

CG2028
Computer Organization

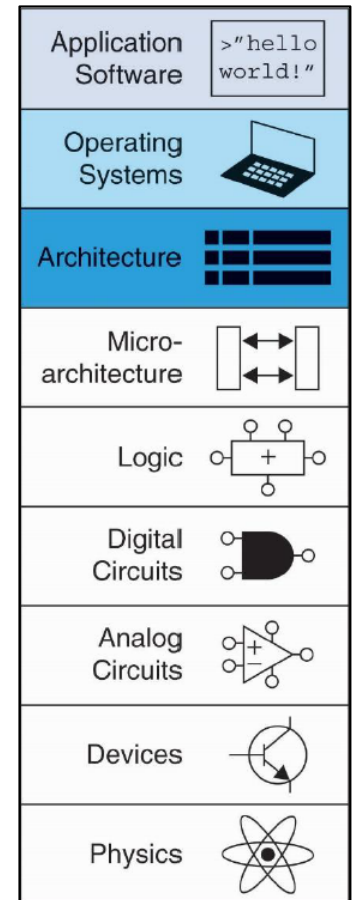
Lecture 1: Introduction & Microprocessor Concepts

Dr Henry Tan, ECE, NUS
E-mail: eletanh@nus.edu.sg



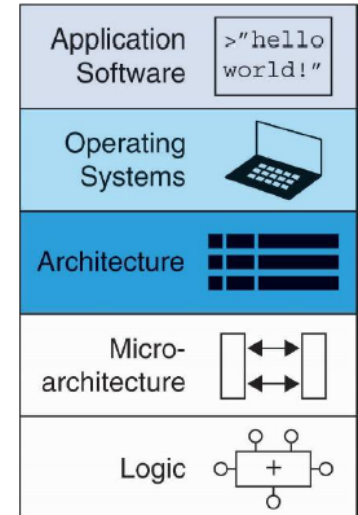
What is Computer Organization?

- Aka Microarchitecture (μ arch or μ arch):
 - Internal implementation of a computer at the register & functional unit level
 - Has to abide by the rules set out in the Computer Architecture
- Computer Architecture, aka Instruction Set Architecture (ISA)
 - Programmer's abstraction of a computer
 - Dictates what a computer can & cannot do
 - In software: instruction set & assembly language (asm)
 - In hardware: technological specifications
 - It is the contract that binds the software & the hardware



What you will learn in this module

- Computer organization basic concepts:
 - Processor microarchitecture
 - datapath & controller design
 - Memory system fundamentals
 - Pipelining basics
- ARM microprocessor instruction set & assembly language (asm)
 - How to write efficient microprocessor programs using asm
- In the assignment, you will design & implement an efficient asm solution to an engineering problem



Learning Outcomes

- (a) appreciate computer organization concepts, including processor microarchitecture, addressing, caches & pipelining.
- (b) apply knowledge of microprocessor concepts to program a microprocessor or microcontroller using assembly language to implement an efficient solution to an engineering problem.

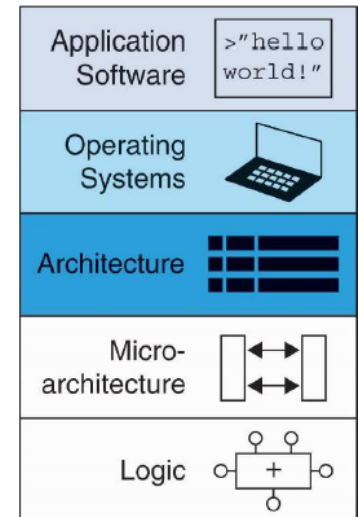
Teaching Team

➤ Lecturers:

- Part I*: Dr Henry Tan
- Part II: Dr Rajesh Panicker

➤ Teaching/Lab Assistants:

- Dr Gu Jing (Tutor)
- Ms Eda Koksai (TA for Lab)
- TBC (~5 Graduate Assistants)



**Acknowledgement: The content of Part I is partly adopted from the original CG2028 material created by my predecessor, Assoc Prof Tham Chen Khong*

Syllabus

Lecture	Topics
1	Intro to module, ARM, LPC1769 & microprocessor concepts <ul style="list-style-type: none">ARM core & system-on-chipfunctional units: I/O, memory, processor with registersmemory organization, addressing & operationsinstruction execution & sequencingRISC vs CISC instruction sets
2	Intro to ARMv7-M Instruction Set & asm <ul style="list-style-type: none">asm instructions: load/store, move, arithmetic operations
3	<ul style="list-style-type: none">branch, conditional execution, logical & stack operations
4 - 6	Microarchitecture, caching & pipelining <ul style="list-style-type: none">processor microarchitecture, datapath & controller designmemory hierarchy, mapping techniques & cachespipelined execution of instructions

Tutorial, Lab Sessions & Assignments

➤ Tutorial Sessions:

- Lecture weeks 2 – 7 (i.e. NUS week 8 – 13)

➤ Lab Sessions (*@ Digital Electronics Lab E4-03-07*)

- Lecture week 3: Assembly Language Programming (3 hrs)
- Lecture week 5: Microarchitecture, Caching & Pipelining (3 hrs)
- Lecture week 6: Assignment Assessment (30 mins)

➤ Assignment

- Lecture weeks 3 – 6:
ARM Assembly Language & Microarchitecture Assignment

Assessments

Assignment (50%):

During Lab #3, Lecture week 6 (i.e. NUS week 12)
@ Digital Electronics Lab (E4-03-07)
30 mins per team of two (schedule to be announced)

Final Quiz (50%):

Saturday 13 Nov 2021 (i.e. NUS week 13)
11:00 am to 12:00 pm
Venue: to be announced

A few references – **no** single recommended textbook.

➤ Supplementary Reading:

- Joseph Yiu, *The Definitive Guide to the ARM Cortex-M3*, Newnes, 2010 (eBook downloadable from NUS library)
- Carl Hamacher, Zvonko Vranesic, Safwat Zaky, Naraig Manjikian: *Computer Organization & Embedded Systems*, 6th Edition, McGraw-Hill, 2012. ISBN 978-007-108900-5.
Note: This book addresses the older ARM7TDMI (ARMv4T) & other processors, while this module focuses on **ARMv7-M** architecture instead. There are some differences.
- Yifeng Zhu, *Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language & C*, 2nd edition, E-Man Press LLC, 2015
- David Harris & Sarah Harris, *Digital Design & Computer Architecture: ARM Edition*, 1st Edition, Morgan Kaufmann Publishers, 2015
- ARM, *ARM[®]v7-M Architecture Reference Manual*
- NXP, *UM10360 LPC17xx User Manual* (Chapter 34)

Introduction to ARM & ARM Cortex-M3 (NXP LPC1769)

CG2028

Dr Henry Tan, ECE, NUS
E-mail: eletanh@nus.edu.sg



ARM Cortex-M3 & ASM

➤ Objectives:

- I. Understand the motivation behind learning ARM & 32-bit ARM Cortex-M3
- II. Differentiation between ARM processor & ARM-based MCUs, e.g. NXP LPC1769 System-on-Chip (SoC)
- III. Introduction to NXP LPC1769 & its key features

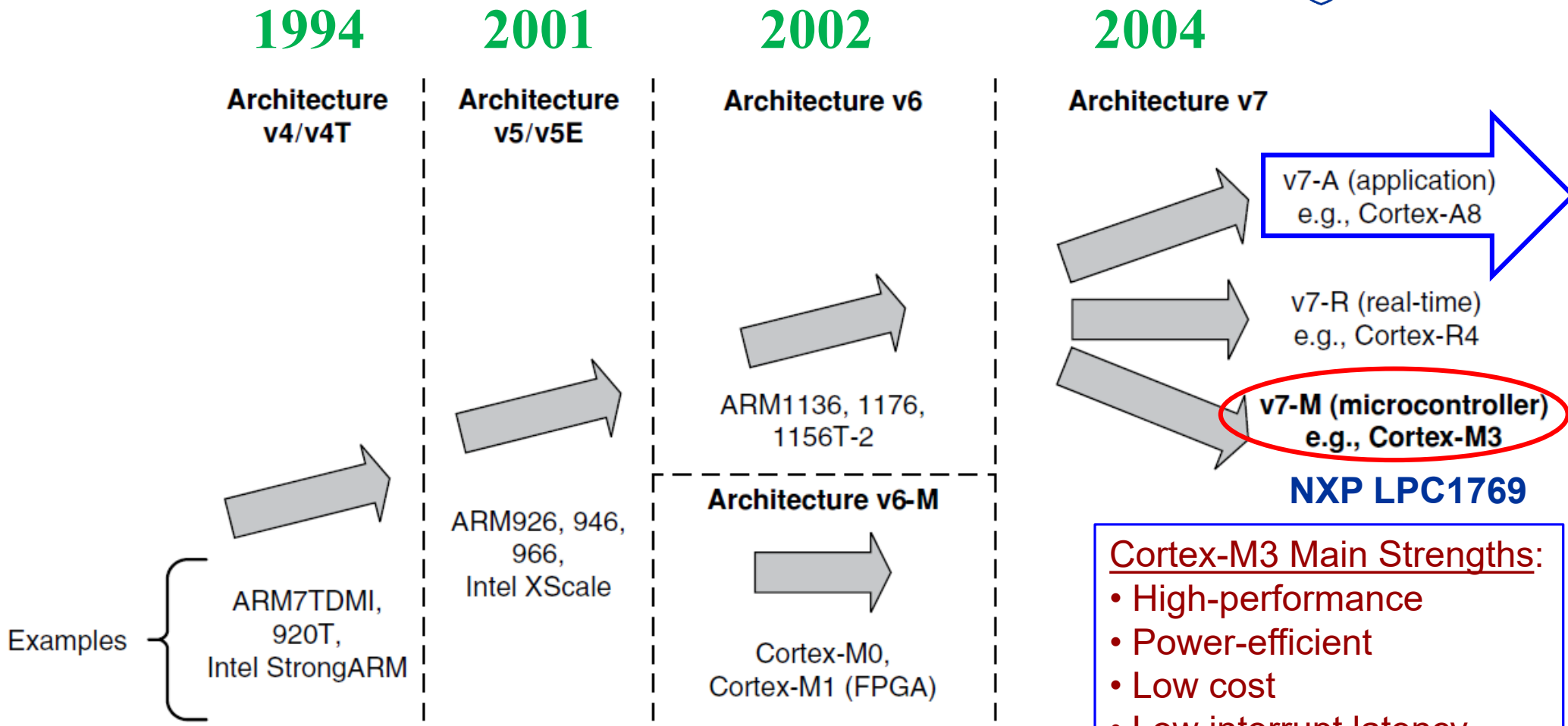
Advanced RISC Machine (ARM)

- Primary business – selling easily integratable IP cores, which allows licensees to create, manufacture & sell their own CPUs, μ P, μ C (MCUs) & SoCs based on the cores
- ARM has since become the most popular embedded processing architecture in the world

As of Feb 2020, 160 billion ARMs shipped !

