CZ1006
CE1006

Data Organisation in Memory

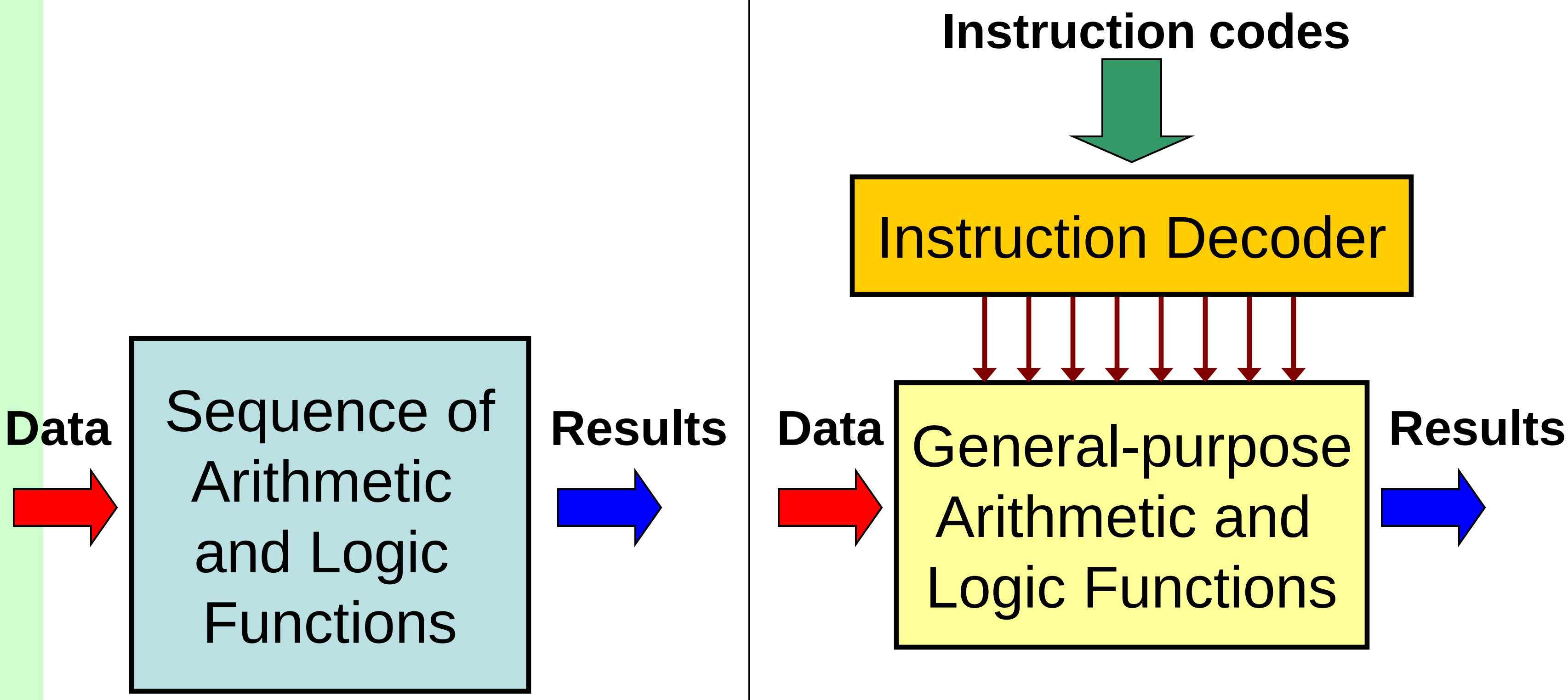
A/P Goh Wooi Boon

Data Organisation in Memory

Role of Memory in Computing

Learning Objectives (2.1a)
1. Contrast the programming in hardware and software approaches to computing 2. Describe the von Neumann's stored program concept. 3. Describe the role of memory in computing. 4. Describe the characteristics and function of different data storage elements in the memory hierarchy

