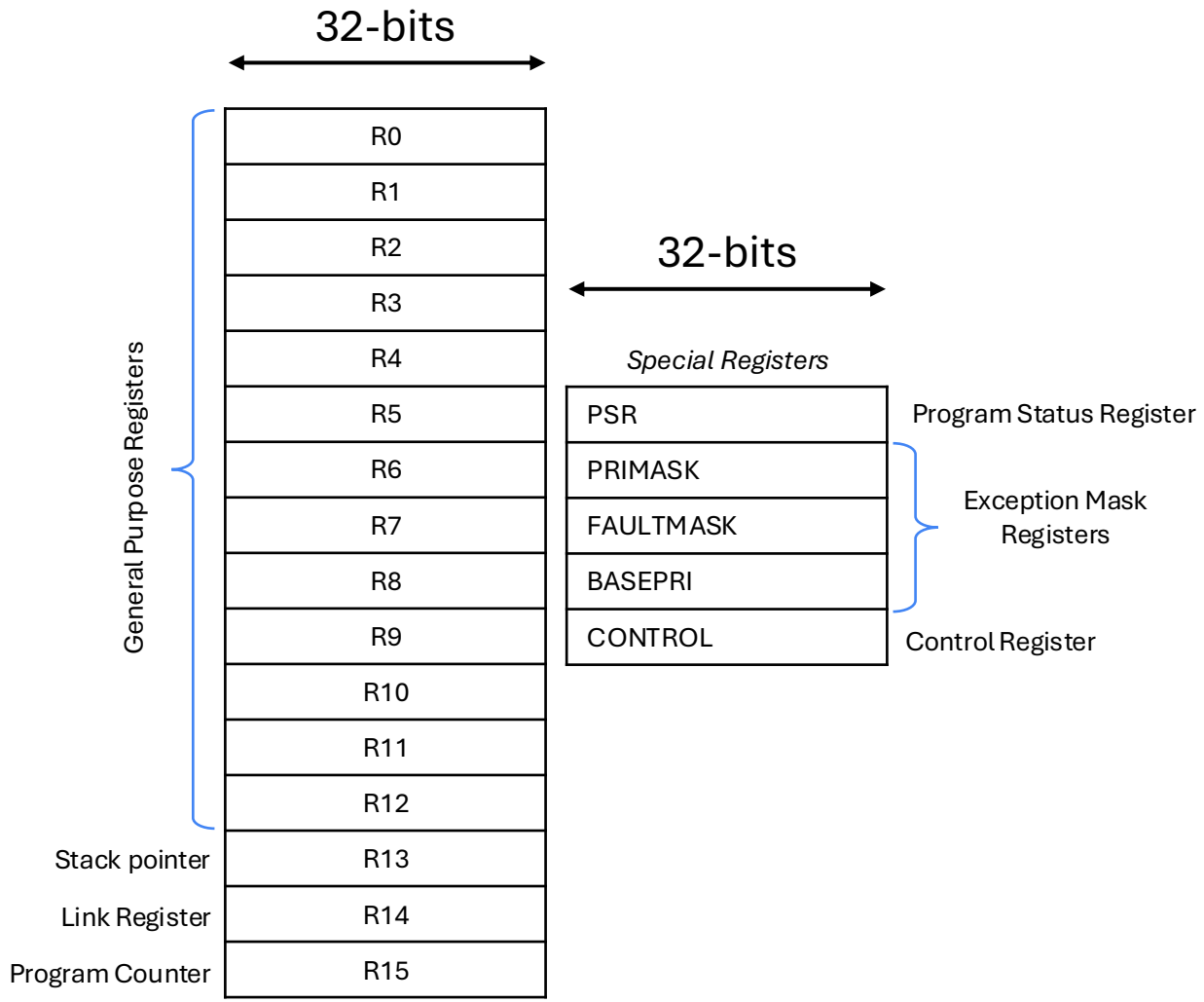
Processor Registers

- Registers are high-speed storage inside of the processor
- In this case, the register stores **32-bit values**
- **R0-R12** are general-purpose registers
- **R13:** Stack Pointer
  - Main Stack Pointer (MSP)
  - Process Stack Pointer (PSP)
- **R14:** Link Register (LR)
- **R15:** Program Counter (PC)
- Special Registers
  - Program Status Register (PSR)
  - Base Priority Mask Register (BASEPRI)
  - Priority Mask Register (PRIMASK)
  - Fault Mask Register (FAULTMASK)
  - Control Register (Control)



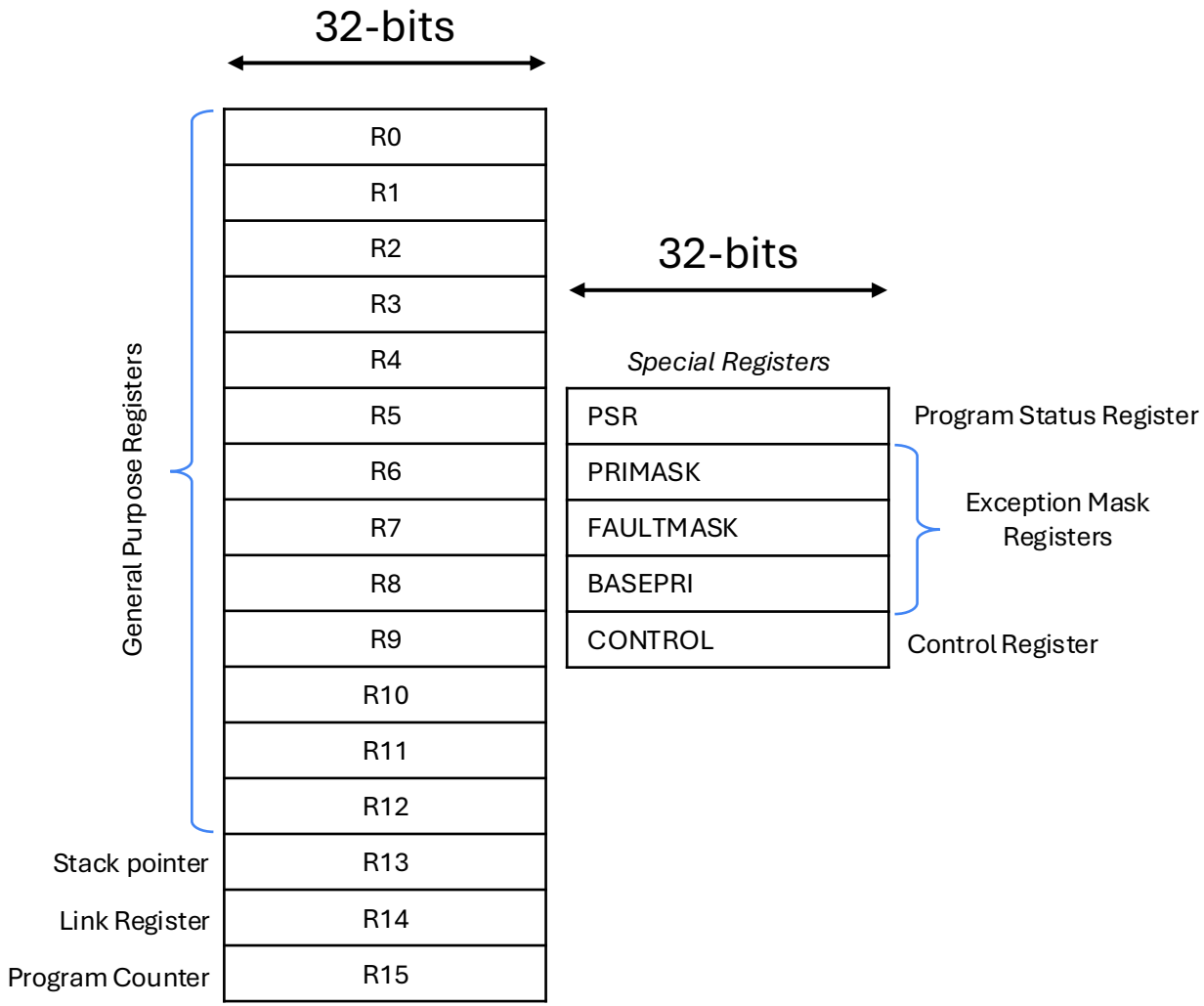
Processor Registers

- Registers are high-speed storage inside of the processor
- In this case, the register stores **32-bit values**
- **R0-R12** are general-purpose registers
- **R13:** Stack Pointer
  - Main Stack Pointer (MSP)
  - Process Stack Pointer (PSP)
- **R14:** Link Register (LR)
- **R15:** Program Counter (PC)
- Special Registers
  - Program Status Register (PSR)
  - Base Priority Mask Register (BASEPRI)
  - Priority Mask Register (PRIMASK)
  - Fault Mask Register (FAULTMASK)
  - Control Register (Control)



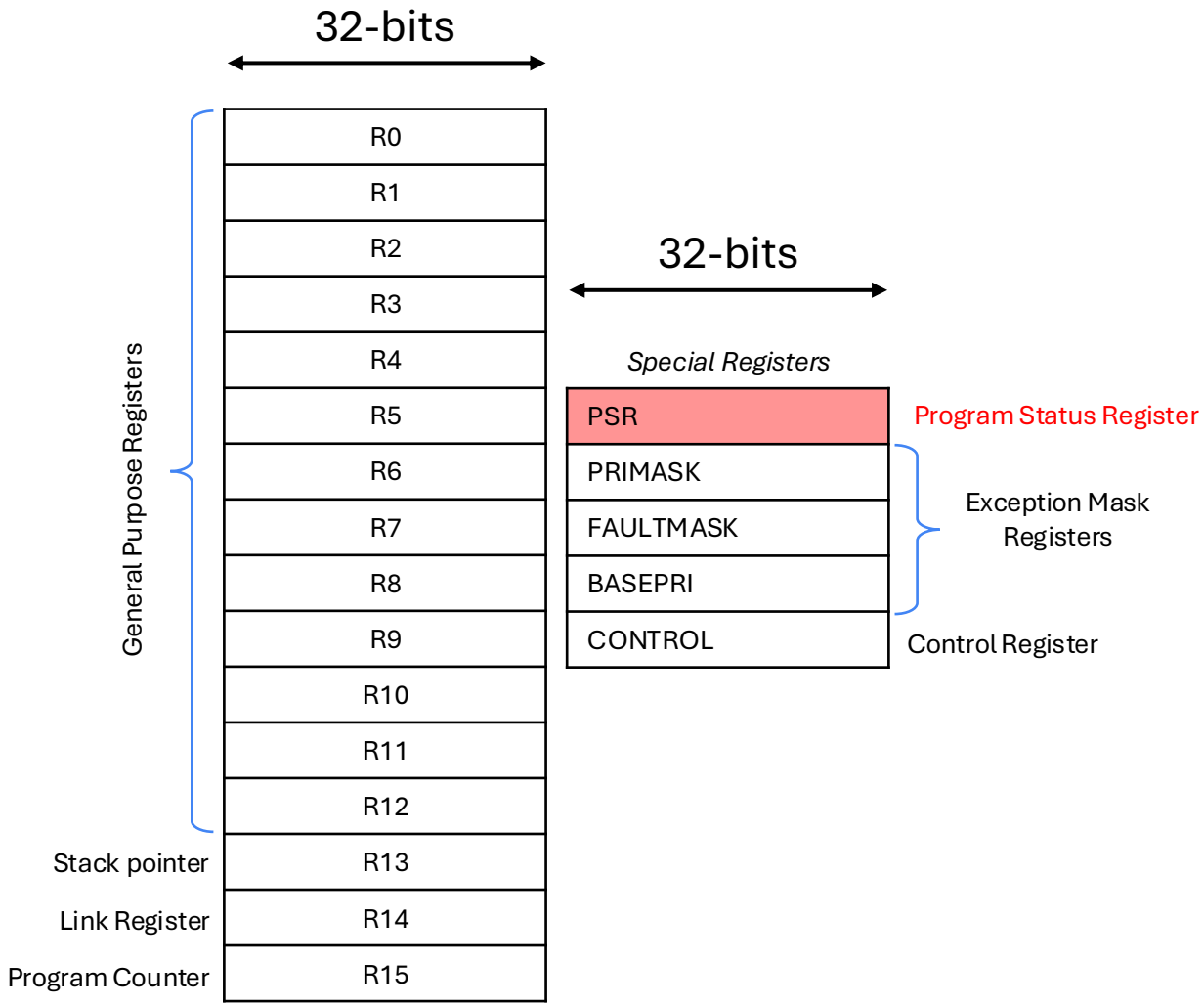
Processor Registers (R13, R14, R15)

- **Stack:** Last-in-first-out temporary storage.
- **Stack Pointer (SP):** points to the 32-bit data on the top of the stack.
- **Link Register (LR) :** Used to store the return location for function
- **Program Counter (PC):** Points to the next instruction to be fetched from memory



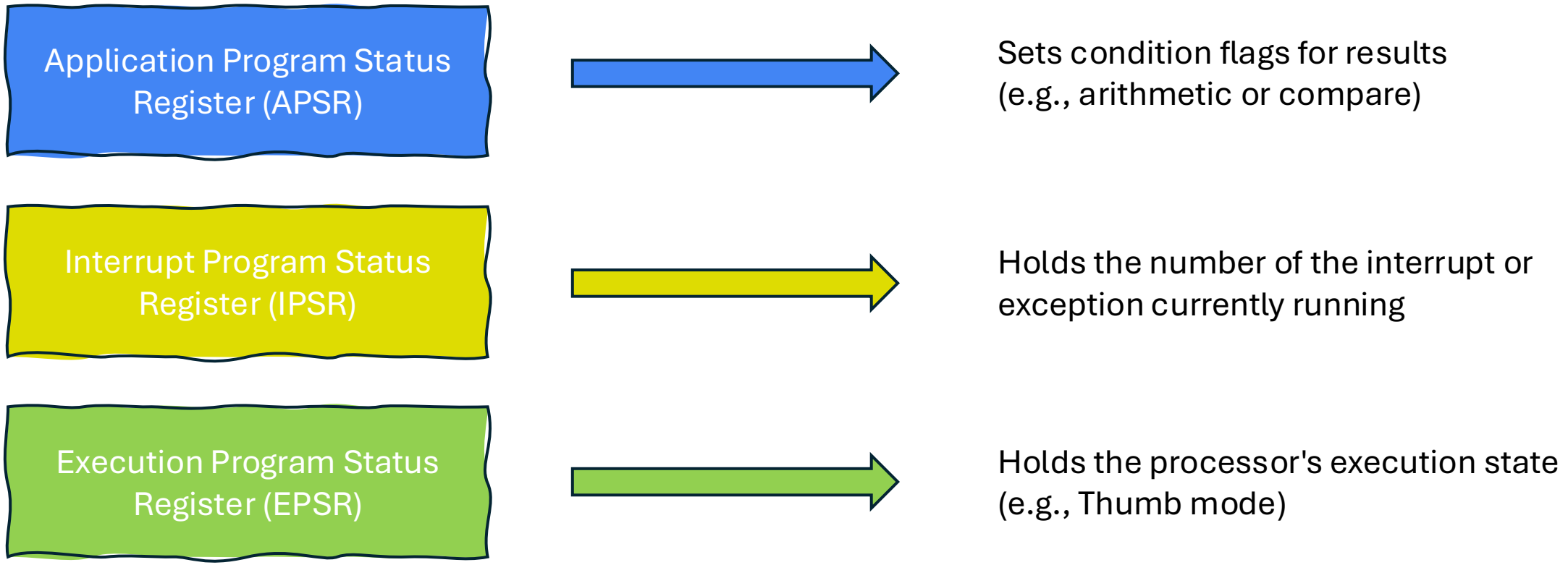
Processor Registers

- Registers are high-speed storage inside of the processor
- In this case, the register stores **32-bit values**
- **R0-R12** are general-purpose registers
- **R13:** Stack Pointer
  - Main Stack Pointer (MSP)
  - Process Stack Pointer (PSP)
- **R14:** Link Register (LR)
- **R15:** Program Counter (PC)
- Special Registers
  - **Program Status Register (PSR)**
  - Base Priority Mask Register (BASEPRI)
  - Priority Mask Register (PRIMASK)
  - Fault Mask Register (FAULTMASK)
  - Control Register (Control)

