Embedded System (ES)

Lecturer: Dr. Cheng-Kai Lu
Phone: (02)7749-3554
Office: TD302/BAIR Lab
Email: cklu@ntnu.edu.tw

國立臺灣師範大學
NATIONAL TAIWAN NORMAL UNIVERSITY

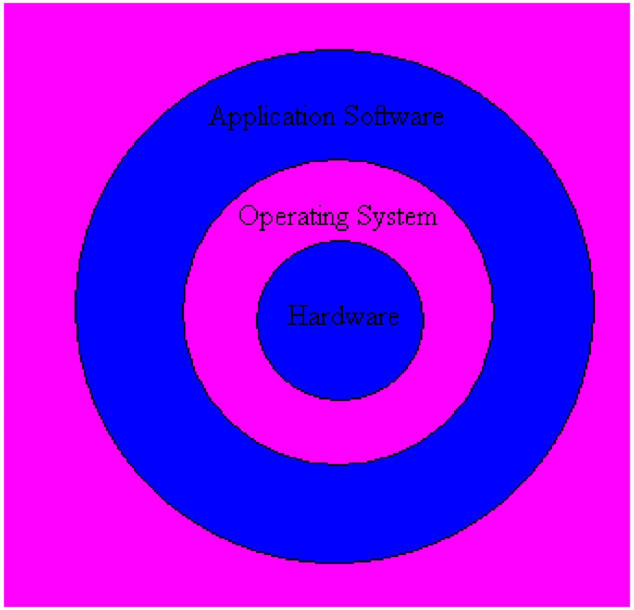
Outline

In this lecture, we will cover:

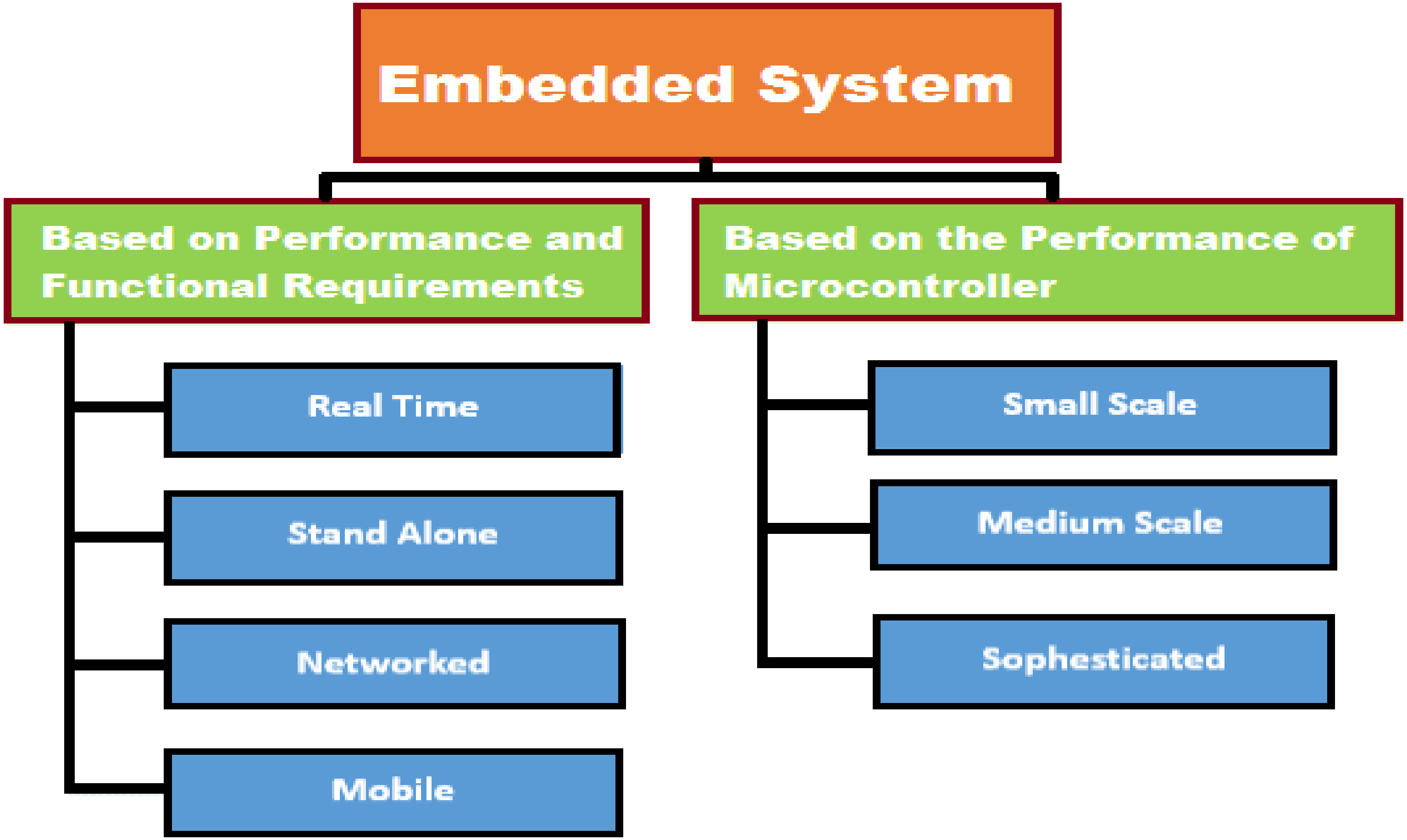
- Review on the **important points** which were covered in the previous lecture
- Pointers
- Microprocessors/Microcontrollers