Approaches to Computing



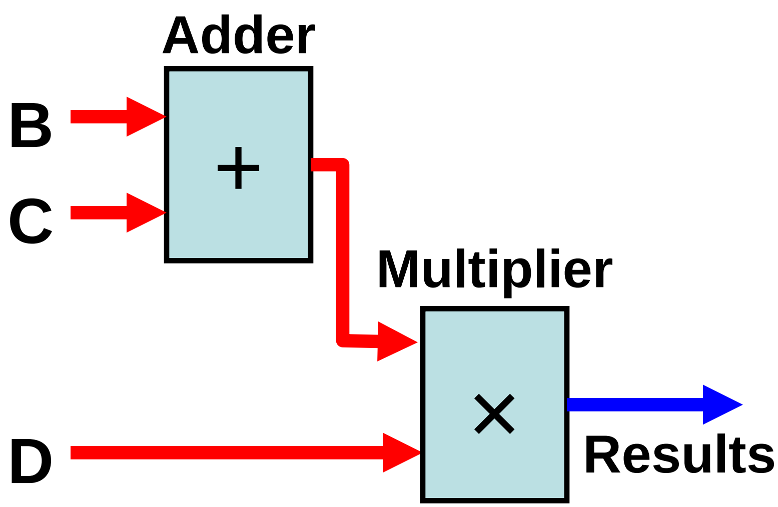
Programming in Hardware

Programming in Software

Approaches to Computing

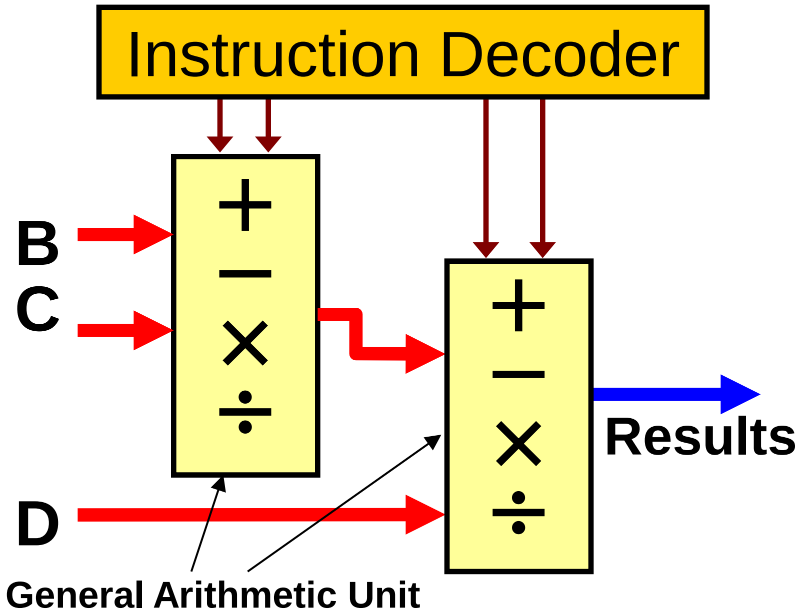
Implementing A = (B+C)×D

Fast computation but very inflexible



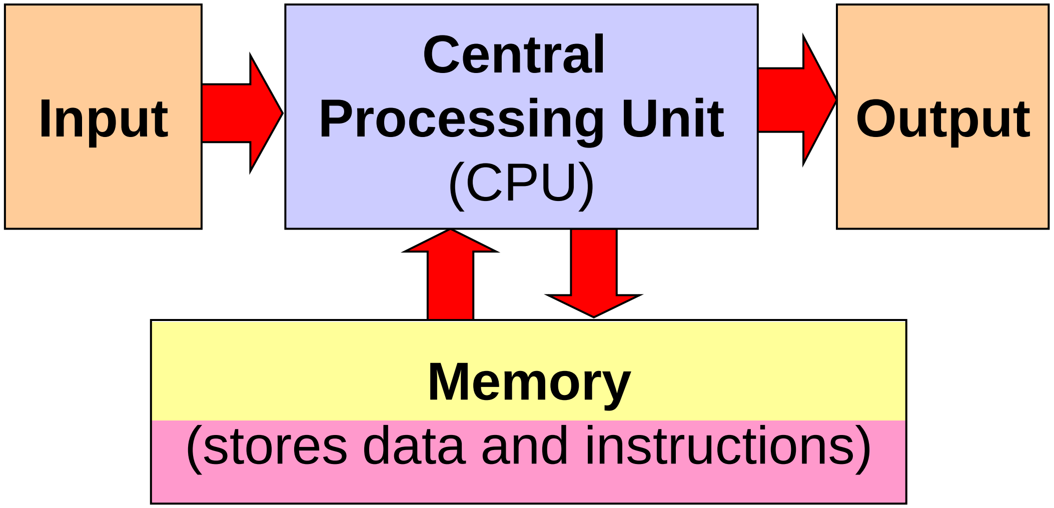
Programming in Hardware

Slower and more complex but easily programmable



Programming in Software

The Stored Program Concept



- Most modern day computer design are based on von Neumann's stored program concept:

1. Both data & instructions are stored in the same memory
2. Contents of memory are addressable by location, without regard to data type
3. Execution occurs sequentially (unless explicitly modified)

Code, Data and Memory

- What is code and what is data?
  - **Code** is a sequence of instructions.
  - **Data** are values these instructions operate on.

- What is the memory?
  - Its a sequential list of addressable storage elements for storing both instructions and data.

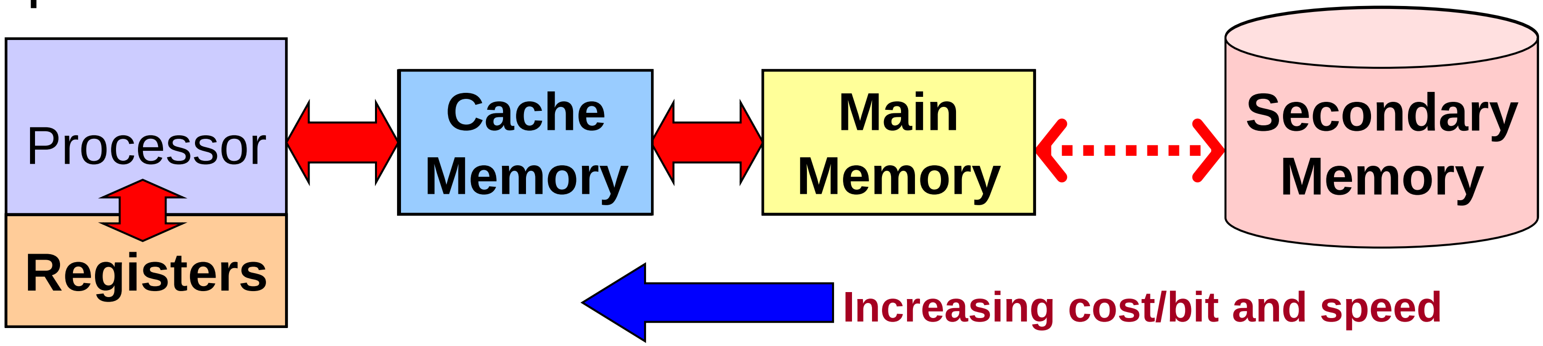
e.g. B+C

Address	Content	
0x000	+	**Instruction**
0x001		
0x002		
:	:	
:	:	Code Memory
0x100	B	**Data**
0x101	C	**Data**
:	:	Data Memory

Memory

Memory Hierarchy

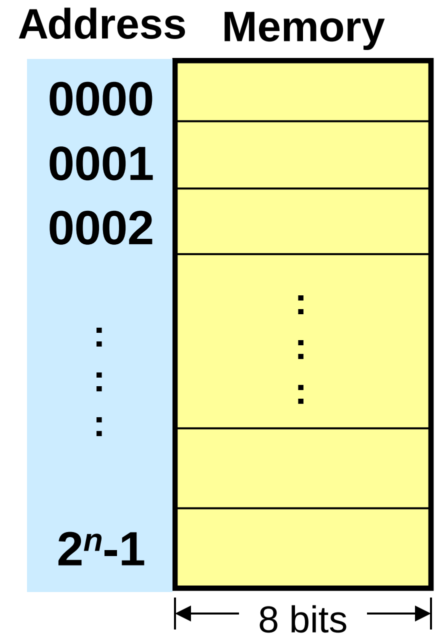
- Memories are generally organized in levels of increasing speed and cost/bit.