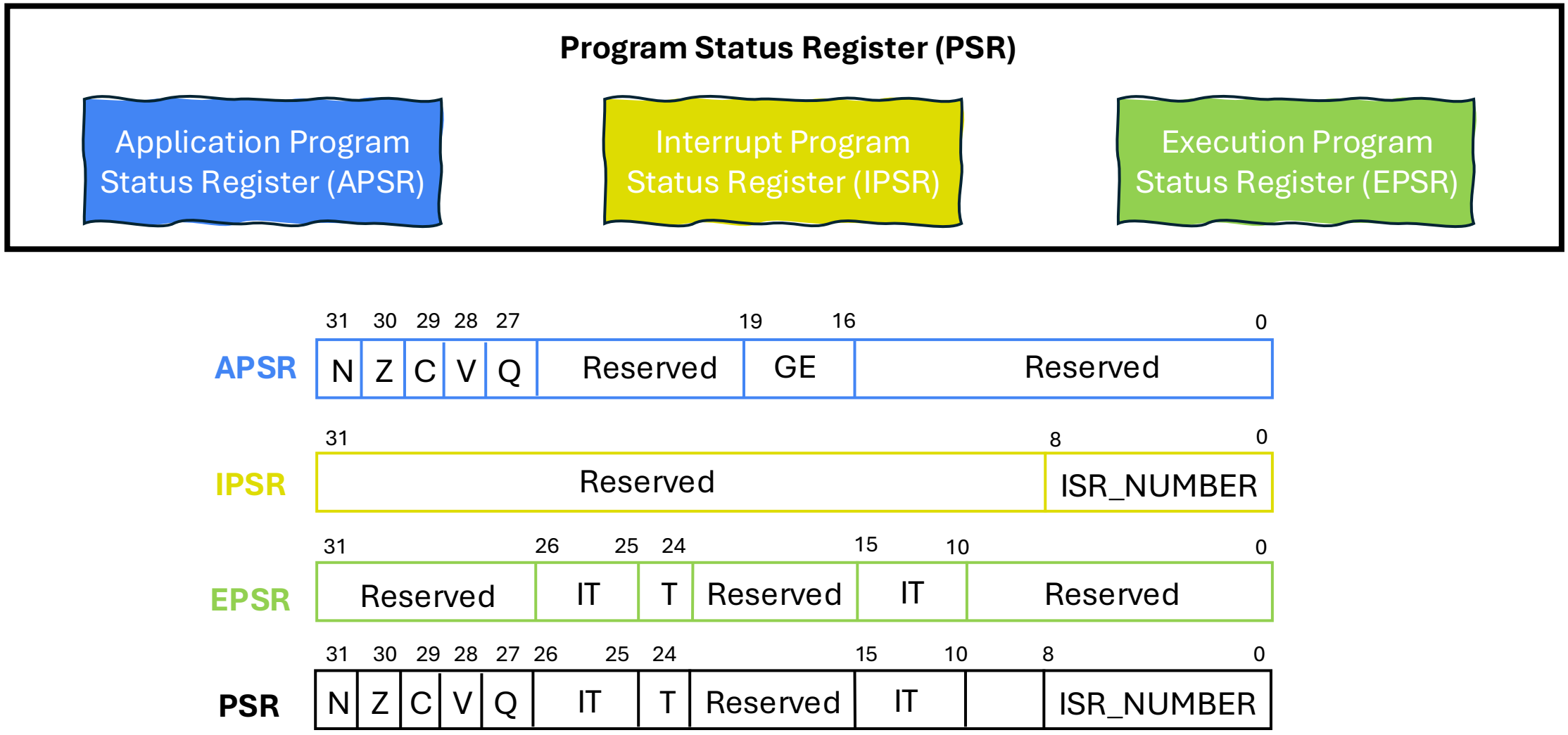


Program Status Register

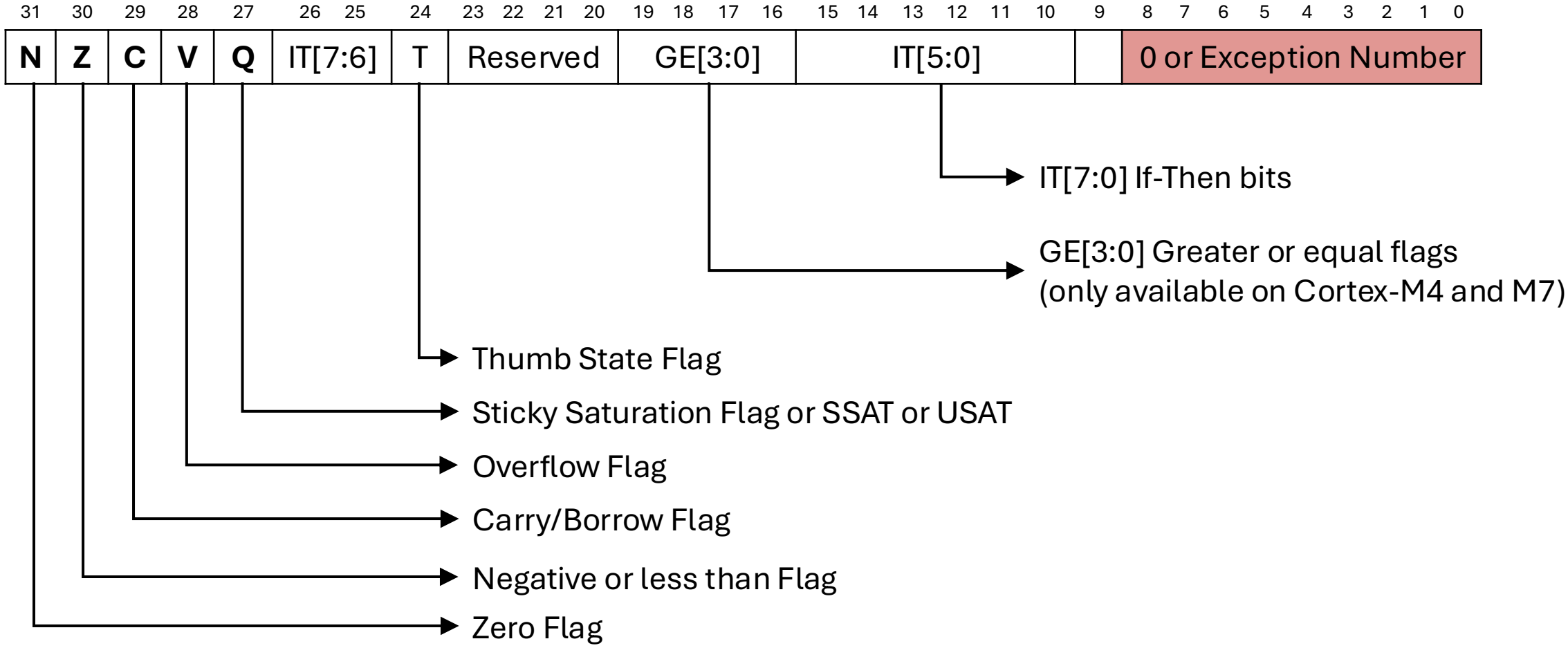
There are three status registers



Program Status Register



Program Status Register (PSR)



Application Binary Interface (ABI)

Defines:

How compiled programs should interface at the binary level…how functions receive arguments and return values

Some Basic Rules:

1. Registers R0, R1, R2, and R3 are used to pass input parameters into a C function
2. Functions must preserve the values of registers R4-R11
3. Return parameter is placed in R0
4. Push and pop an even number of registers to maintain an 8-byte alignment in the stack.

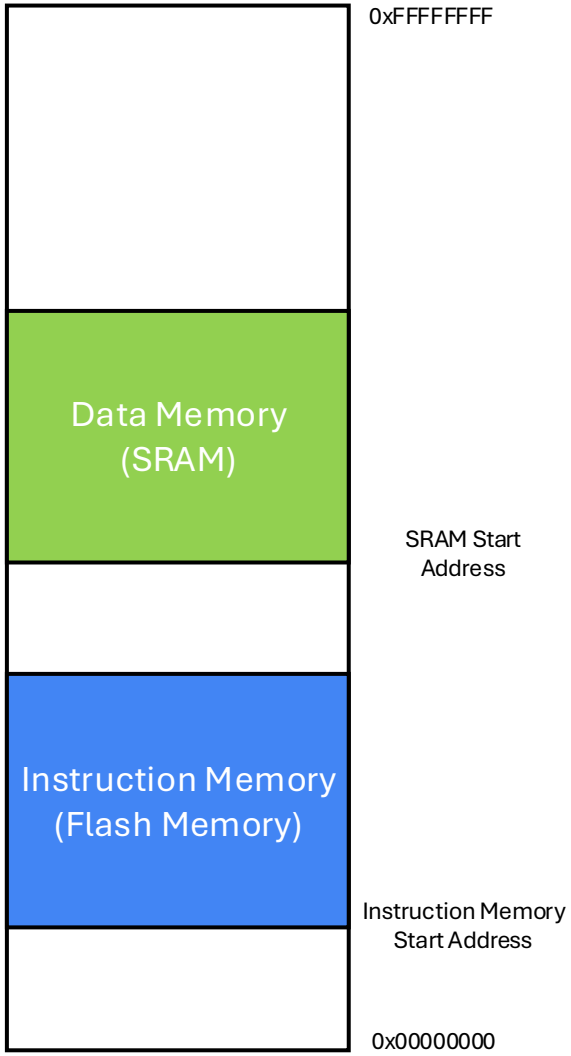
What are we learning about today?

- Instruction Set Architecture (ISA)
- Memory Organization
- Processor Registers and the Application Binary Interface (ABI)
- **Loading Code and Data into Memory**
- Instruction (Assembly)
  - Arithmetic and Logic Instructions
  - Bitwise Logic
  - Move, Loads, and Stores
  - Flow Control and Branching
- Indexing Problem

Loading Code and Data into Memory

Let's say we have the following C program…

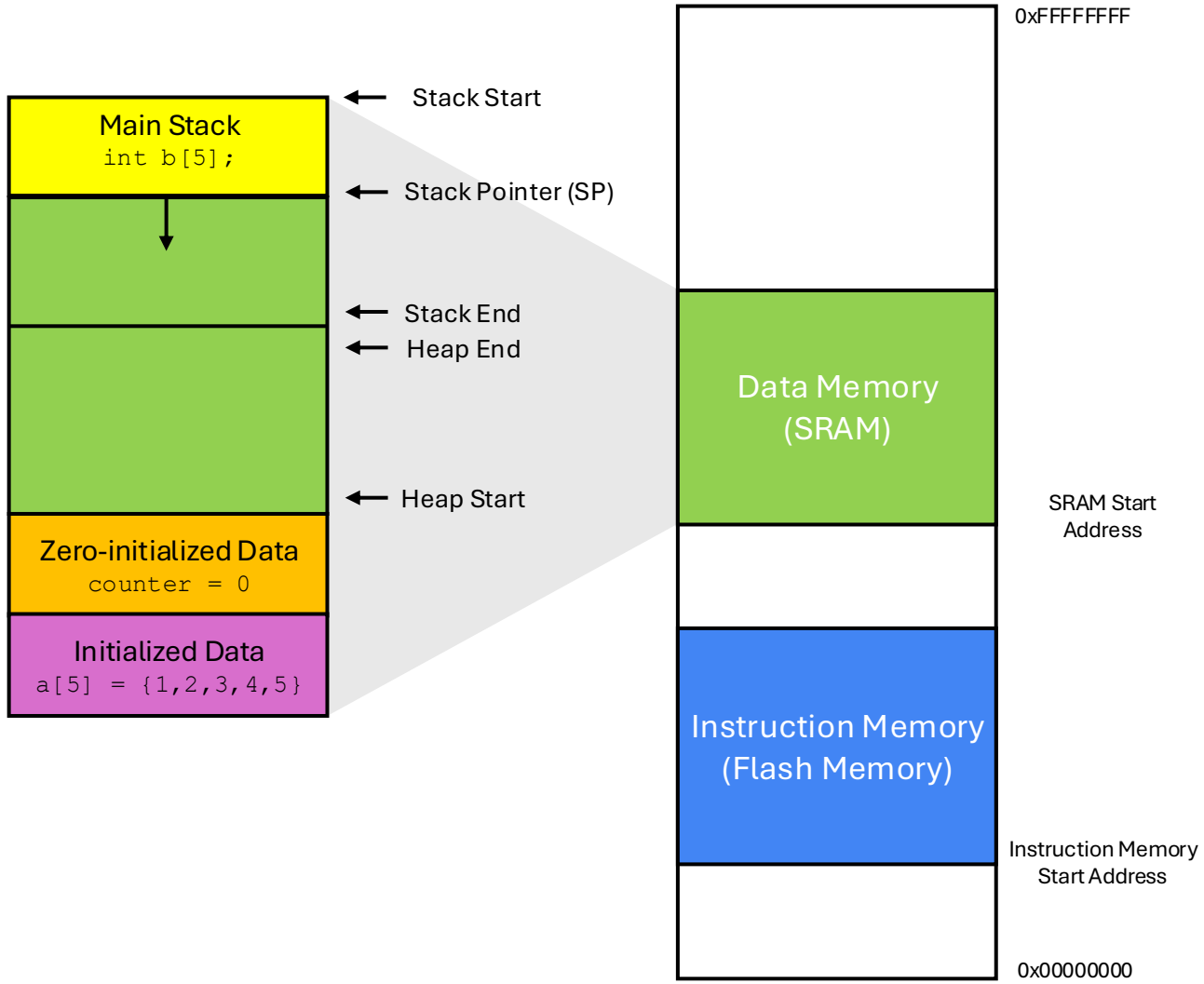
```
int counter;

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);
}
```



Loading Code and Data into Memory

Let's say we have the following C program...

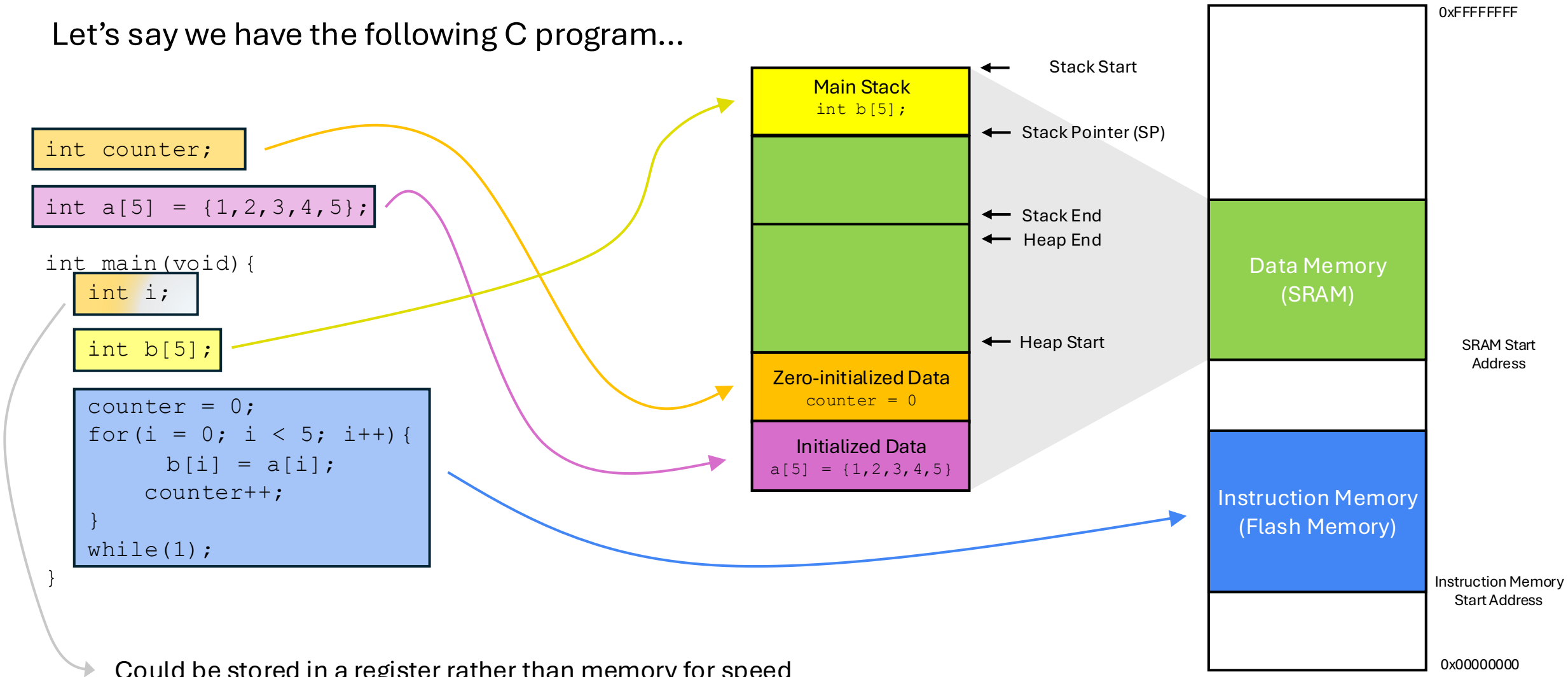
```
int counter;

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);
}
```



Loading Code and Data into Memory

Let's say we have the following C program...