Product Family	Designed For	Example Use Cases
Cortex-A	Open Application Platforms; High Performance; Power Efficiency	Mobile Devices, Set-top Boxes, for Full-Featured OSs
Cortex-R	Real-time System; Mission-Critical; High Processing Power & Reliability; Low Latency	Airbags/Braking Systems, HDD/SSD Controllers
Cortex-M	Microcontroller Embedded System; Low Power Consumption; Low Cost	IoT Wireless Sensor Nodes, 3D Printers, Home Appliances

ARM Architecture Evolution



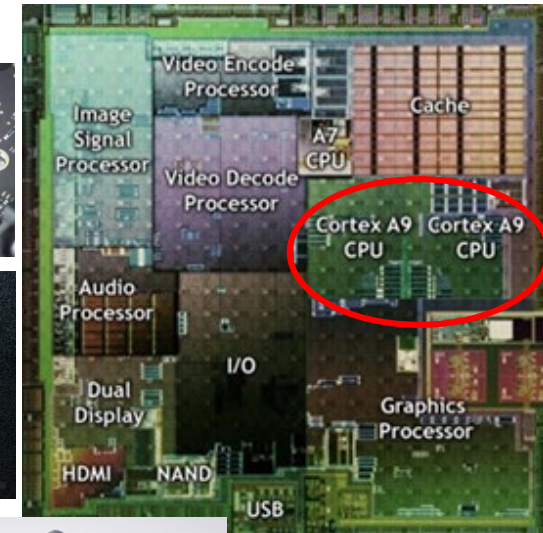
- Cortex-M3 Main Strengths:**
- High-performance
 - Power-efficient
 - Low cost
 - Low interrupt latency
 - Ease-of-use
 - Ease-of-integration

Note: First Cortex-M3 processor shipped in 2004

ARM SoCs in Popular Devices

➤ Apple M1/A15 Bionic SoC (2021)

- CPU: Hexa-core (ARM-based Custom CPU)
- Instruction Set: compatible to **ARMv8.4-A**
- In Apple M1 Macbook Pro/Air, iMac, Mac Mini; iPhone 13, 13 Pro/Pro Max/Mini, iPad Mini 6



➤ Samsung Exynos 2100 SoC (2021)

- CPU: Octa-core (Custom X1, Cortex A78 & A55)
- Instruction set: **ARMv8.2-A**
- In Galaxy S21 Ultra 5G



➤ Qualcomm Snapdragon 898 SoC (~2022)

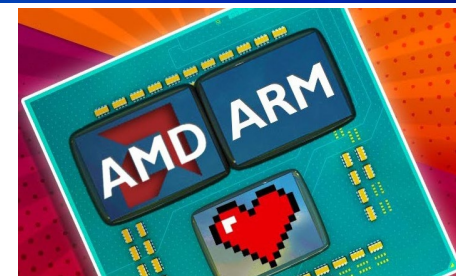
- CPU: Octa-core (Custom X2, Cortex-A710 & A510)
- Instruction Set: **ARMv9-A**
- In Samsung Galaxy S22 Ultra 5G, Galaxy Tab S8; OnePlus 10/10 Pro



➤ Microsoft/AMD: SQ1, SQ2/K12, Opteron (2020/2016)

- In MS Azure cloud servers & Surface PCs (Laptop3, Pro X)
- In A1100 series servers

(NB: 64-bit Only Available From Around 2013)



THE CORTEX-M3 PROCESSOR VERSUS CORTEX-M3-BASED MCUs (e.g. NXP LPC1769)

The Cortex-M3 processor is the central processing unit (CPU) of a microcontroller chip. In addition, a number of other components are required for the whole Cortex-M3 processor-based microcontroller. After chip manufacturers license the Cortex-M3 processor, they can put the Cortex-M3 processor in their silicon designs, adding memory, peripherals, input/output (I/O), and other features. Cortex-M3 processor-based chips from different manufacturers will have different memory sizes, types, peripherals, and features.

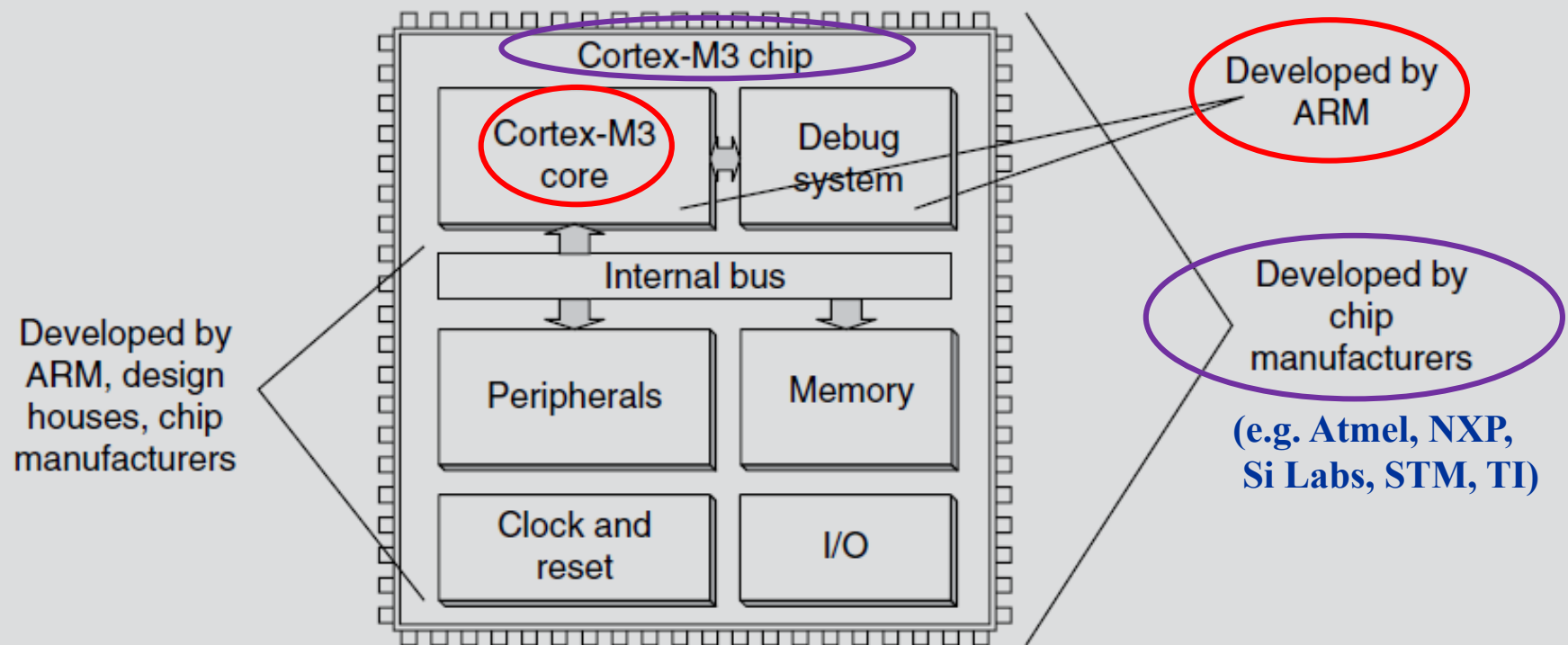
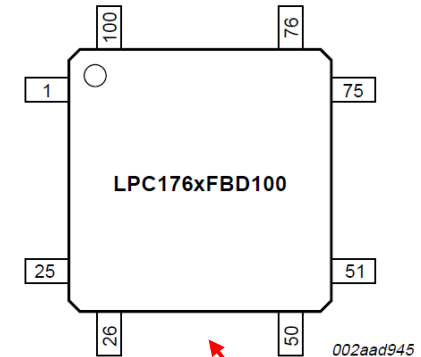


FIGURE 1.1

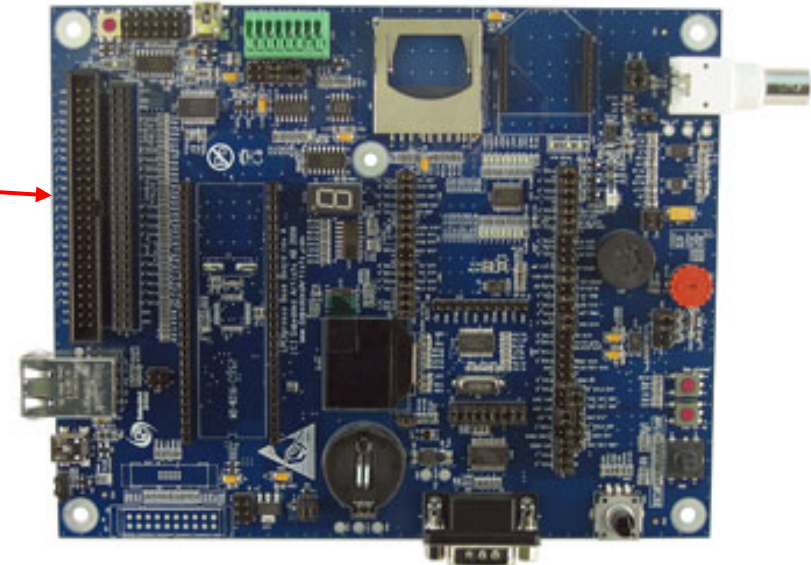
The Cortex-M3 Processor versus the Cortex-M3-Based MCU.