- **Registers** Very fast access but limited numbers within CPU. Operates at CPU clock rate (size: 2-128 registers)
- **Cache Memory** Fast access static RAM close to CPU. Typical access time 3-20nS (size: up to 512 kB)
- **Main memory** Usually dynamic RAM or ROM (for program storage). Typical access time 30-70nS. (size: up to 16GB)
- **Secondary Memory** Non always random access but non-volatile. Maybe be based on magnetic or flash technology. Typical access time 0.03-100mS. (size: up to 4TB)

Characteristics of Main Memory

- **Fix-sized** (typically 8-bit) storage location accessible at high speed and in **any order**.
- Each byte-sized location has a unique **address** that is accessed by specifying its binary pattern on the **address bus**.



- Memory size is dependent on number of lines (*n*) in the address bus. (Memory size = 2^n bytes).
- Memory stores both **data** and **instructions.**

  Consecutive locations used to store multi-byte data.

Summary

- Main memory contents are accessed using unique addresses.
- In the von Neumann architecture, a single memory stores both instructions and data.
  - However, instructions and data are usually kept in different areas in memory.
- Each addressable memory location stores a fixed number of data bits, normal 8 bits (i.e. a byte).
  - Storing data that are larger than a byte size requires the use consecutive memory locations.

Data Organisation in Memory

Number Representation

Learning Objectives (2.1b)

1. Describe the different C numeric data types and their characteristics.
2. Describe the concept of numeric range and its implications to data size.
3. Describe how multi-byte numbers are stored in memory.

Data Representation in Memory

- Programming languages like C has many different data types.
  - Numbers
  - Characters
  - Boolean
  - Arrays
  - Structures
  - Pointers
- How are these variables stored in memory?
  - Knowledge of their representation in memory allows us to find efficient ways to access them in our program.
  - Note: ANSI C programming language will be used as an example.

Number Representation

- ANSI C data types that represent numbers come in various varieties (basic type, sign & size).
- There a two **basic types**. Whole number or **integer** and **floating point** number.
  - Integer, declared as `int` (e.g. 32,676)
  - Floating point, declared as `float` (e.g. 3.2676×10^4)
  - Floating point numbers are useful for scientific calculations and has issue of trading off **precision** and **range** for a given size (in bits).

    (to be covered in later lectures in Computer Arithmetic)
  - Some CPUs are only integer-based and make use of a additional floating point unit (FPU) to support floating point computations.

Number Representation (cont)

- Floating point numbers are always signed.
- Integers can be either **signed** or **unsigned**.
  - Signed integer, declared as `int` (e.g. -1)
  - Unsigned integer, declared as `unsigned int` (e.g. 255)
  - Most processors interpret signed numbers using the **2's complement** representation.

2's complement	Unsigned
$0111\ 1111_2$ = (**127**)	$0111\ 1111_2$ = (**127**)
$1111\ 1111_2$ = (**-1**)	$1111\ 1111_2$ = (**255**)

- Use unsigned numbers where possible to increase the positive range (e.g. counting a population).

Number Representation (cont)

- The **range** can be increased by using more bytes to represent the number.

Type	Bytes	Bits	Range	
signed char	1	8	-128 -> +127	
unsigned char	1	8	0 -> +255	
short int	2	16	-32,768 -> +32,767	(+/-32KB)
unsigned short int	2	16	0 -> +65,535	(64KB)
unsigned int	4	32	0 -> +4,294,967,295	(4GB)
int	4	32	-2,147,483,648 -> +2,147,483,647	(+/-2GB)
long int	4	32	-2,147,483,648 -> +2,147,483,647	(+/-2GB)
long long int	8	64	-(2^63) -> (2^63)-1	
float	4	32		
double	8	64		
long double	12	96		

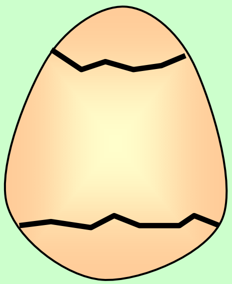
ANSI C numeric data types and their respective size and range

