CZ1106
CE1106

Chapter 2

Data Organisation in Memory

©2020, SCSE/NTU

1

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Role of Memory in Computing

Learning Objectives (2.1)

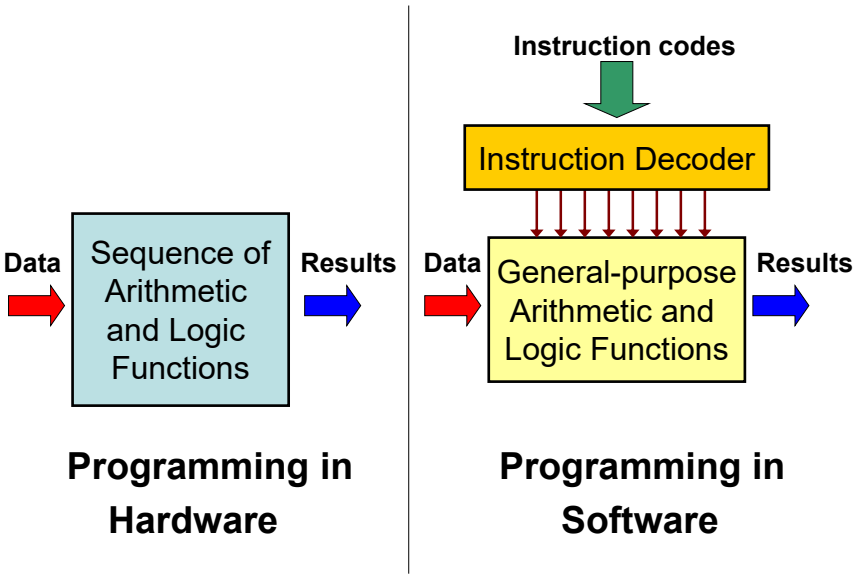
1. Describe the concept of programming in software.
2. Describe the von Neumann's stored program concept.
3. Describe the role of memory in computing.
4. Describe the characteristics and function of different data storage elements in the memory hierarchy.

©2020, SCSE/NTU

2

CZ1106
CE1106

Approaches to Computing



Programming in Hardware

Programming in Software

©2020 SCSE/NTU

3

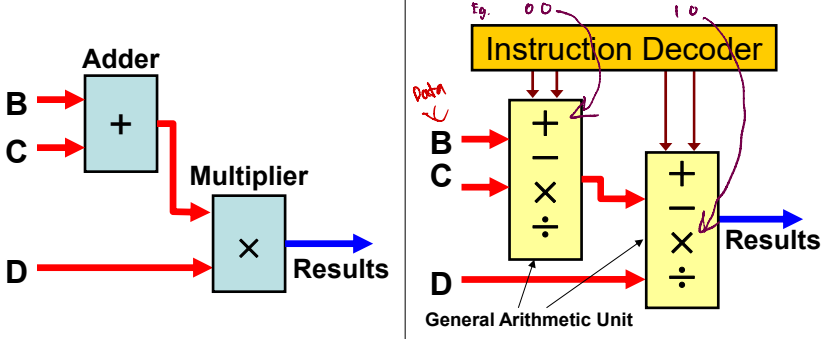
CZ1106
CE1106

Approaches to Computing

Implementing A = (B+C)×D

Fast computation but very inflexible

Slower and more complex but easily programmable



Programming in Hardware

Programming in Software

©2020 SCSE/NTU

4

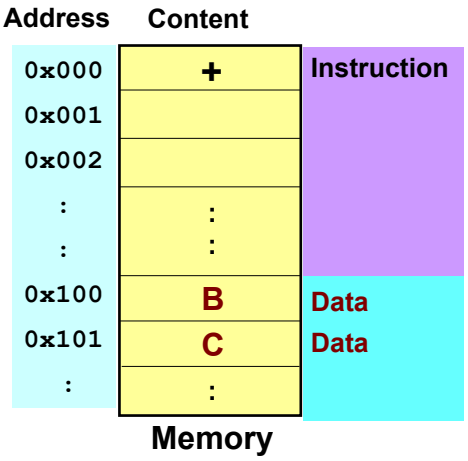
CZ1106 CE1106

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.

1 memory location = 8 bits

e.g. B+C

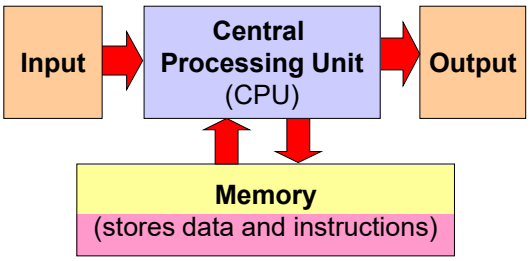


©2020 SCSE/NTU

5

CZ1106 CE1106

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)

↳ follows through order of addresses (unless have jump statement)

©2020 SCSE/NTU

6

lower to higher address

Standard int = 4 bytes

CZ1106
CE1106

CX1106

Multi-byte Number Ordering

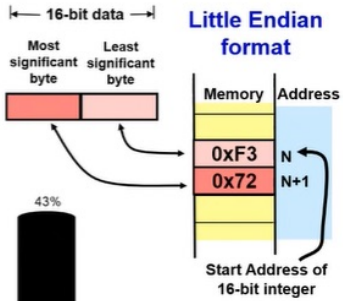
must not be 1

2 bytes

Address	Content in Memory
0x0100	0xF3
0x0101	0x72
0x0102	0x82
0x0103	:
0x0104	:

8 bits

Find a **positive short integer** in the memory map if **Little Endian** byte-ordering is used.

