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 (µarch or uarch):
  - Internal implementation of a computer at the register & functional unit level
  - Has to abide by the rules set out in the Computer Architecture

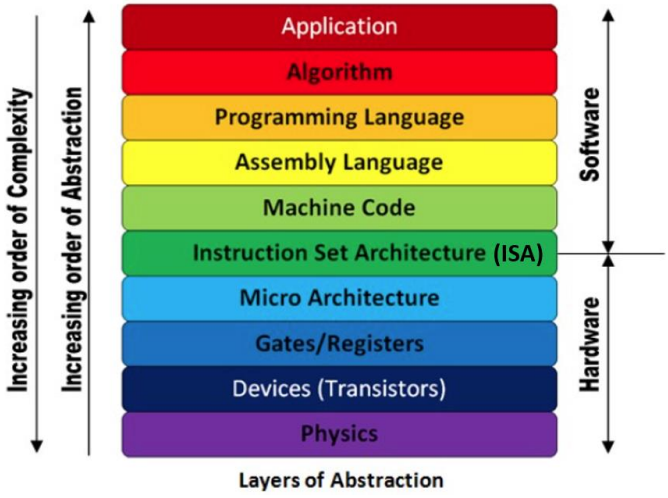
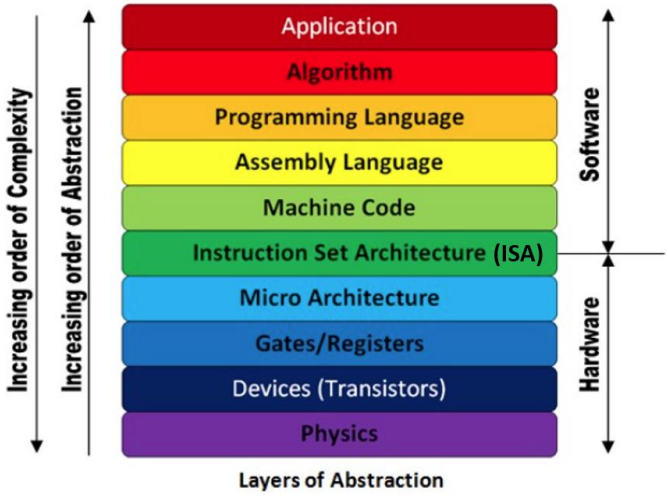


Image: https://www.secplicity.org/2018/09/19/understanding-the-layers-of-a-computer-system/

- Computer Architecture, aka Instruction Set Architecture (ISA)
  - Programmer's abstraction of a computer
  - Defines how the CPU is controlled by the software
  - Specifies what the CPU can do & how it gets done
  - In software: instruction set & assembly language (asm)
  - In hardware: technological specifications

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) apply knowledge of microprocessor concepts and program a microprocessor using assembly language to develop an efficient solution to an engineering problem

(b) apply knowledge of computer organization concepts, such as processor microarchitecture, addressing, caches and pipelining

(c) design a solution comprising sequential and parallel logical processing steps

(d) use an Integrated Development Environment (IDE) to write assembly language programs

Teaching Team

- Lecturers:
  - Part I: Henry Tan (ARM asm)
  - Part II: Ankit Srivastava (µarch, caching, pipelining)
- Teaching/Lab Assistants:
  - Tutors & TA for Lab: Hou Linxin, Ni Qingqing
  - Lab Graduate Assistants: ~4, TBC

Acknowledgement: The content of Part I is partly adopted from the original EE2028 material created by my predecessor, Assoc Prof Tham Chen Khong

Syllabus

Lecture	Topics
1	Intro to course, ARM & microprocessor concepts ▪ ARM cores, Cortex-M4 (STM32L475) & Discovery Kit overview ▪ functional units: I/O, memory, processor with registers ▪ memory organization, addressing & operations ▪ instruction execution & sequencing
2	Intro to ARMv7E-M Instruction Set & asm ▪ glossary, memory allocation, load/store, move, arithmetic ops
3	▪ compare, branch, conditional exe, IT block, logic & stack ops
4 - 6	Microarchitecture, caching & pipelining ▪ processor microarchitecture, datapath & controller design ▪ memory hierarchy, mapping techniques & caches ▪ pipelined execution of instructions

Tutorial & Lab Sessions

- Tutorial Sessions:
  - Weeks 2 – 6

- Lab Sessions:
  - Venue: Digital Electronics Lab (E4-03-07)
  - Week 3: STM32CubeIDE & Discovery Kit Familiarization (3 hrs)
  - Week 4: Assignment Briefing & Discussion (3 hrs)
  - Week 6: Assignment Assessment (30 mins)

This schedule is tentative and subject to change

Assessments

- Assignment (50%):
  - Weeks 4 – 6: ARM Assembly & Microarchitecture Assignment
  - Assessment during Lab #3, (i.e. Week 6)
    - ✓ Venue: Digital Electronics Lab (E4-03-07)
    - ✓ Duration: 30 mins per team of two (schedule to be announced)

- Final Quiz (50%):
  - *First Tuesday after Recess (i.e. Week 7)*
  - Date: **1 Oct 2024** (Tuesday)
  - Time: **2.30 pm to 3.45 pm** (75 minutes)
  - Venue: **LT7A** (not our usual LT!)
  - Format: **Pen & Paper, Open book (hardcopy), no devices allowed**

All Dates & Time are Tentative

Introduction to Microprocessors, ARM & ARM Cortex-M4 (STM32L475)

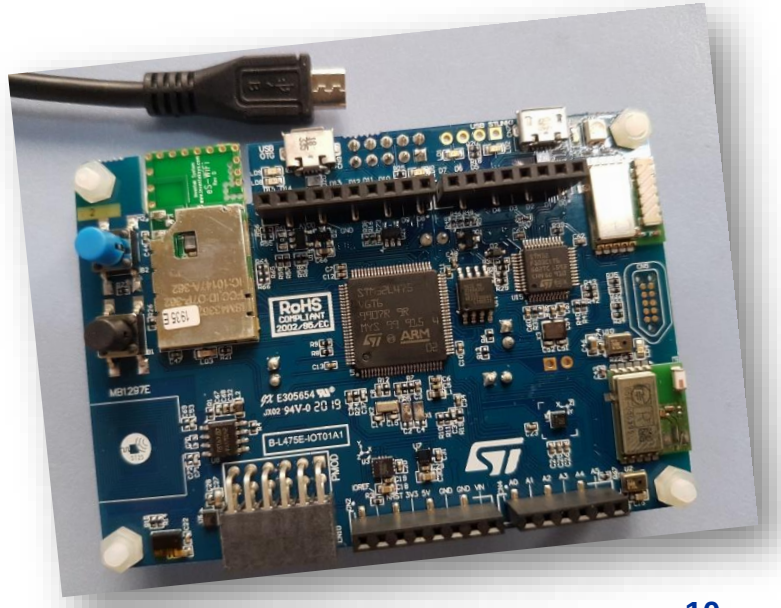
CG2028

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

Introduction to Microprocessors & ARM

NUS National University of Singapore

- Objectives:
  - Understand the motivation behind learning ARM & 32-bit ARM Cortex-M4 (e.g. STM32L475)
  - Differentiation between ARM processor & ARM-based MCUs (e.g. Cortex-M4 processor vs Cortex-M4-based STM32L475)
  - Introduction to STM32 Discovery Kit & its key features



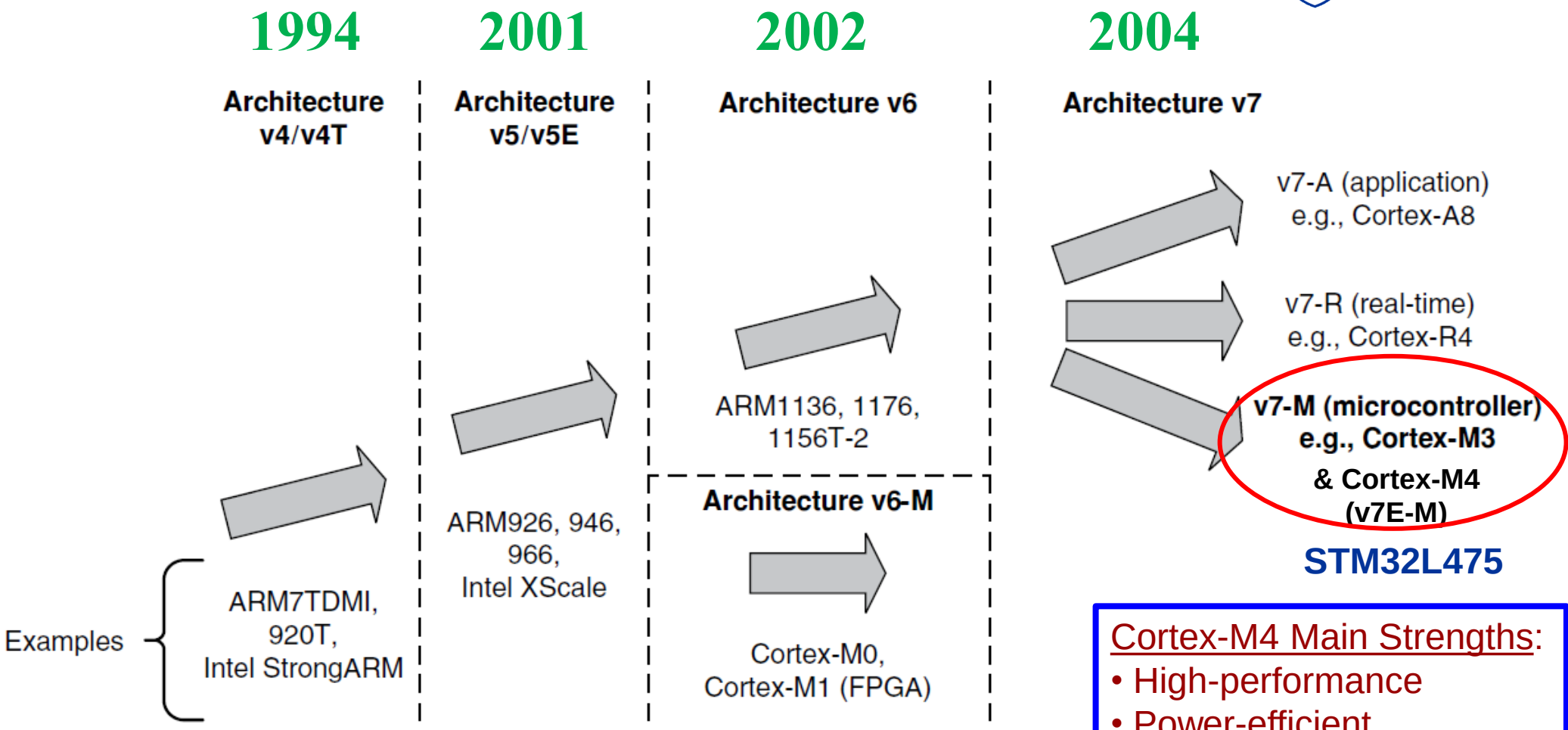
Advanced RISC Machine (ARM)

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

As of Feb 2024, more than 280 billion ARMs shipped!