NXP LPC1769 Introduction_1

- Operates at 120 MHz
- 100 pin packaging
 - Many of these pins have 4 functions
 - Desired function is programmable
(via pin connect block, PINSEL)
- **LPCXpresso board** with IDE & **Baseboard** allow convenient access to LPC1769 functionalities
 - Rich software library provided



Pin configuration LQFP100 package



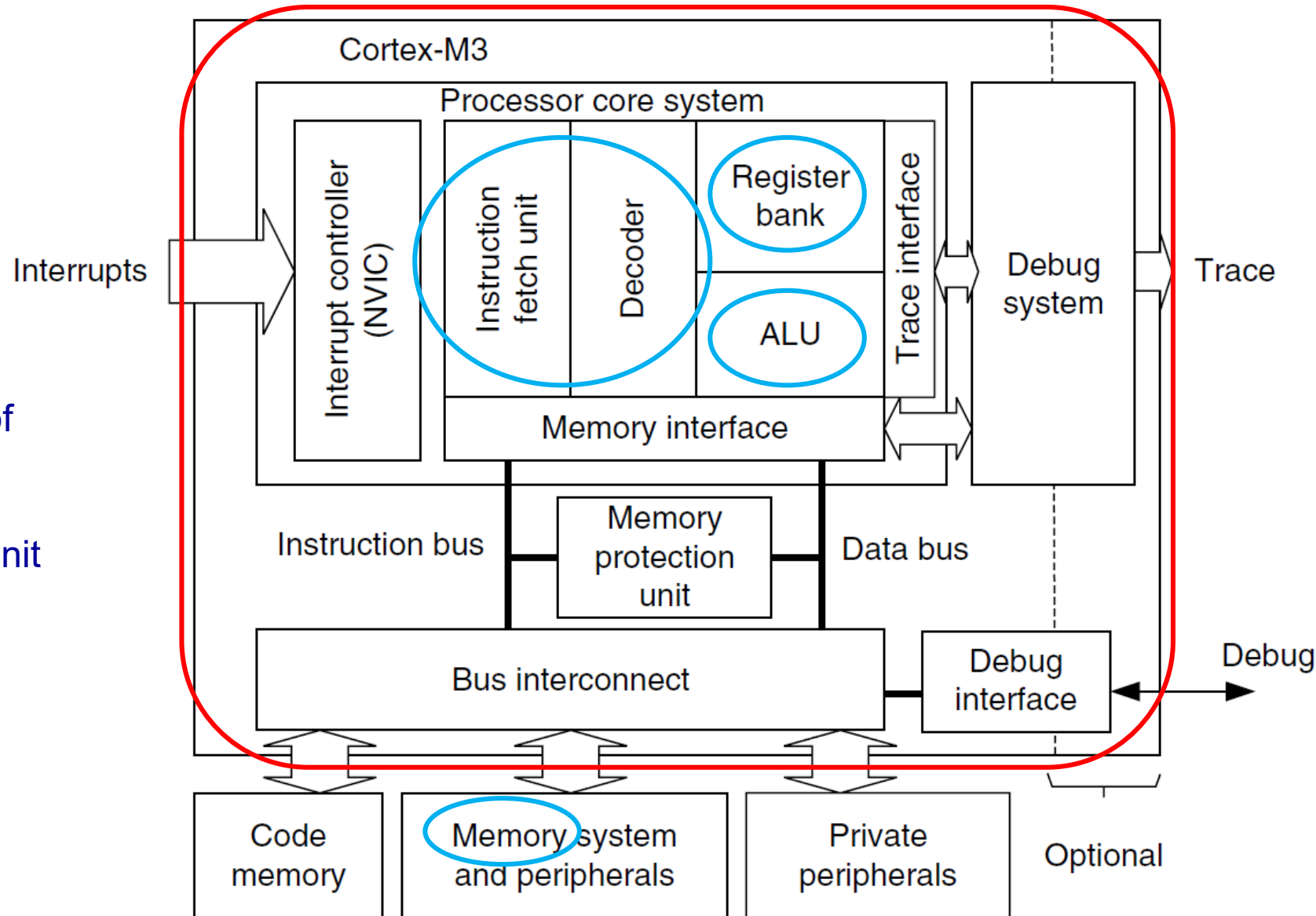
NXP LPC1769 Introduction_2

- Cortex-M3 processor (**32-bit**):
 - 32-bit word length
 - 32-bit data path
 - 32-bit register bank (consisting of 16 registers)
 - 32-bit memory interfaces
- Uses a modified Harvard architecture:
 - **Separate** instruction bus & data bus, & a 3rd bus for peripherals
 - Instructions & data accesses can take place **concurrently**
 - Performance of the processor increases, because data accesses do not affect the instruction pipeline
- Incorporates a **3-stage pipeline**
- 512 KB of Flash Memory
- 64 KB on-chip SRAM includes:
 - 32 kB of SRAM on the CPU with local code/data bus for high-performance CPU access
 - Two 16 KB SRAM blocks with separate access paths for higher throughput
 - These SRAM blocks may be used for Ethernet, USB, DMA memory & for general purpose instruction & data storage

NXP LPC1769 Introduction_3

- Built-in Nested Vectored Interrupt Controller (NVIC)
- Fast General-Purpose Input Output (GPIO) ports
- Ethernet MAC
- A USB interface that can be configured as either Host, Device, or OTG (On The Go)
- 8 channel general-purpose DMA controller
- 2 SSP controllers, SPI interface, 4 UARTs, 2 CAN channels
- 3 I²C interfaces, 2-input plus 2-output I²S (Inter-IC Sound) interface
- 8 channel 12-bit ADC, 10-bit DAC, motor control PWM
- 4 general-purpose timers
- 6-output general-purpose PWM
- Ultra-low power real-time clock (RTC) with separate battery supply

NXP LPC1769: CPU



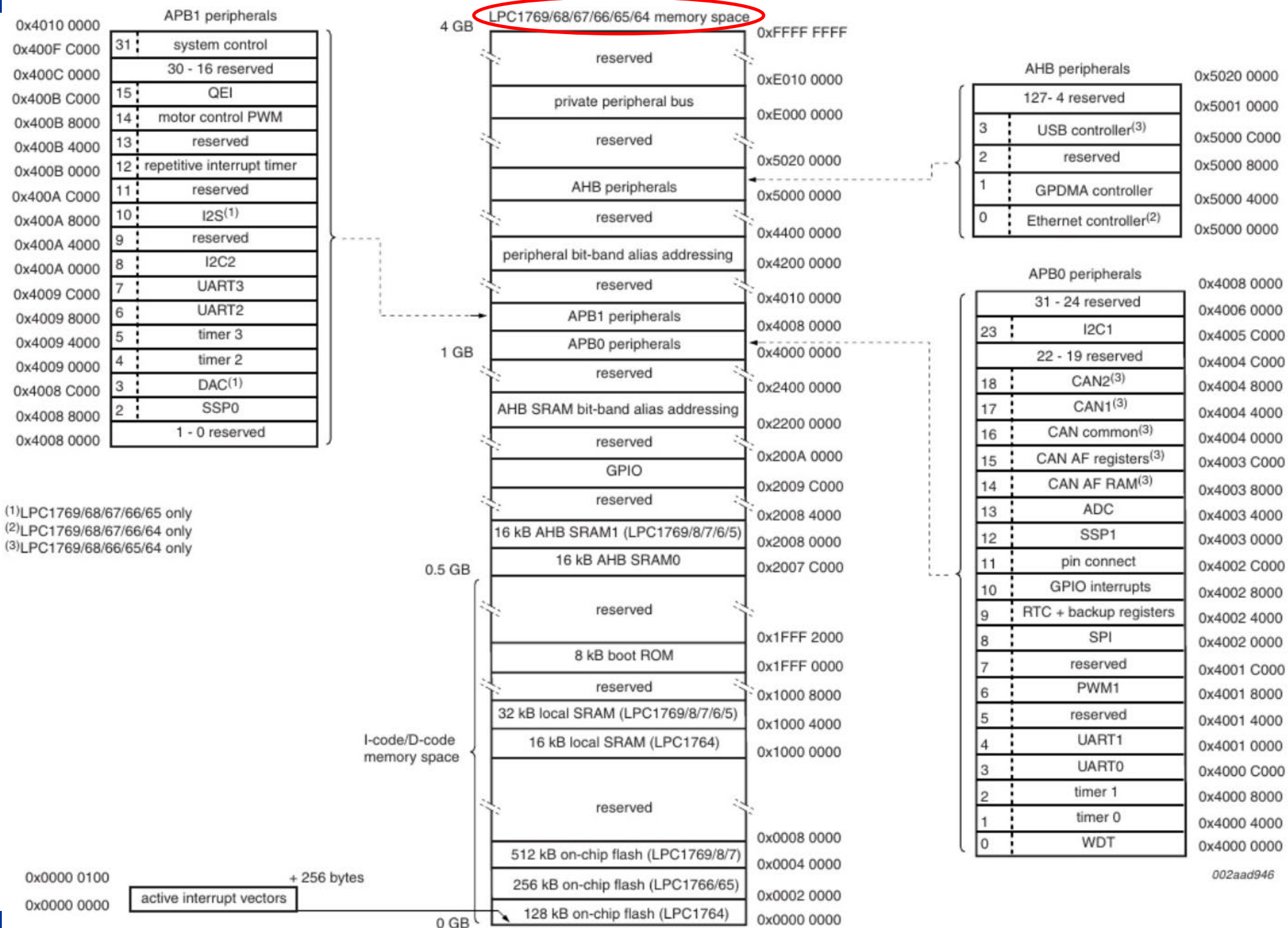
Interests of **Part I** of this module:

- Instruction fetch unit
- Decoder
- ALU
- Register Bank
- Memory

NXP LPC1769: Memory Map

4GB (0xFFFFFFFF) Memory Space

0xFFFFFFFF	System level	Private peripherals including build-in interrupt controller (NVIC), MPU control registers, and debug components
0xE0000000		
0xDFFFFFFF	External device	Mainly used as external peripherals
0xA0000000		
0x9FFFFFFF	External RAM	Mainly used as external memory
0x60000000		
0x5FFFFFFF	Peripherals	Mainly used as peripherals
0x40000000		
0x3FFFFFFF	SRAM	Mainly used as static RAM
0x20000000		
0x1FFFFFFF	CODE	Mainly used for program code. Also provides exception vector table after power up
0x00000000		