Embedded Systems Overview - Recap

- An embedded system
  - employs a combination of hardware & software (a "computational engine") to perform a specific function;
  - is part of a larger system that may not be a "computer";
  - works in a reactive and time-constrained environment.
- Software is used for providing features and flexibility
- Hardware = {Processors, ASICs, Memory,...} is used for performance (& sometimes security)



Classification of ES- Recap



Methods for Representing Data - Recap

- Three of the most common forms of notation
  - Decimal (base 10) 0123456789
  - Hexadecimal (base 16) 0123456789ABCDEF
  - Binary (base 2) 01
- Converting between forms
  - When converting binary to hexadecimal, every group of 4 bits (nibble) represents a hexadecimal digit
  - Examples:

Binary	Hexadecimal
0010	2
0100	4
1010	A

Base in C - Recap

- Syntax in C
  - Computers understand binary
  - The following lines of code are all the same (the complier does not care what base the programmer uses):

```
char x = 2 + 1;
char x = 0b10 + 1;
char x = 0x2 + 1;
char x = 0x02 + 0x01;
```

C for Embedded System- Recap

- Variables
- Arrays
- Strings

Primitive Types and Sizes - Recap

Name	Number of Bytes sizeof()	Range
char	1	0 to 255 or -128 to 127 (Depends on Compiler settings)
signed char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
int	Varies by platform	Varies by platform
int (on TM4C123)	4	-2,147,483,648 to 2,147,483,647
unsigned int (on TM4C123)	4	0 to 4,294,967,295
(pointer)	Varies by platform	Varies by platform
(pointer on TM4C123)	4	Address Space

- Primitive types in C: char, short, int, long, float, double default modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, depends on Compiler settings

Primitive Types and Sizes - Recap

Name	Number of Bytes sizeof()	Range
long	4	-2,147,483,648 to 2,147,483,647
signed long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
long long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	±1.175e-38 to ±3.402e38
double	Varies by platform	
double (on TM4C123)	8	±2.3E-308 to ±1.7E+308

- Primitive types in C: char, short, int, long, float, double default modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, depends on **Compiler settings**

Character Strings in C

- Examples:

```
char myword1[6] = “Helloo”;   //  declare and initialize
char myword2[4]   = “288”;  //  declare and initialize
```

- When defining an array, the array name is the address in memory for the **first** element of the array

Note: myword1[6] does not give room for the NULL byte.

Memory Address	DF00	DF01	DF02	DF03	DF04	DF05	DF06	DF07	DF08	DF09
	myword1						myword2			
Value	‘H’	‘e’	‘l’	‘l’	‘o’	‘o’	‘2’	‘8’	‘8’	‘\0’
Array Index	0	1	2	3	4	5	0	1	2	3

Arrays - Recap

- Be careful of boundaries in C
  - No guard to prevent you from accessing beyond array end
  - **Write beyond array => Potential for disaster**
- No built-in mechanism for copying arrays
- An escape sequence begins with a backslash character (\) to prevent confusion for the compiler.