Product Family	Designed For	Example Use Cases
Cortex-A (64-bit)	Open **Application** Platforms; High Performance; Power Efficiency	Mobile Devices, Set-top Boxes, for Full-Featured OSs
Cortex-R (32-bit)	**Real-time** System; Mission-Critical; High Processing Power & Reliability; Low Latency	Airbags/Braking Systems, HDD/SSD Controllers
Cortex-M (32-bit)	**Microcontroller** Embedded System; Low Power Consumption; Low Cost	IoT Wireless Sensor Nodes, 3D Printers, Home Appliances

ARM Architecture Evolution



Note: First Cortex-M4 processor shipped in 2010

Cortex-M4 Main Strengths:
- High-performance
- Power-efficient
- Low cost
- Low interrupt latency
- Ease-of-use
- Ease-of-integration

THE CORTEX-M4 PROCESSOR VERSUS CORTEX-M4-BASED MCUs (e.g. M4-based STM32L475)

The Cortex-M4 processor is the central processing unit (CPU) of a microcontroller chip. In addition, a number of other components are required for the whole Cortex-M4 processor-based microcontroller. After chip manufacturers license the Cortex-M4 processor, they can put the Cortex-M4 processor in their silicon designs, adding memory, peripherals, input/output (I/O), and other features. Cortex-M4 processor-based chips from different manufacturers will have different memory sizes, types, peripherals, and features.

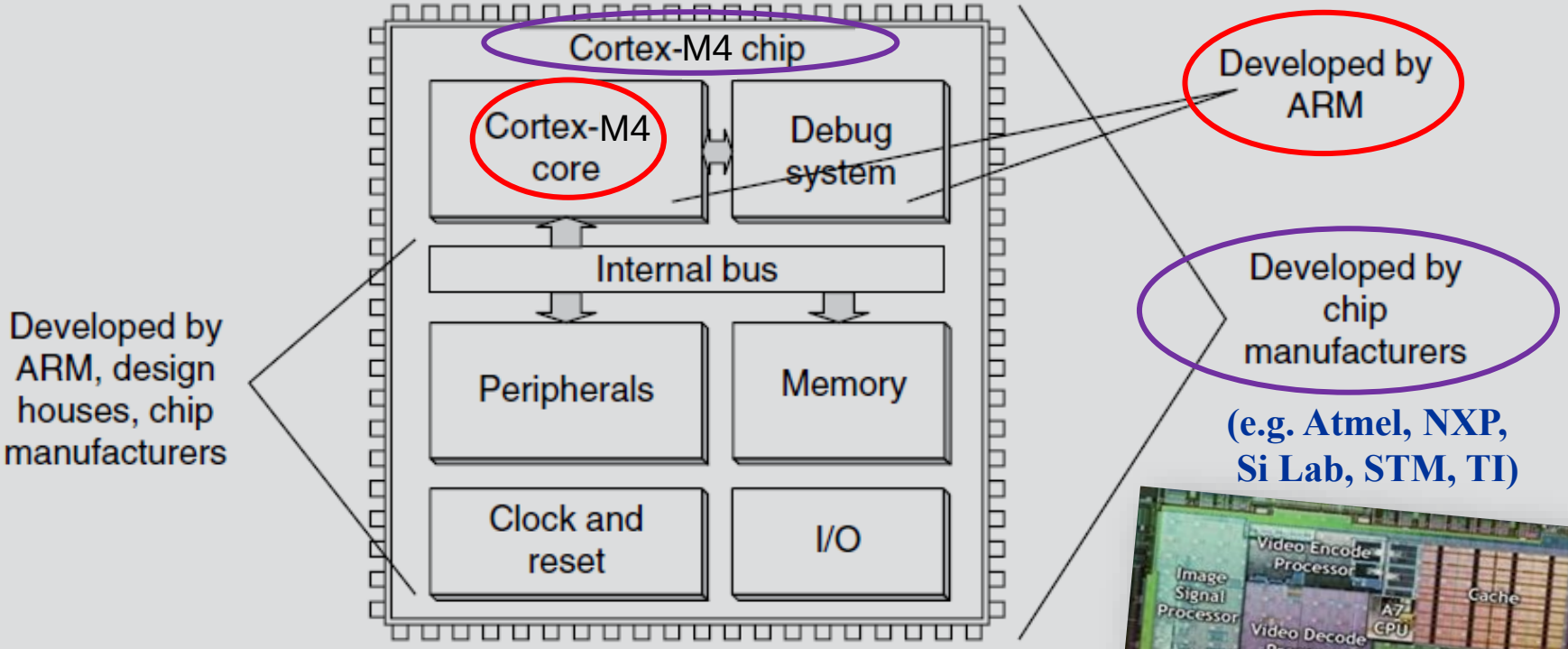
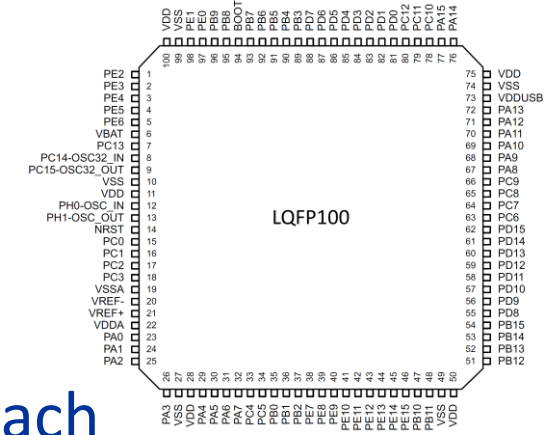


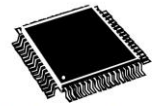
FIGURE 1.1

The Cortex-M4 Processor versus the Cortex-M4-Based MCU.

Image source: https://www.bdti.com/InsideDSP/2011/10/20/NvidiaQualcomm

STM32L475 Introduction





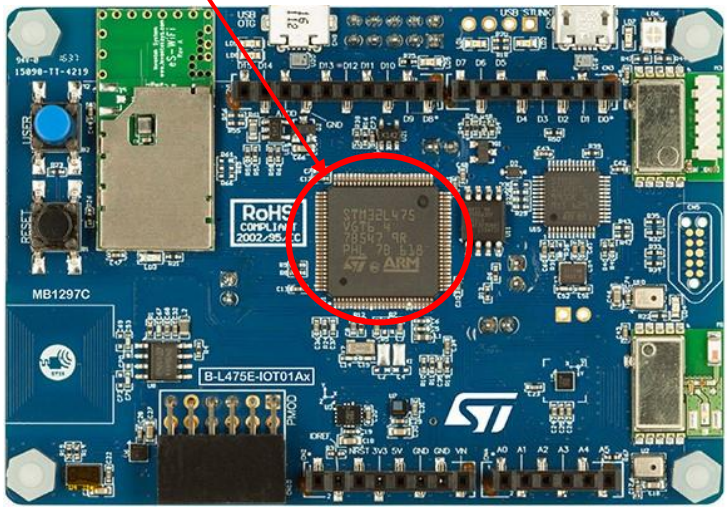
LQFP100 (14 x 14)

- Ultra-low-power ARM Cortex-M4
  Core operates up to 80 MHz

- 100 pin packaging
  - These pins may have up to 8 functions each
  - Desired function is programmable