Microprocessor Concepts

CG2028

Dr Henry Tan, ECE, NUS
E-mail: eletanh@nus.edu.sg



Microprocessor Concepts

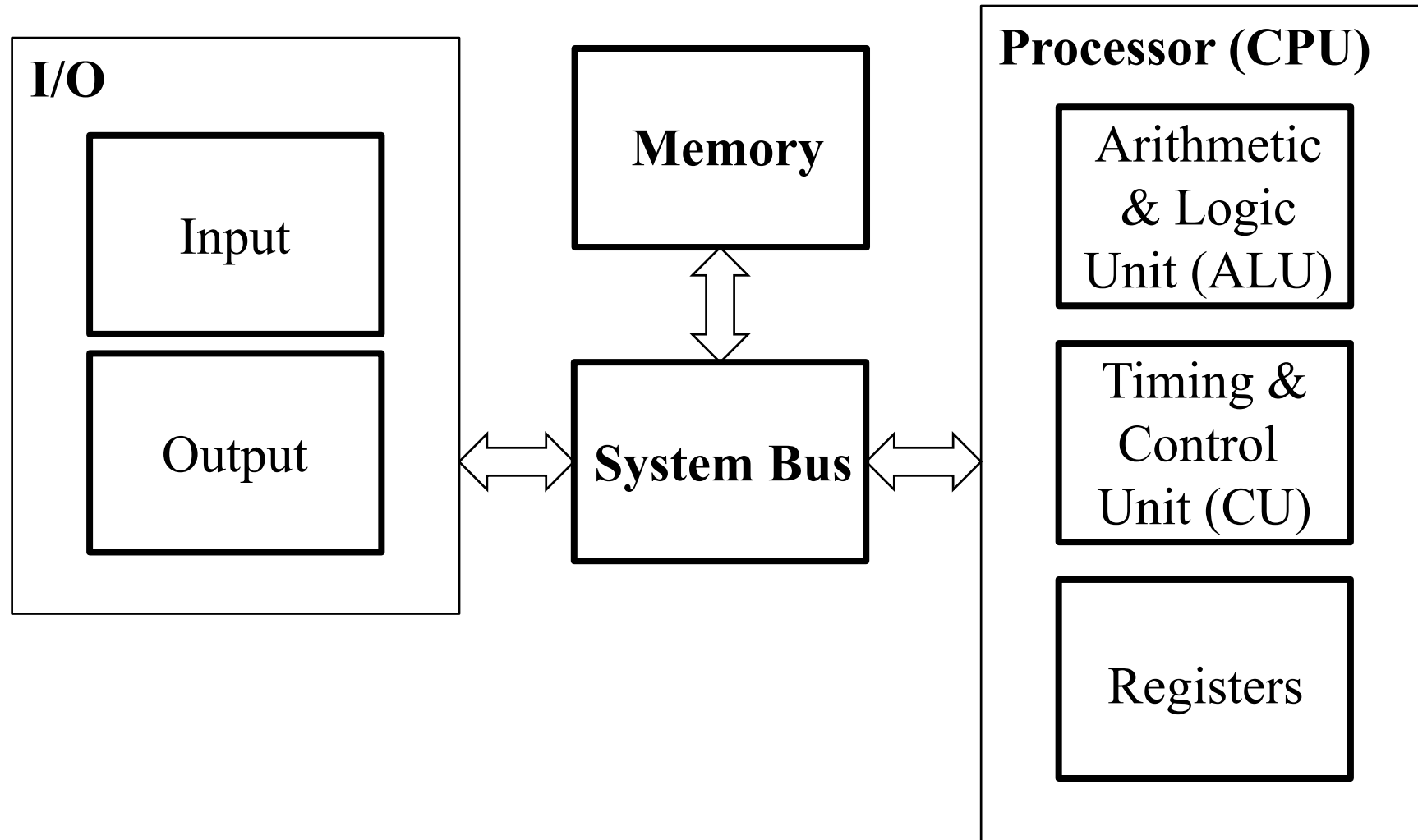
➤ Objectives:

- Understand functional units, memory organization & instruction execution in a computer

➤ Outline:

- 1. Functional units within a computer
- 2. Processor components & their functions
- 3. Memory organization, addressing & operations
- 4. Instructions & sequencing
 - RISC vs CISC instruction sets
 - RISC Program & Storage
 - Instruction execution: fetch- & execute-phase
 - Instruction sequencing: branching & looping, condition codes

1. Functional Units



Functional Units _Typical Examples

➤ Input:

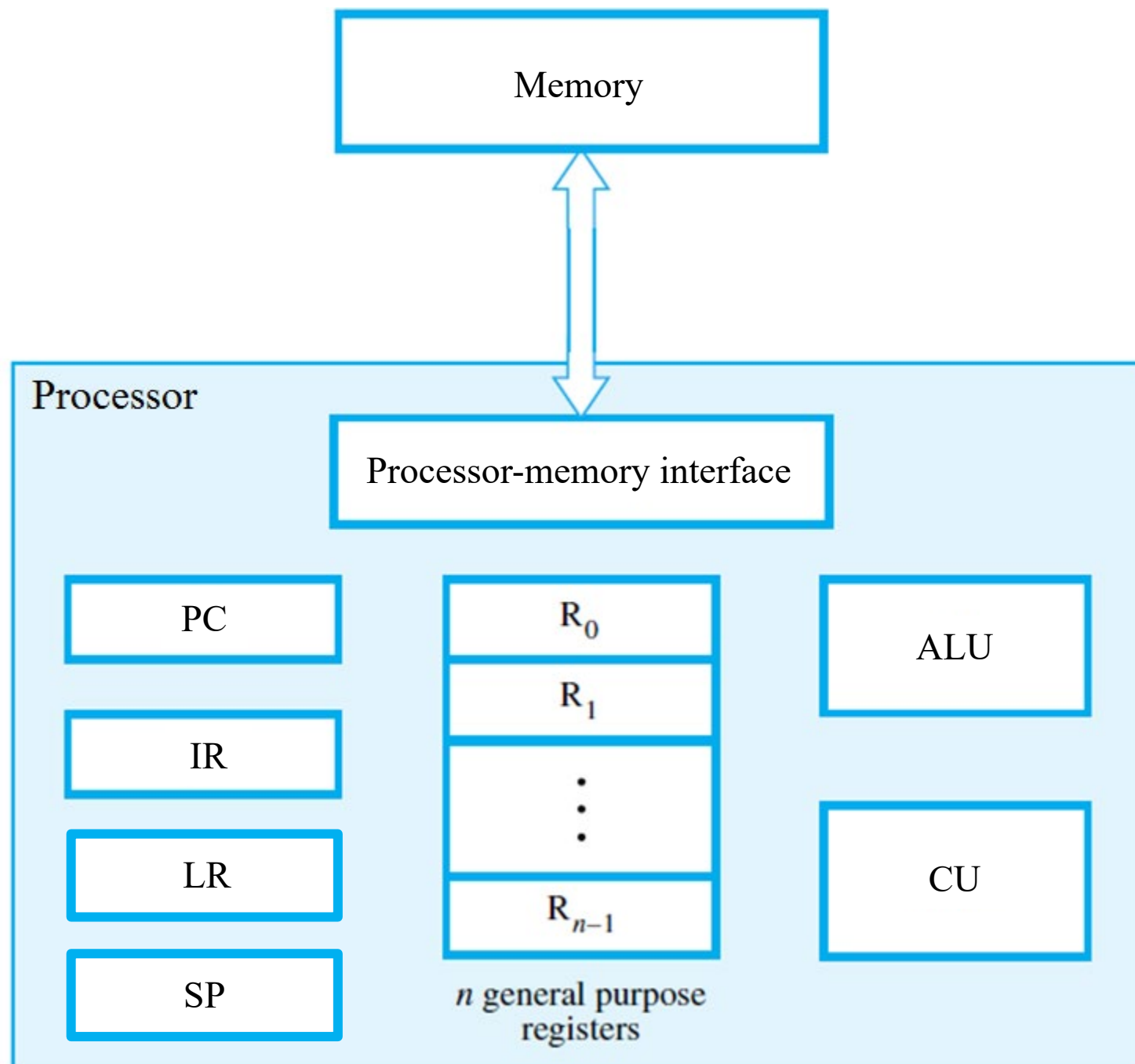
- Keyboard, touchpad, mouse, microphone, camera, sensors, communication lines, the Internet

➤ Output:

- Text & graphics displays, speakers, earbuds, printers

➤ Memory:

- Cache memory (holds sections/blocks)
 - holds part of the program & data currently being executed
 - much smaller & faster than the main & secondary memory
- Main memory (holds pages)
 - Static RAM (SRAM), Dynamic RAM (DRAM)
- Secondary memory (holds files)
 - flash memory devices, magnetic disks, optical disks



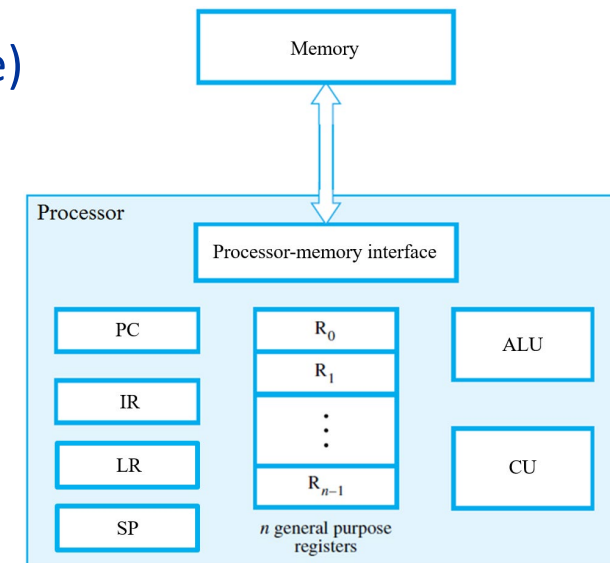
2. Processor Components

- Arithmetic & Logic Unit (ALU) for performing arithmetic & logical operations, on word*-size data operands
- Timing & Control Unit (CU) for fetching program instructions & data from memory, decoding & executing^ them one after another, & transferring the results back to memory, if needed
- Registers (typically 16 or 32), small amounts of fast storage each of which holds one word* of data. Usually classified into general-purpose & special-purpose:
Program Counter (PC), Instruction Register# (IR),
Link Register (LR) & Stack Pointer (SP)

**architecture-dependent; ^by assigning arithmetic & logical tasks to ALU ; #read-only*

The Registers

- The General purpose registers hold **data[%] & addresses**
- The **PC** register holds the **memory address** of the current/next instruction (i.e. fetch/execute phase)
- The **IR** holds the **current instruction**
- The **LR** stores the **return address** when a **subroutine/function** is called
- The **SP** holds the **memory address** of the last (most recent) data in the stack memory