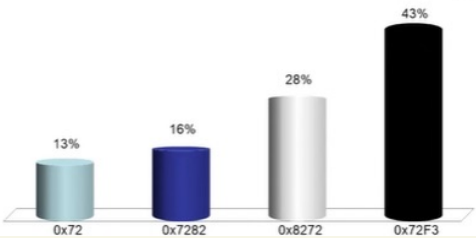


A. 0x72

B. 0x7282

C. 0x8272

D. ✓ 0x72F3

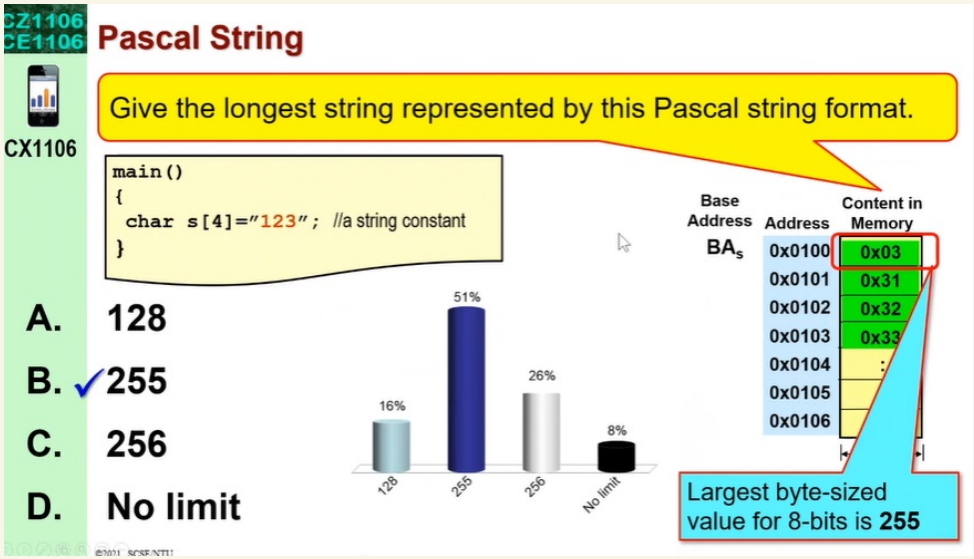


©2021. SCSE/NTU

A is wrong as it is only 1 byte

B is Big Endian byte-ordering

C is not a positive number



256 − 1 = 255

need the 0 to say that this string has no characters

∴ 0 is a value number to describe string

CZ1106
CE1106

Interpreting Data in Memory

CX1106

Address	Memory
N	0x00
N+1	0x41

Which interpretation is **not possible** for the memory contents shown?

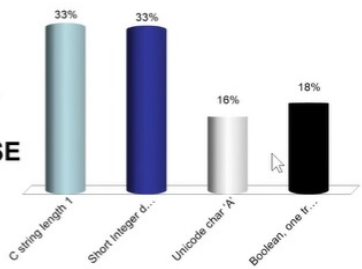
A. C string of length 1 ✓

B. Short integer with decimal value of 16640

C. One Unicode character representing letter 'A'

D. 2 Boolean variables, one TRUE and one FALSE

only non-zero value ⇒ not necessarily 1



©2021. SCSE/NTU

A. The null character (C string terminator) should be at a higher memory address

E.g.

0x41
0x00

← needs to be here

B. Using Little Endian interpretation, we get a short integer (2 bytes) of hexadecimal value 0x4100.

⇒ equivalent to decimal 16640

$0x4100 = (4 \times 16^3) + (1 \times 16^2) + (0 \times 16^1) + (0 \times 16^0)$

$= 16384 + 256 = 16640$

C. 0041 = 16-bit unicode for ASCII value 0x41 = 'A'

⇒ allow 1 bit, 2 bits, 4 bits description of character

⇒ in this case Big Endian interpretation

CZ1106
CE1106

CX1106

Pointers

Based on the given memory map below, what is the size (in bits) of the `short int` pointer `p`?

```
main()
{
 short int x;
 short int *p;
 x = 1;
 p = &x;
```

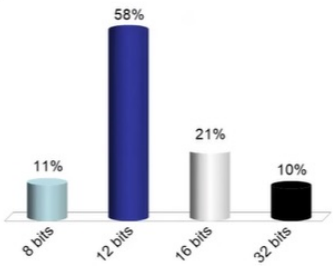
12-bit address still needs **two bytes** to store pointer address.