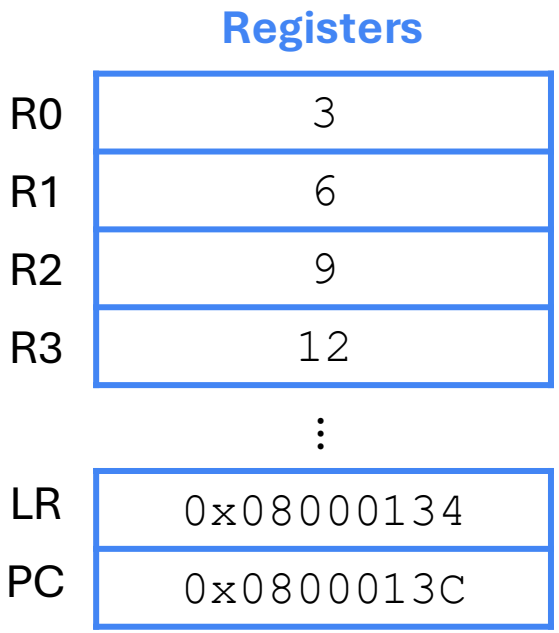
```
int counter;

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);
}
```

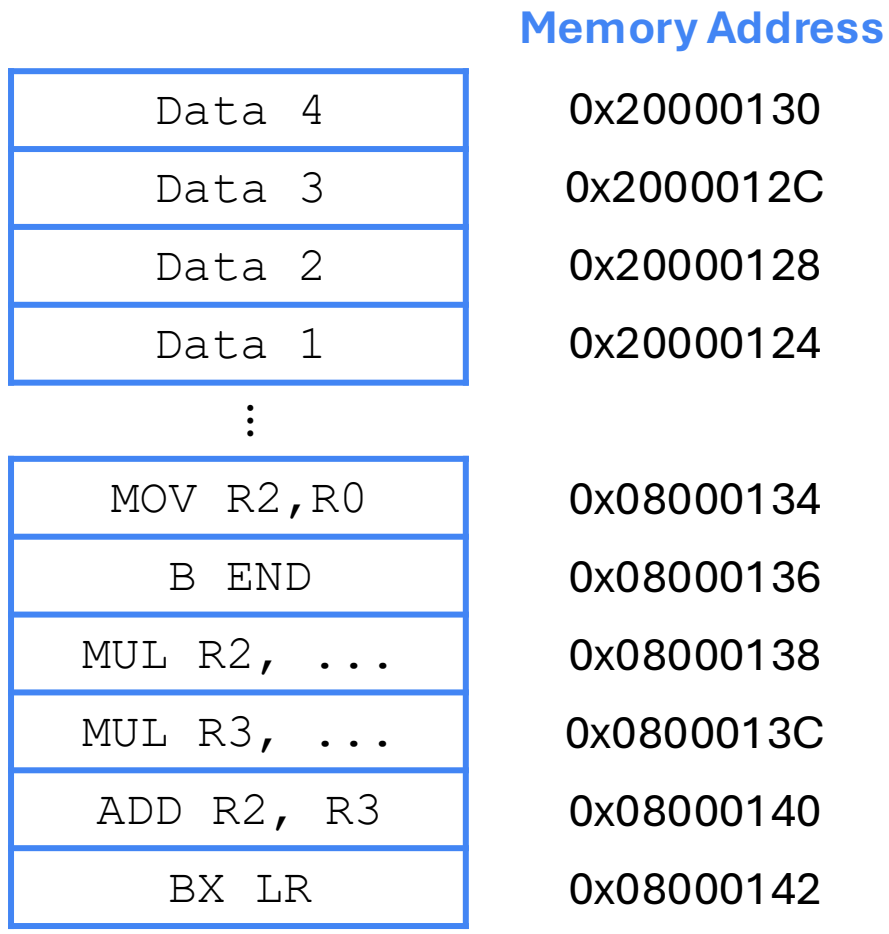


Could be stored in a register rather than memory for speed

Recap

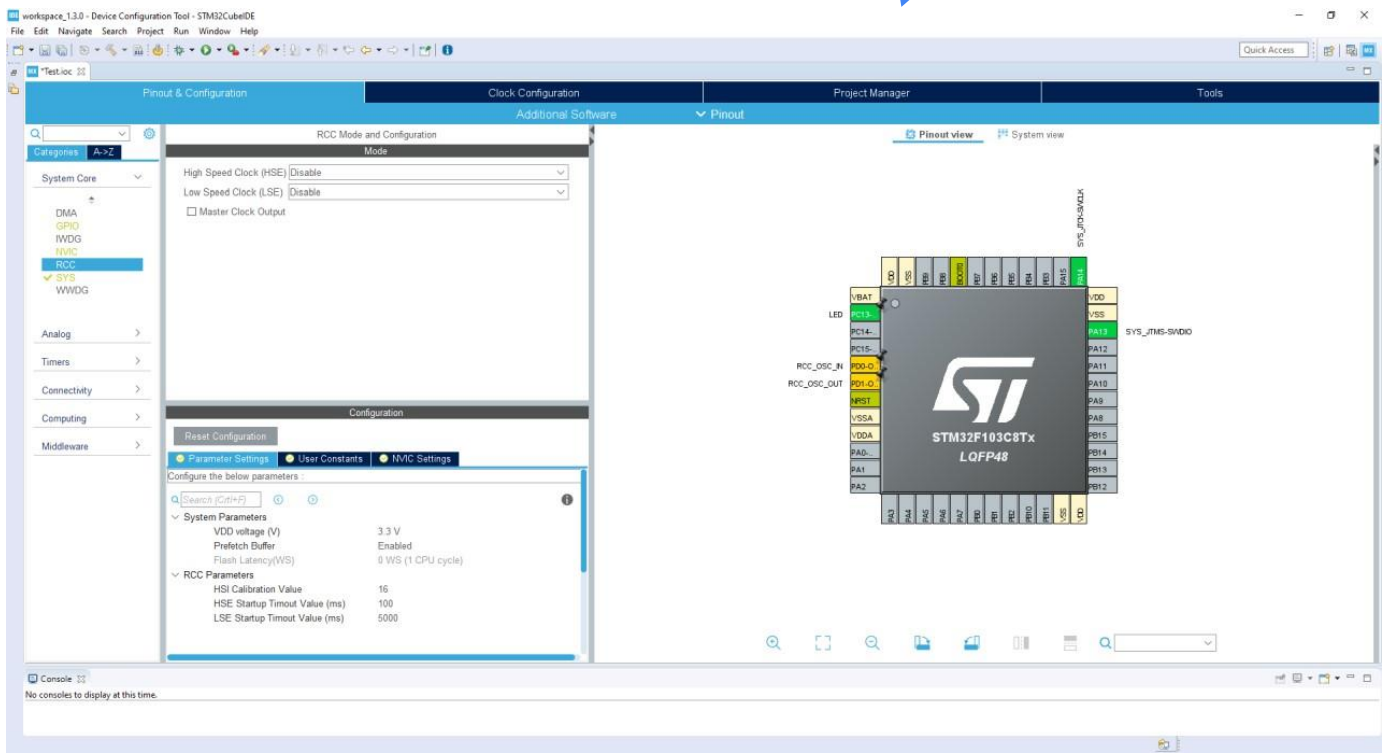
- Processor Register R0-R15
- Byte Addressable Memory





Software Development Process

We are using the STM32CubeIDE



What is going on behind the scenes?

Levels of Program Code

C Program

```
int main(void){
    int i;
    int total = 0;
    for(i=0; i < 10; i++){
            total += 1;
    }
    while(1); // Dead loop
```

High-Level Language

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
```

Assembly Language

Assemble

Machine Program

```
0010000010000000
0010000000000000
1110000000000000
0100010000000001
0001110001000000
0010100000010100
1101110011111011
1011111100000000
1110011111111110
```

Hardware Representation

Review of Instruction Encoding

Convert assembly to machine code

Instructions

`movs r0, #10` ⟹ **001 00 000 00001010**

`movs r1, #0` ⟹ **001 00 001 00000000**

ARMv7-M Architecture Reference Manual

Encoding T1 All versions of the Thumb ISA.

`MOVS <Rd>,#<imm8>` Outside IT block.

`MOV<c> <Rd>,#<imm8>` Inside IT block.

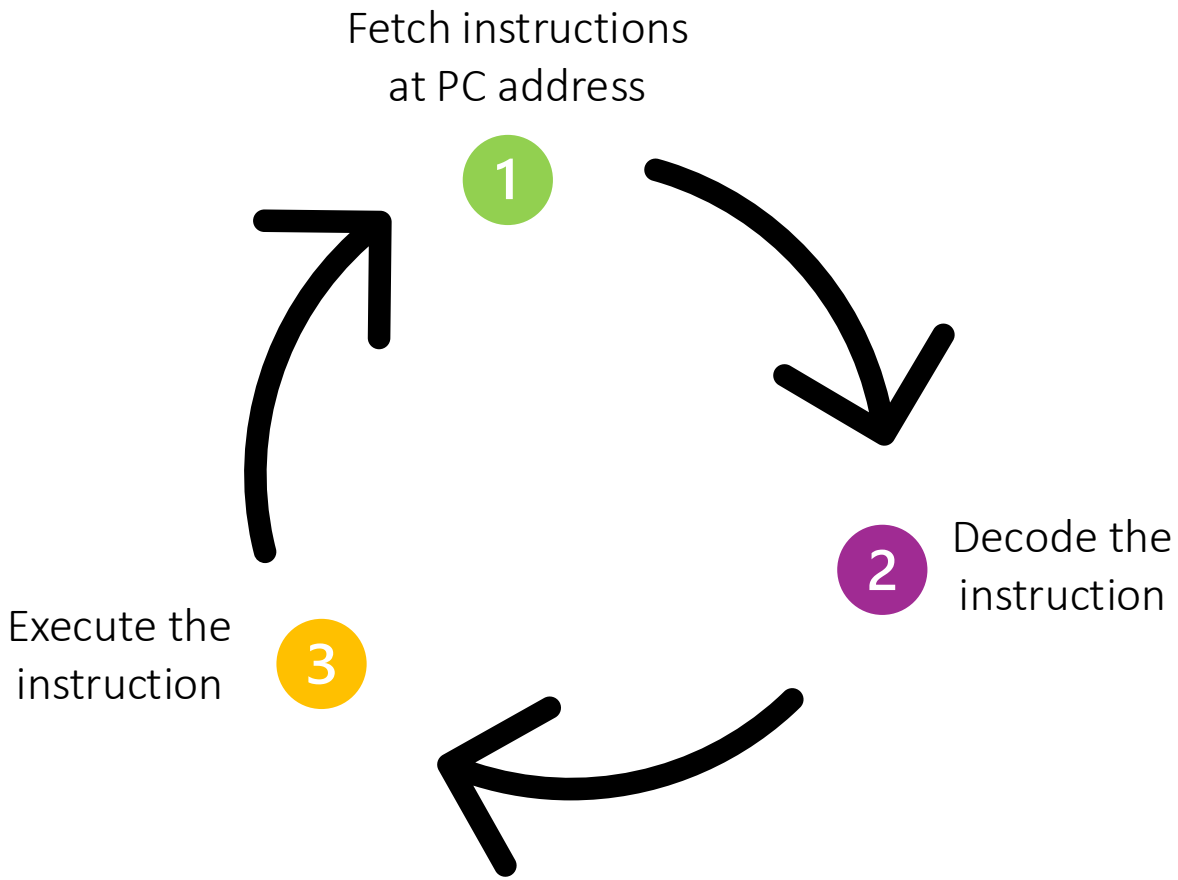
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

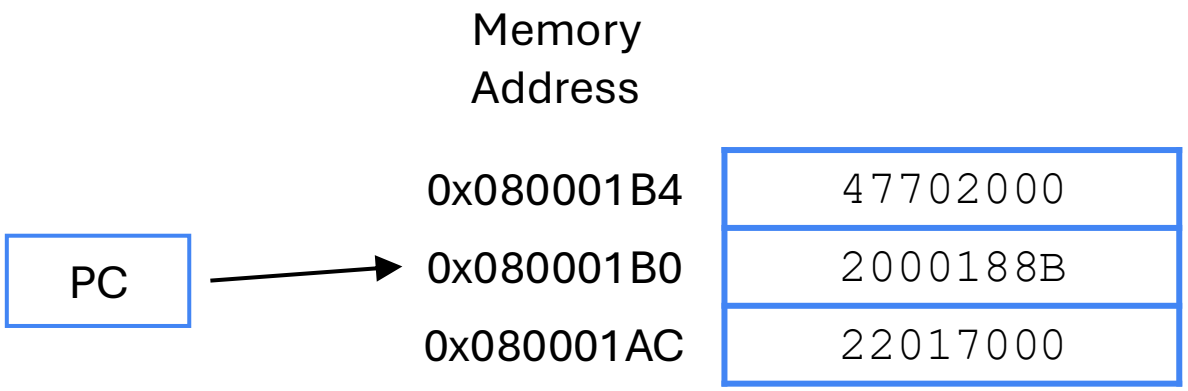
`d = UInt(Rd); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32); carry = APSR.C;`

2100 200A	0x08000008
	0x08000004
	0x08000000

Program Execution

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





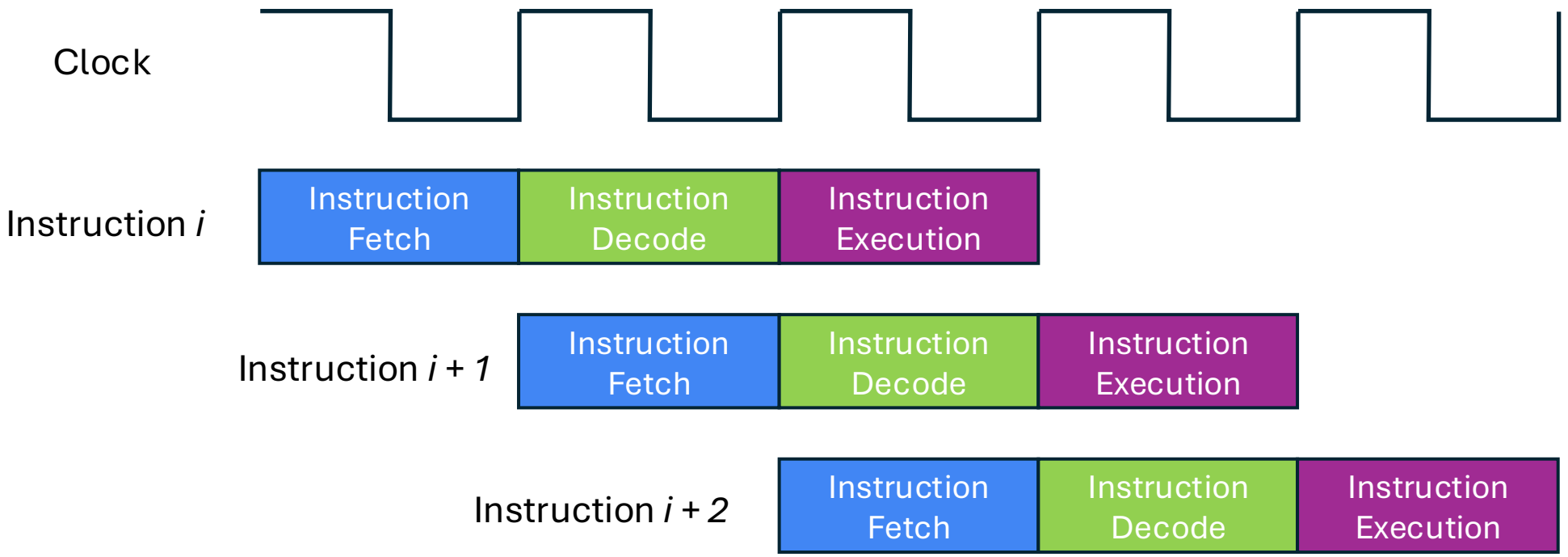
The PC always **fetches a word (32 bits)**

The PC always **increments the address by 4**

Three-stage Pipeline: Fetch, decode, execution

Pipelining allows hardware resources to be fully utilized.

One 32-bit instruction or two 16-bit instructions can be fetched.

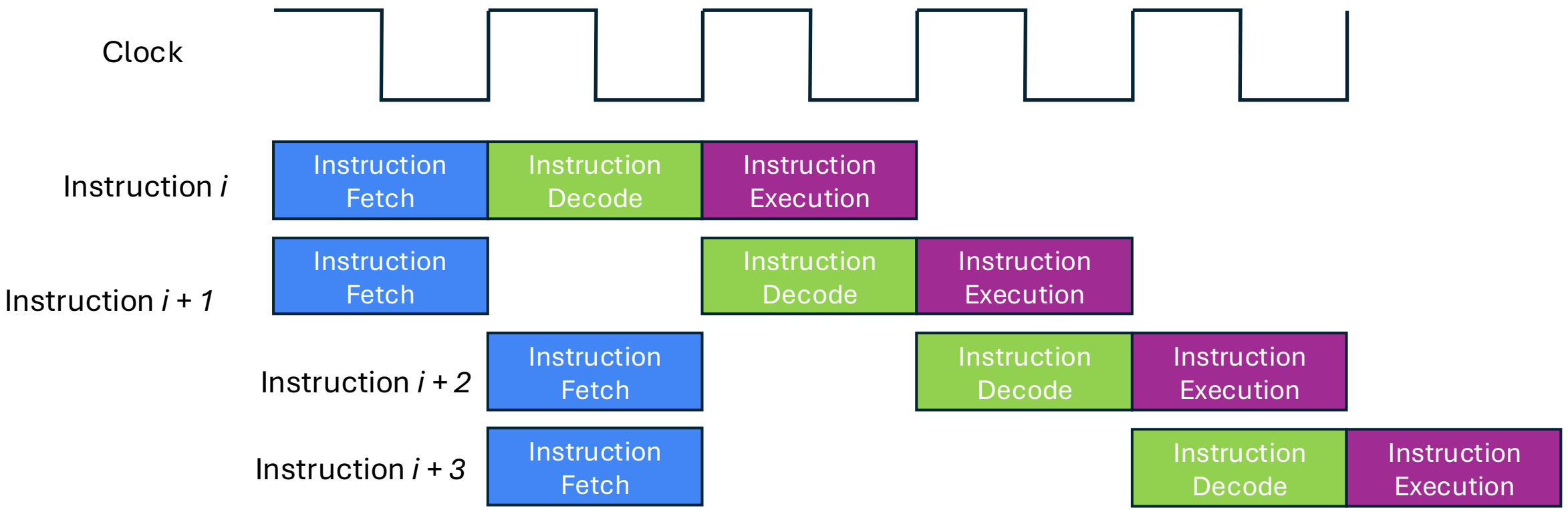


Pipeline of 32-bit instructions

Three-stage 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

>> 30 second brainstorm <<

When the core fetches the next instruction how does it know if it is one 32-bit instruction or two 16-bit instructions?

Answer

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

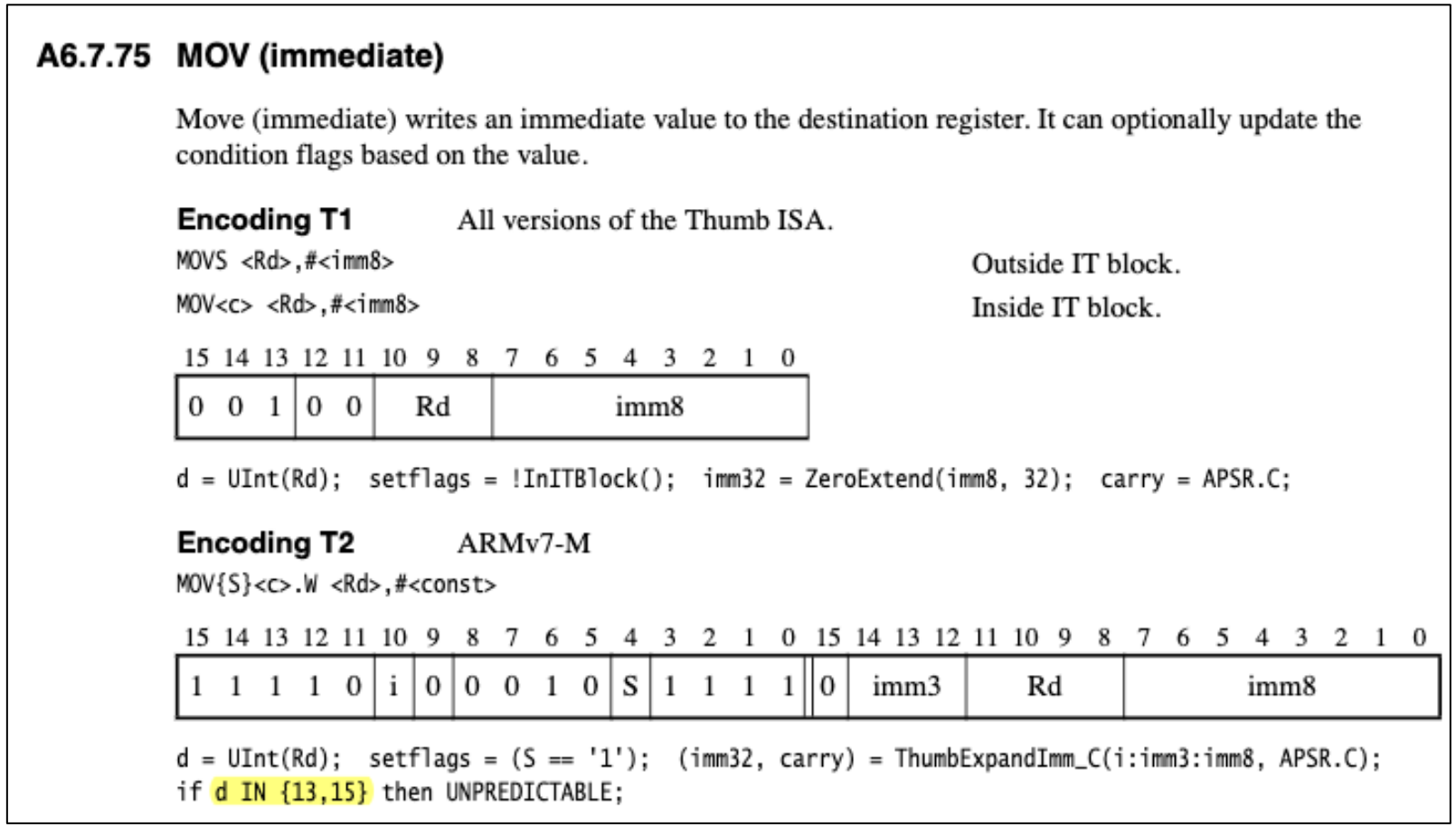
Thus, there is a little bit of combinational logic that decodes the bits[15-11] to determine if it is a 32-bit instruction or two 16-bit instructions.

Instruction Encoding

Which encoding should you use (T1 or T2)?

`MOVS R3, #15` → T1

`MOVS R9, #15` → T2



A6.7.75 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. It can optionally update the condition flags based on the value.

Encoding T1 All versions of the Thumb ISA.

`MOVS <Rd>,#<imm8>` Outside IT block.

`MOV<c> <Rd>,#<imm8>` Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Encoding T2 ARMv7-M

`MOV{S}<c>.W <Rd>,#<const>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	0	0	0	1	0	S	1	1	1	1	0	imm3				Rd				imm8						