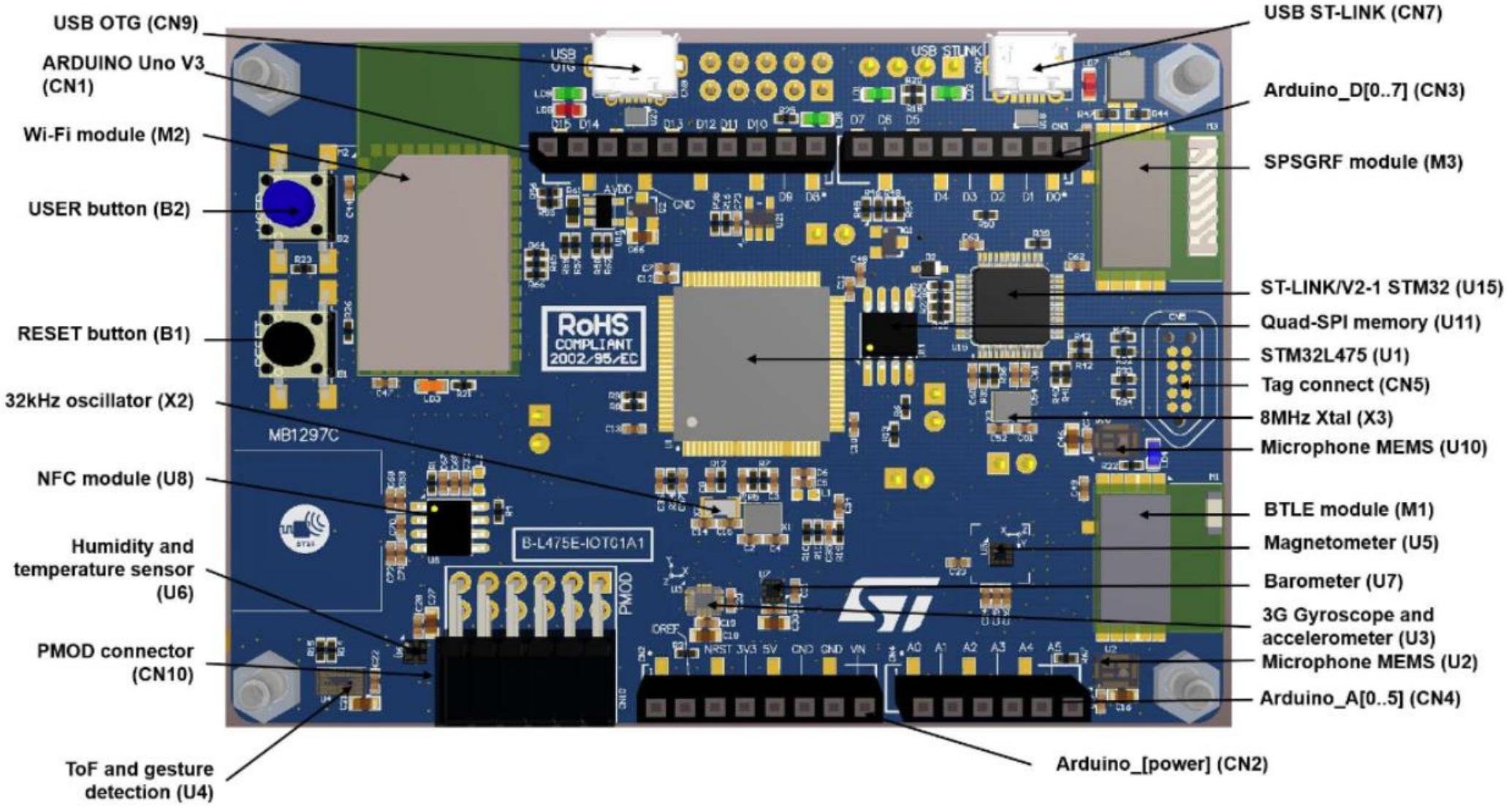
- 1 MB Flash, 128 KB SRAM

- Cortex-M4 processor (32-bit):
  - 32-bit word length
  - 32-bit data path
  - 32-bit register bank
  - 32-bit memory interfaces (i.e. no. of address bits)

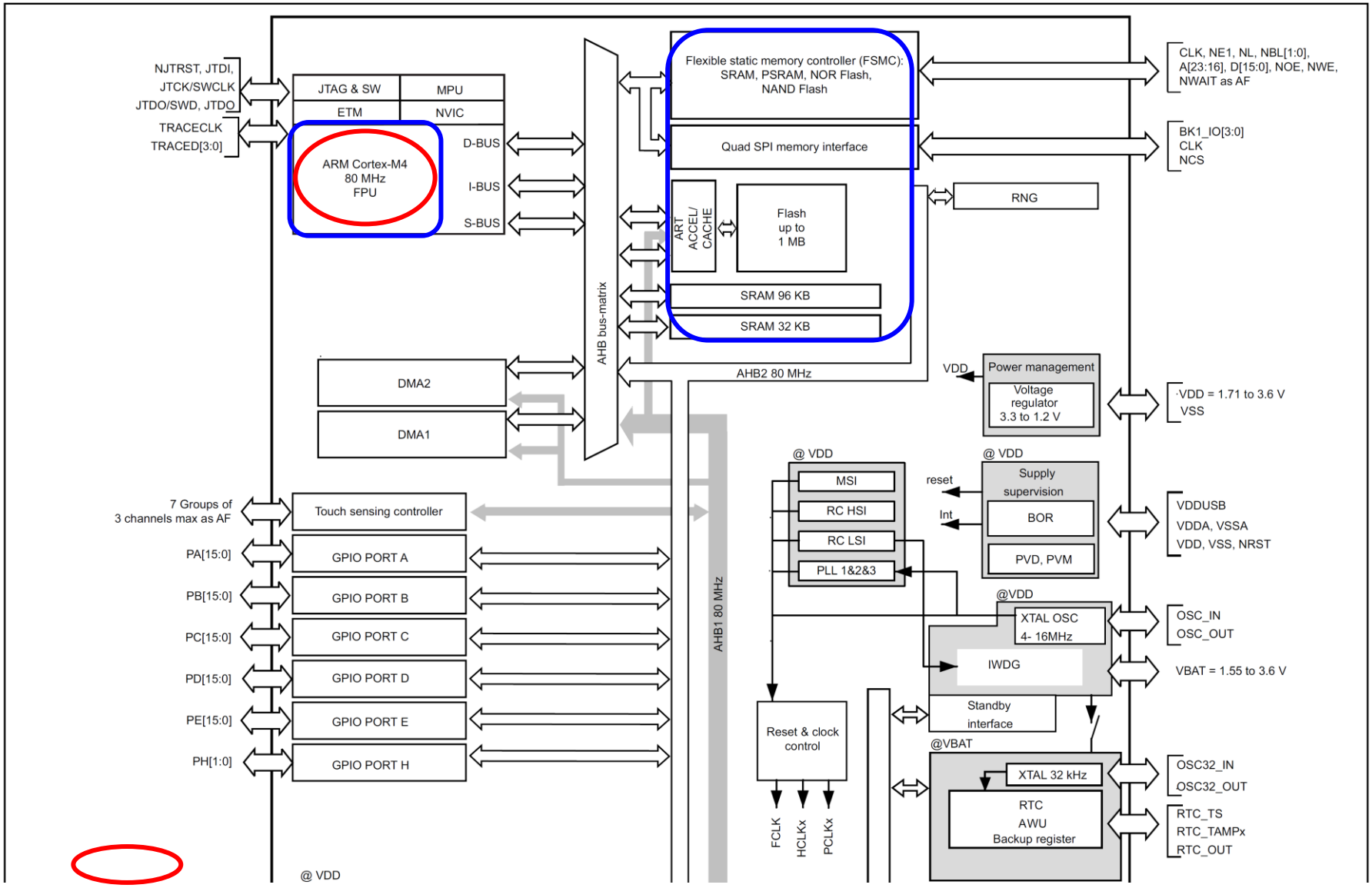


STM32 Discovery Kit

Cortex-M4-Based STM32L475 in STM32 Discovery Kit:



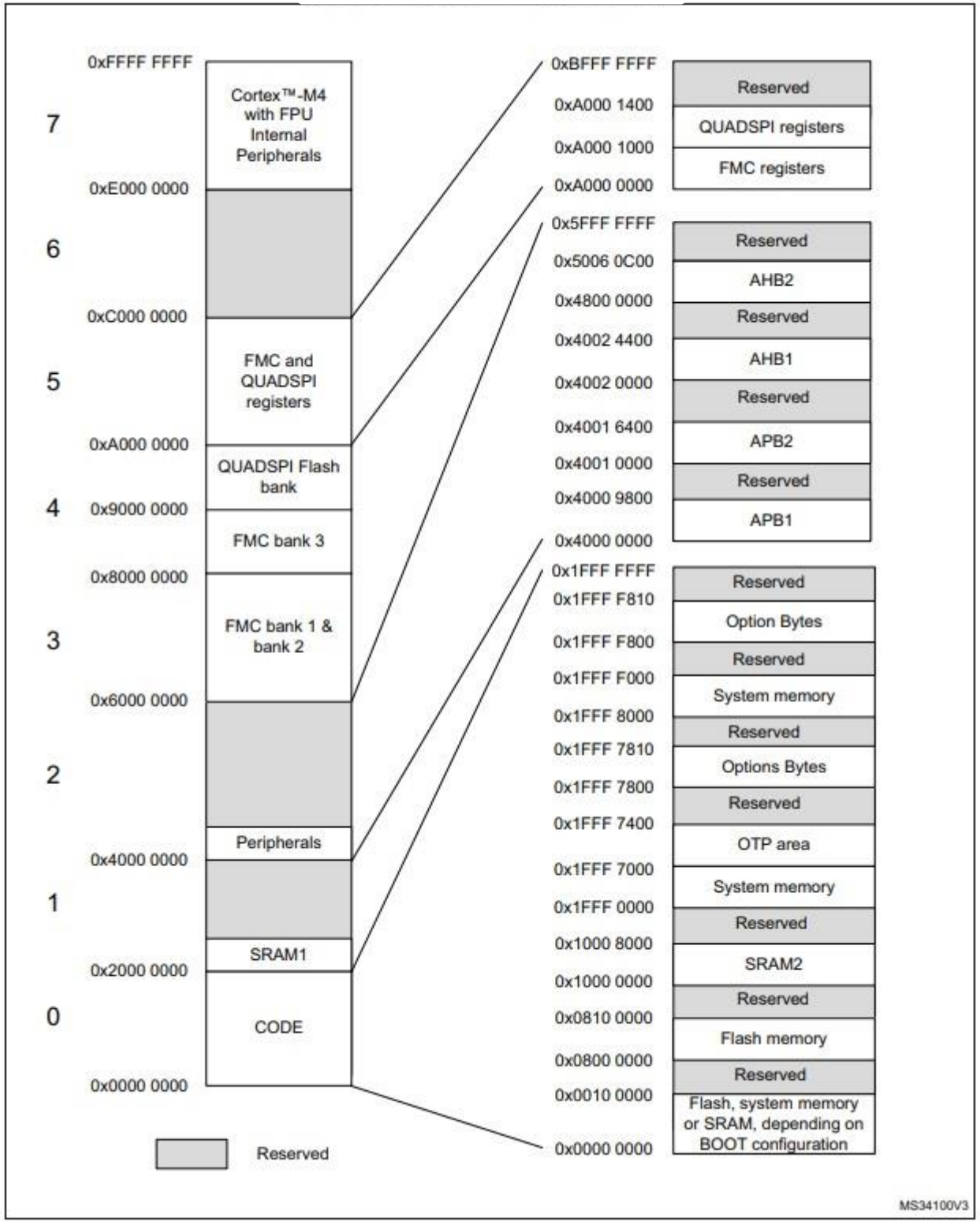
STM32 Discovery Kit Block Diagram:



CG2028 Part I:

CG2028 Part II:

STM32L475 Introduction (Memory Map)



Microprocessor Concepts

CG2028

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

Microprocessor Concepts

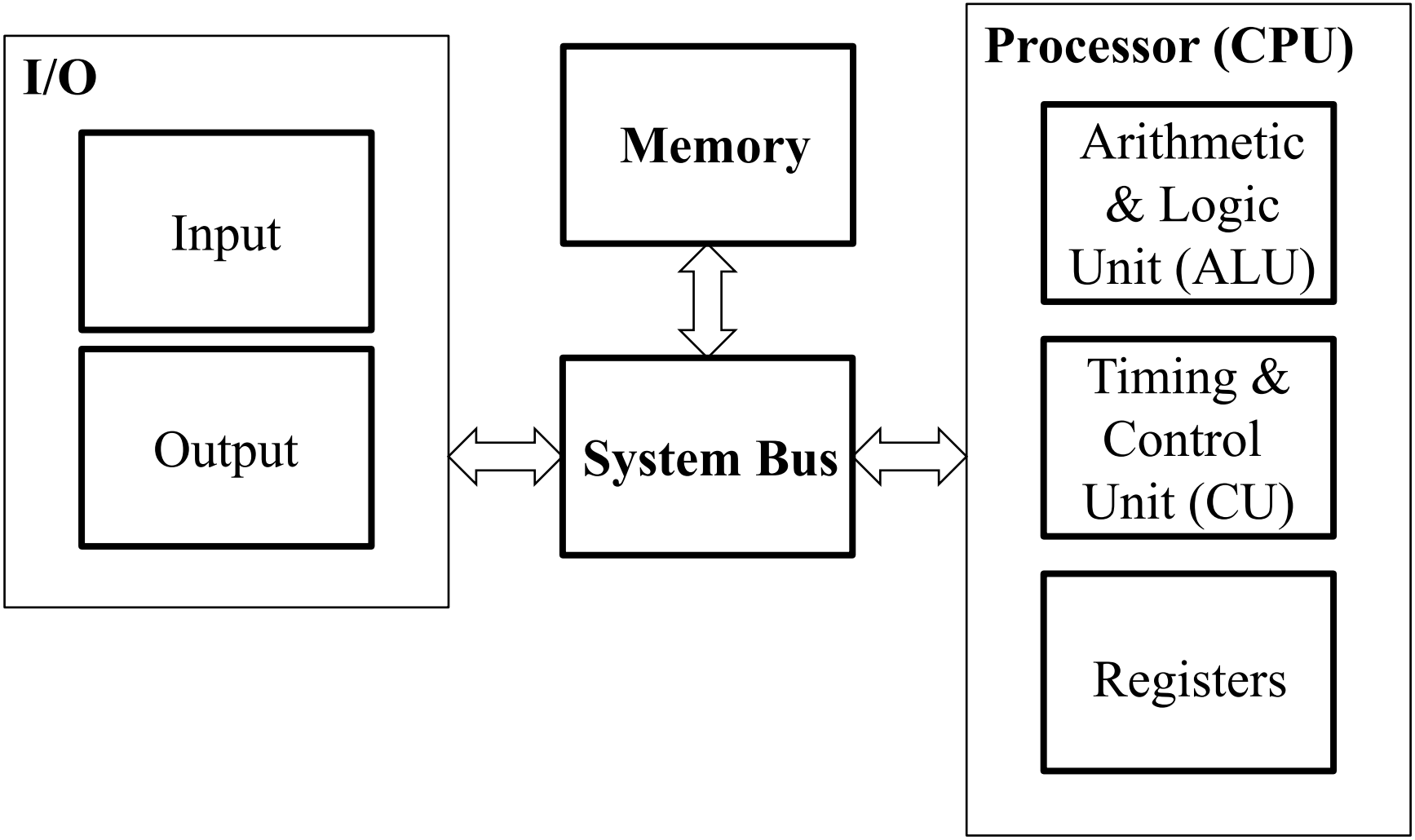
NUS National University of Singapore

- Objectives:

  To understand the essential microprocessor concepts of how a computer system operates, and appreciate software programming and hardware interfacing in an embedded system

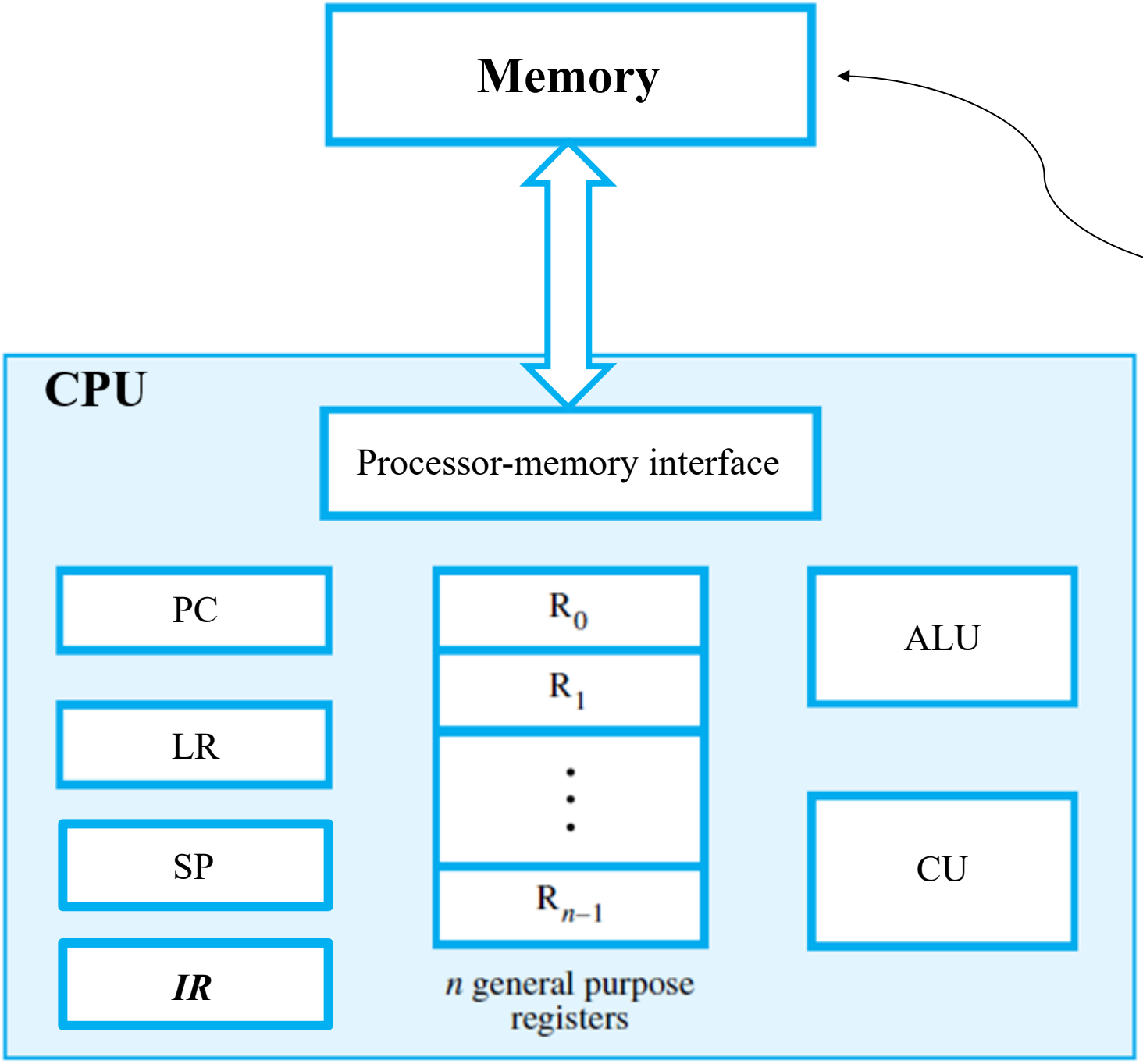
- Outline:
  - 1. Functional units within a computer
  - 2. Processor components & their functions
  - 3. Memory organization, addressing & operations
  - 4. Generic RISC Instructions execution & sequencing

1. Functional Units

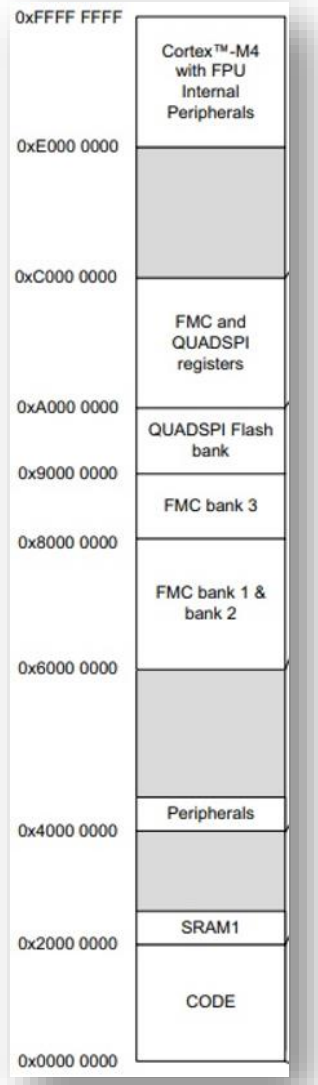


Functional Units: Typical Examples

- Input:
  - Keyboard, touchpad, mouse, microphone, camera, sensors, communication lines, the Internet
- Output:
  - Displays, speakers, earbuds, (3D) printers
- Memory:
  - Cache memory (holds sections/blocks)
    - much smaller & faster than the main & secondary memory
    - holds sections of the program & data currently being executed
  - Main memory (holds pages)
    - Static RAM (SRAM), Dynamic RAM (DRAM)
  - Secondary memory (holds files)
    - flash memory devices, magnetic disks, optical disks