Address	Content in Memory
0x200	0x00
0x201	0x01
0x202	0x00
0x203	0x02
0x204	:
0x205	:
0x206	:

← 8 bits →

CPU has 12-bit address

A. 8 bits
B. 12 bits
✓C. 16 bits
D. 32 bits



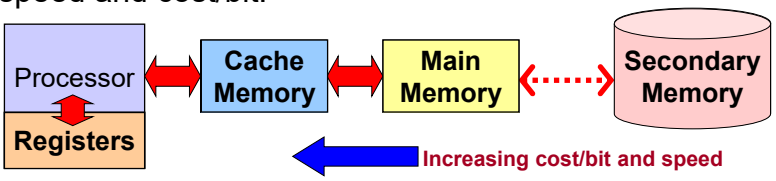
©2021, SCSE/NTU

B. can't squeeze 12 bit memory to 2 bytes
⇒ goes by 8, 16, 32 sequence (even though not fully used)

CZ1106 CE1106

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	Not always random access but non-volatile. Maybe be based on magnetic or flash technology. Typical access time 0.03-100mS. (size: up to 4TB)

©2020 SCSE/NTU

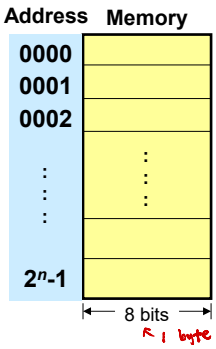
7

7

CZ1106 CE1106

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**.



Access content by putting pattern of address on the address bus; memory system then activate to allow access

Main memory → Processor (read content)
⇒ read memory operation

Processor → main memory
⇒ write memory operation

- Memory size is dependent on number of lines (n) in the address bus. (Memory size = 2^n bytes). ⇒ address lines
- Memory stores both **data** and **instructions. Consecutive** locations used to store multi-byte data.

©2020 SCSE/NTU

8

Memory module with it pins to connect to the address and data bus

Quiz:

Which memory type is the data storage on a nano SIM card?

Ans: Secondary Memory

CZ1106
CE1106

Summary

- Programming in software provides flexibility.
- Functionality specified by bit patterns in the instructions.
- Instructions are stored sequentially in memory.
- Programs are executed by fetching these instructions one at a time from memory.
- Memory stores both instructions and data.
- Memory contents are addressable by a unique address.
- Each unique address stores a fixed number of bits (i.e. 8 bits or a byte)

©2020 SCSE/NTU

9

CZ1106
CE1106

10

©2020 SCSE/NTU

10

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Number Representation

Learning Objectives (2.2)

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.

©2020, SCSE/NTU

11

CZ1106
CE1106

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 and process them in our program.
  - Note: Lecture notes will use ANSI C programming language as an example.

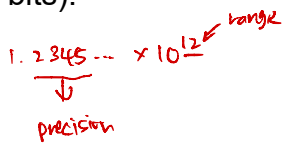
©2020, SCSE/NTU

12

CZ1106 CE1106

Number Representation

- ANSI C data types representing 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 an additional floating point unit (FPU) to support floating point computations.



©2020 SCSE/NTU

13

CZ1106 CE1106

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**)

Sign bits (MSB)

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

Learn More: Google "Signed number representation"

©2020 SCSE/NTU

14

Signed : 1111 1111 = -(?)
0000 0000
+ 1
0000 0001 = -1

CZ1106 CE1106

Number Representation (cont)

- The **range** can be increased by using more bytes to represent the number.
- Use appropriate suffix (`short` and `long`) to specify required size.
- Use appropriate size to reduce memory requirements of your program.

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	-(2^32) -> (2^32)-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

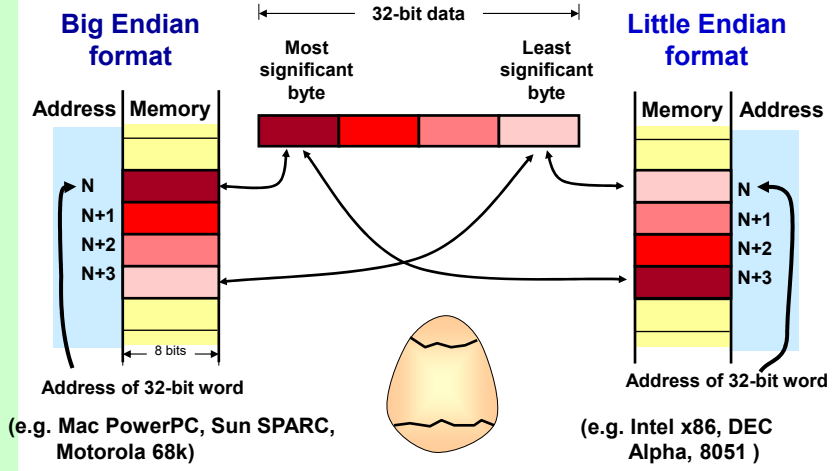
©2020, SCSE/NTU

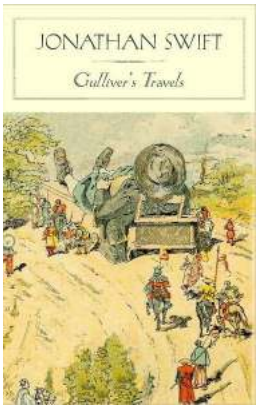
15

CZ1106 CE1106

Data Organization in Memory

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





Learn More: Google "Byte ordering"

©2020, SCSE/NTU

16

CZ1106
CE1106

Summary

- Numbers are represented in a variety of ways.
  - Number of bits.
  - Integer or Floating Point.
  - Signed or Unsigned.
- For a given data size (i.e. number of bits), range and precision must trade off against each other.
- Multi-byte integers are stored using either Big or Little Endian byte-ordering.