[%] user data or copy of machine instruction

Notes: In our **32-bit** ARM context:
PC = R15, LR = R14 & SP = R13

3. Memory Organization

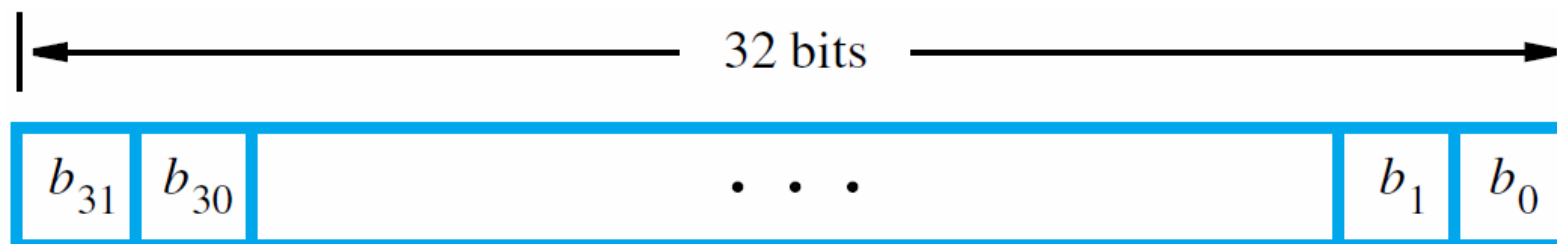
- Memory consists of many millions of **cells**
- Each cell holds a bit of information, 0 or 1
- Information usually handled in larger units: **words/bytes**
- A **word** is a group of n consecutive bits
 - a **word length** is usually between 16 & 64 bits
- The **memory** is thus a **collection of consecutive words** of the size specified by the word length


Notes: In our **32-bit** ARM context:

- word = 32 bits, halfword = 16 bits. *(both are architecture-dependent)*
- byte = 8 bits *(each byte is addressable in the memory)* ; nibble = 4 bits
- hexadecimal: 0x0-0xF (4 bits) (0b0000-0b1111 in binary; 0-15 in decimal)

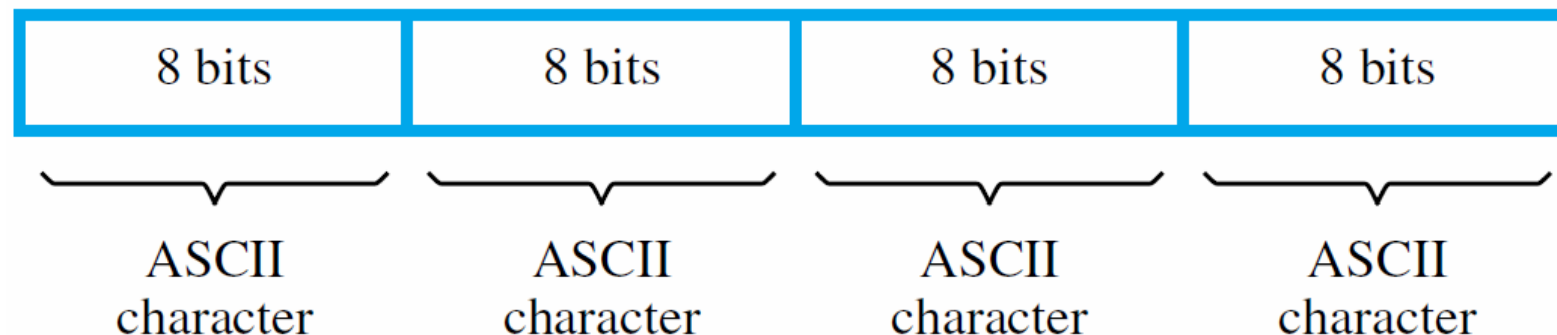
3.1 Word & Byte Encoding

- Consider the common word length of 32 bits
- Such a **word** in memory may store:
 - a machine **instruction** for a program
(or part of it, as some machine instructions may require more than one consecutive words for encoding)
 - or, **data** for the program :
 - a 32-bit unsigned/signed integer (i.e. MSB b_{31} is the sign bit) or
 - four 8-bit bytes (e.g. ASCII character codes)



 Sign bit: $b_{31} = 0$ for positive numbers
 $b_{31} = 1$ for negative numbers

(a) A signed integer



(b) Four characters

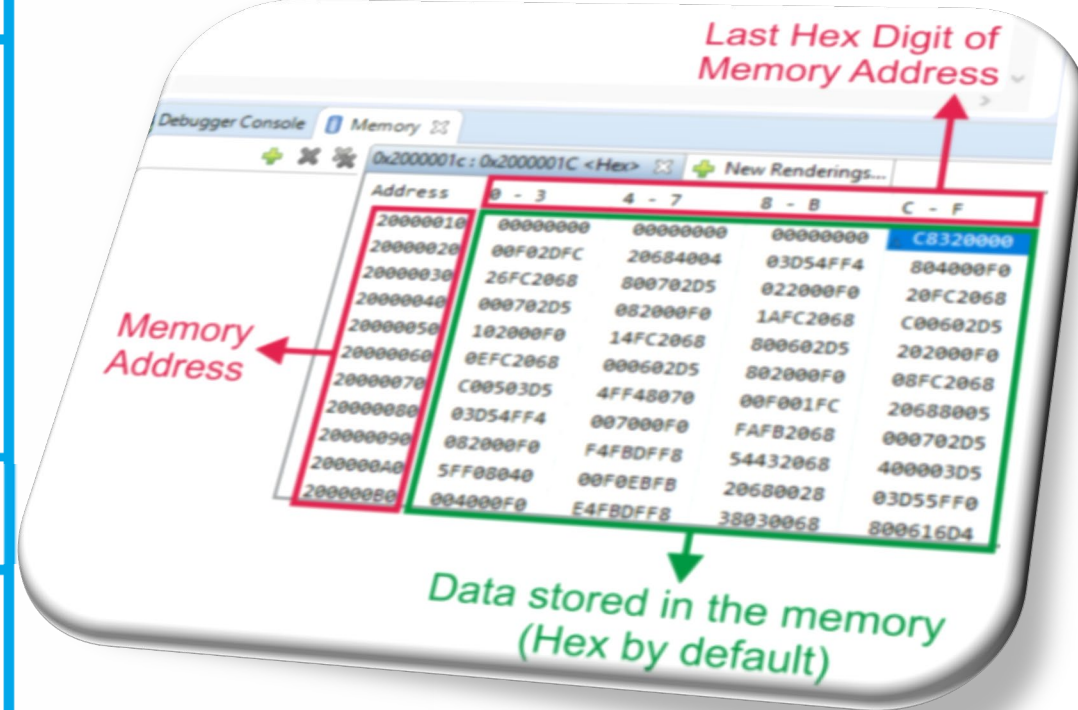
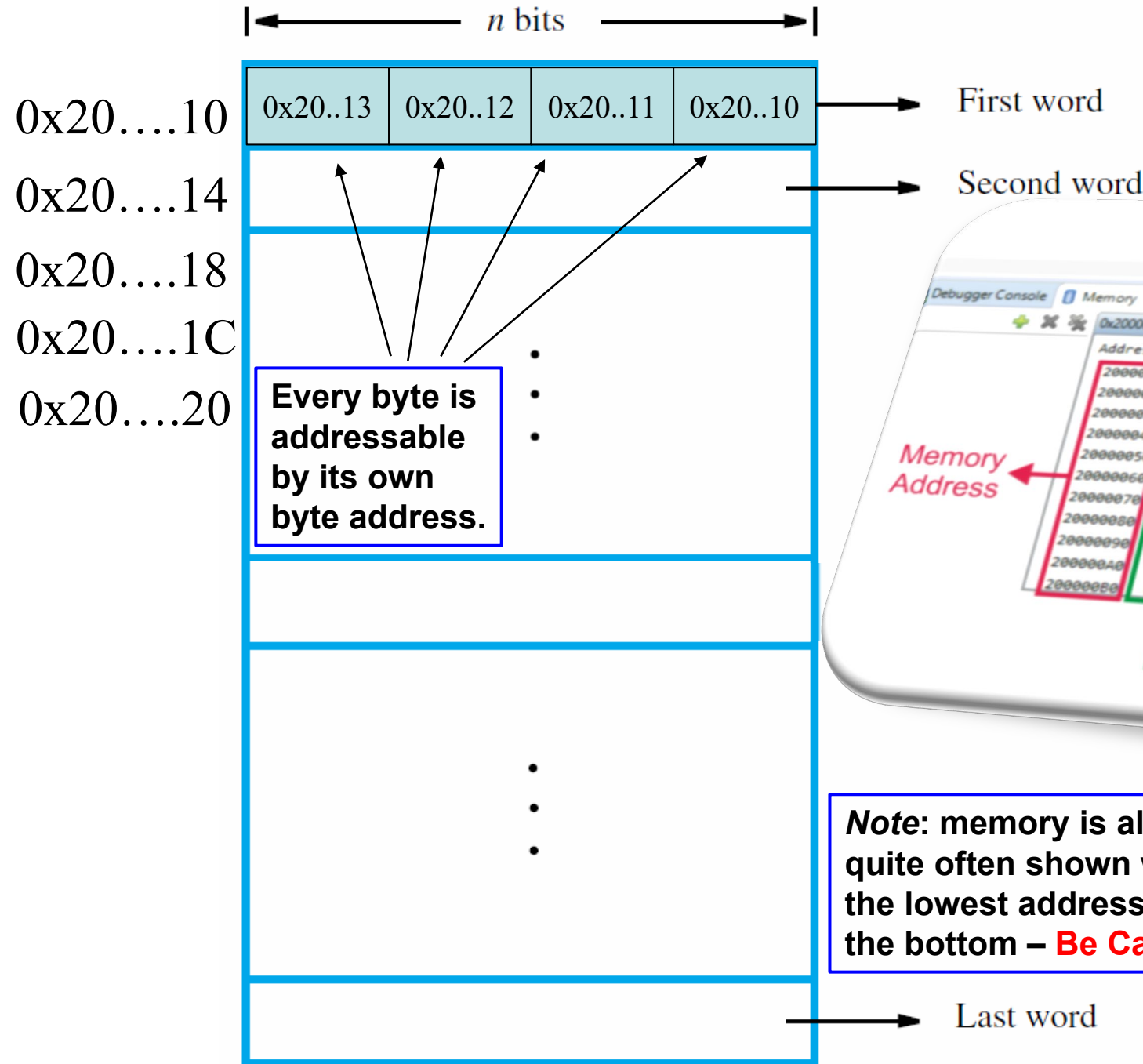
3.2 Addresses for Memory Locations

- To store or retrieve items of information, each **memory location** needs a **distinct address** in the memory map
- Numbers 0 to $2^k - 1$, (k = number of address bits), are used as addresses for successive locations in the memory
- Hence, these 2^k locations constitute the **address space**, i.e. the **range of memory addresses** available to programs
- In hardware, this 2^k range is referred to as **memory size**,
e.g.: If $k = 20 \rightarrow 2^{20}$ or 1M locations,
If $k = 32 \rightarrow 2^{32}$ or **4G locations** (i.e. >4 billions locations!)