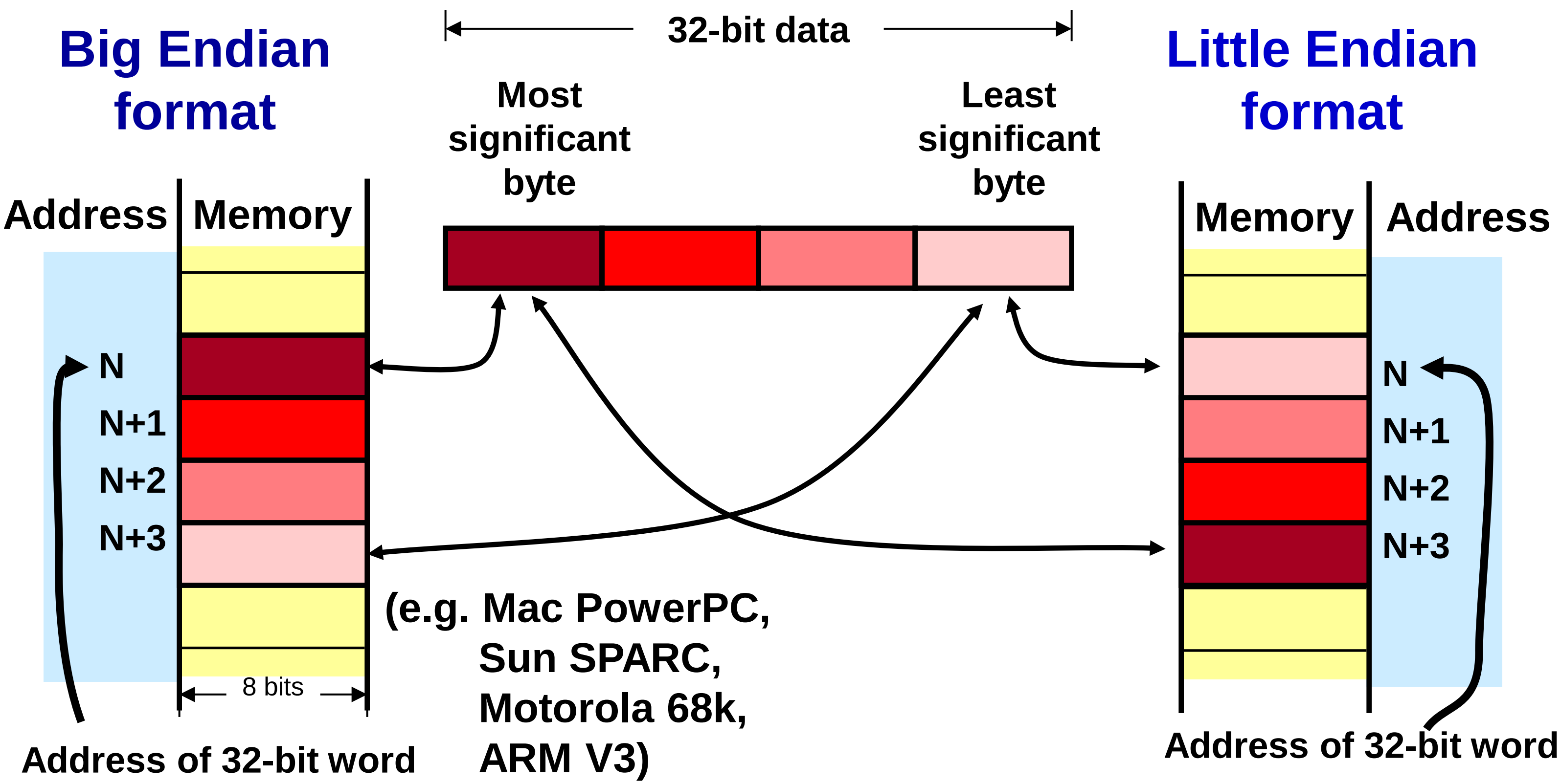
Note: These sizes may change depending of the processor and compiler

- Use appropriate suffix (**`short`** and **`long`**) to specify required size. Use appropriate size to reduce memory requirements of your program.

Data Organization in Memory

- How is a 32-bit integer stored in memory?
- There are two ways, depending on the **byte-ordering** of the data in memory





(e.g. Mac PowerPC, Sun SPARC, Motorola 68k, ARM V3)

(e.g. Intel x86, DEC Alpha, 8051)

Summary

- There are 2 basic number data types in C, namely integer (`int`) and floating point (`float`).
- The **2's complement** number representation is generally used to interpret signed integer values.
  - With the representation of signed values, the positive range for a given number of bits is halved.
- The varying byte-ordering of multi-byte number storage in memory gives rise to the **Big Endian** or **Little Endian** formats.

Data Organisation in Memory

Characters, Boolean, Arrays and Strings

Learning Objectives (2.1c)
1. Describe the char data type and its representations.
2. Describe the Boolean data type and its implementation.
3. Describe the representation of arrays in memory.
4. Describe the C and Pascal strings storage in memory.

Data Representation in Memory

- Programming languages like C has many different data types.
  - Numbers
  - **Characters**
  - **Boolean**
  - **Arrays (and Strings)**
  - Structures
  - Pointers

Character Representation

- Computer not only process numbers but handles character data (e.g. text processing, print display).
  - In ANSI C, a character type is declared as **`char`**.
  - A char variable required **one byte** of memory storage
  - Char data in a computer are stored in **binary** but they are transformed into representative characters through some encoding standard.
  - The most common character encoding standard is the 7-bit **ASCII** code.
  - There are many other character encoding standards - DEC's Sixbit (6-bit), IBM's EBCDIC (8-bit) and Unicode (16-bit), etc.

ASCII

- American Standard Code for Information Interchange (ASCII) is a **7-bit code** for representing characters.
  - Lower case, upper case and digits are **contiguous**. This makes it easy to range check a character’s category and also transpose it from one case to another.
  - A byte is normally used to store a ASCII character and MSB could be used for **parity error checking**.

e.g. digits from 0x30 to 0x39

LS \ MS	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	
F	SI	US	/	?	O	_	o	DEL

7-bit ASCII Table

SIXBIT

- DEC's SIXBIT character code was popular in the 1960's and 70's.
  - Six-bit character format was popular with DEC's PDP-8 and PDP-10 computers, which used **18-bit** and **36-bit** processors respectively.
  - Not used for general text processing as it lacks control characters like CR and LF.
  - Six-character names such as **filenames** and **assembler symbols** were stored in a single 36-bit word of PDP-10,

MS \ LS	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
1	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
2	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
3	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_