Memory
CPU
Processor-memory interface
PC
LR
SP
IR
R_0
R_1
R_{n-1}
n general purpose registers
ALU
CU

STM32L475's

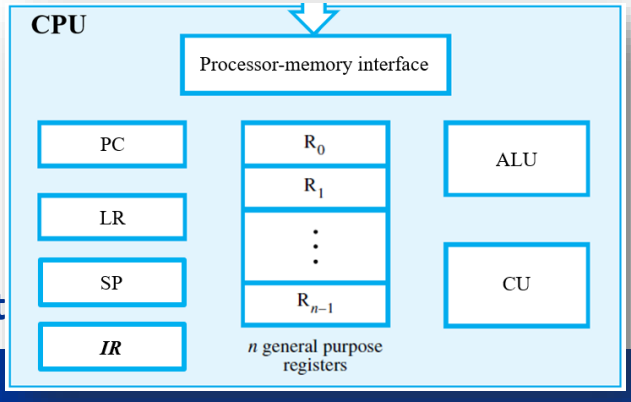

0xFFFF FFFF
Cortex™-M4 with FPU Internal Peripherals
0xE000 0000
0xC000 0000
FMC and QUADSPI registers
0xA000 0000
QUADSPI Flash bank
0x9000 0000
FMC bank 3
0x8000 0000
FMC bank 1 & bank 2
0x6000 0000
Peripherals
0x4000 0000
SRAM1
0x2000 0000
CODE
0x0000 0000

2. Processor Components

- Arithmetic & Logic Unit (ALU) for performing arithmetic & logic operations, usually 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 a bank of 16 *or 32* of them), small amounts of fast storage where each one holds one word* of data. Usually classified into general-purpose (e.g. R0-R12) & special-purpose:
  Program Counter (PC), Link Register (LR),
  Stack Pointer (SP) & *Instruction Register*# (*IR*)

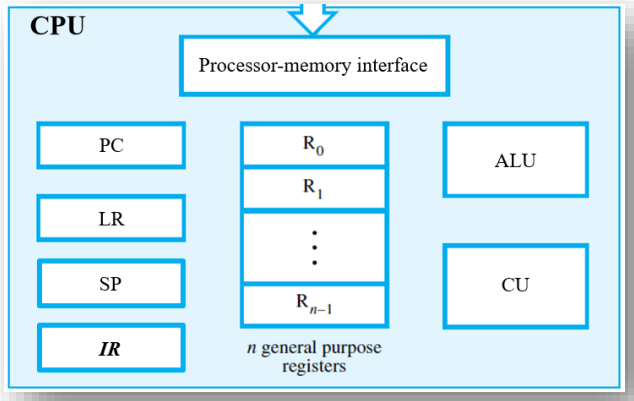


**architecture-dependent;*
^by assigning arithmetic & logic tasks to ALU;
#in CU, read-only by CU

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 LR holds the **return memory address** when a **subroutine/function** is called
- The SP holds the **memory address** of the last (most recent) data in the stack memory
- The ***IR*** holds the **current instruction**

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



[%] *user data or a copy of machine instruction*

3. Memory Organization

- Memory consists of many billions of cells
- Each cell holds a bit of information, 0 or 1
- Information is 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

	Memory
0x40	Load R2, A
0x44	Load R3, B
0x48	Add R4, R2, R3
0x4C	Store R4, C

Notes: In our 32-bit ARM context,
a word = 32 bits,
halfword = 16 bits *(both are architecture-dependent)*

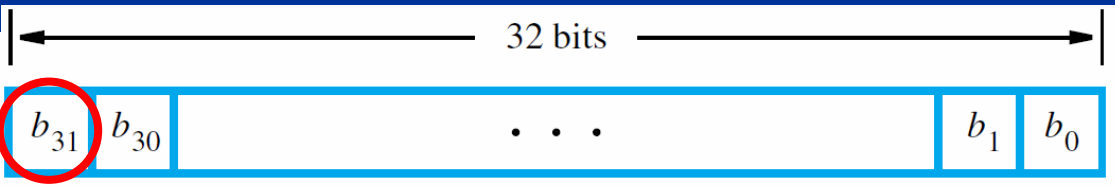
3.1 Word & Byte Encoding

- 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 word for encoding)

  or
  - **data** for the program, which may be, e.g.:
    - a 32-bit unsigned/signed integer (i.e. MSB b_{31} is the sign bit)

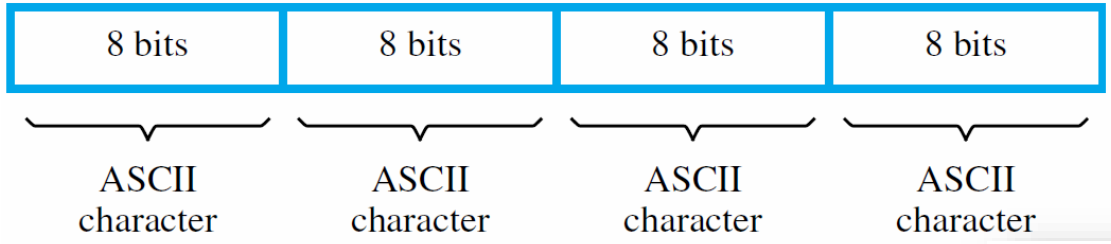
    or
    - four 8-bit bytes (e.g. ASCII character codes)

    or
    - two 16-bit wide characters, one float, half a double, etc.



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

(a) A signed integer



(b) Four characters

ASCII Table

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Source:
https://bournetocode.com/projects/GCSE_Computing_Fundamentals/pages/3-3-5-ascii.html

3.2 Addresses for Memory Locations

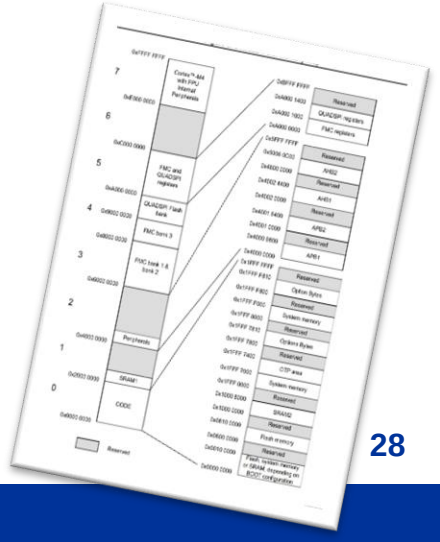
- To store or retrieve items of information, **each *memory location*** has a **distinct address** in the memory map

- If there are k number of address bits,
  - there are 2^k memory locations, known as the **address space**
  - i.e. the **range of memory addresses** available to programs is 2^k
  - i.e. addresses 0 to 2^k-1 form this address space
  - In hardware, this 2^k range is referred to as the **memory size**

e.g.: If $k = 20 \rightarrow 2^{20}$ or >1M locations,

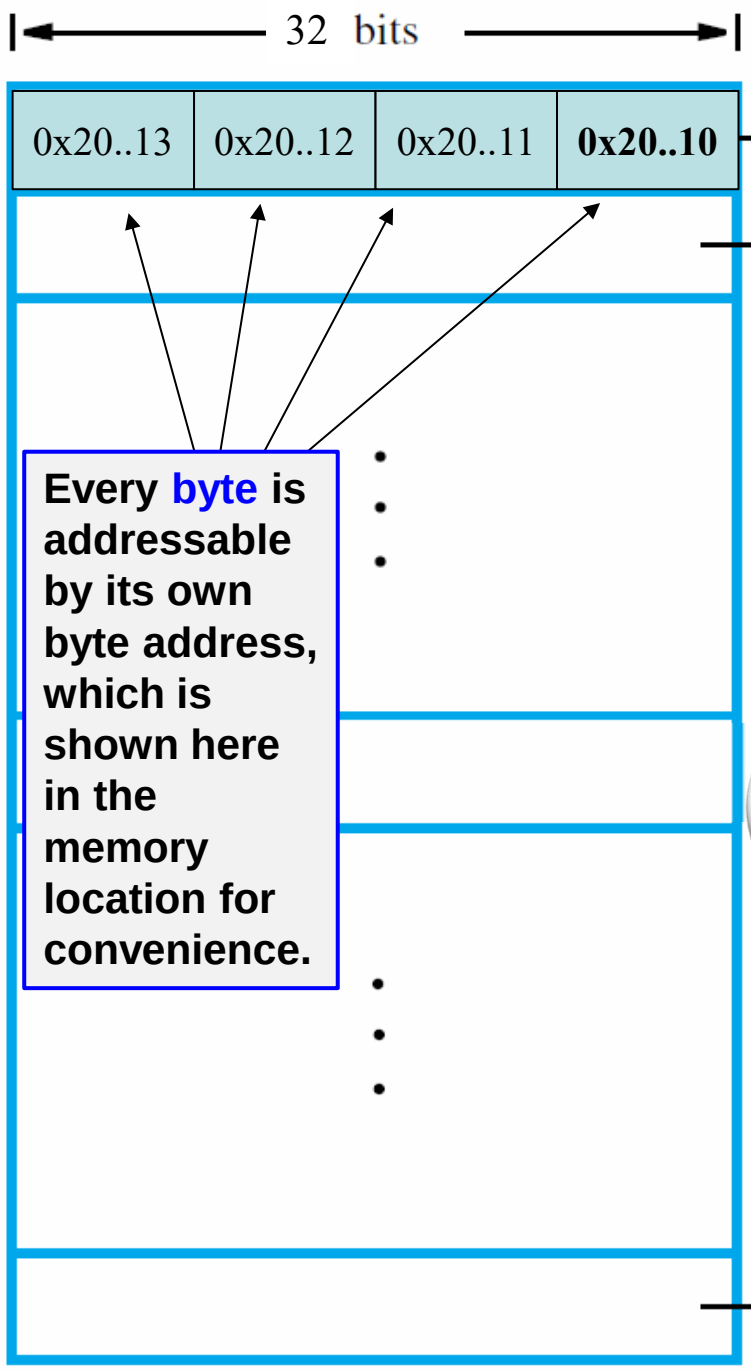
If $k = 32 \rightarrow$ **2^{32} or >4G locations**