Structure and Union - Recap

Use of union inside of a struct

```
struct {
  char *name;       4
  int flags;        4
  short s_type;     2
  union {
    short val;      2
    float fval;     4 //largest member of union u
    char  cval;     1
  } u; //largest member defines a union's size
} symtab;
      Just sum the size of each struct member.
      symtab size is: 4+4+2+4 = 14 bytes
```

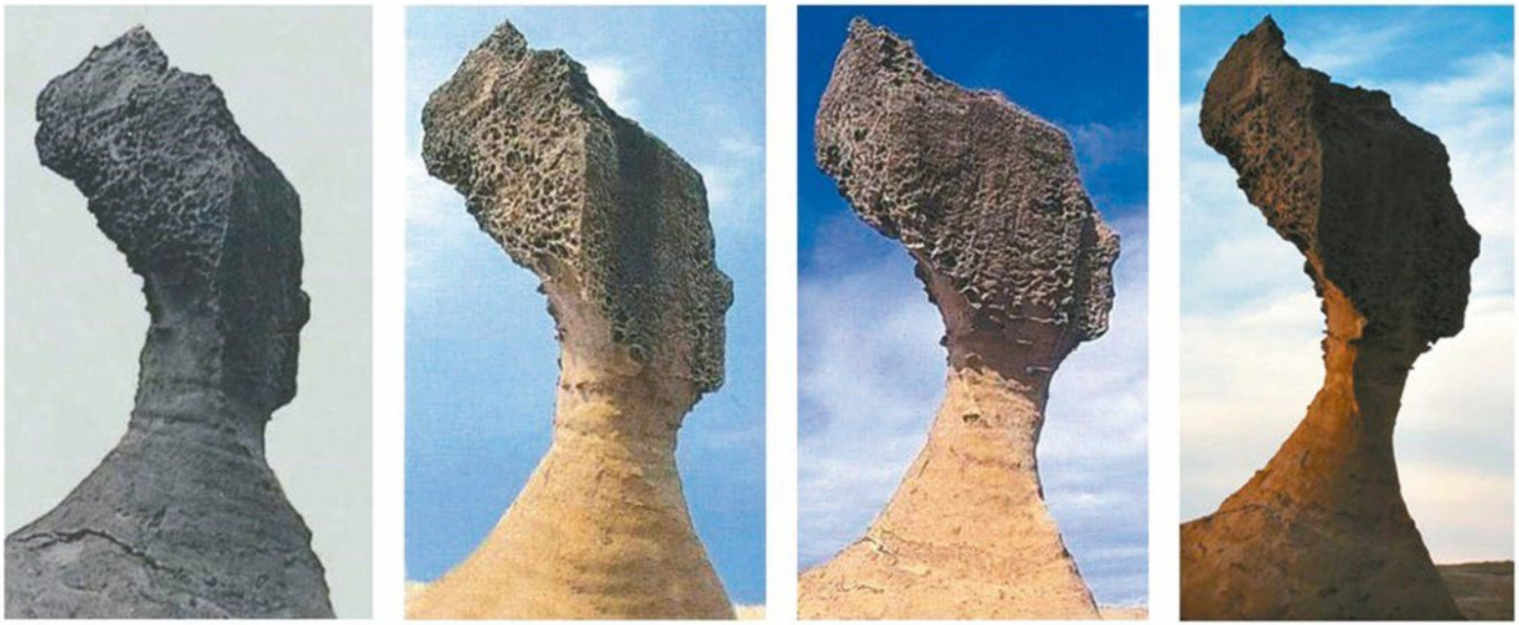
POINTERS

Pointers

What is a pointer:

- A **variable** whose value is interpreted as a location in memory.
- A variable that can be dereferenced (i.e. you can go to the place in memory indicated by the pointer's value).

Weathering and Erosion



1969 年　1980 年　1990 年　2010 年

Pointers: All the same size

Queen's Head

20744 (Yehliu)

Address same size (5 digits)

College of Technology and Engineering Building

10644 (Taipei City)

Pointers

- You should understand these basic operations:
  1. Set the value of a pointer to the address of a variable
  2. Dereference a pointer
  3. Set or Read the value of a dereferenced pointer
  4. Increment a pointer

- Pointers are declared using the * character

```
Examples:
int* ptr1;           // pointer to type int
int *ptr2;           // alternative declaration
char* ptr3;          // pointer to type char
int** ptr4;          // pointer to an int pointer
```

Why Does This Matter in Embedded Systems?

- Memory Allocation Awareness: In low-level programming (such as writing firmware), knowing where variables are stored is crucial.
- Endianness: The byte order of short i = 5; might appear differently depending on whether the system is Big-Endian or Little-Endian.
- Pointer Manipulation: This is essential for direct memory access (DMA) and peripheral register access.

*Big-Endian and Little-Endian are two different ways of storing multi-byte data in memory.

- Big-Endian: The most significant byte (MSB) is stored at the lowest memory address.
- Little-Endian: The least significant byte (LSB) is stored at the lowest memory address.

Importance of Endianness in Embedded Systems

- **Endianness** determines how multi-byte data is stored in memory.
- Many **microcontrollers** and **processors** use different endianness.
  - ARM Cortex-M processors default to Little-Endian.
  - Some DSPs (Digital Signal Processors) and PowerPC architectures use Big-Endian.
- Peripheral devices (e.g., sensors, network interfaces) may send data in a different format.
  - Network Byte Order (Big-Endian): Used in Ethernet, TCP/IP, and CAN bus.