Byte Addressability

- Information is usually represented by words
- But the **word** length may range from 16 to 64 bits
- Impractical to assign an address to each **bit**
- Since **byte** size is always 8 bits
- As such, memory is always **byte-addressable**, i.e.
an address is assigned to each byte
- If **byte locations** have addresses ending with 0x **0, 1, 2 or 3** for a word length of 32 bits, the **word locations** will have addresses ending with 0x **0, 4, 8 or C** since 4 bytes made up a 32-bit-word!



Note: memory is also quite often shown with the lowest address at the bottom – Be Careful!

LPC1769: Memory Map

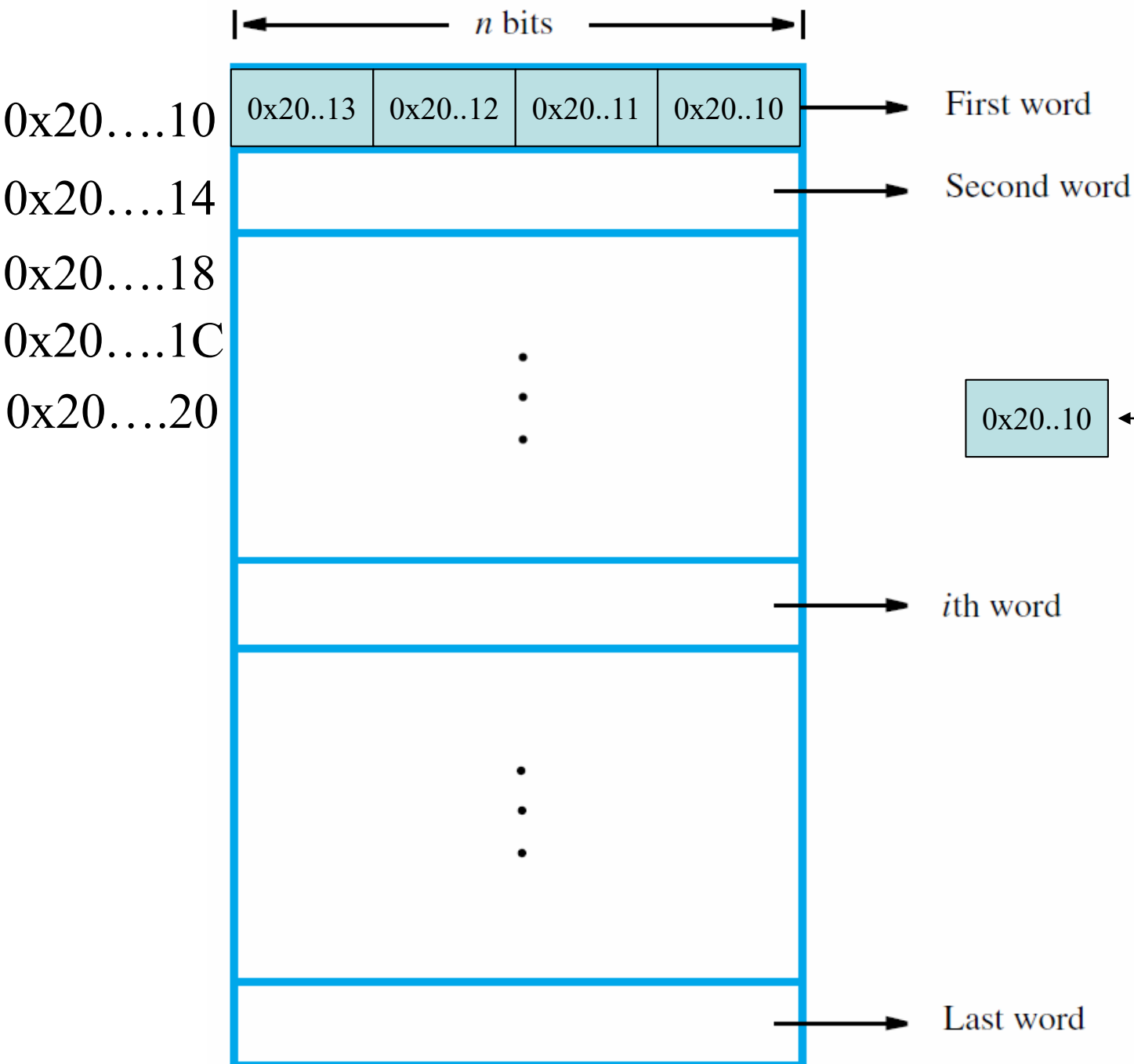
4GB (0xFFFFFFFF) Memory Space

Address Range	Component	Description
0xFFFFFFFF	System level	Private peripherals including built-in interrupt controller (NVIC), MPU control registers, and debug components
0xE0000000 - 0xDFFFFFFF	External device	Mainly used as external peripherals
0xA0000000 - 0x9FFFFFFF	External RAM	Mainly used as external memory
0x60000000 - 0x5FFFFFFF	Peripherals	Mainly used as peripherals
0x40000000 - 0x3FFFFFFF	SRAM	Mainly used for program code. Also provides exception vector table after power up
0x20000000 - 0x1FFFFFFF	CODE	

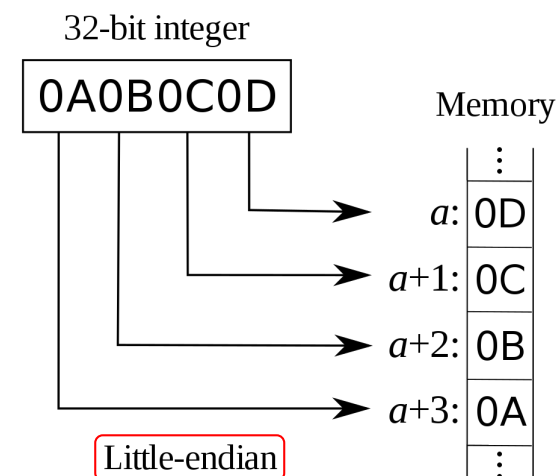
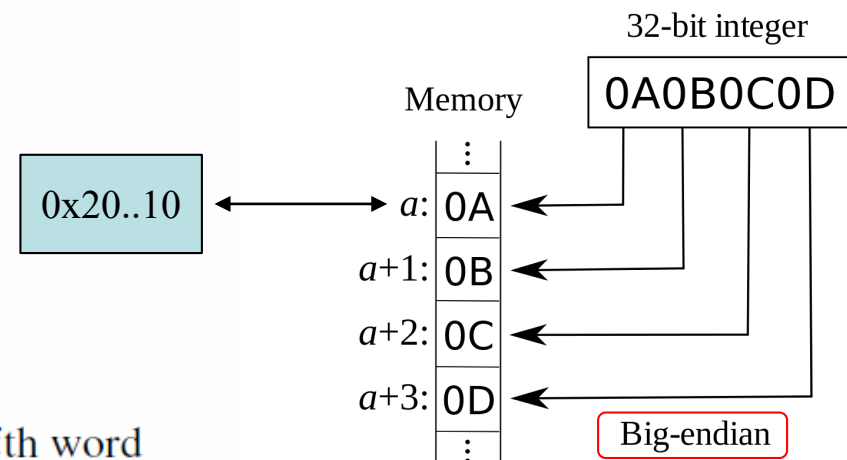
- Describes the **order** in which a sequence of bytes are stored in computer memory; or more generally, the internal ordering of a sequence of bytes
- **BIG** endian: places the ***most significant*** byte ***first*** (into the lowest address byte) & the least significant byte last
- **little** endian: places the ***least significant*** byte ***first*** (into the lowest address byte) & the most significant byte last

Notes: In our **32-bit** ARM context:

- Control accesses & Instruction fetches are always little endian
- Data accesses can be big or little endian, depending on endianness setting

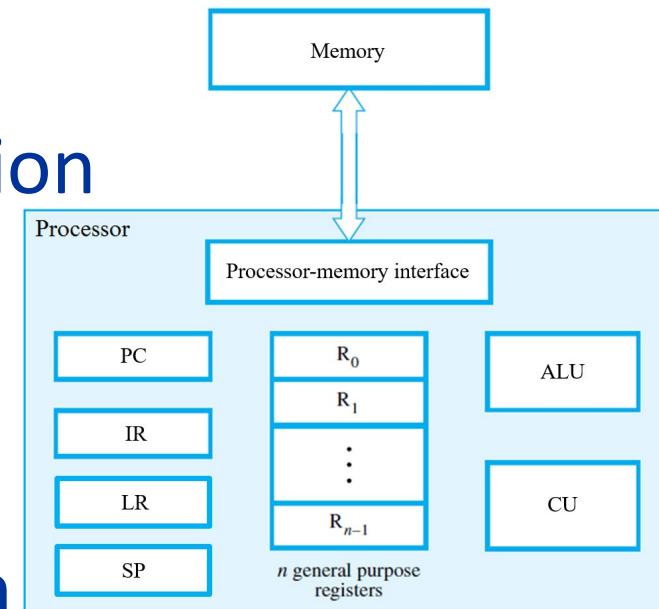


Endianness



3.3 Memory Operations

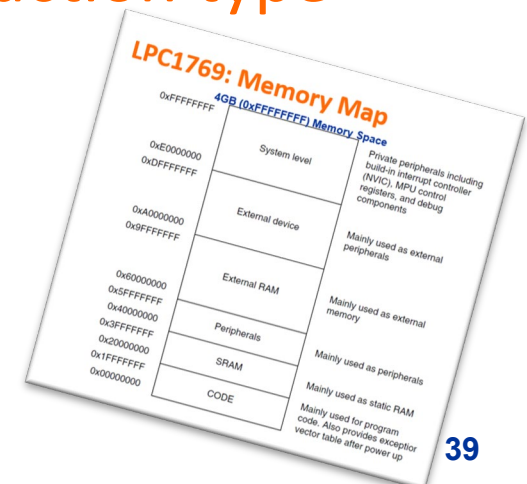
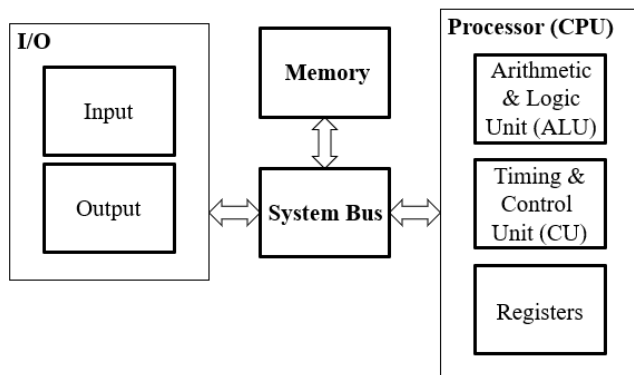
- CU initiates **transfer** of both instructions & data between memory & the processor registers
- (low-level*) **Read** memory operation: contents at the memory address location given by processor is **retrieved**
- (low-level) **Write** memory operation: contents at the given memory location is **overwritten** with the given data