SIXBIT Table

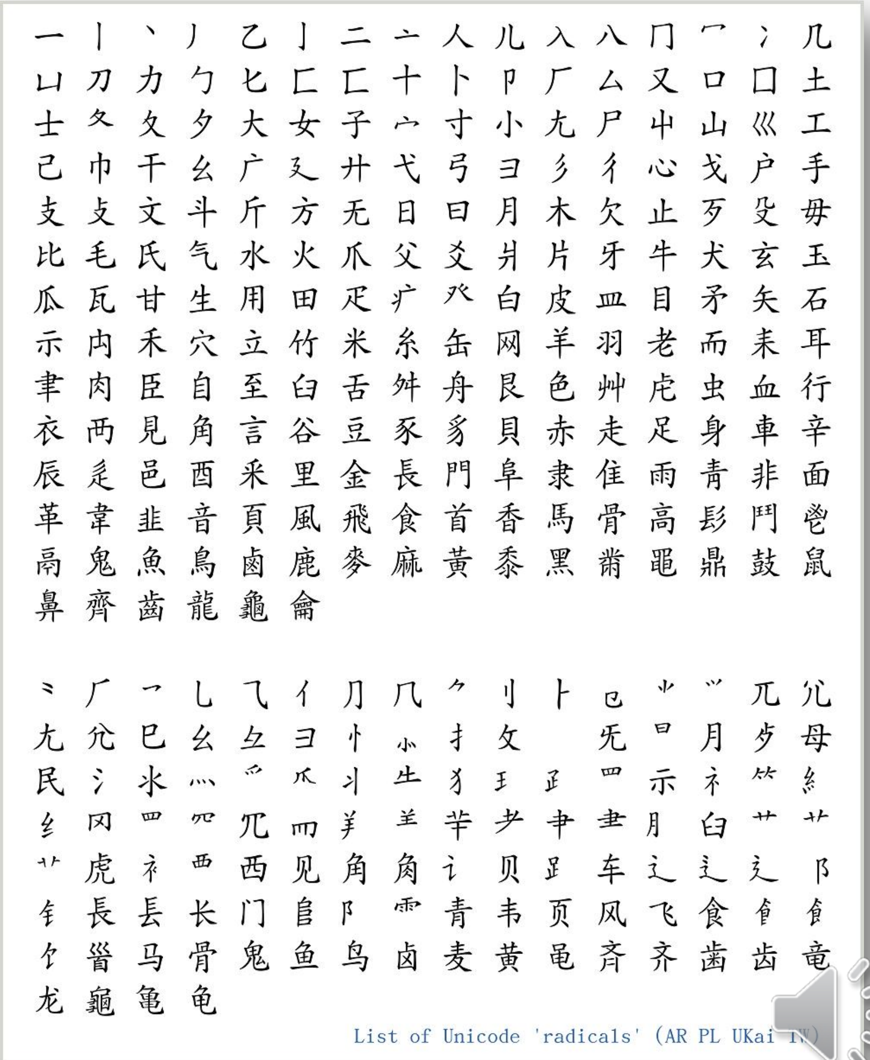
Unicode

- Developed to handle text expressions for all living languages in the world.
  - A **16-bit** character encoding that is downward compatible with ASCII.
  - Adopted by technologies such as XML, Java programming language, Microsoft .NET Framework and many operating systems.

Unicode radicals for KangXi

214 radicals to support Chinese, Japanese and Korean ideographs

(U+2F00 – U+2FDF)



List of Unicode 'radicals' (AR PL UKai TW)

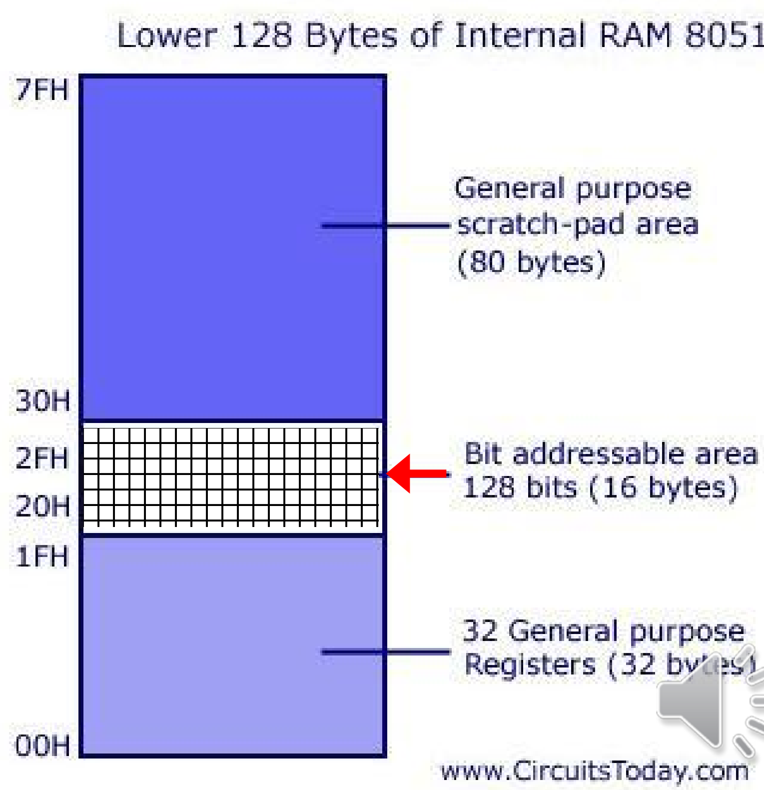
Boolean Representation

- Boolean variables have only 2 states.
- The Boolean type was made available in ANSI C (after 1999) as `_Bool` with the `stdbool.h` header file.
- Values assigned in C: **False** = 0, **True** = non-zero (e.g. 1)
- Memory storage for Boolean variables is **inefficient** as most implementation use a byte (minimum memory unit) to store a 1-bit Boolean value.
- Some embedded processors (e.g. 8051) with limited memory resources have bit-addressable memory.

8051 - Bit Addressable Memory

- The 8051 processor support a small portion of bit addressable memory.
  - In embedded applications, data variables often have Boolean values related to ON-OFF status of discrete sensors (e.g. switches).
  - Bit addressable memory provide an efficient way to handle such information.