©2020, SCSE/NTU

17

CZ1106
CE1106

18

©2020, SCSE/NTU

18

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Character and Boolean Representation

Learning Objectives (2.3)

1. Describe the char data type and its representations.
2. Describe the Boolean data type and its implementation.

©2020, SCSE/NTU

19

CZ1106
CE1106

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
- Data in a computer is 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.

```
main()
{
 char c;    //declare a character variable

 c = 'a'; //assign character 'a' to variable

}
```

- There are many other character encoding standards - DEC's Sixbit (6-bit), IBM's EBCDIC (8-bit) and Unicode (16-bit), etc.

Learn More: Google "Character Codes"

©2020, SCSE/NTU

20

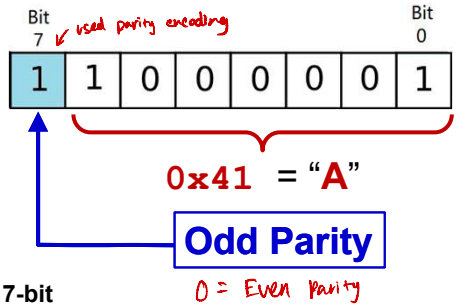
CZ1106 CE1106

ASCII

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

MS / LS	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



Learn More: Google "Parity bit and ASCII"

+32 to change from upper case to lowercase

©2020 SCSE/NTU

21

CZ1106 CE1106

SIXBIT

- DEC's SIXBIT character code was popular in the 1960's and 70's.
  - Not used for general text processing as it lacks control characters like CR and LF.
  - Six-bit character format was popular with DEC's PDP-8 and PDP-10 computers, which used 18-bit and 36-bit processors respectively.

LS / MS	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

©2020 SCSE/NTU

22

CZ1106
CE1106

Unicode

- more commonly used as ASCII only has 7 bits ∴ limited combinations

- Developed to handle text expressions for all major living languages in the world.
  - An **8,16 or 32-bit** character encoding standard that is downward compatible with ASCII.
  - Adopted by technologies such as XML, Java programming language, Microsoft .NET Framework and many operating systems.
  - Unicode 13.0 was launched on 20 March 2020. It contains a total of 143,859 characters.

Unicode radicals for KangXi
214 radicals to support Chinese, Japanese and Korean ideographs (U+2F00 – U+2FDF)

Learn More: Google "Unicode" or "Unicode characters"

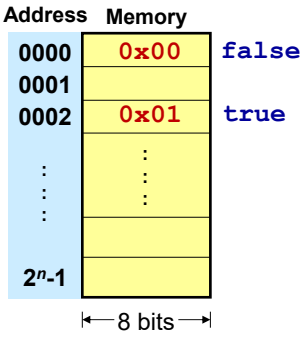
©2020 SCSE/NTU

23

CZ1106
CE1106

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.
  - The 8051 processor has 128 bit-addressable memory locations.



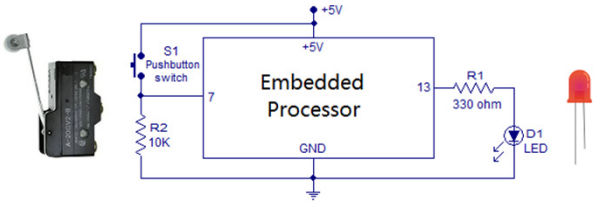
©2020 SCSE/NTU

24

CZ1106
CE1106

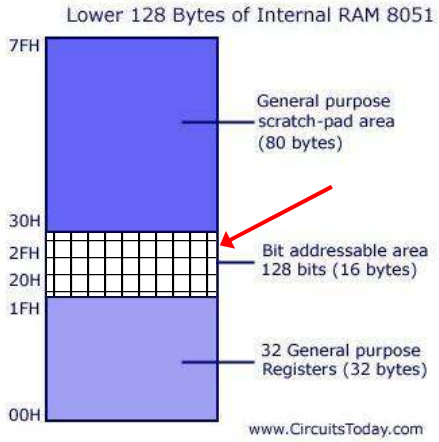
8051 - Bit Addressable Memory

- The 8051 processor support a small portion of bit-addressable memory.
- In embedded applications, data variables are often related to ON-OFF status of discrete sensors and output (e.g. switches or LEDs)



- Bit-addressable memory provide an efficient way to handle such information.

8051 8-bit Processor



©2020 SCSE/NTU

25

CZ1106
CE1106

Summary

- Characters data type is used to handle text.
- They are byte-sized.
- They need an encoding standard (ASCII being the most common).
- In the C programming language they are declared using `char`.

- Boolean data types are used to represent true and false states.
- They are usually stored using a whole byte.
- In ANSI C, the `false` value is given by the numeric value 0.
- The `true` value is taken as any non-zero value.

©2020 SCSE/NTU

26

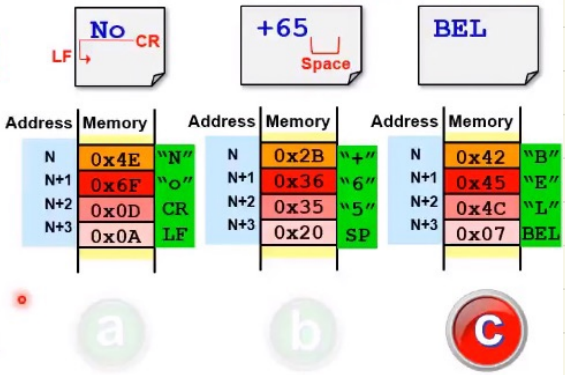
CZ1106
CE1106

Quiz: Text File in Memory

- Which of the memory contents below is **least likely** to be a segment of memory containing a plain text file?