**low-level refers to the operation at hardware level*

4. Instructions & Sequencing

- Basic **types of instruction** a computer must support:
- **data transfers** to & from the **memory**, e.g. *load, store*
 - **arithmetic & logical** operations on data, e.g. *add, or*
 - program **sequencing & control**, e.g. *branch*
 - **input/output** transfers,
 - but they are usually **memory-mapped**, hence, they can be considered to be the same as the first instruction type



4.1 RISC vs CISC Instruction Sets

➤ Nature of instructions distinguishes computers

Reduced Instruction Set Computers

vs

Complex Instruction Set Computers

RISC (e.g. ARM, Apple, Samsung)

Simple Instruction Type; Reduced Set (30-40)

Fixed Instruction Length: One Word

One Cycle/Instruction (Except Load & Store)

Instructions Accessing Memory: Load & Store

Arithmetic/Logic Operands Must Be In Registers

Limited Addressing Mode

Highly Pipelined – Pipelining Is Easy

Software-Centric

CISC (e.g. Intel, AMD, Motorola)

Complex; Extended Set (100-200)

Variable Instruction Length: Multi-Word

Multiple Cycles/Instruction (4-120)

Almost All Instructions From The Set

Allow Direct Memory Operations

Compound Addressing Mode

Less Pipelined – Difficult

Hardware-Centric; Complex Processor

RISC → simpler, faster, more power efficient, cheaper !

4.2 RISC Instruction Set (Load & Store)

- Each RISC instruction occupies a single word
- An **instruction** specifies an **operation** (e.g. ADD)
& the locations of its **data operands** (e.g. R1)

E.g. 32-bit instruction word in IR:

opcode ...	registers ...	value of #immX (X-bit)
------------	---------------	-------------------------

- A Load & Store architecture means:
 - only **Load & Store** instructions are used to access memory operands
 - operands for arithmetic/logical instructions must be in **Registers**, or one of them may be given explicitly (hard-coded) in the instruction word, i.e. **#imm** (aka **immediate**), e.g. #4 (*will be covered later*)

A RISC Program Example

- Consider a high-level language statement:

$$C = A + B$$

where A, B, & C correspond to memory locations, i.e.

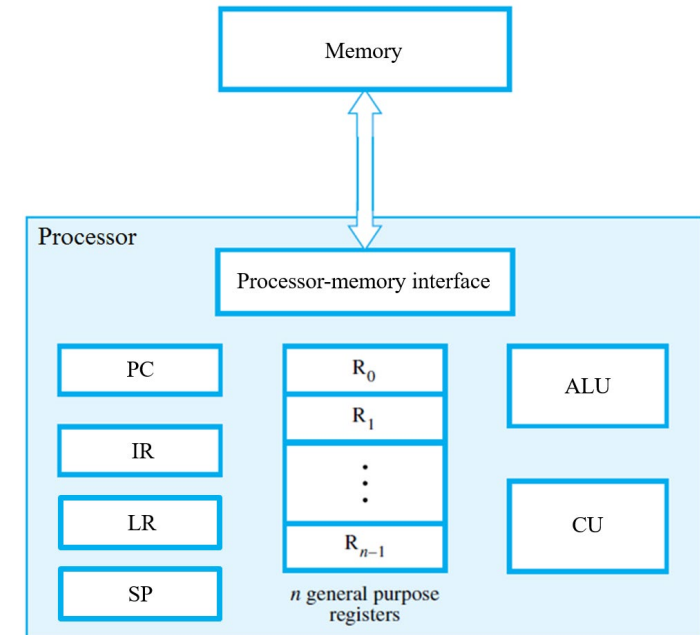
$$[C] \leftarrow [A] + [B]$$

- Tasks:
 - load contents of locations A & B to registers,
 - compute sum, &
 - transfer result from register to location C

A RISC Program Example

- Sequence of simple *generic* RISC asm program for tasks:

```
Load    R2, A
Load    R3, B
Add     R4, R2, R3
Store   R4, C
```

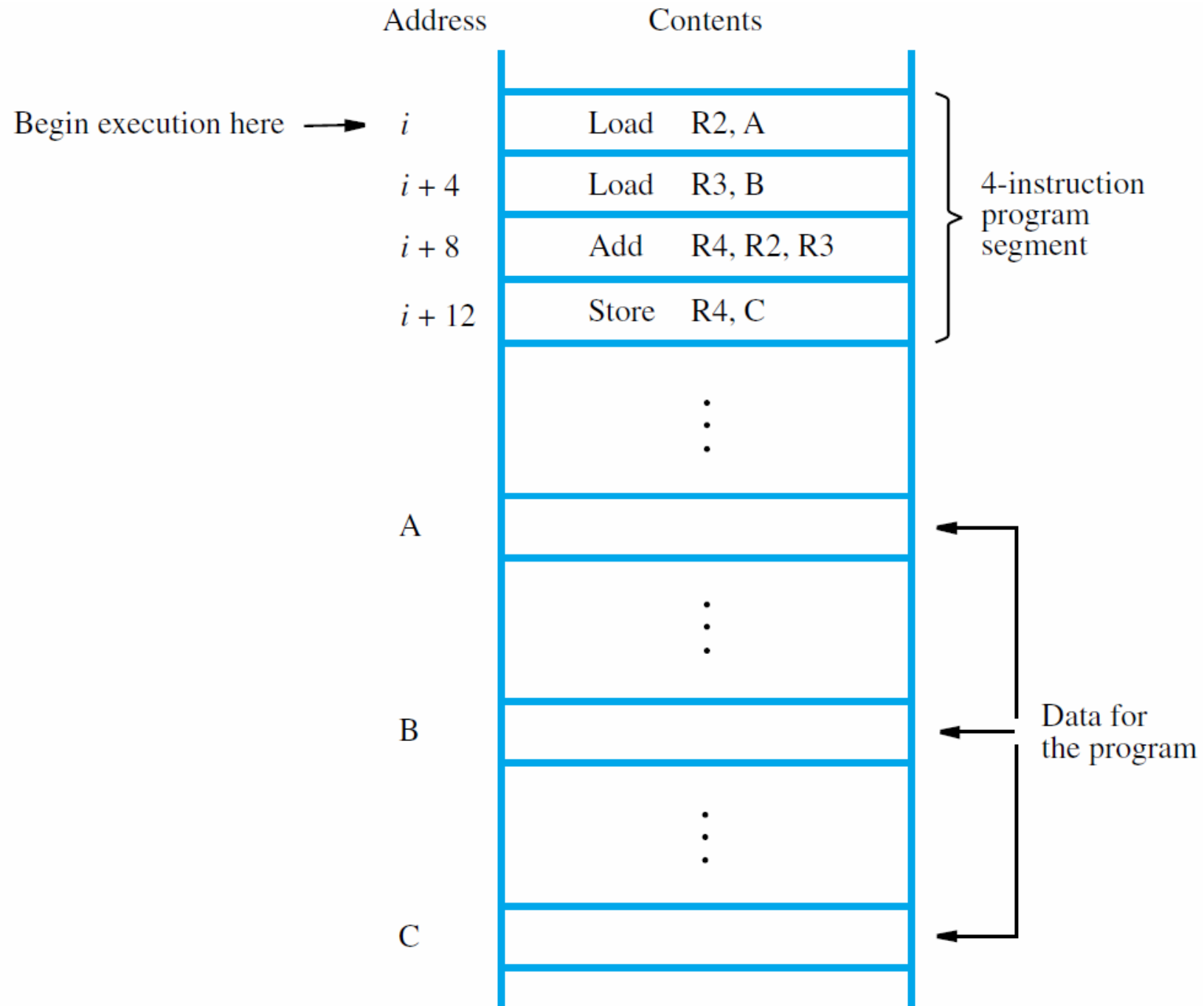


- Load instruction: transfers data to Register
- Store instruction: transfers data to the memory

Load, Store destinations differ despite the same operand order!

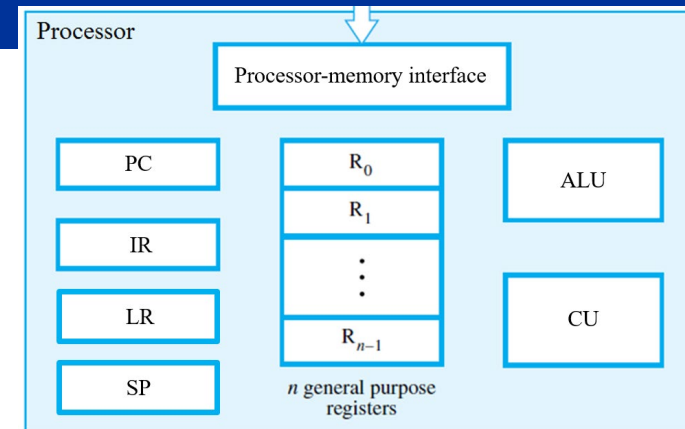
4.3 RISC Program Storage

- Consider the preceding 4-instruction program, how is it stored in the memory?
(32-bit word length, byte-addressable)
- Place first RISC instruction word at address i
- Remaining instructions are at $i + 4$, $i + 8$, $i + 12$
- For now, assume that Load/Store instructions specify desired operand address ***directly*** (*more addressing modes in the next lecture ...*)



4.4 RISC Program Execution

➤ How is the RISC program executed?