8051 8-bit Processor

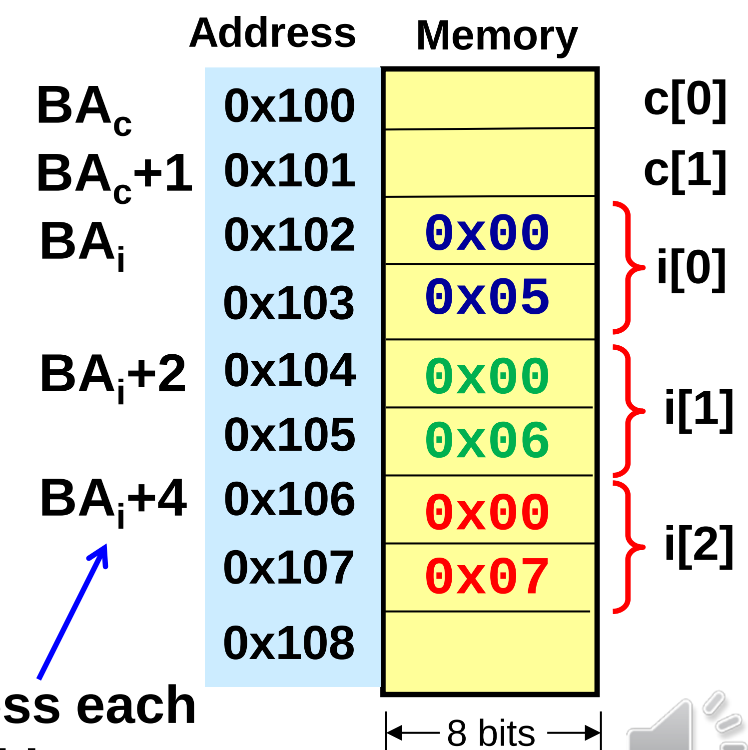


Array Representation

- A linear array is a consecutive area in memory storing a series of homogenous data type.
  - Elements of an array are accessed through appropriate offset from the **base address** (BA) of the array.

```
main()
{
  char c[2];          //2 element char array c
  short int i[3];     //3 element integer array i
  i[0] = 5;           //assign values to array i
  i[1] = 6;
  i[2] = 7;

}
```



Offset from base address to access each element of the arrays c and i

String Representation

- A string in a C program is an array of characters terminated by a null (**`0x00`**) character.

```
main()
{
 char s[4]="123";  //a string constant
}
```

Base Address	Address	Content in Memory
BA_s	0x0100	0x31
	0x0101	0x32
	0x0102	0x33
	0x0103	0x00
	0x0104	:
	0x0105	:
	0x0106	:

← 8 bits →

- An alternative is the **Pascal string**, which stores the length of the string at the start of the string.

Which method is better?

Ref: Google Search "C strings vs Pascal strings"

Nested Array

- Each element of an array can itself be an array.
  - General principles of array allocation and referencing hold for nested array.
  - An array of arrays is created.

```
main()
{
 int k[3][2];  //a 3x2 integer array
}
```

Offset from BA	Address	Element in Memory
BA_k	0x0100	k[0][0]
BA_k+4	0x0104	k[0][1]
BA_k+8	0x0108	k[1][0]
BA_k+12	0x010C	k[1][1]
BA_k+16	0x0110	k[2][0]
BA_k+20	0x0114	k[2][1]
	0x0118	:

← 32 bits →

- The offset from BA for element **k[a][b]**

  = `sizeof(int)*((2*a)+b)`

Summary

- The **char** data type is stored as a byte and interpreted using the **ASCII** encoding.
- Boolean values in C are a byte with the values of **0** for **False** and **non-zero** value for **True**.
- An **array** is constructed using **consecutive** memory locations and is accessed using its **base address**, which is the starting address.
- A **C string** is an array of characters that is terminated with a **NULL** byte value of 0.

Data Organisation in Memory

Pointers and Structures

Learning Objectives (2.1d)
1. Describe the representation of pointers in memory. 2. Describe the representation of structures in memory. 3. Describe data alignment considerations for efficient access of objects in structures.

Data Representation in Memory

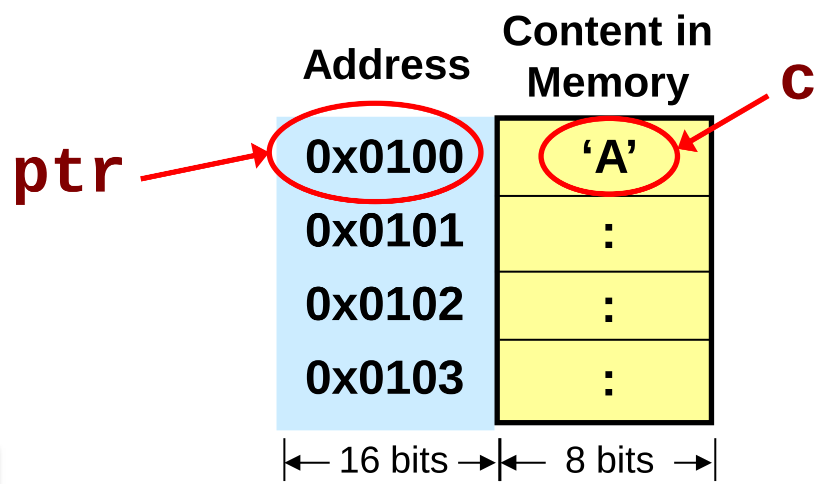
- Programming languages like C has many different data types.
  - Numbers
  - Characters
  - Boolean
  - Arrays
  - Structures
  - Pointers

Pointers Representation

- Pointers in C provides a mechanism for referencing memory variables, elements of structures and arrays.
- C pointers are declared to point to a particular data type.
- Regardless of data type, the value of the pointer is an **address** and its **size is fixed**, i.e. the number of bytes needed to specify an address (this varies with processor).

```
main()
{
  char c ;        //char variable c
  char *ptr;      //char pointer ptr

  c = 'A';        //assign value 'A' to c
  ptr = &c;       //ptr gets address of c
}
```