❑ IPv4 address 192.168.1.1 (0xC0A80101)Stored in memory as: C0 A8 01 01 (Big-Endian) vs. 01 01 A8 C0 (Little-Endian)

- Incorrect handling leads to **data misinterpretation.**

Handling Endianness in Embedded Systems

Issue	Impact	Solution
Incorrect byte order	Wrong sensor readings or corrupted data	Use byte swapping functions (`__builtin_bswap32`)
Mixed-endian systems	Communication failure	Always check MCU documentation
External peripherals	Registers may have fixed endianness	Use structured data access with shifts & masks

Pointers

- Setting the pointer to the address of a variable
  - & is the address operator
  - **&myVariable** is the address of **myVariable**

	Address	Value
	0xFFFF_FFFF	**0x00**
i	**0xFFFF_FFFE**	**0x05**
	0xFFFF_FFFD	
	0xFFFF_FFFC	
	0xFFFF_FFFB	
ip	**0xFFFF_FFFA**	

```
short i = 5;
short* ip = &i;
```

Pointers

- Setting the pointer to the address of a variable
  - & is the address operator
  - **&Variable** is the address of **Variable**

	Address	Value
	0xFFFF_FFFF	0x00
i	0xFFFF_FFFE	0x05
	0xFFFF_FFFD	0xFF
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
ip	0xFFFF_FFFA	0xFE

```
short i = 5;
short* ip = &i;
```

Pointers

- To dereference a pointer, use the * operator before the pointer's variable name
- Dereference means to "go to" the location in Memory

	Address	Value
	0xFFFF_FFFF	0x00
i	0xFFFF_FFFE	0x05
	0xFFFF_FFFD	0xFF
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
ip	0xFFFF_FFFA	0xFE
	0xFFFF_FFF9	0x00
x	0xFFFF_FFF8	0x05

```
short i = 5;
short* ip = &i;
short x = *ip;
// x == i == 5
```

Pointers

- To set the value of Memory after dereferencing a pointer
- Means "go to" the location indicated by the pointer variable, and place a value at that location.

	Address	Value
	0xFFFF_FFFF	0x00
i	0xFFFF_FFFE	0x05 0x07
	0xFFFF_FFFD	0xFF
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
ip	0xFFFF_FFFA	0xFE
	0xFFFF_FFF9	
	0xFFFF_FFF8	

```
short i = 5;
short* ip = &i;
*ip = 7;
```

Pointers

- **WARNING!** A * operator is used for both declaring and for dereferencing a pointer.

```
int i = 5;
int *ip = &i; // declare
*ip = 7;        // dereference
```

國立臺灣師範大學
NATIONAL TAIWAN NORMAL UNIVERSITY

Pointers

- Pointer Reassignment: A pointer can change which variable it points to.
- Dereferencing a Pointer: Modifying the value of the variable a pointer points to.
- Multiple Pointers to the Same Object: Different pointers can reference the same variable.

	Address	Value
	0xFFFF_FFFF	0x00
i	0xFFFF_FFFE	0x07
	0xFFFF_FFFD	0xFF
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
ip	0xFFFF_FFFA	0xF8
	0xFFFF_FFF9	0x00
j	0xFFFF_FFF8	0x03

```
short i = 5;
short* ip = &i;
// Pointer 'ip' initially points to 'i'
*ip = 7;
// 'i' is now modified to 7 through 'ip'
short j = 3;
ip = &j; // Now 'ip' points to 'j'
```

Key Takeaways

- **Pointers Can Be Reassigned**
  - Initially, ip points to i, but later it is reassigned to j.
- **Dereferencing a Pointer Affects the Variable It Points To**
  -*ip = 7; modifies i because ip was pointing to i at that moment.
- **Multiple Pointers Can Point to the Same Object**
  - We could have another pointer short* p2 = &i; while ip was still pointing to i.

- **Practical Implications in Embedded Systems**

❑ **Modifying Registers**

- Microcontrollers often access hardware registers using pointers.

e.g.

```
volatile uint16_t *gpio = (uint16_t *)0x40021000;
*gpio = 0x01;  // Set GPIO pin using a pointer
```

❑ **Dynamic Memory Allocation**

-When allocating memory dynamically, pointers can be reassigned to manage different parts of memory.

❑ **Linked Lists or Data Structures**

- Pointers are frequently used to build and traverse linked lists, trees, etc.