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



BEL = control character ; controls a function in the terminal

∴ C least likely to be a plain text file

↑ not acceptable character in text file

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Array, String and Structure Representations

Learning Objectives (2.4)

1. Describe the representation of arrays in memory.
2. Describe the C and Pascal strings storage in memory.
3. Describe the representation of structures in memory

©2020, SCSE/NTU

27

CZ1106
CE1106

Array Representation

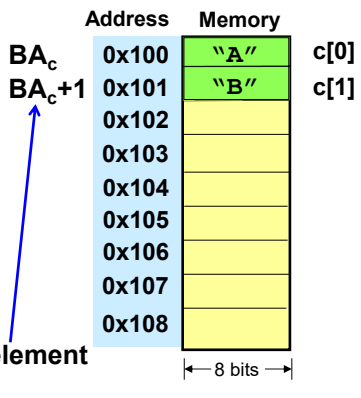
- A linear array is a consecutive area in memory storing a series of homogenous data type.
- Elements of an array are accessed through appropriate offset from the **base address** (BA) of the array.

- We only know base address
- need to know size of array

↳ same

```
main()
{
  char c[2];      //2 element char array c
  c[0]="A";       //assign "A" to 1st element
  c[1]="B";       //assign "B" to 2nd element

}
```



Offset from base address (BA$_c$) to access each element of the array c

©2020, SCSE/NTU

28

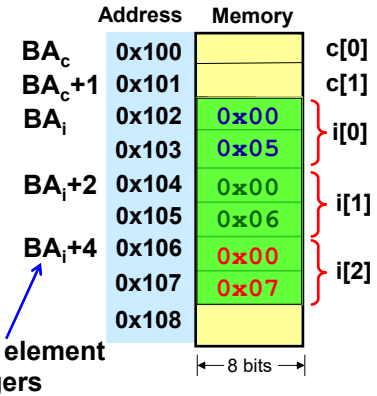
CZ1106 CE1106

Array Representation (cont)

- Computation of the address of array element must take into account its data type size.
- Address is computed by multiplying element number by size of data type before addition to the base address (BA).

```
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 (n × 2) from base address (BA_i) to access each element n of the array `i` consisting of 2 byte-sized short integers

©2020 SCSE/NTU

29

CZ1106 CE1106

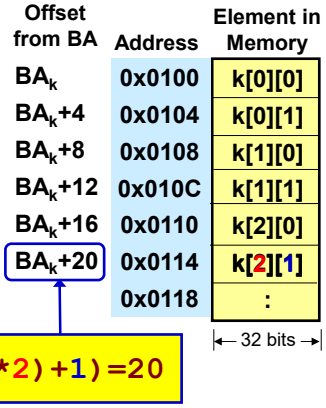
Nested Array

- 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 then created.

↳ 2 dimension array

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

Still only 1 base address



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

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

datatype column

4*((2*2)+1)=20

©2020 SCSE/NTU

30

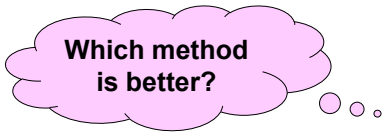
CZ1106 CE1106

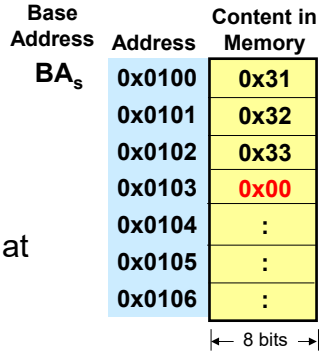
String Representation

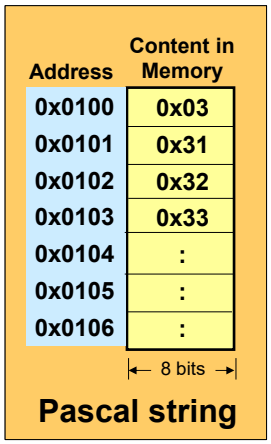
- A string in a C program is an array of characters terminated by a null (`0x00`) character.

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

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







Learn more: Google Search "C strings vs Pascal strings"

31

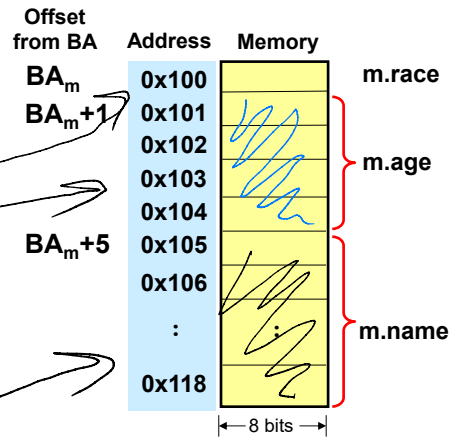
CZ1106 CE1106

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;
}
```

must declare structure to memory variable or else no memory will be allocated



©2020 SCSE/NTU

32

CZ1106
CE1106

Summary

- Arrays stores a series of similar data types in consecutive memory locations.
  - Elements are accesses using offsets from a base address.
  - Offset computation must take into account size of data type.
- Structures stores a series of dissimilar data types in consecutive memory locations.
  - Memory space for structure data type is only allocated when structure variables are declared.