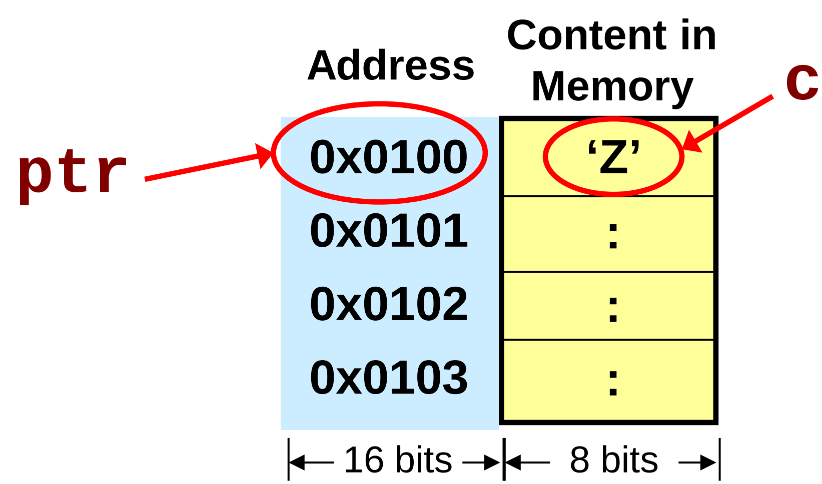
Variable	Size (Bytes)
c	1
ptr	2

Note: 1. Assume address of **c** = 0x0100
2. Assume addresses in this CPU are specified using 2 bytes.

Dereferencing a Pointer

- When properly initialized, a pointer contains the start address where the memory variable can be found.
- We can use the dereferencing operator (*) to copy a value to the address pointed to by the pointer **ptr**.
- Knowing the start address (base address) of a array or structure makes it easy to access their different elements.

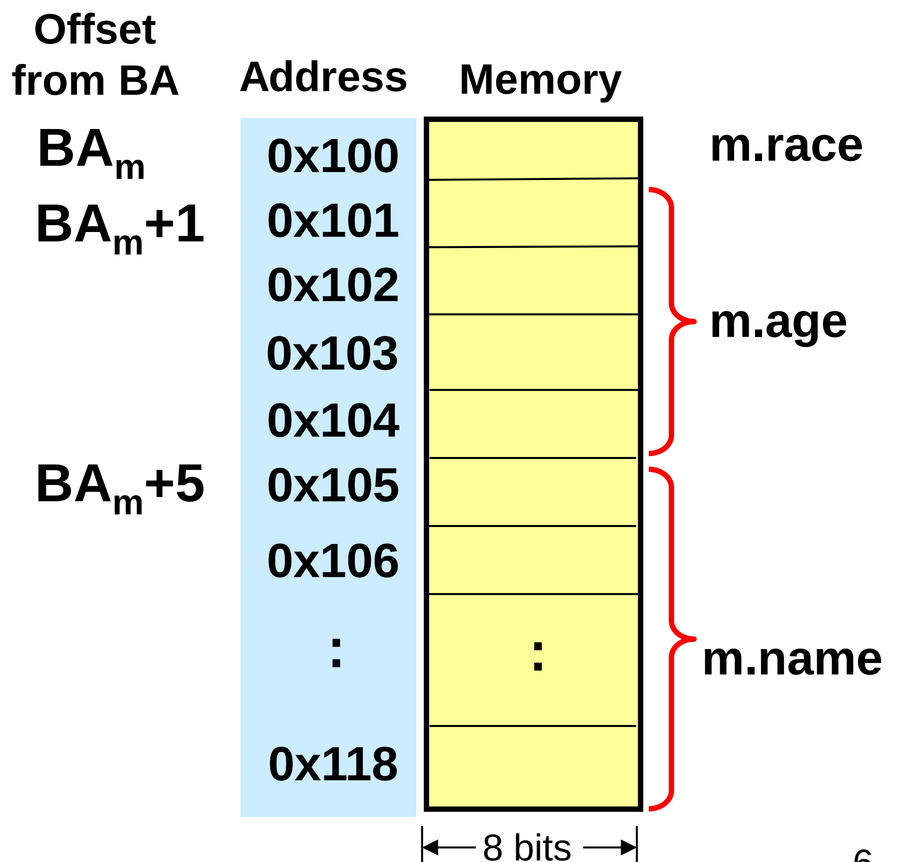
```
main()
{
  char c ;          //char variable c
  char *ptr;        //char pointer ptr
  c = 'A';          //assign value 'A' to c
  ptr = &c;         //ptr gets address of c
  *ptr = 'Z';       //use dereferencing
                    //operator on ptr to give
                    // variable c value 'Z'
```



Structure Representation

- Structure allows new data types to be created by combining objects of different types.
- Each data type in a **declared** structure variable occupies predefined consecutive locations based on data type size.

```
main()
{
 struct Man ;  //define structure Man
 {
  char race;
  int age;
  char name[20];
};
 Man m;
}
```