```
d = UInt(Rd);  setflags = (S == '1');  (imm32, carry) = ThumbExpandImm_C(i:imm3:imm8, APSR.C);
if d IN {13,15} then UNPREDICTABLE;
```

>> 30 second brainstorm <<

The ISA makes a distinction between “low” and “high” processor registers.

Why?

Answer

16-bit instructions have a limited number of bits that can be used to denote registers.

Many 16-bit instructions are limited to 3-bit destination registers or 3-bit source registers.

This means that 16-bit instructions cannot easily operate on registers R8-15 (i.e. the “high” registers).

In contrast most of the 32-bit instructions allow for operation on both the “high” and “low” registers.

What are we learning about today?

- Instruction Set Architecture (ISA)
- Memory Organization
- Processor Registers and the Application Binary Interface (ABI)
- Loading Code and Data into Memory
- **Instruction (Assembly)**
  - Arithmetic and Logic Instructions
  - Move, Loads, and Stores
  - Flow Control and Branching
- Indexing Problem

Common Instructions

ARMv7-M Architecture Reference Manual

- A4.3 Branch instructions
- A4.4 Data-processing instructions
  - A4.4.1 Standard data-processing instructions
  - A4.4.2 Shift instructions
  - A4.4.3 Multiply instructions
  - A4.4.4 Saturating instructions
  - A4.4.5 Packing and unpacking instructions
  - A4.4.6 Miscellaneous data-processing instruction
  - A4.4.7 Divide instructions
- A4.5 Status register access instructions
- A4.6 Load and store instructions
- A4.7 Load/store multiple instructions

Table A4-2 Standard data-processing instructions

Mnemonic	Instruction	Notes
ADC	Add with Carry	-
ADD	Add	Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
ADR	Form PC-relative Address	First operand is the PC. Second operand is an immediate constant. Thumb supports a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.
AND	Bitwise AND	-
BIC	Bitwise Bit Clear	-
CMN	Compare Negative	Sets flags. Like ADD but with no destination register.
CMP	Compare	Sets flags. Like SUB but with no destination register.
EOR	Bitwise Exclusive OR	-
MOV	Copies operand to destination	Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. See *Shift instructions* on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.
MVN	Bitwise NOT	Has only one operand, with the same options as the second operand in most of these instructions.
ORN	Bitwise OR NOT	-
ORR	Bitwise OR	-
RSB	Reverse Subtract	Subtracts first operand from second operand. This permits subtraction from constants and shifted registers.
SBC	Subtract with Carry	-
SUB	Subtract	Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.
TEQ	Test Equivalence	Sets flags. Like EOR but with no destination register.
TST	Test	Sets flags. Like AND but with no destination register.

ARM Cortex-M Assembly Language

Assembly language instructions have four fields

Label	Opcode	Operands	Comment
`Func`	`MOV`	`R0, #100`	`; this sets R0 to 100`
	`BX`	`LR`	`; this is a function return`

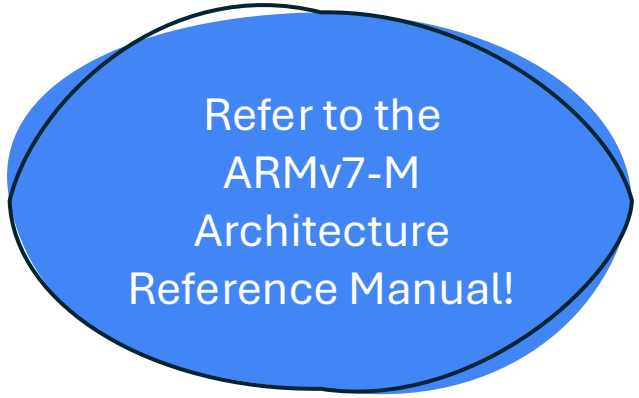
Symbol used to describe assembly instruction

`Ra Rd Rm Rn Rt` and `Rt2` represent registers
`{Rd,}` represents an optional destination register
`#imm12` represents a 12-bit constant, 0 to 4095
`#imm16` represents a 16-bit constant, 0 to 65535
`operand2` represents the flexible second operand

`{cond}` represents an optional logical condition
`{type}` encloses an optional data type
`{S}` is an optional specification that this instruction sets the condition code bits
`Rm {, shift}` specifies an optional shift on Rm
`Rn {, #offset}` specifies an optional offset to Rn

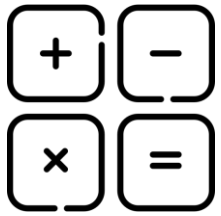
Types of Assembly Instructions Supported

- Arithmetic and logic
  - Add, Subtract, Multiply, Divide, Shift, Rotate
- Data Movement
  - Load, Store, Move
- Compare and Branch
  - Compare, Test, If-then, Branch, Compare and Branch on Zero
- Miscellaneous
  - Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier



What are we learning about today?

- Instruction Set Architecture (ISA)
- Memory Organization
- Processor Registers and the Application Binary Interface (ABI)
- Loading Code and Data into Memory
- Instruction (Assembly)
  - **Arithmetic and Logic Instructions**
  - Bitwise Logic
  - Move, Loads, and Stores
  - Flow Control and Branching
- Indexing Problem



Commonly Used Arithmetic Operations

ADD {Rd,} Rn, Op2	Add	Rd = Rn + Op2
ADC {Rd,} Rn, Op2	Add with carry	Rd = Rn + Op2 + Carry
SUB {Rd,} Rn, Op2	Subtract	Rd = Rn – Op2
SBC {Rd,} Rn, Op2	Subtract with carry	Rd = Rn – Op2 – (1– Carry)
RSB {Rd,} Rn, Op2	Reverse subtract	Rd = Op2 – Rn
MUL {Rd,} Rn, Rm	Multiply	Rd = Rn × Rm *
MLA Rd, Rn, Rm, Ra	Multiply and accumulate	Rd = (Ra + (Rn × Rm))*
MLS Rd, Rn, Rm, Ra	Multiply and subtract	Rd = (Ra – (Rn × Rm))*
SDIV {Rd,} Rn, Rm	Signed divide	Rd = Rn/Rm
UDIV {Rd,} Rn, Rm	Unsigned divide	Rd = Rn/Rm

* Returns the 32 least significant bits of the result

Addition Instruction

```
ADD{cond} {Rd,} Rn, #imm12

ADD R0, #1              ; R0 = R0 + 1
ADD R0, R1, #10         ; R0 = R1 + 10
ADDEQ R1, R1, #5        ; R1 = R1 + 5 if Z=1 (zero flag)
```

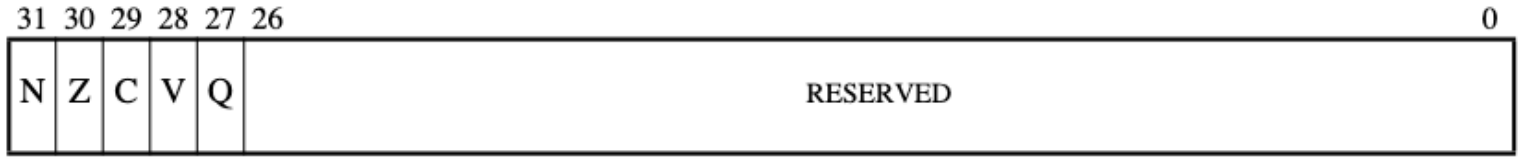
Recall…

`{S}` is an optional specification that this instruction sets the condition code bits

For most instructions, we can add the suffix `S` to update the N, Z, C, V bit flags of the APSR register

A2.3.2 The Application Program Status Register (APSR)

Program status is reported in the 32-bit Application Program Status Register (APSR), where the defined bits break down into a set of flags as follows:



APSR bit fields are in the following two categories:

- Reserved bits are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.

- Flags that can be set by many instructions:

  N, bit [31] Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.

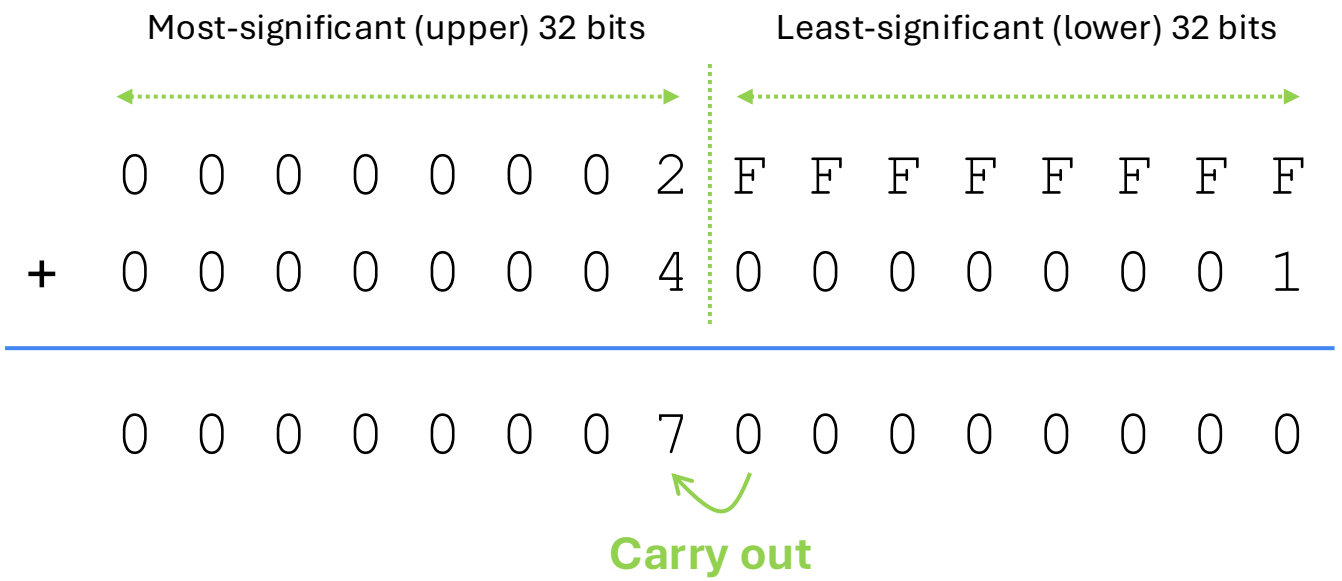
  Z, bit [30] Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  C, bit [29] Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  V, bit [28] Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  Q, bit [27] Set to 1 if an `SSAT` or `USAT` instruction changes (saturates) the input value for the signed or unsigned range of the result.

Example of a Condition Flag: 64-bit Addition



A register can only store 32 bits so…a 64-bit integer needs two registers.

Split 64-bit addition into two 32-bit additions

Example of a Condition Flag: 64-bit Addition