©2020, SCSE/NTU

33

CZ1106
CE1106

Quiz: Accessing Array Elements

- What is the start address of the array variable `A[3]` if the base address of the array `A` is hexadecimal `0x100`?

```
        :
long int A[5];
A[3] = 1;
```

Since `long int` is 4 bytes, the start addresses of `A[3]` should be:
BA+(3x4) = `0x100` + `0x00C` = `0x10C`
Note: 12 bytes from `0x100` is `0x10C` (not `0x112`)
(addresses use concise hexadecimal notation)

`0x112` `0x10C` `0x103` `0x100`

a c d

34

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Pointer Representation

Learning Objectives (2.5)

1. Describe the representation of pointers in memory.

©2020 SCSE/NTU

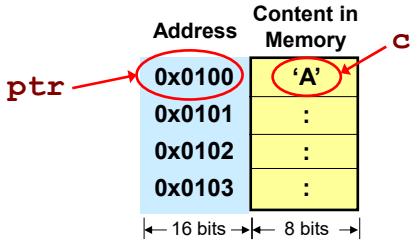
35

CZ1106
CE1106

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.
- The value of a pointer is an **address**. Its **size** is **fixed** (regardless of data type).
- The size of the pointer depends on the processor's address range.

```
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 CPU are specified using 16 bits (2 bytes).

©2020 SCSE/NTU

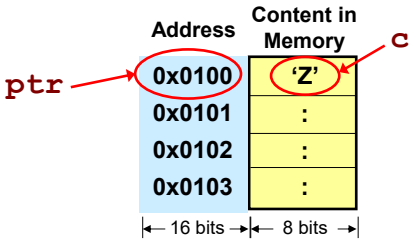
36

CZ1106 CE1106

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 an 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'
```



©2020 SCSE/NTU

37

CZ1106 CE1106

Summary

- Pointer is a variable whose value is an address.
- The size of a pointer variable is independent of data type it is pointing to.
- The size of a pointer is dependent on the addressable memory space of the processor.

©2020 SCSE/NTU

38

CZ1106 CE1106

Quiz: Memory Allocation

- A C compiler is compiling for a CPU with a 32-bit address bus. How many bytes in memory will it allocate for the variable declarations shown?

```
        :
short int I;  2 bytes
short int *P; 4 bytes
```

The **short int** data type is 2 bytes. A pointer data type represents an address and will require a memory size that matches the number of bits needed to specify a unique address (i.e. 32 bits or 4 bytes). Therefore the total memory allocated by the C compiler will be **2 + 4**, which is **6** bytes.

2 bytes **4 bytes** **6 bytes** **8 bytes**

a b c d

CZ1106
CE1106

Chapter 2

Data Organisation in Memory

Data Alignment

Learning Objectives (2.6)

1. Describe the concept of data alignment.
2. Describe data alignment considerations for efficient access and storage of structure variables in memory.

©2020, SCSE/NTU

39

CZ1106
CE1106

Data Alignment

- Most computer systems have restrictions on the allowable address for accessing various 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

← no data alignment issue

↖ in hexadecimal e.g. this is 16 in decimal

©2020, SCSE/NTU

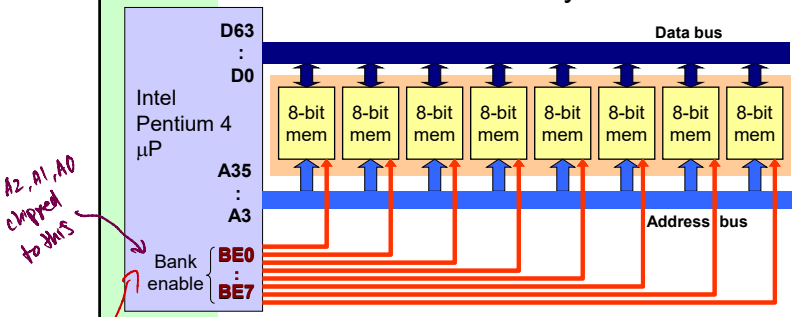
40

CZ1106 CE1106

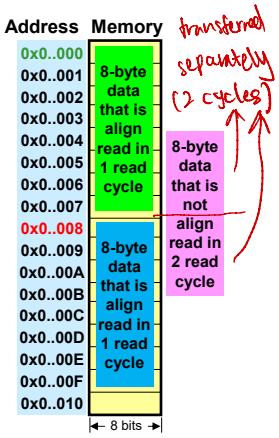
Memory Interfacing & Data Alignment

(Case Study: Intel Pentium 4)

- Main memory consist of multiple 8-bit memory modules required to make up 64-bit data word size of processor.
- Data width is selected by additional control lines (**BE0-BE7**).



- Only one address pattern can exist on address bus during data transfer. 8-byte data needs **two memory cycles** if not aligned to 8-byte address boundary.