Data Alignment

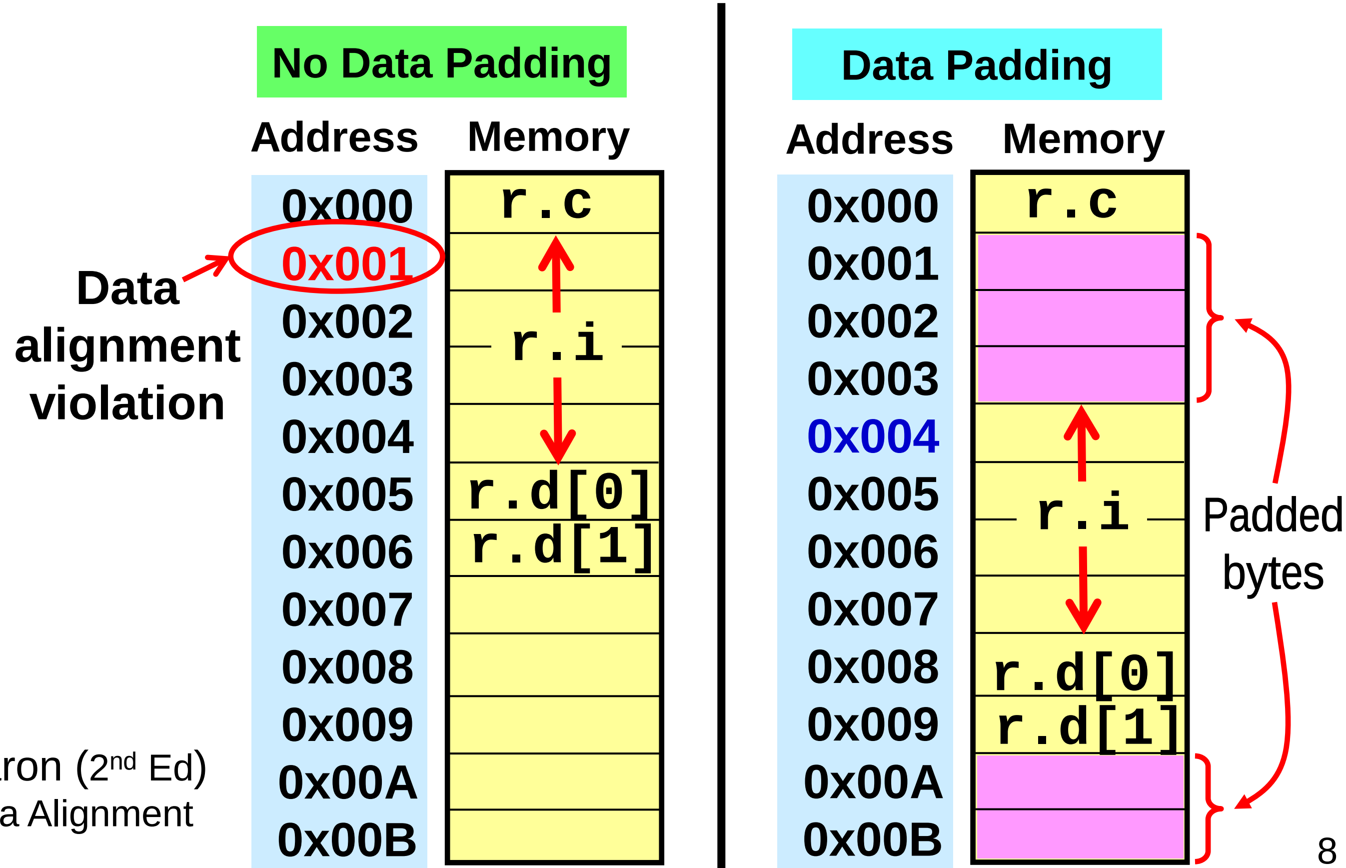
- Most computer systems place restrictions on the allowable access address of different data types.
  - Multi-byte data (e.g. **`int`**, **`double`**) must be aligned to addresses that are multiple of values such as 2, 4, or 8.
  - Programs written with Microsoft (Visual C++) or GNU (gcc) and compiled for a 64-bit Intel processor will use the following data alignment enforcement:

Data Type	Size (Byte)	Example of allowable start addresses due to alignment
char	1	0x..0000, 0x..0001, 0x..0002
short	2	0x..0000, 0x..0002, 0x..0004
int	4	0x..0000, 0x..0004, 0x..0008
float	4	0x..0000, 0x..0004, 0x..0008
double	8	0x..0000, 0x..0008, 0x..0010
pointer	8	0x..0000, 0x..0008, 0x..0010

Data Structure Alignment

- **Data padding** (addition of meaningless bytes) is used by compilers to ensure alignment of different data types within a structure.
- Padded bytes are added between end of last data structure and the start of the next to maintain alignment.

```
struct rec1 {
  char c;
  int i;
  char d[2];
 }
 rec1 r;
   :
 rec1 t[10];
```

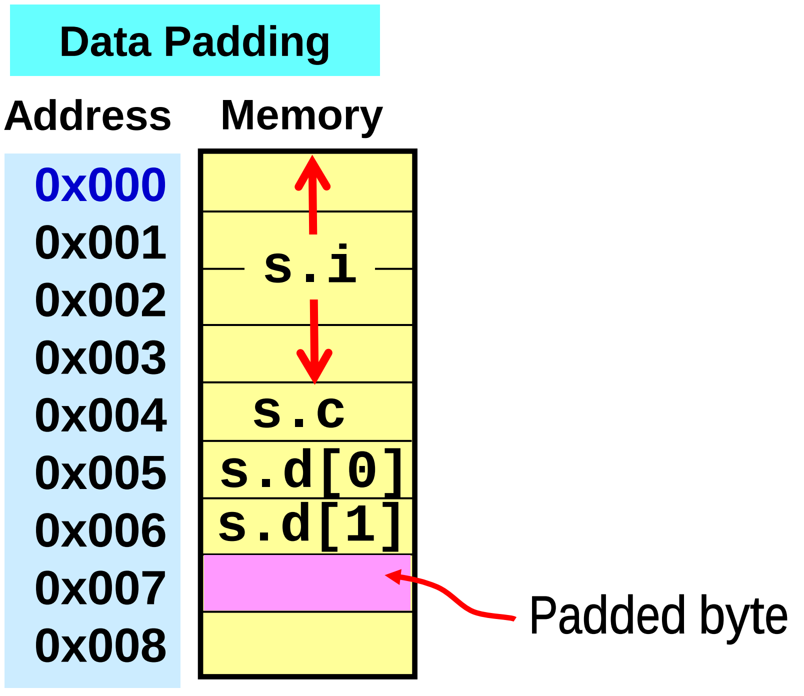


Ref: Bryant & O'Hallaron (2nd Ed) section 3.9.3 - Data Alignment

Data Structure Alignment (cont)

- Structures should be constructed with data alignment constraints in mind.
  - **Rearrange the order** of data objects in the structure based on their size to minimize data padding where possible.

```
struct rec2 {
  int i;
  char c;
  char d[2];
 }
 rec2 s;
```



Ref: Bryant & O'Hallaron (1st Ed) section 3.10 - Alignment

Summary

- The pointer data type is related to **addresses** and their size must match the size of the address range of the CPU.
- Non-homogenous elements of a **structure** are also stored in consecutive memory locations.
- The data alignment of multi-byte elements like integers must be handled using data padding.