```
Start:
      ; C = A + B
      ; Two 64-bit integers A (r1, r0) and B (r3, r2)
      ; Result C (r5, r4)
      ; A = 00000002FFFFFFFF
      ; B = 0000000400000001

      MVN r0, #0            ;A's lower 32 bits equivalent to 0xFFFFFFFF
      MOV r1, #2            ;A's upper 32 bits
      MOV r2, #1            ;B's lower 32 bits
      MOV r3, #4            ;B's upper 32 bits

      ; Add A and B
      ADDS r4, r2, r0       ;C[31…0] = A[31…0] + B[31…0], update Carry
      ADC r5, r3, r1        ;C[63…32] = A[64…32] + B[64…32] + Carry

Stop: B Stop
```

What are we learning about today?

- Instruction Set Architecture (ISA)
- Memory Organization
- Processor Registers and the Application Binary Interface (ABI)
- Loading Code and Data into Memory
- Instruction (Assembly)
  - Arithmetic and Logic Instructions
  - **Bitwise Logic**
  - Move, Loads, and Stores
  - Flow Control and Branching
- Indexing Problem

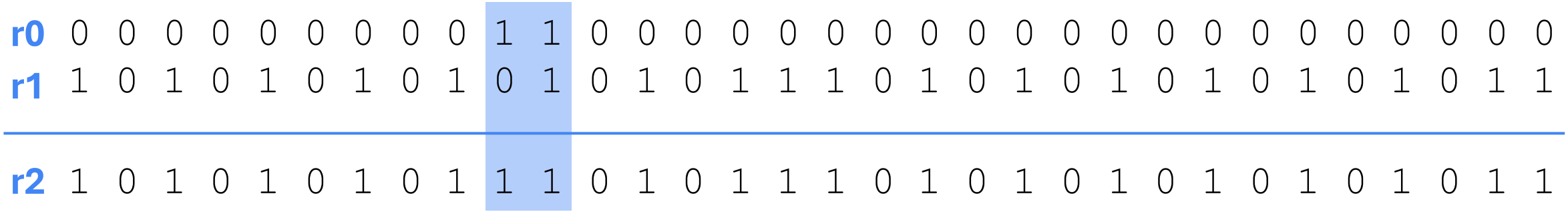
Bitwise Logic

AND {Rd,} Rn, Op2	Bitwise Logic AND	Rd = Rn & Op2
ORR {Rd,} Rn, Op2	Bitwise Logic OR	Rd = Rn | Op2
EOR {Rd,} Rn, Op2	Bitwise Logic Exclusive OR	Rd = Rn ⊕ Op2
ORN {Rd,} Rn, Op2	Bitwise Logic NOT OR	Rd = Rn | ~Op2
BIC {Rd,} Rn, Op2	Bit Clear	Rd = Rn & ~Op2
BFC Rd, #lsb, $width	Bit Field Clear	Rd[lsb+width-1: lsb] = 0
BFI Rd, Rn, #lsb, #width	Bit Field Insert	Rd[(width+lsb-1):lsb] = Rd[(width-1):0]
MVN Rd, Op2	Move NOT, logically negate all bits	Rd = 0xFFFFFFFF **EOR** Op2

Example of Bitwise Logic

Bitwise Logic OR : ORR r2, r0 r1

r0	r1	r2
0	0	0
0	1	1
1	0	1
1	1	1



Example of Bitwise Logic

Bit Clear : r2 = r0 & ~r1

Step 1:

r1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
~r1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0

Step 2:

r0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
~r1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
r2	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0

What are we learning about today?

- Instruction Set Architecture (ISA)
- Memory Organization
- Processor Registers and the Application Binary Interface (ABI)
- Loading Code and Data into Memory
- Instruction (Assembly)
  - Arithmetic and Logic Instructions
  - Bitwise Logic
  - **Move, Loads, and Stores**
  - Flow Control and Branching
- Indexing Problem

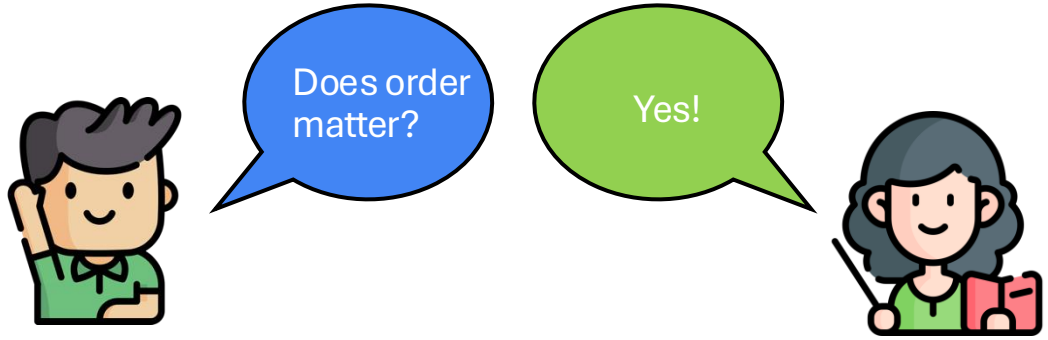
Moving Data Between Registers

MOV Rd, Op2	Rd = Op2
MVN Rd, Op2	Move NOT, logically negate all bits
MRS Rd, spec_reg	Move from special register to general register
MSR spec_reg, Rm	Move from general register to special register

```
MOV r4, r5          ; Copy r5 to r4
MVN r4, r5          ; r4 = bitwise logical NOT of r5
MRS r0, APSR        ; Copy condition flags to r0
MSR PRIMASK, r1     ; Copy r1 to PRIMASK (1 = disable interrupts)
```

Move Immediate Number to Register

MOV Rd, #const	Move
MOVW Rd, #imm16	Move Wide
MOVT Rd, #imm16	Move Top



Example : Load a 32-bit number into a register

```
                          r0 = 0x0000 0000
MOVW r0, #0x4321   ⟶      r0 = 0x0000 4321
MOVT r0, #0x8765   ⟶      r0 = 0x8765 4321
```

- MOVW will **"zero out"** the upper halfword
- MOVT **will not** modify the lower halfword

lower 16 unaffected

```
MOVT,r0, #0x8765   ⟶      r0 = 0x8765 xxxx
MOVW r0, #0x4321   ⟶      r0 = 0x0000 4321
```

upper 16 overwritten

>> 60 second brainstorm <<

What does this function do?
Assume R0 is 20 when the function is called