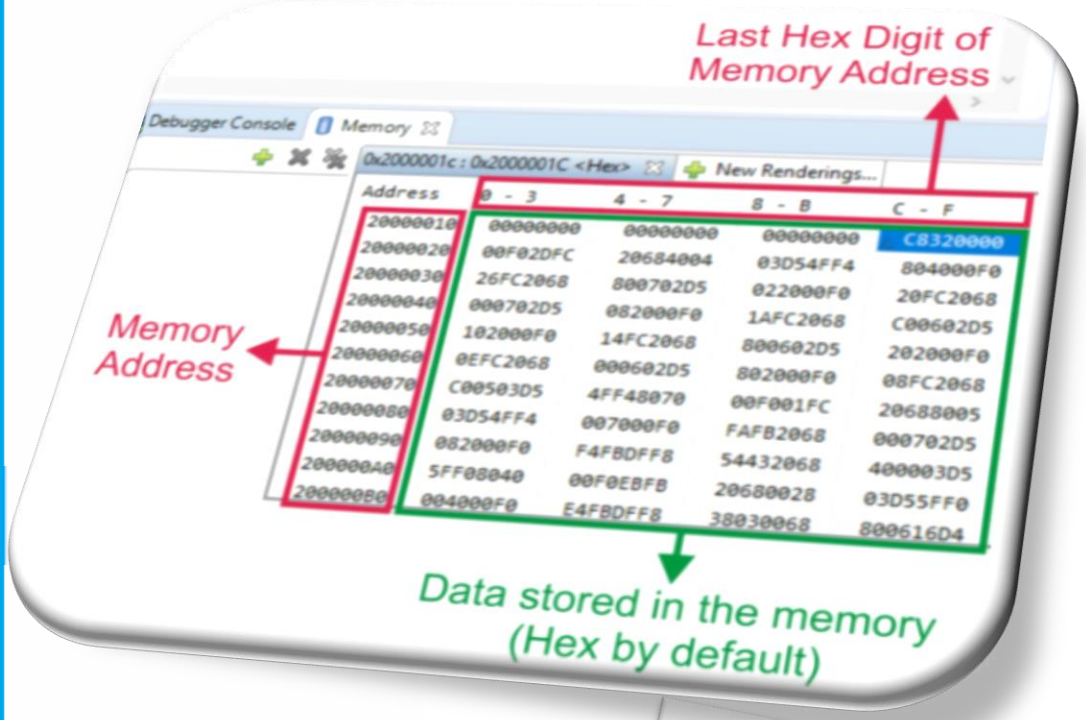
(i.e. >4 billion memory locations!)



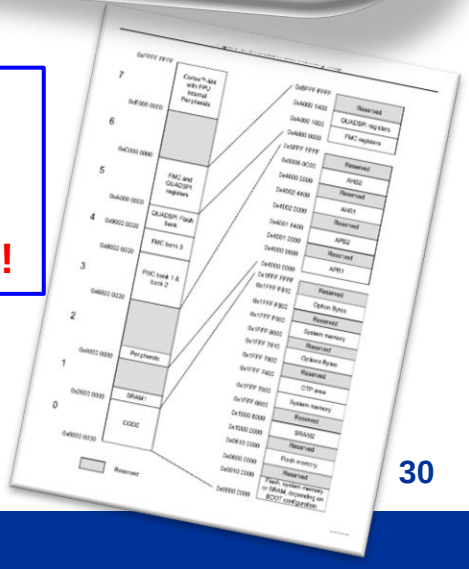
Byte Addressability

- 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**
- The **word address** is that of the **lowest address byte**
- e.g. if the **byte addresses** of 4 consecutive bytes (of a word) end with 0x..**0, 1, 2 & 3** in a 32-bit system, the **word address** is 0x..**0,**
- since every word is 4 bytes in a 32-bit system, the addresses of subsequent words will end with **4, 8, C, 0,** *and this pattern repeats*





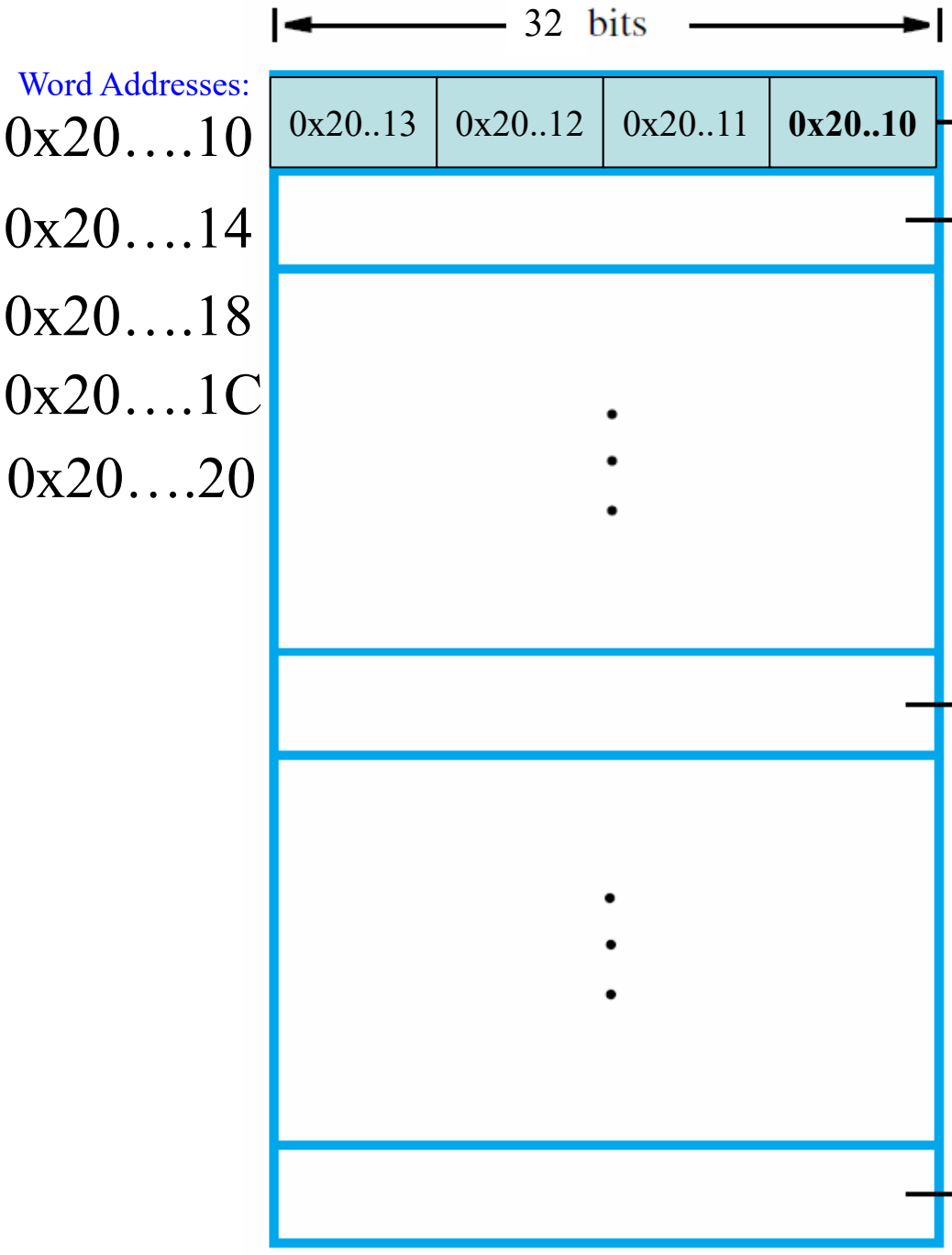
***Note*: memory is also quite often shown with the lowest address at the bottom – Be Careful!**

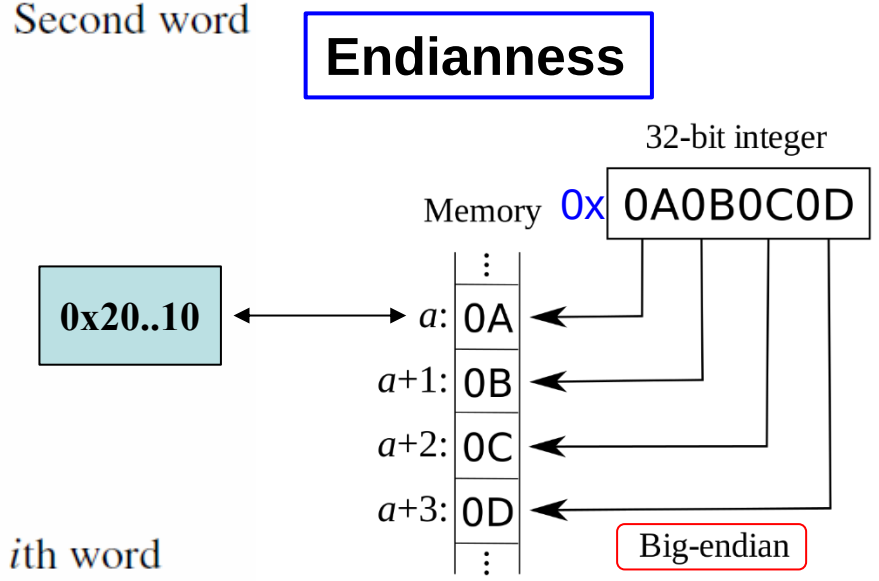


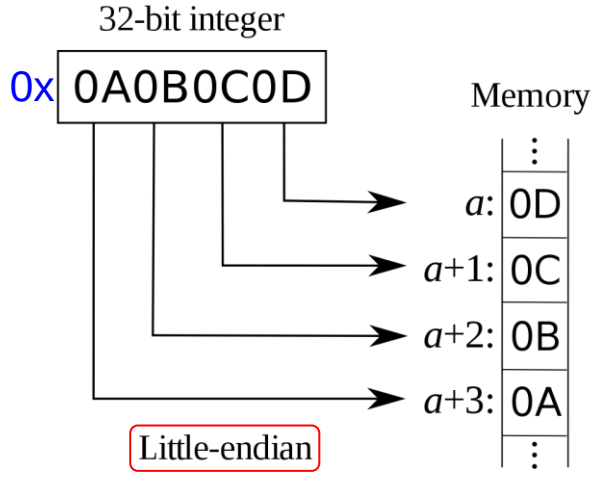
Endianness

- 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


32 bits
Word Addresses:
0x20….10
0x20….14
0x20….18
0x20….1C
0x20….20
0x20..13
0x20..12
0x20..11
0x20..10
First word
Second word
ith word
Last word