Learn More: Google "memory and data alignment"

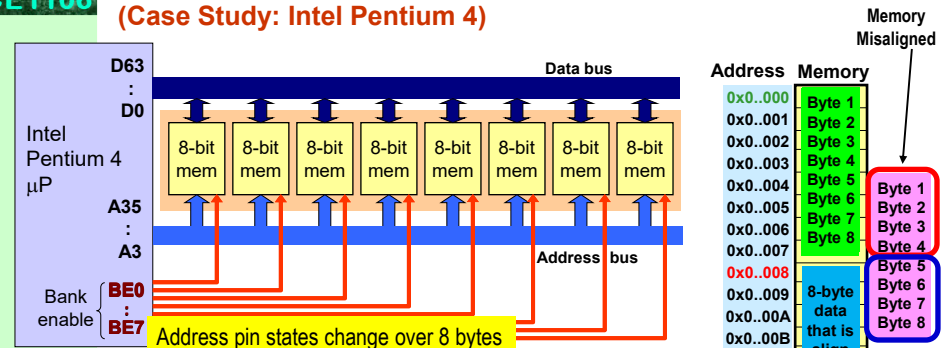
©2020, SCSE/NTU

41

CZ1106 CE1106

Address Mapping of 64-bit data

(Case Study: Intel Pentium 4)



Memory Align

Address Pins	A_{35}	A_{34}	.....	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_{00}
Byte 1	0	0	.....	0	0	0	0	1	0	0
Byte 2	0	0	.....	0	0	0	0	1	0	1
Byte 3	0	0	.....	0	0	0	0	1	1	0
Byte 4	0	0	.....	0	0	0	0	1	1	1
Byte 5	0	0	.....	0	0	0	1	0	0	0
Byte 6	0	0	.....	0	0	0	1	0	0	1
Byte 7	0	0	.....	0	0	0	1	0	1	0
Byte 8	0	0	.....	0	0	0	1	0	1	1

Note:
Address pin A_{03} cannot be in states 0 and 1 simultaneously to transfer a mis-aligned word in a single memory cycle.

Note: Lines BE0-BE7 implements address bits A_{02} to A_{00} and determine which byte(s) in the 8 possible addresses are accessed during memory transfer.

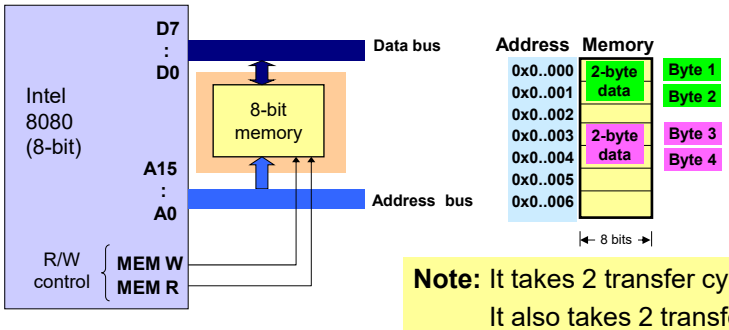
42

CZ1106 CE1106

Data Alignment and Data Bus Width

(Case Study: An 8-bit Processor - Intel 8080)

- The extent of data alignment is dependent on the width of the processor's data bus.
- A processor with an **8-bit** data bus does not have any data alignment issues. Multi-byte data can be fetched from **any address**.
- A processor with a **16-bit** (i.e. 2 bytes) data bus only needs to align 2, 4 or 8 byte-sized data to **even addresses** (e.g. `0x0000`, `0x0002`, `0x0004`, etc).



Note: It takes 2 transfer cycles to read Bytes 1 and 2
It also takes 2 transfer cycles to read Bytes 3 and 4

©2020 SCSE/NTU

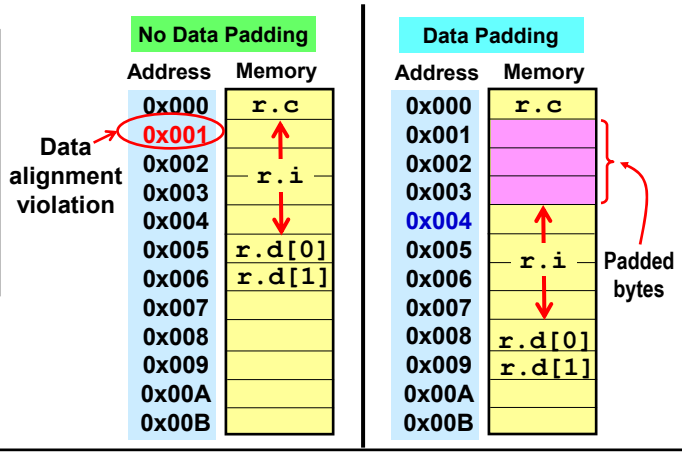
43

CZ1106 CE1106

Data Alignment with Structures

- **Data padding** (addition of meaningless bytes) is used by compilers to ensure alignment of different data types within a structure.