```
Func MOV   R1, #100
     MUL   R0, R0, R1
     ADD   R0, #100
     BX    LR
```

```
R1 = 100
R0 = R0 * R1 = 20 * 100
R0 = R0 + 100
Return from the function … R0
holds the value 2100
```

Load and Store Addressing Modes

A6.5 Memory accesses

The following addressing modes are commonly permitted for memory access instructions:

Offset addressing

The offset value is added to or subtracted from an address obtained from the base register. The result is used as the address for the memory access. The base register is unaltered.

The assembly language syntax for this mode is:

`[<Rn>,<offset>]`

Pre-indexed addressing

The offset value is applied to an address obtained from the base register. The result is used as the address for the memory access, and written back into the base register.

The assembly language syntax for this mode is:

`[<Rn>,<offset>]!`

Post-indexed addressing

The address obtained from the base register is used, unaltered, as the address for the memory access. The offset value is applied to the address, and written back into the base register.

The assembly language syntax for this mode is:

`[<Rn>],<offset>`

Load

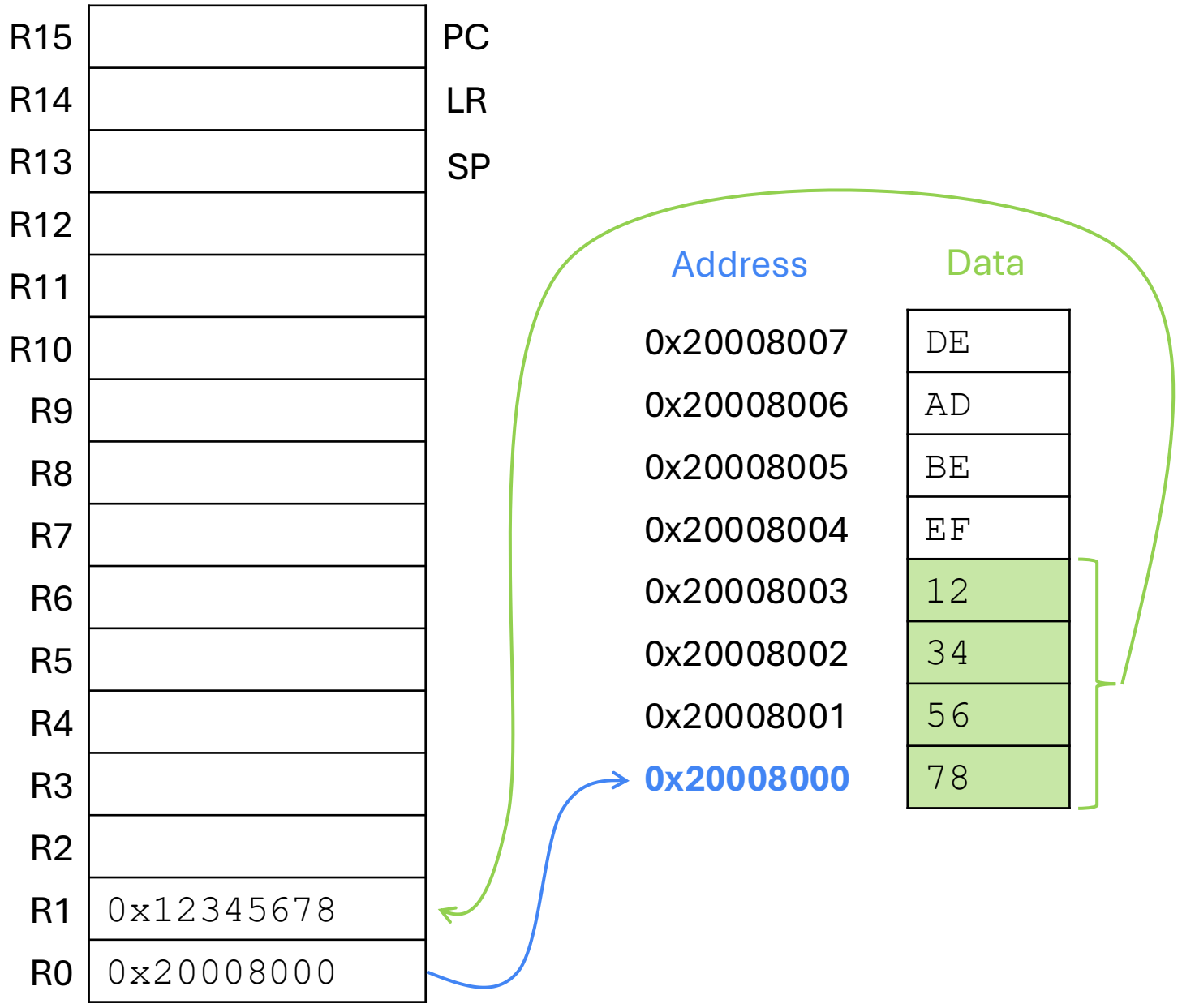
LDR r3, [r1]

- Use the value stored in register "r1" as the address for a location in memory
- Load (copy) the value stored in the memory location defined by “r1” and place it in register “r3”

Load

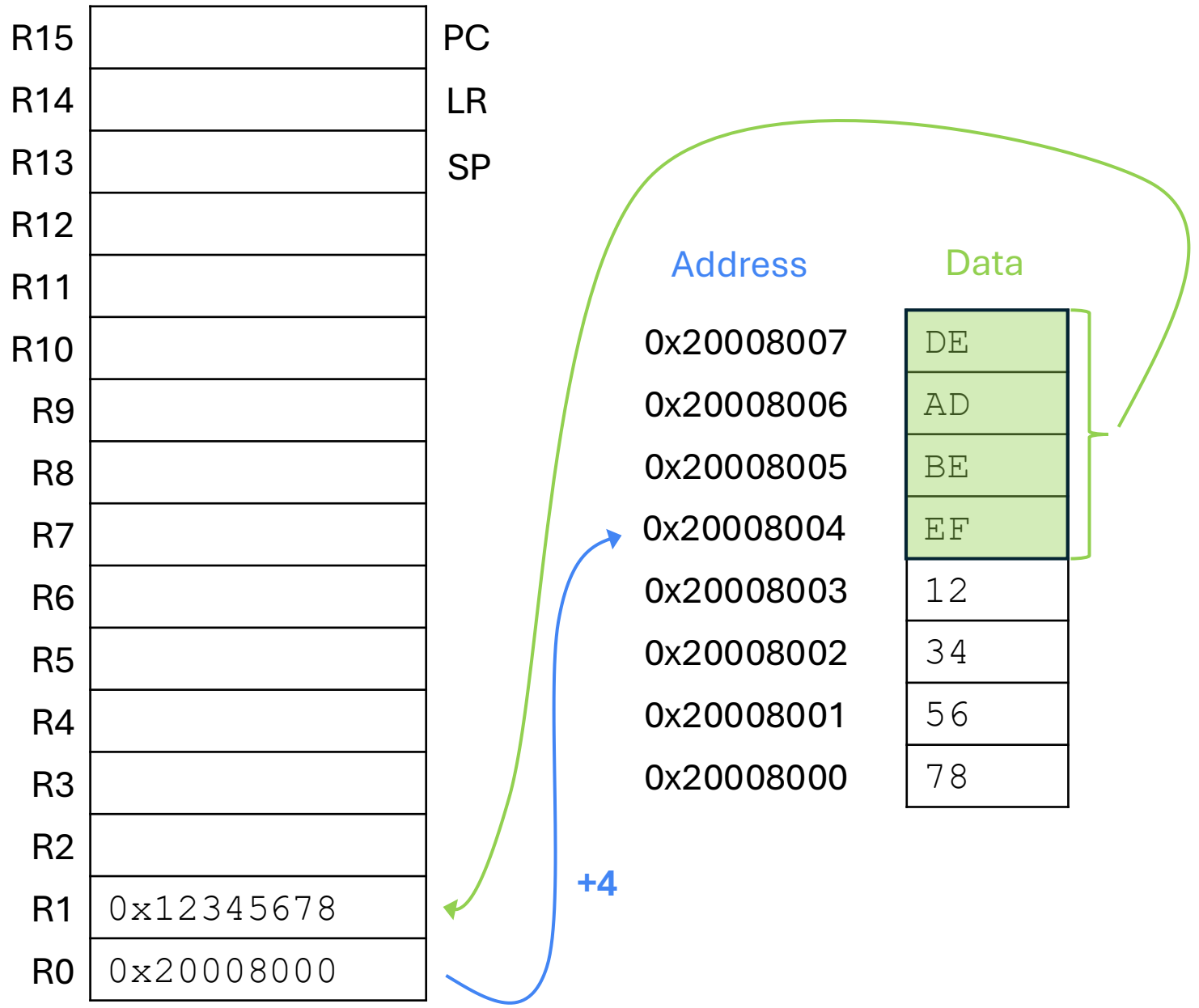
LDR r1, [r0]

- Use the value stored in register ”r0” as the address for a location in memory
- Load (copy) the value stored in the memory location defined by “r0” and place it in register “r1”



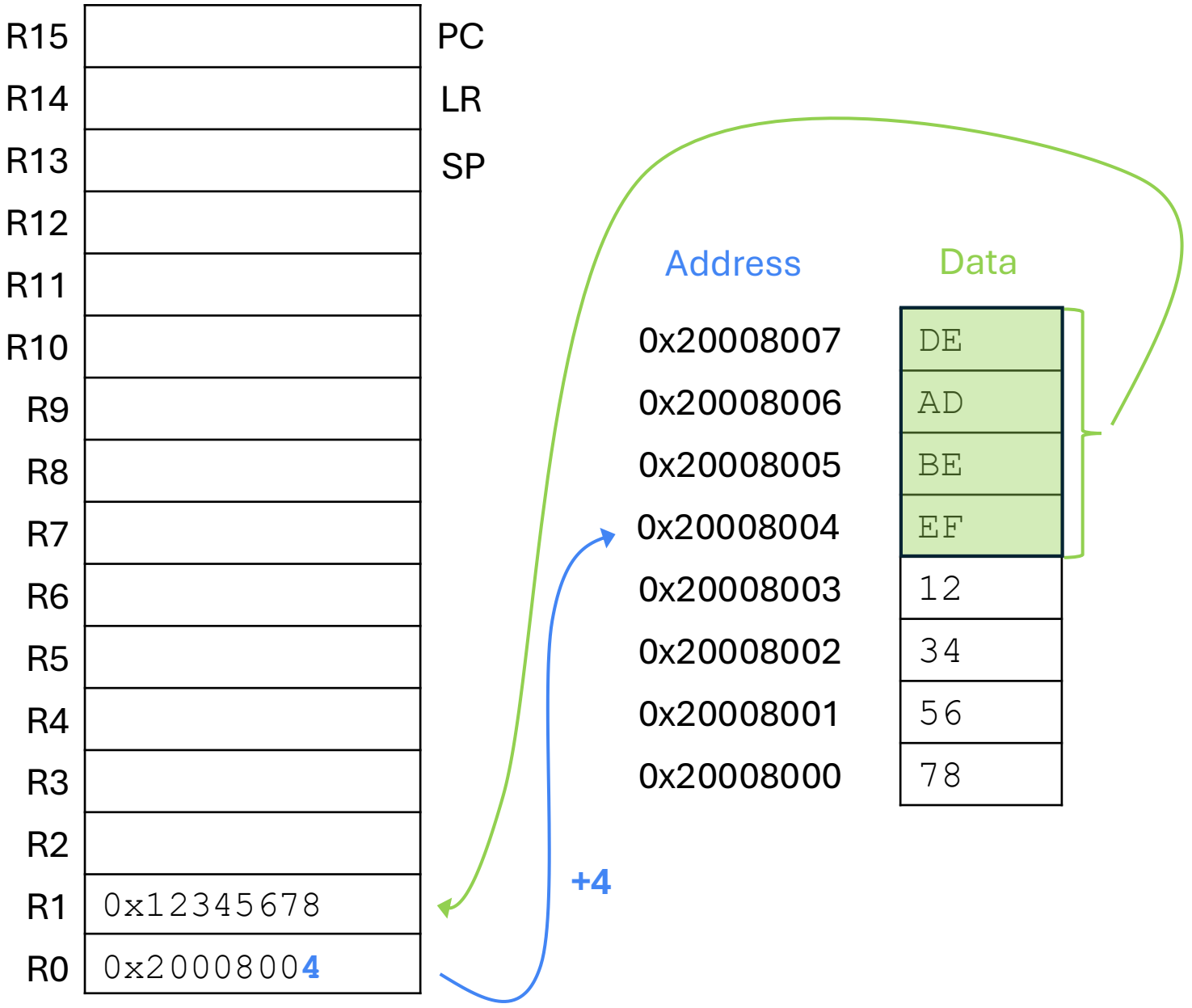
Load: Pre-index

LDR r1, [r0, #4]



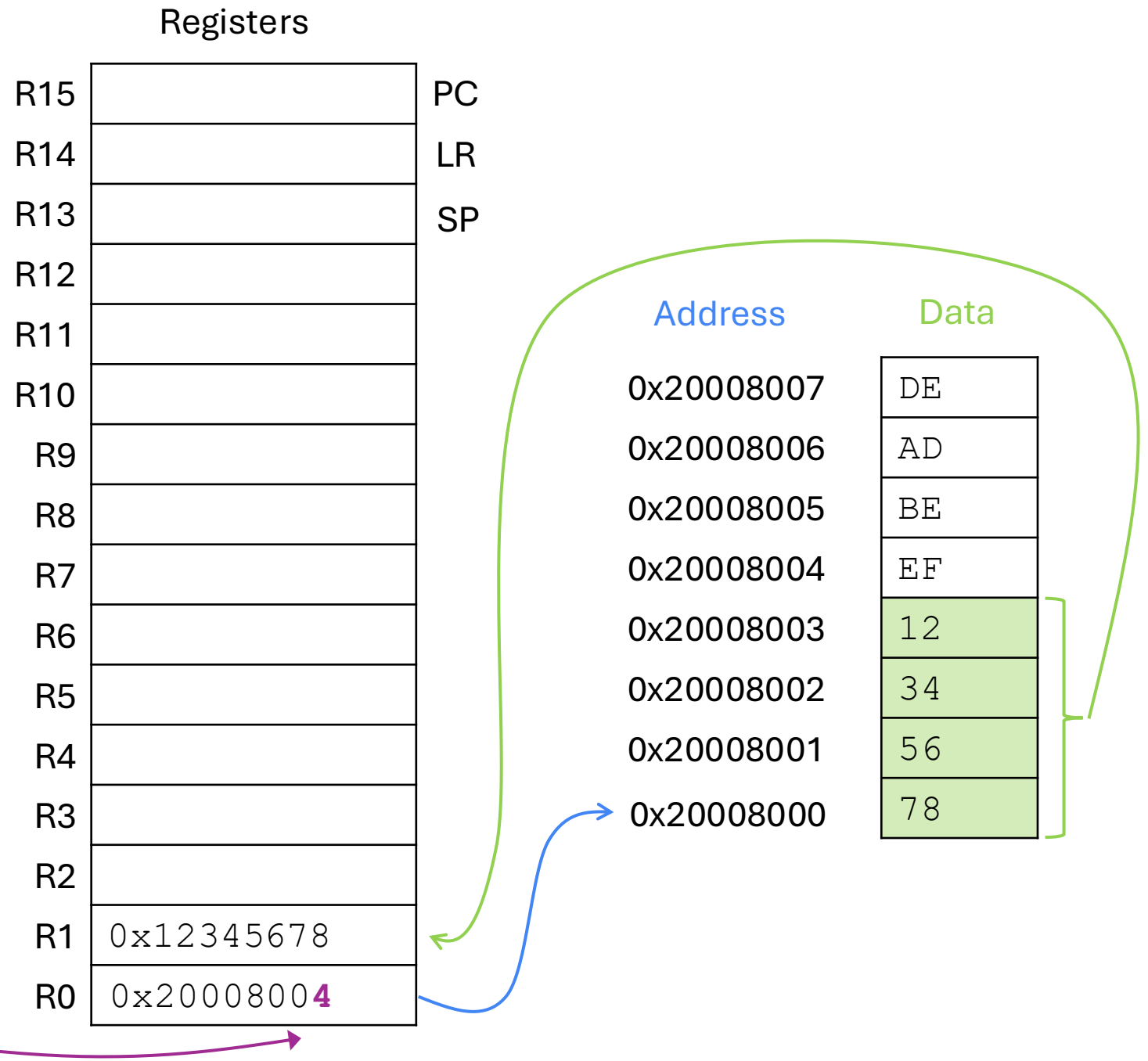
Load:
Pre-Index w/ Update

LDR r1, [r0, #4]!



Load: Post-Index

LDR r1, [r0], #4



Post index in the register after loading from memory

Recap

Index Format	Example	Equivalent
No Index	LDR r1, [r0]	r1 ← memory[r0] r0 is unchanged
Pre-index	LDR, r1, [r0, #4]	r1 ← memory [r0 + 4] r0 is unchanged
Pre-index with update	LDR r1, [r0, #4]!	r1 ← memory[r0 + 4] r0 ← r0 + 4
Post-index	LDR r1, [r0], #4	r1 ← memory[r0] r0 ← r0 + 4

What are we learning about today?

- Instruction Set Architecture (ISA)
- Memory Organization
- Processor Registers and the Application Binary Interface (ABI)
- Loading Code and Data into Memory
- Instruction (Assembly)
  - Arithmetic and Logic Instructions
  - Bitwise Logic
  - Move, Loads, and Stores
  - **Flow Control and Branching**
- Indexing Problem

Branching

A **branch instruction** is used to change the flow of a program execution from a normal sequential order

B	label	Causes a branch to label
BL	label	Branch with link
BLX	Rm	Branch with link and exchange
BX	Rm	Branch and exchange

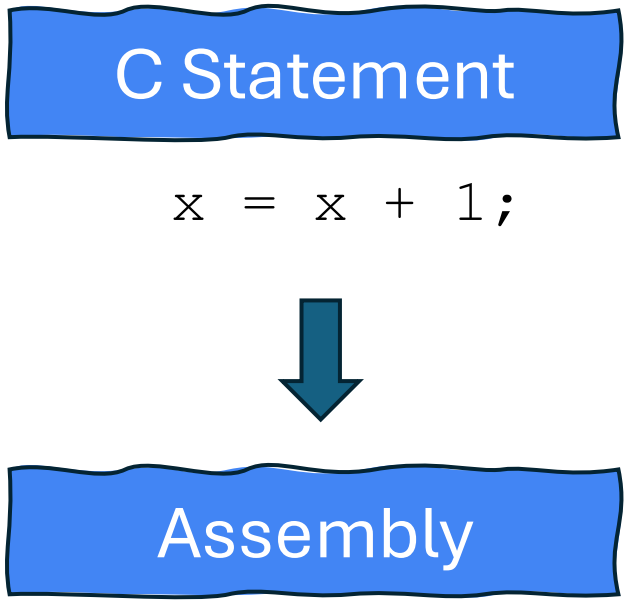
Branch + Conditional

	Instruction	Description	Flags tested
Unconditional Branch	**B** label	Branch to label	none
Conditional Branch	**BEQ** label	Branch if **EQ**ual	Z = 1
	BNE label	Branch if **N**ot **E**qual	Z = 0
	BGT label	Branch if **G**reater **T**han	Z = 0 & N = V
	⋮		
	BLE label	Branch if **L**ess than or **E**qual	Z = 1 or N = !V
	BMI label	Branch if **MI**nus (Negative)	N = 1

Comparison Instructions

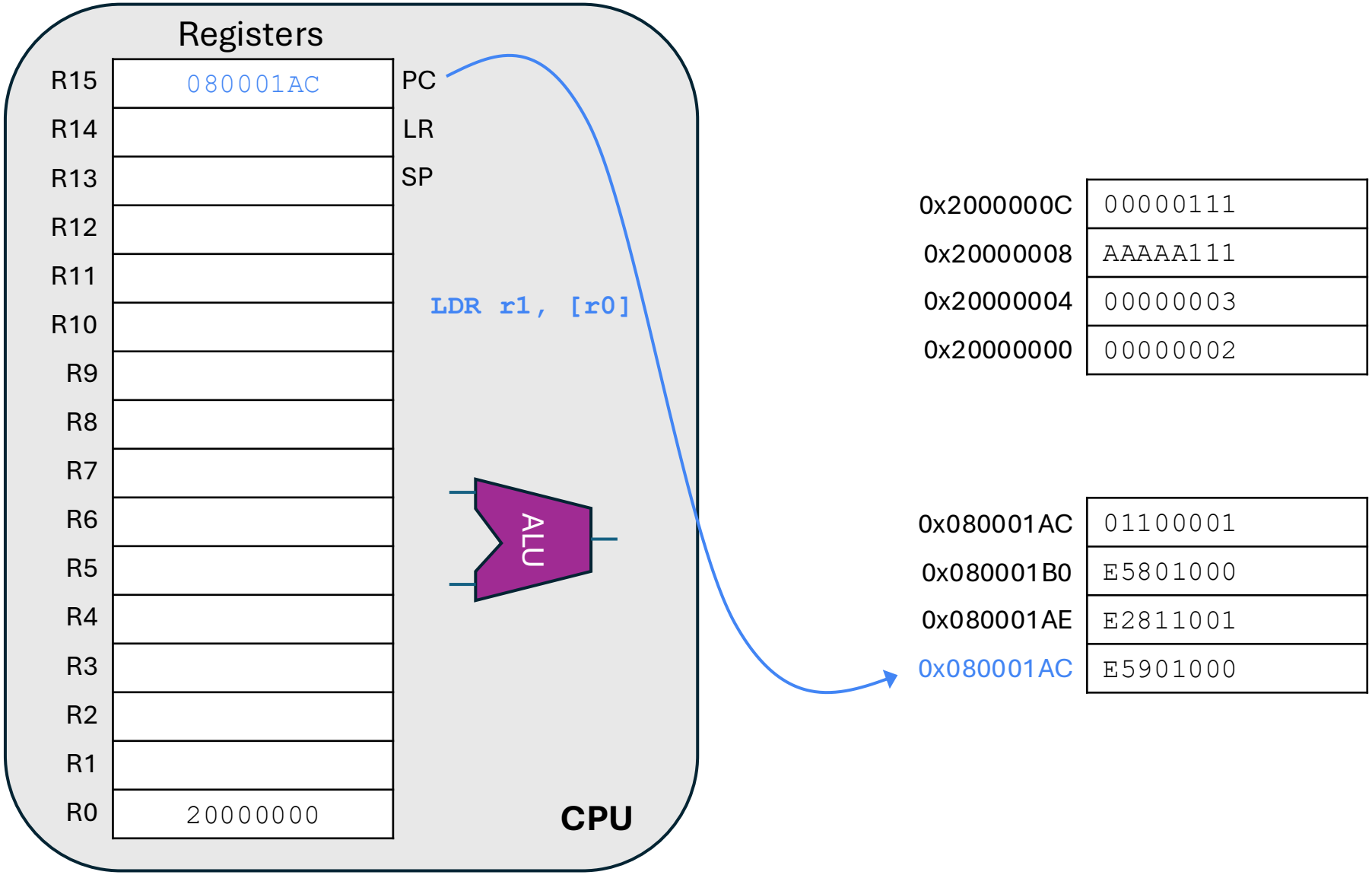
Instruction	Operands	Description	Flags
CMP	Rn, Op2	Compare	N,Z,C,V
CMN	Rn, Op2	Compare Negative	N, Z, C, V
TEQ	Rn, Op2	Test Equivalence	N, Z, C
TST	Rn, Op2	Test	N, Z, C

Example: C vs. Assembly

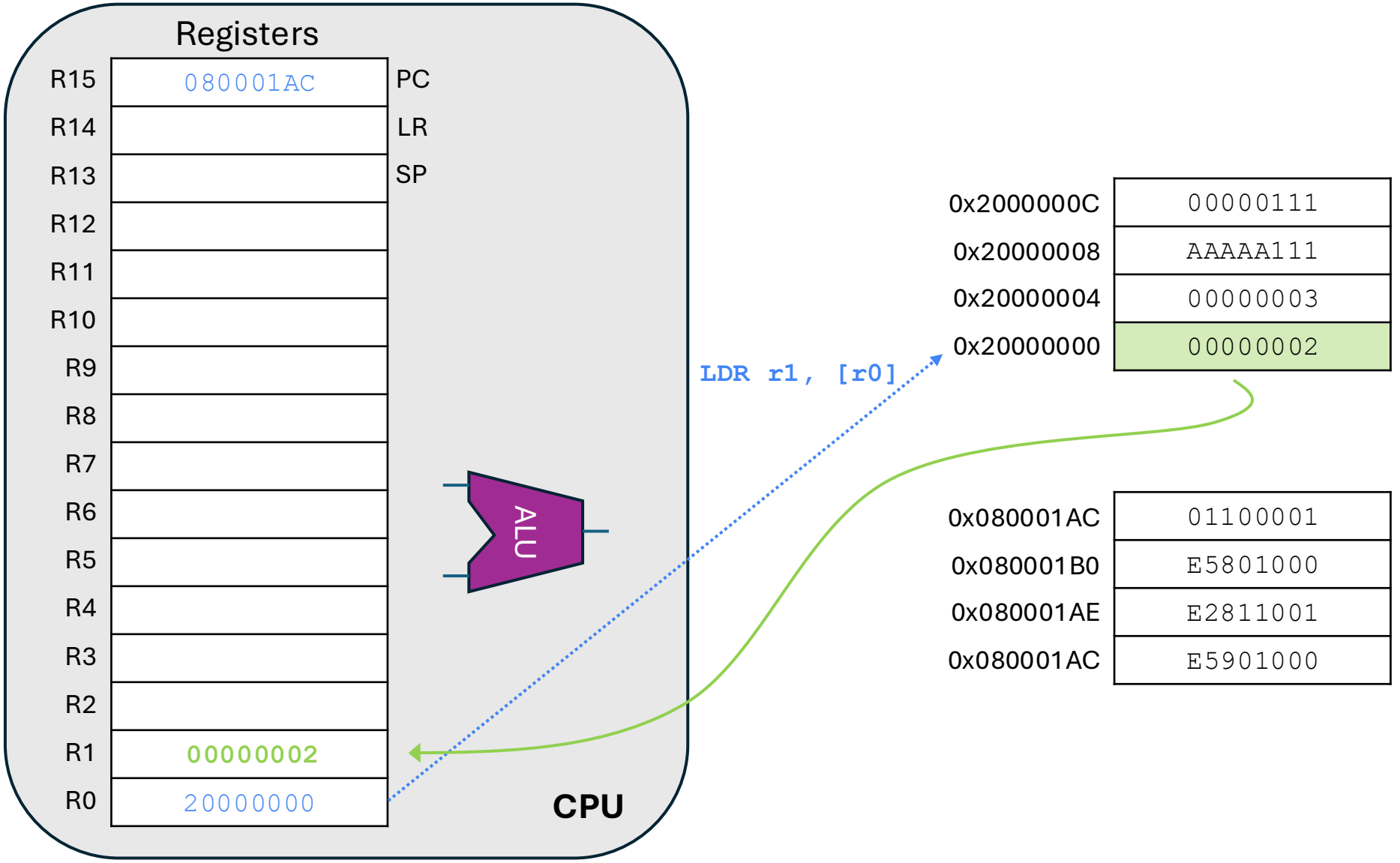