Endianness
32-bit integer
Memory
0x
0A0B0C0D
0x20..10
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
Big-endian

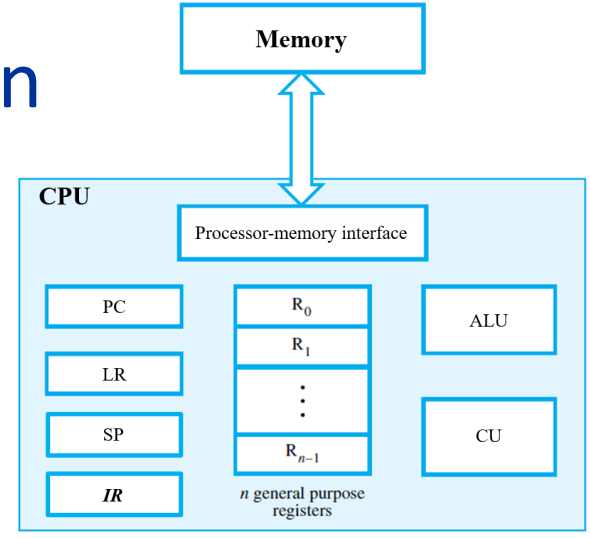

32-bit integer
0x
0A0B0C0D
Memory
a: 0D
a+1: 0C
a+2: 0B
a+3: 0A
Little-endian

3.3 Memory Operations

- CU initiates the **transfer** of both instructions & data between memory & the processor registers

- (low-level*) ***Read*** memory operation: contents at the memory address location given by the processor is **retrieved**

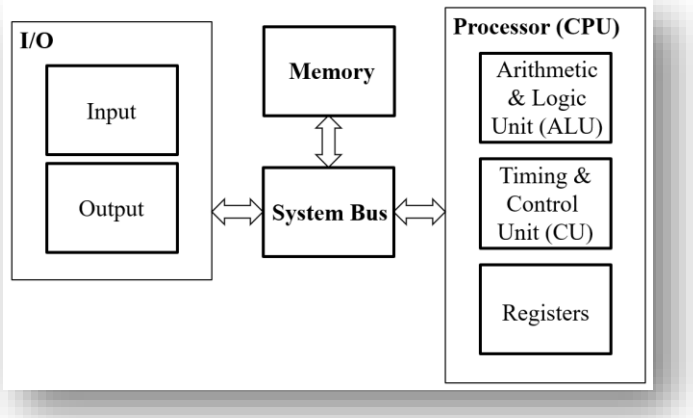
- (low-level) ***Write*** memory operation: contents at the given memory location is **overwritten** with the given data

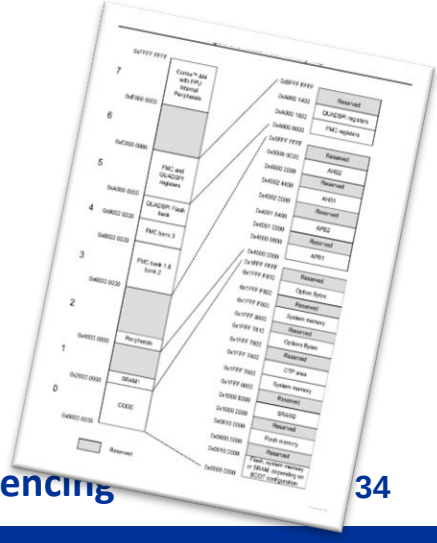


** low-level refers to the operation at the hardware level*

4. RISC Instructions Exec & Sequencing

- Basic **types of instructions** a computer must support:
  - **data transfers** to & from the **memory**, e.g. *load, store*
  - **arithmetic & logic** operations on data, e.g. *add, xor*
  - 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





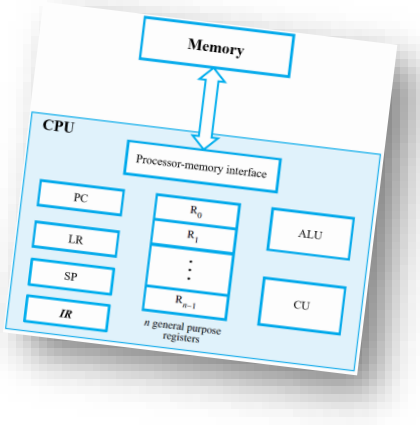
4.1 RISC Instruction Set Characteristics

- Most RISC instructions occupy a single word
- RISC **instruction** specifies the **operation** (e.g. ADD)
  & the locations of its **data operands** (e.g. R1)

*32-bit instruction word in **IR** :*

opcode …	registers …	value of #imm*X* (*X*-bit)

- RISC uses Load & Store architecture, i.e.
  - only **Load** & **Store** instructions have access to the memory operands
  - operands for arithmetic & logic 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)*



4.2 A RISC Program Example

- Consider a high-level language statement:

$$C = A + B$$

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

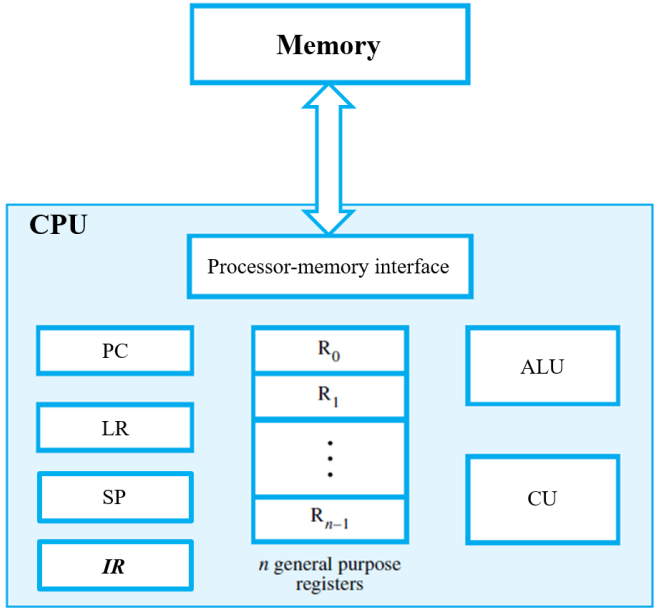
$$[C] \leftarrow [A] + [B]$$

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

A *generic* RISC Program Example

- A possible simple ***generic*** RISC asm program for the required tasks could be:

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



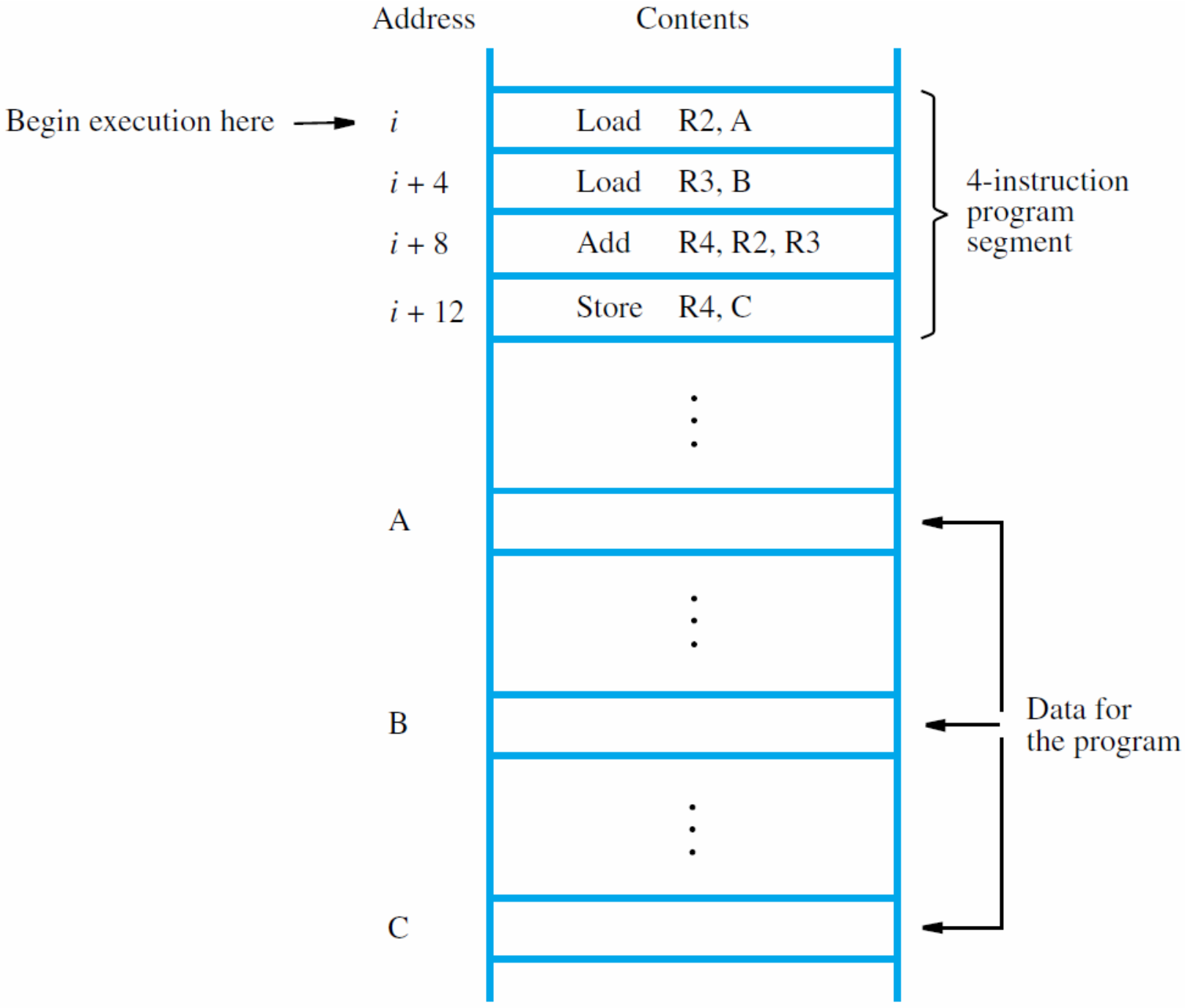
- Load instruction: transfers data to Register
- Store instruction: transfers data to the memory

Load, Store destinations differ despite the same operand order!