1. Address (e.g. i) for first instruction is placed in PC
2. CU **fetches & executes** instructions, one after another
→ straight-line sequencing
3. During the execution of each instruction, **PC** register is always **incremented by 4 *by default***, unless told otherwise (e.g. in branching & looping)
i.e. content of **PC** is $i + 16$ after the fourth instruction
(i.e. *Store R4,C* in the example program) is executed

Instruction Execution (Fetch-Execute)

➤ **Fetch**-phase:

- CU reads **value in PC** for the address of current instruction
- CU transfers & places the machine instruction **into IR**

➤ **Execute**-phase:

- CU **decodes** machine instruction in IR
- The **specified operation** is performed in steps, e.g. CU transfers operands, ALU performs arithmetic/logical ops
- Concurrently, **PC is incremented**, pointing at the next instruction, ready for the next fetch

Details of Instruction Execution_1



Memory		Registers	
0x40	Load R2, A	R2	
0x44	Load R3, B	R3	
0x48	Add R4, R2, R3	R4	
0x4C	Store R4, C	PC	0x40
	...	IR	Load R2, A
A	2		
B	3		
C			

Fetch

Execute

Memory		Registers	
0x40	Load R2, A	R2	2
0x44	Load R3, B	R3	
0x48	Add R4, R2, R3	R4	
0x4C	Store R4, C	PC	0x44
	...	IR	Load R2, A
A	2		
B	3		
C			

Details of Instruction Execution_2

Memory		Registers	
0x40	Load R2, A	R2	2
0x44	Load R3, B	R3	
0x48	Add R4, R2, R3	R4	
0x4C	Store R4, C	PC	0x44
	...	IR	Load R3, B
A	2		
B	3		
C			

Fetch

Execute

Memory		Registers	
0x40	Load R2, A	R2	2
0x44	Load R3, B	R3	3
0x48	Add R4, R2, R3	R4	
0x4C	Store R4, C	PC	0x48
	...	IR	Load R3, B
A	2		
B	3		
C			

Details of Instruction Execution_3

Memory		Registers	
0x40	Load R2, A	R2	2
0x44	Load R3, B	R3	3
0x48	Add R4, R2, R3	R4	
0x4C	Store R4, C	PC	0x48
	...	IR	Add R4, R2, R3
A	2		
B	3		
C			

Fetch

Execute

Memory		Registers	
0x40	Load R2, A	R2	2
0x44	Load R3, B	R3	3
0x48	Add R4, R2, R3	R4	5
0x4C	Store R4, C	PC	0x4C
	...	IR	Add R4, R2, R3
A	2		
B	3		
C			

Details of Instruction Execution_4



Memory		Registers	
0x40	Load R2, A	R2	2
0x44	Load R3, B	R3	3
0x48	Add R4, R2, R3	R4	5
0x4C	Store R4, C	PC	0x4C
	...	IR	Store R4, C
A	2		
B	3		
C			

Fetch

Straight-line Sequencing

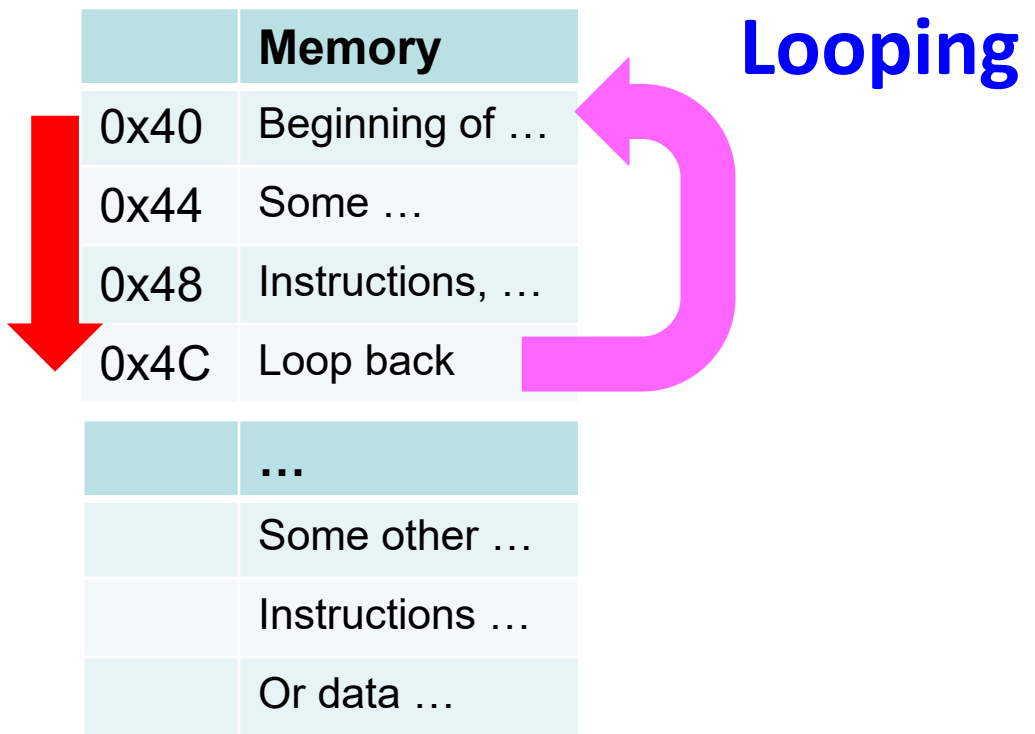
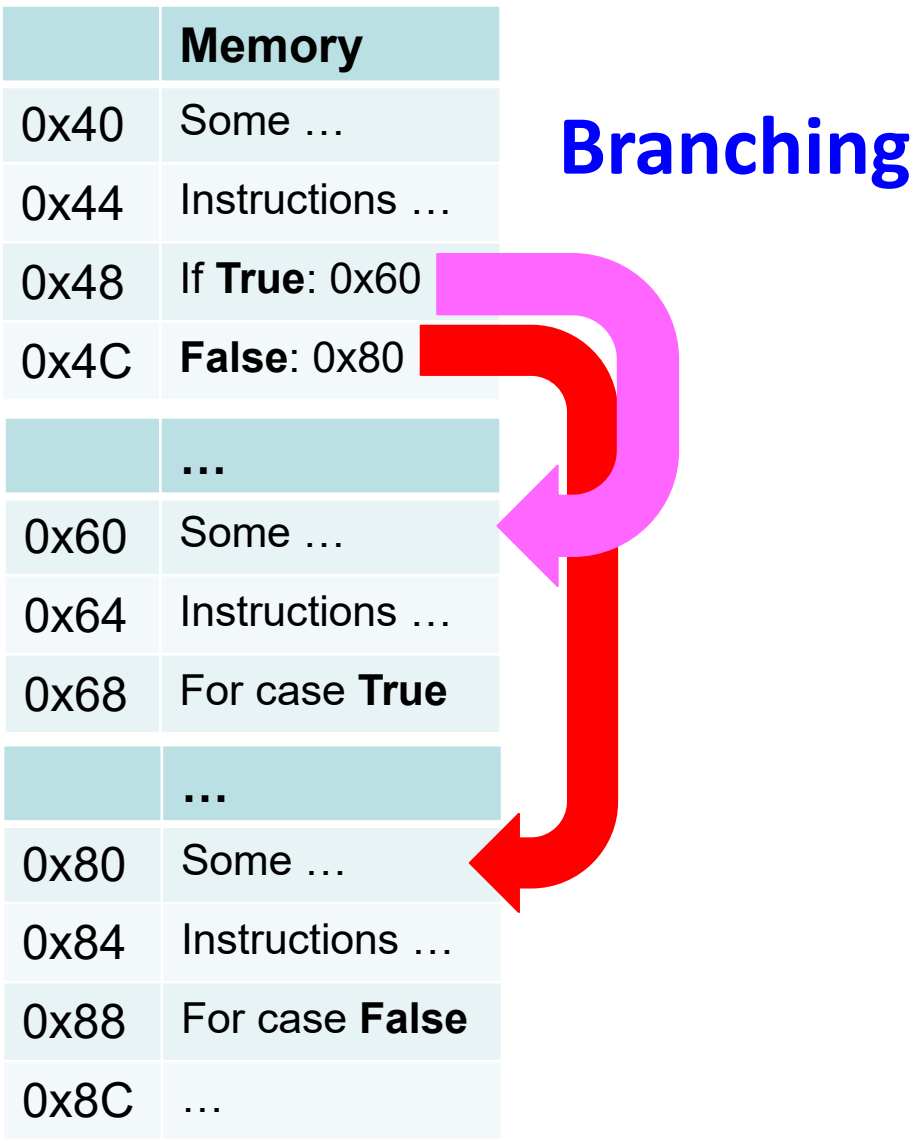
Execute

Memory		Registers	
0x40	Load R2, A	R2	2
0x44	Load R3, B	R3	3
0x48	Add R4, R2, R3	R4	5
0x4C	Store R4, C	PC	0x50
	...	IR	Store R4, C
A	2		
B	3		
C	5		

4.5 RISC Branching & Looping

- To execute another part of a program, e.g. when a certain condition is true, we can use a **conditional branch** instruction
- A **branch** instruction works by
 - placing the **address** containing the required instruction into **PC**
 - then fetching & executing that instruction, & all subsequent ones if any, **until** the end of program or another branching occurs
- If there are operations that need to be performed repeatedly, a **loop** can be implemented by:
 - **branching repeatedly** to the **start of a section** of a program **until** a terminating condition is satisfied.

Branching & Looping Examples



Branching & Looping: Condition Codes

- Processor maintains the information on arithmetic or logical operation results to affect subsequent conditional branches, thus altering the program flow
- Condition code **flags** in a status register:

N (negative)	1 if result negative, else 0
Z (zero)	1 if result zero, else 0
C (carry)	1 if carry-out occurs, else 0 (for unsigned int)
V (overflow)	1 if overflow occurs, else 0 (for signed int)

THE END
Questions?