```
; Assume the memory address of x is stored in r0

LDR r1, [r0]     ; load value of x from memory
ADD r1, r1, #1   ; x = x + 1
STR r1, [r0]     ; store x into memory
```

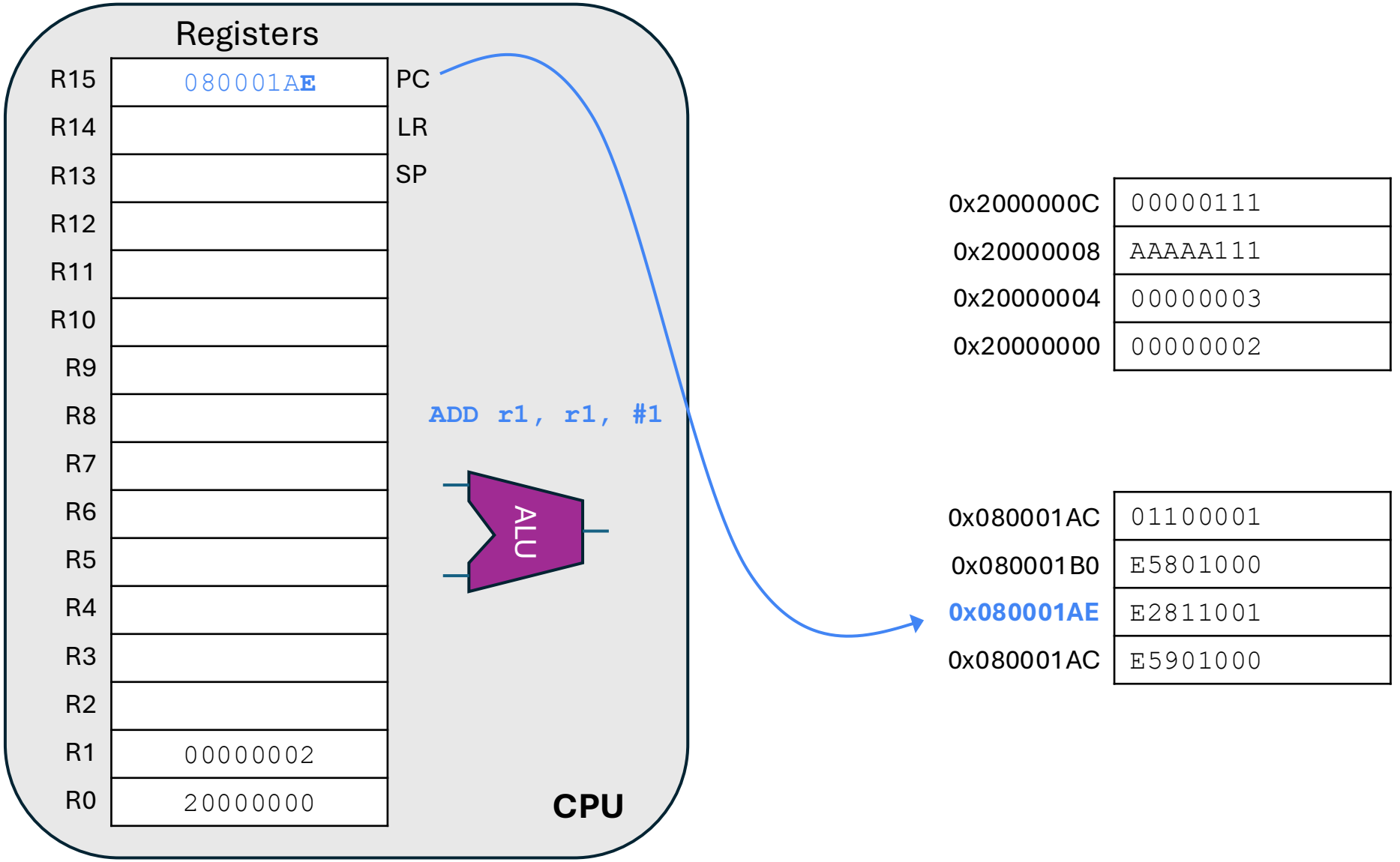
Example: Fetch and Decode Instruction



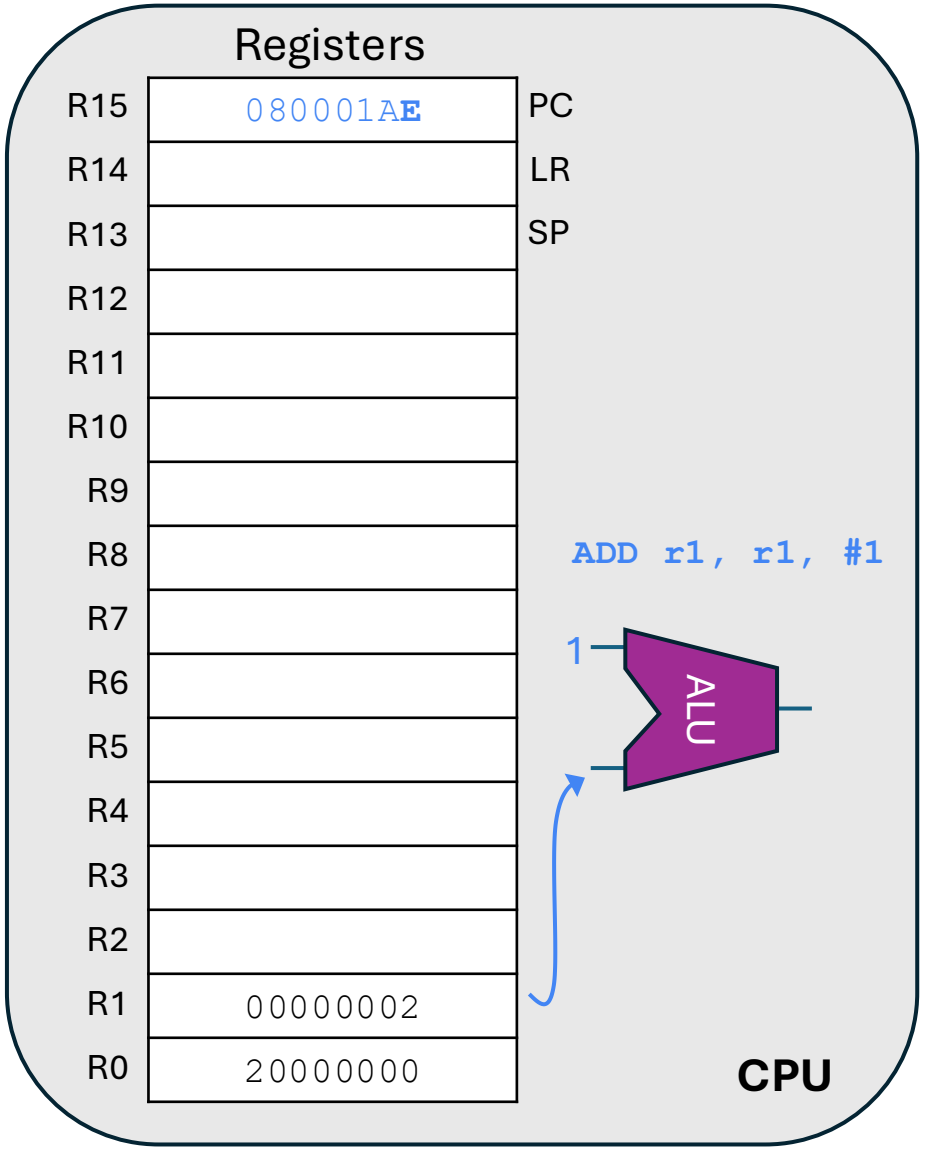
Example: Load



Example: Fetch and Decode Instruction



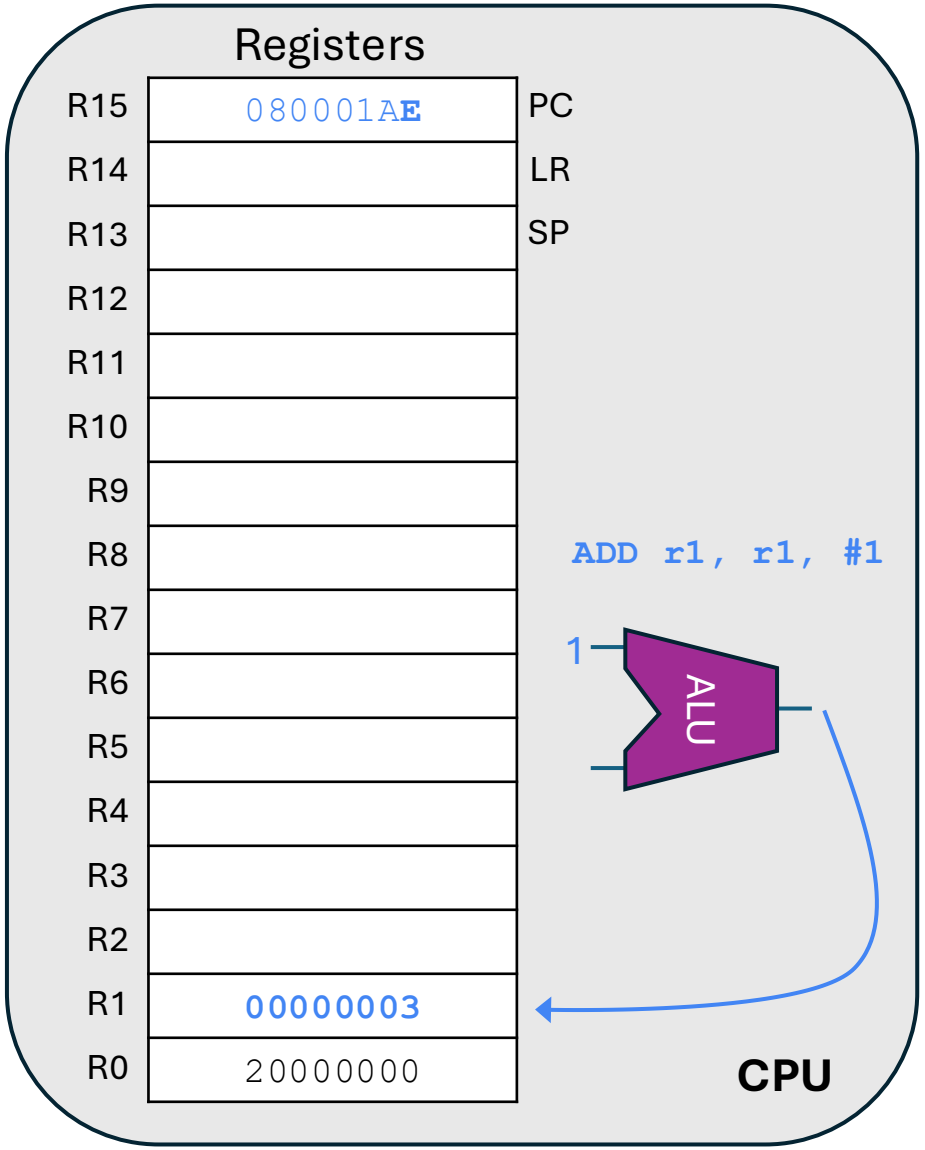
Example: ADD



0x2000000C	00000111
0x20000008	AAAAA111
0x20000004	00000003
0x20000000	00000002

0x080001AC	01100001
0x080001B0	E5801000
0x080001AE	E2811001
0x080001AC	E5901000

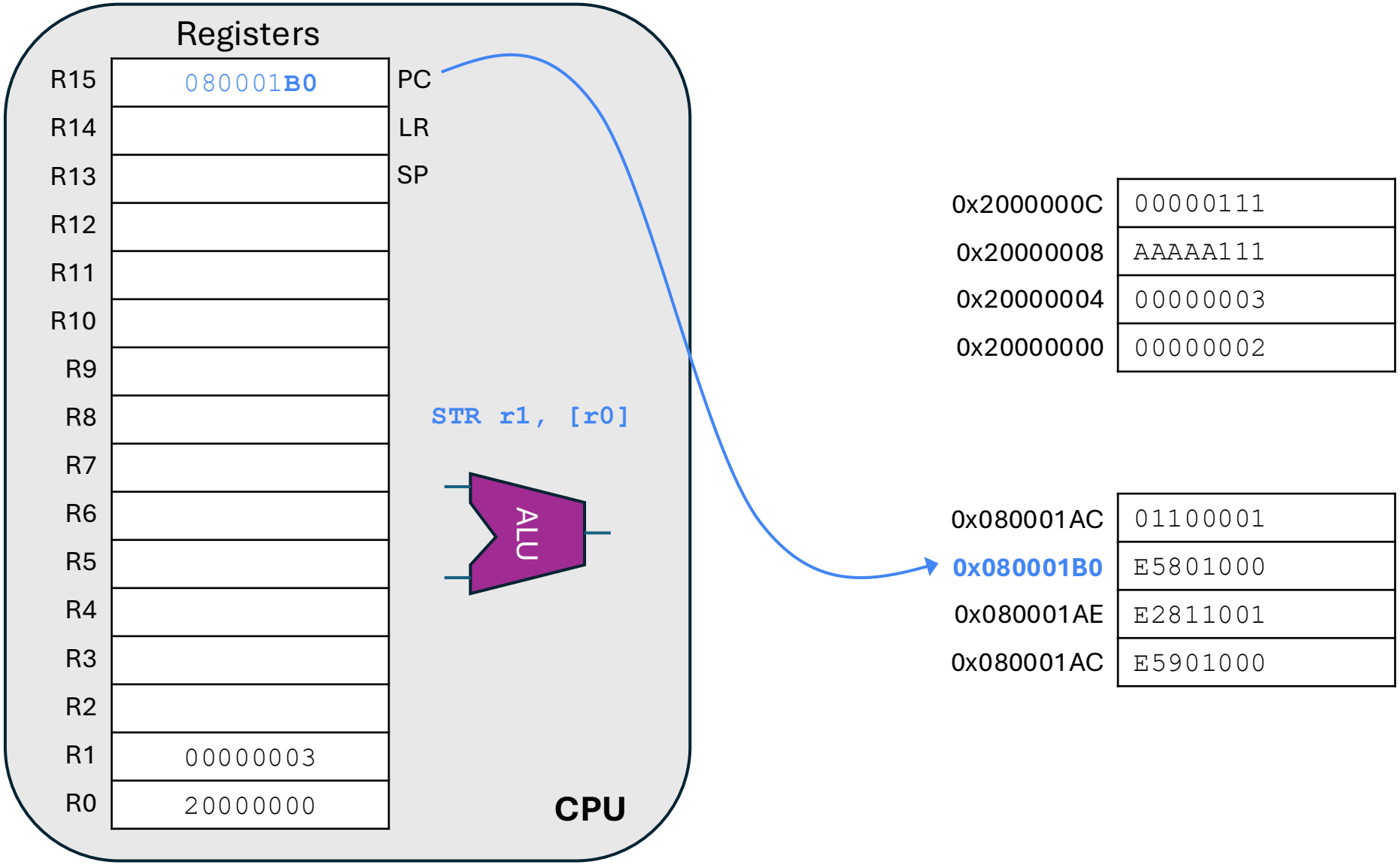
Example: ADD



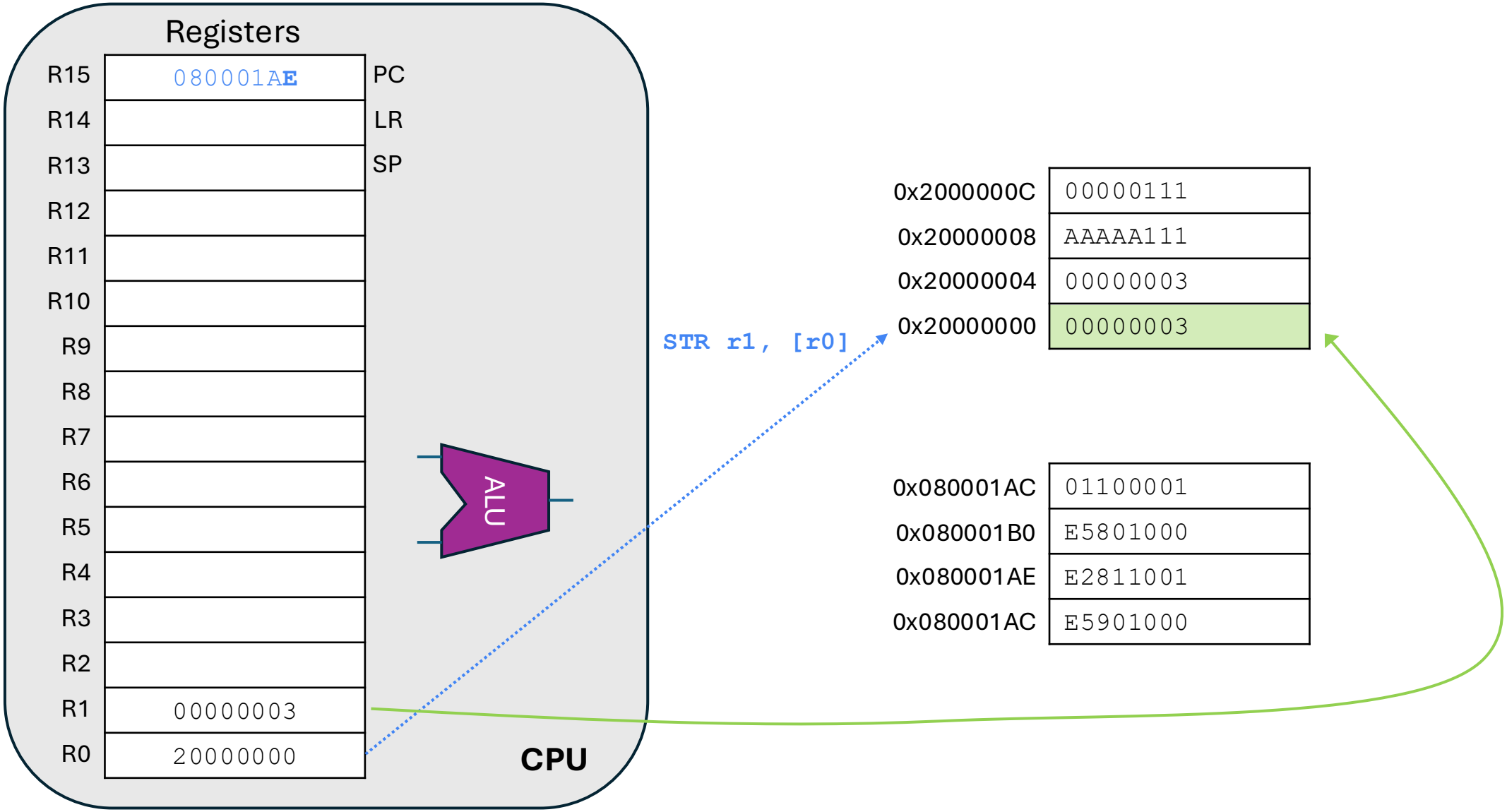
0x2000000C	00000111
0x20000008	AAAAA111
0x20000004	00000003
0x20000000	00000002

0x080001AC	01100001
0x080001B0	E5801000
0x080001AE	E2811001
0x080001AC	E5901000

Example: Fetch and Decode Instruction



Example: Store



Group Exercise

Given the following assembly program and memory table, fill in the missing contents (data in hex) for the unknown ***addresses***. The memory locations are ***byte addressable***. Each memory address below stores a ***word.*** Write all answers in hex.