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



©2020 SCSE/NTU

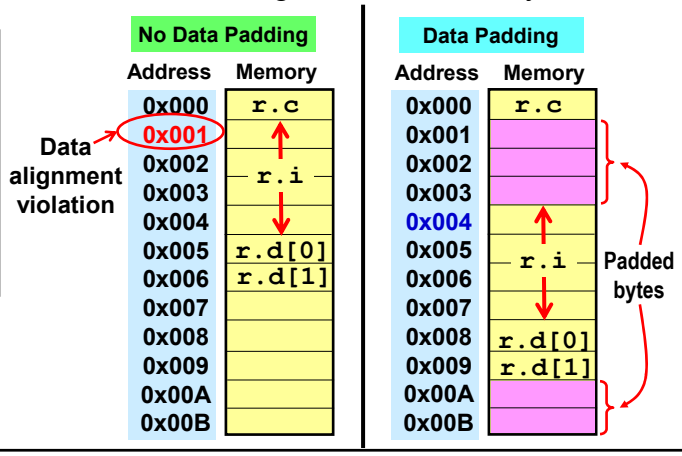
44

CZ1106 CE1106

Data Alignment with Structures

- **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 structure element and the start of the next to maintain alignment in an array of structures.

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



©2020 SCSE/NTU

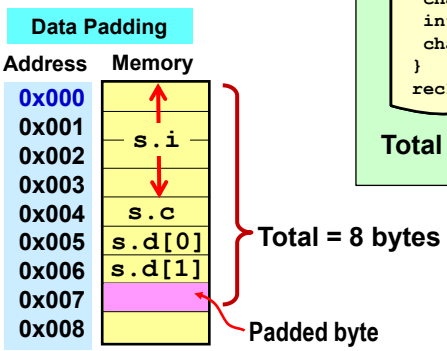
45

CZ1106 CE1106

Data Alignment with Structures (cont)

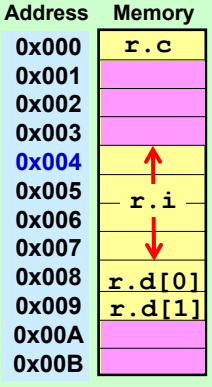
- Structures should be constructed with data alignment constraints in mind.
- **Rearrange the order** of data objects in the structure based on their size to minimize data padding where possible.

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




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

Total = 12 bytes



Before re-arranging structure (previous)

Learn More: Google "Structure and Padding in C"

©2020 SCSE/NTU

46

Continue from Slide 45

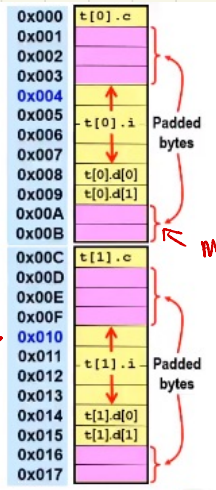
CZ1106
CE1106

Data Type	Size (Byte)	Example of allowable start addresses due to alignment
int	4	0x..000, 0x..004, 0x..008, 0x..00C, 0x..010, 0x..014, 0x..018, 0x..01C,

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

An array of rec1

proper data alignment



ent and the
es.

must have or will have data alignment violation

©2020, SCSE/NTU

CZ1106
CE1106

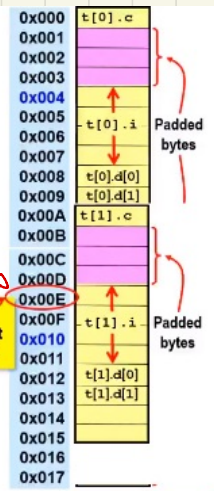
Data Type	Size (Byte)	Example of allowable start addresses due to alignment
int	4	0x..000, 0x..004, 0x..008, 0x..00C, 0x..010, 0x..014, 0x..018, 0x..01C,

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

An array of rec1

not allowed

Data alignment violation



ent and the
es.

©2020, SCSE/NTU

Continue from Side 46

CZ1106 CE1106

Quiz: Data Alignment

- How many bytes will be allocated for variables `S1` and `S2`? Assume multi-byte data requires even address data alignment and memory allocation starts at address `0x000`.

```
struct rec {
  char C;
  short int X;
}
rec S1,S2;
```

switching these will still result in padding

Address	Memory
0x000	S1.C
0x001	padding
0x002	S1.X
0x003	
0x004	S2.C
0x005	padding
0x006	S2.X
0x007	
0x008	

8 bytes

2 bytes (a) 4 bytes (b) 6 bytes (c) 8 bytes (C)

CZ1106 CE1106

Quiz: Data Alignment

- How many bytes will be allocated for variables `S1` and `S2`? Assume multi-byte data requires even address data alignment and memory allocation starts at address `0x000`.

```
struct rec {
  char C;
  short int X;
}
rec S1,S2;
```

Character `C` of `S1` occupies byte at `0x000`. But next available memory location `0x001` is an odd address, a padded byte will be added to ensure the two-byte variable `X` starts at as even address `0x002`. Structure `S1` takes 4 bytes. The same is repeated for `S2` starting at `0x004`. Total allocation is 8 bytes.

2 bytes (a) 4 bytes (b) 6 bytes (c) 8 bytes (C)

©2020, SCSE/NTU

CZ1106
CE1106

Summary

- Data alignment restricts where in memory multi-byte data can be stored.
  - Data must be aligned to addresses that are multiple of the size of their data type.
  - The extent of the data alignment restriction depends on the size of the processor's data bus.
- Structure data types should consider data alignment issues.
  - Data padding is used to meet alignment restrictions.

©2020, SCSE/NTU

47

Strings are considered characters or byte sized values

⇒ even if you have a string of 4 bytes, it is still treated as individual bytes ⇒ use instructions to fetch bytes

⇒ fetching multi byte values like 2, 4 bytes
↳ have to take care of alignment