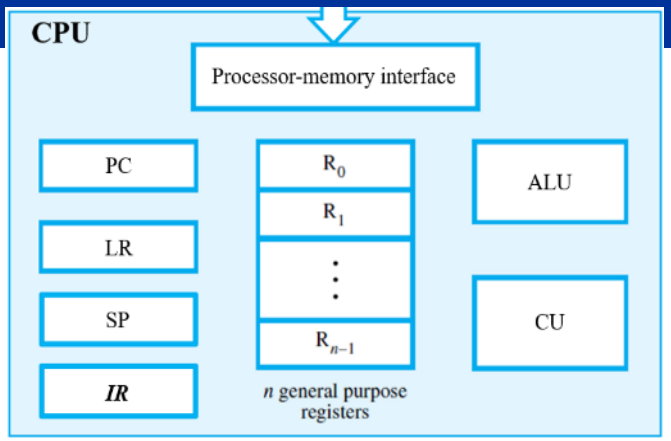
4.3 RISC Program Storage

- Consider the preceding 4-instruction program, how is it stored in the memory? (assuming 32-bit word length, byte-addressable)
  - Place the first RISC instruction word at address *i*
  - Remaining instructions are at *i* + 4, *i* + 8, *i* + 12 (i.e 0xC)
  - For now, assume that the Load & Store instructions specify the desired operand addresses ***directly***

(more addressing modes to be covered later…)


Address
Contents
Begin execution here
i
i + 4
i + 8
i + 12
Load R2, A
Load R3, B
Add R4, R2, R3
Store R4, C
4-instruction program segment
A
B
C
Data for the program

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. before branching & looping)
   i.e. content of **PC is *i* + 16** (0x10) after the execution of the 4th instruction (i.e. *Store R4, C* in the example)

Instruction Execution (Fetch-Execute)

NUS National University of Singapore

➢ **Fetch**-phase:

- CU reads value in PC for the address of current instruction
- CU transfers and 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/logic ops
- Concurrently, PC is incremented/updated, pointing at the next instruction, ready for the next fetch

Details of Instruction Execution_1

	Memory
0x40	Load R2, A
0x44	Load R3, B
0x48	Add R4, R2, R3
0x4C	Store R4, C

	...
A	2
B	3
C	

	Registers
R2	
R3	
R4	
PC	0x40
IR	Load R2, A

Fetch

Execute

	Memory
0x40	Load R2, A
0x44	Load R3, B
0x48	Add R4, R2, R3
0x4C	Store R4, C

	...
A	2
B	3
C	

	Registers
R2	2
R3	
R4	
PC	0x44
IR	Load R2, A

Details of Instruction Execution_2

	Memory
0x40	Load R2, A
0x44	Load R3, B
0x48	Add R4, R2, R3
0x4C	Store R4, C

	...
A	2
B	3
C	

	Registers
R2	2
R3	
R4	
PC	0x44
IR	Load R3, B

Fetch

Execute

	Memory
0x40	Load R2, A
0x44	Load R3, B
0x48	Add R4, R2, R3
0x4C	Store R4, C

	...
A	2
B	3
C	

	Registers
R2	2
R3	3
R4	
PC	0x48
IR	Load R3, B

Details of Instruction Execution_3

	Memory
0x40	Load R2, A
0x44	Load R3, B
0x48	Add R4, R2, R3
0x4C	Store R4, C

	...
A	2
B	3
C	

	Registers
R2	2
R3	3
R4	
PC	0x48
IR	Add R4, R2, R3

Fetch

Execute

	Memory
0x40	Load R2, A
0x44	Load R3, B
0x48	Add R4, R2, R3
0x4C	Store R4, C

	...
A	2
B	3
C	

	Registers
R2	2
R3	3
R4	5
PC	0x4C
IR	Add R4, R2, R3

Details of Instruction Execution_4

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

	Registers
R2	2
R3	3
R4	5
PC	0x4C
IR	Store R4, C

Fetch

Straight-line Sequencing

Execute

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

	Registers
R2	2
R3	3
R4	5
PC	0x50
IR	Store R4, C

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

- **Selection (aka *Branching*)** instructions (think *if*, *if-else, switch-case, continue, break, goto* statements in C language) work by:
  - **placing the address** containing the required instruction **into PC**
  - then fetching & executing that instruction (& all subsequent ones, if any) **until** the end of the program or another branching occurs
  - this **same process** is used in **function/subroutine calls**

- **Iteration (aka *Looping*)** instructions that need to be performed repeatedly, (think *for*, *while, do-while* statements) work by:
  - **branching repeatedly** to the **start of a section** of a program **until** a terminating condition is satisfied
  - in other words, the **address** containing the **first instruction** of the section to be repeated is placed **into PC**

Branching & Looping Examples

Branching

	Memory
0x40	Some …
0x44	Instructions …
0x48	If **True**: 0x60
0x4C	**False**: 0x80
	…
0x60	Some …
0x64	Instructions …
0x68	For case **True**
	…
0x80	Some …
0x84	Instructions …
0x88	For case **False**
0x8C	…

Looping

	Memory
0x40	Beginning of …
0x44	Some …
0x48	Instructions, …
0x4C	Loop back
	…
	Some other …
	Instructions …
	Or data …

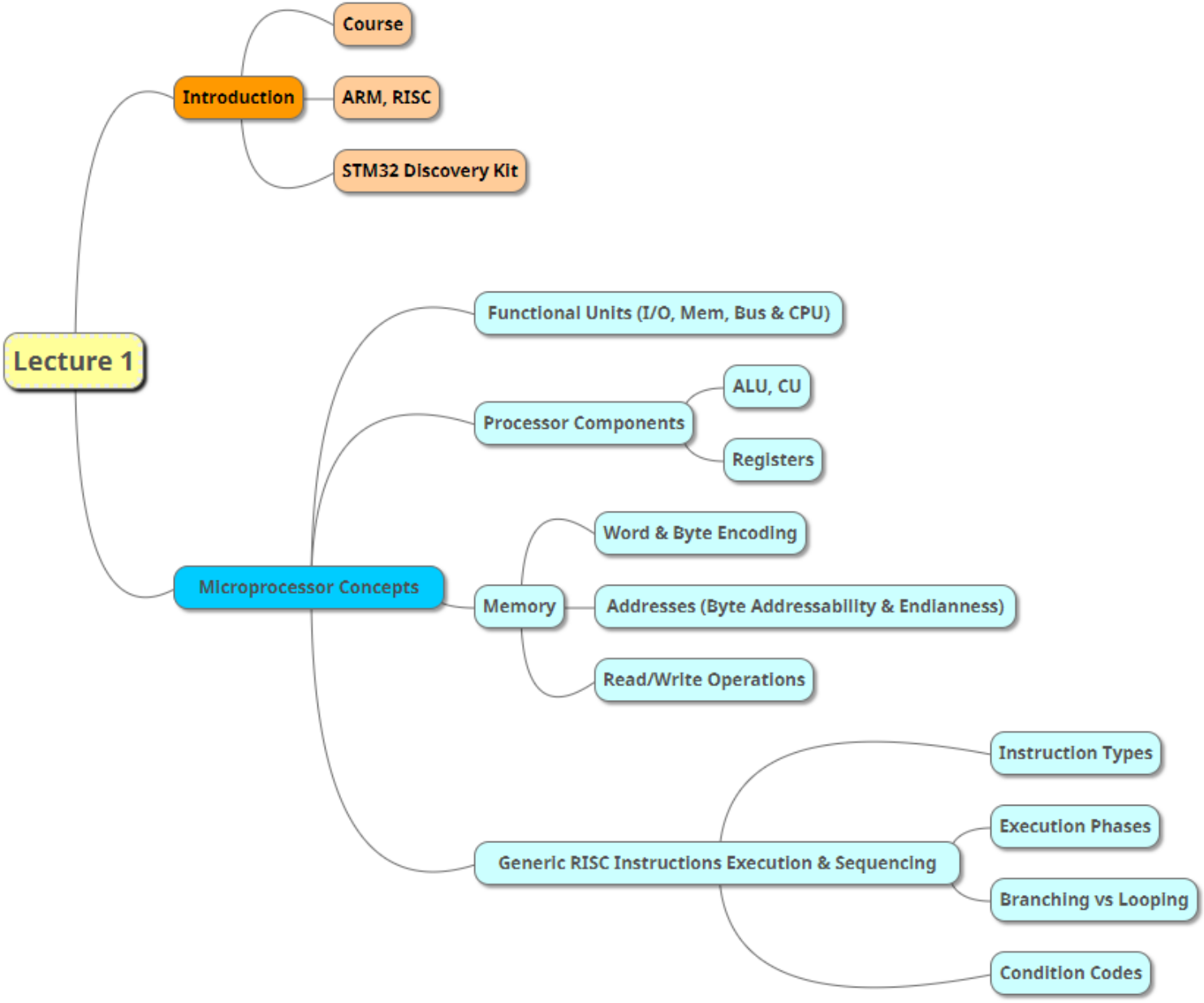
Branching & Looping: Condition Codes

- Processor maintains the information on arithmetic or logic 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 ops)
V	(overflow)	1 if overflow occurs, else 0 (for signed ops)

In Summary:



THE END