```
.global start
_start:

        MOV r0, #152
        LDR r1, [r0]

jump:

        STR r1, [r0], #2
        STR r1, [r0, #2]!
        STR r1, [r0, #4]

        ADD r1, r1, #4
        ADD r0, r0, #4

        B jump
```

Memory

Address	Contents
…	…
0x98	0x00000064
0x9C	
0xA0	
0xA4	
0xA8	
0xAC	
0xB0	
0xB4	

Group Exercise

Given the following assembly program and memory table, fill in the missing contents (data in hex) for the unknown ***addresses***. The memory locations are ***byte addressable***. Each memory address below stores a ***word.*** Write all answers in hex.

```
.global start
_start:

        MOV r0, #152            r0 = 0x98
        LDR r1, [r0]            r1 = 0x00000064

jump:

        STR r1, [r0], #2        Mem[0x98] = 0x00000064 ; r0 = 0x9A
        STR r1, [r0, #2]!       r0 = 0x9C ; Mem[0x9C] = 0x00000064
        STR r1, [r0, #4]        Mem[r0+4] = Mem[0xA0] = 0x00000064

        ADD r1, r1, #4          r1 = r1 + 4 = 0x00000068
        ADD r0, r0, #4          r0 = r0 + 4 = 0xA0

        B jump
```

Cycle #1

Memory

Address	Contents
…	…
0x98	0x00000064
0x9C	0x00000064
0xA0	0x00000064
0xA4	
0xA8	
0xAC	
0xB0	
0xB4	

Group Exercise

Given the following assembly program and memory table, fill in the missing contents (data in hex) for the unknown ***addresses***. The memory locations are ***byte addressable***. Each memory address below stores a ***word.*** Write all answers in hex.

```
.global start
_start:

        MOV r0, #152
        LDR r1, [r0]

jump:

        STR r1, [r0], #2
        STR r1, [r0, #2]!
        STR r1, [r0, #4]

        ADD r1, r1, #4
        ADD r0, r0, #4

        B jump
```

Cycle #1

Memory

Address	Contents
…	…
0x98	0x00000064
0x9C	0x00000064
0xA0	0x00000064
0xA4	
0xA8	
0xAC	
0xB0	
0xB4	

Cycle #2

Memory

Address	Contents
…	…
0x98	0x00000064
0x9C	0x00000064
0xA0	0x00000068
0xA4	0x00000068
0xA8	0x00000068
0xAC	
0xB0	
0xB4	

Cycle #3

Memory

Address	Contents
…	…
0x98	0x00000064
0x9C	0x00000064
0xA0	0x00000068
0xA4	0x00000068
0xA8	0x0000006C
0xAC	0x0000006C
0xB0	0x0000006C
0xB4	

Cycle #4

Memory

Address	Contents
…	…
0x98	0x00000064
0x9C	0x00000064
0xA0	0x00000068
0xA4	0x00000068
0xA8	0x0000006C
0xAC	0x0000006C
0xB0	0x00000070
0xB4	0x00000070

Example: Complete Assembly Program

```
.global _start
.space 8
data:
        .byte 0x12, 20, 0x20, -1
_start:

load:
        mov r0, #0
        mov r4, #0
        movw r1, #:lower16:data
        movt r1, #:upper16:data

loop:
        ldrb r2, [r1], #1
        add r4, r4, r2
        add r0, r0, #1
        cmp r0, #4
        bne loop

deadloop:
        b deadloop
_end:
```

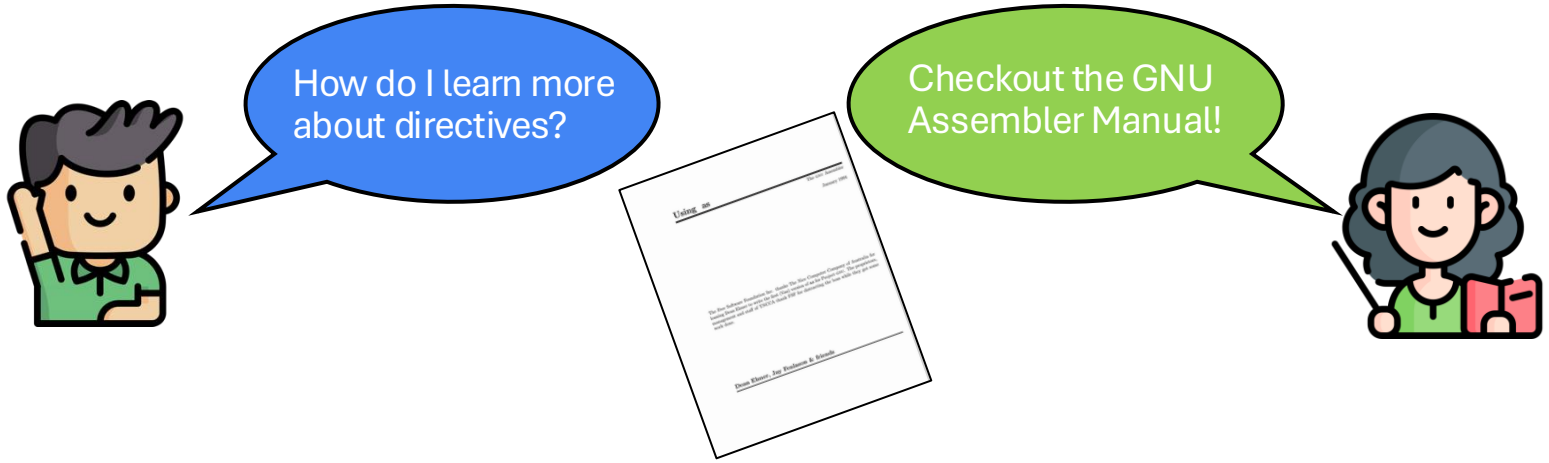
Directives

Labels

Code

Directives tell the "assembler" what to do

Directives



Directive	Short Description
.space size , fill	Emits size bytes, each of value fill. Both size and fill are absolute expressions. If fill is omitted, fill is assumed to be zero.
.byte expressions	Expects zero or more expressions. Each expression is assembled into the next byte
#:lower16:data	Extract the lower 16 data bits
#:upper16:data	Extract the upper 16 data bits

Assembly Program Example

```
.global _start
.space 8
data:
        .byte 0x12, 20, 0x20, -1
_start:

load:
        mov r0, #0
        mov r4, #0
        movw r1, #:lower16:data
        movt r1, #:upper16:data

loop:
        ldrb r2, [r1], #1
        add r4, r4, r2
        add r0, r0, #1
        cmp r0, #4
        bne loop

deadloop:
        b deadloop
_end:
```

hints that it is a 32 bit encoding

MOV → writes an immediate value to the destination register

MOV**W** → will zero the upper 16 bits

MOV**T** → writes an immediate value to the *top halfword* of the destination register

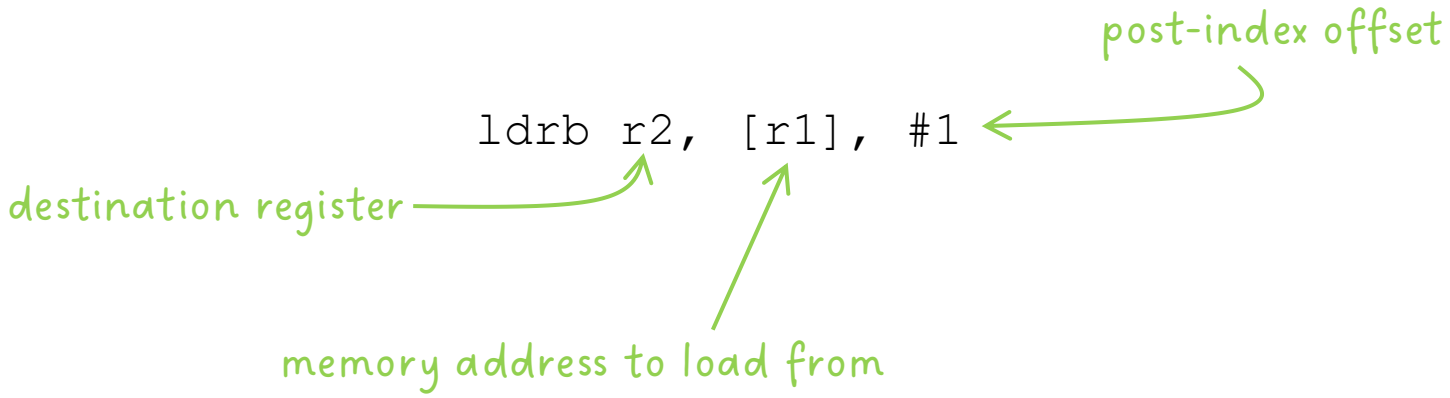
Assembly Program Example

```
.global _start
.space 8
data:
        .byte 0x12, 20, 0x20, -1
_start:

load:
        mov r0, #0
        mov r4, #0
        movw r1, #:lower16:data
        movt r1, #:upper16:data

loop:
        ldrb r2, [r1], #1
        add r4, r4, r2
        add r0, r0, #1
        cmp r0, #4
        bne loop

deadloop:
        b deadloop
_end:
```

LDRB → Load Register Byte (register) calculates an address from a base register value and an offset register value

Loads a byte from memory, zero-extends it to form a 32-bit word and writes it to a register.

The offset register value can be shifted left by 0, 1, 2, or 3 bits



Assembly Program Example

```
.global _start
.space 8
data:
        .byte 0x12, 20, 0x20, -1
_start:

load:
        mov r0, #0
        mov r4, #0
        movw r1, #:lower16:data
        movt r1, #:upper16:data

loop:
        ldrb r2, [r1], #1
        add r4, r4, r2
        add r0, r0, #1
        cmp r0, #4
        bne loop

deadloop:
        b deadloop
_end:
```

CMP → Compare (immediate) subtracts an immediate value from a register value. It updates the condition flags based on the result and discards the result.

B → Branch causes a branch to a target address.

BNE → Branch if not equal

Group Exercise

```
.global _start
.space 8
data:
        .byte 0x12, 20, 0x20, -1
_start:

load:
        mov r0, #0
        mov r4, #0
        movw r1, #:lower16:data
        movt r1, #:upper16:data

loop:
        ldrb r2, [r1], #1
        add r4, r4, r2
        add r0, r0, #1
        cmp r0, #4
        bne loop

deadloop:
        b deadloop
_end:
```

1. What does this program do?
2. What is the final value of registers R0, R2, and R4?
3. What does ***deadloop*** do? Why include it?

Acknowledgements

- These lecture slides were adapted from the University of Michigan's EECS 373 course (many thanks to Alanson Sample!)

- These lecture slides also leverage materials from the following books:
  - Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C
    - Yifeng Zhu
  - The Definitive Guide to ARM Cortex – M3 and Cortex – M4 Processors
    - Joseph Yiu
  - Introduction to ARM Cortex – M Microcontrollers, Embedded Systems
    - Jonathan W. Valvano