Pointer incrementing and decrementing

- Incrementing (++) and decrementing (--)a pointer
  - Increments/decrements by the size of the **"sub-type"**
- Example:
  - int* increment by 4 (ints are 4 bytes)
  - char* increment by 1 (chars are 1 byte)

```
int* ip = 0x1000;      // sizeof(int) == 4
char* cp = 0x1000;     // sizeof(char) == 1
        Ip++;
        cp++;
// ip == 0x1004 and cp = 0x1001
```

	Address	Value
	0xFFFF_FFFF	0x00
	0xFFFF_FFFE	0x00
	0xFFFF_FFFD	0x10
ip	0xFFFF_FFFC	0x04
	0xFFFF_FFFB	0x00
	0xFFFF_FFFA	0x00
	0xFFFF_FFF9	0x10
cp	0xFFFF_FFF8	0x01

Primitive Types and Sizes

Name	Number of Bytes sizeof()	Range
char	1	0 to 255 or -128 to 127 (Depends on Compiler settings)
signed char	1	-128 to 127
unsigned char	1	0 to 255
short	2	-32,768 to 32,767
unsigned short	2	0 to 65,535
int	Varies by platform	Varies by platform
int (on TM4C123)	4	-2,147,483,648 to 2,147,483,647
unsigned int (on TM4C123)	4	0 to 4,294,967,295
(pointer)	Varies by platform	Varies by platform
(pointer on TM4C123)	4	Address Space

- Primitive types in C: char, short, int, long, float, double **default** modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, it depends on Compiler settings

Primitive Types and Sizes

Name	Number of Bytes sizeof()	Range
long	4	-2,147,483,648 to 2,147,483,647
signed long	4	-2,147,483,648 to 2,147,483,647
unsigned long	4	0 to 4,294,967,295
long long	8	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	4	±1.175e-38 to ±3.402e38
double	Varies by platform	
double (on TM4C123)	8	±2.3E-308 to ±1.7E+308

- Primitive types in C: char, short, int, long, float, double default modifier on primitive types is **signed** (not unsigned)
- Note: char does not have a standard default, it depends on **Compiler settings**

Pointers Arithmetic

```
typedef struct {
   char x;
   char y;
} coord_t;

coord_t* coord_ptr;
coord_t c_array[4];

coord_ptr = c_array;
```

Location	Variable Name	Value
0xFFFF_FFFF	**coord_ptr**	
0xFFFF_FFFE		
0xFFFF_FFFD		
0xFFFF_FFFC		
0xFFFF_FFFB	**c_array[3].y**	
0xFFFF_FFFA	**c_array[3].x**	
0xFFFF_FFF9	**c_array[2].y**	
0xFFFF_FFF8	**c_array[2].x**	
0xFFFF_FFF7	**c_array[1].y**	
0xFFFF_FFF6	**c_array[1].x**	
0xFFFF_FFF5	**c_array[0].y**	
0xFFFF_FFF4	**c_array[0].x**	
0xFFFF_FFF3		
0xFFFF_FFF2		
0xFFFF_FFF1		

Pointers Arithmetic

```
typedef struct {
   char x;
   char y;
} coord_t;

coord_t* coord_ptr;
coord_t c_array[4];

coord_ptr = c_array;
```

Location	Variable Name	Value
0xFFFF_FFFF	coord_ptr	0xFF
0xFFFF_FFFE		0xFF
0xFFFF_FFFD		0xFF
0xFFFF_FFFC		0xF4
0xFFFF_FFFB	c_array[3].y	
0xFFFF_FFFA	c_array[3].x	
0xFFFF_FFF9	c_array[2].y	
0xFFFF_FFF8	c_array[2].x	
0xFFFF_FFF7	c_array[1].y	
0xFFFF_FFF6	c_array[1].x	
0xFFFF_FFF5	c_array[0].y	
0xFFFF_FFF4	c_array[0].x	
0xFFFF_FFF3		
0xFFFF_FFF2		
0xFFFF_FFF1		

Pointers

- Pointers are useful for passing parameters to a function.
  - Especially useful when a variable consume lots of memory

Pass by Reference Example

```
void addThree(short *ptr)
{
  *ptr = *ptr + 3;
}

void main()
{ short x = 5; addThree(&x); // x is now 8
}
```

	Address	Value
	0xFFFF_FFFF	0x00
x	0xFFFF_FFFE	0x08
	0xFFFF_FFFD	0xFF
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
ptr	0xFFFF_FFFA	0xFE
	0xFFFF_FFF9	
	0xFFFF_FFF8	

Example

```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
```

```
*p1 = 20;
*p2 = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;

p3 = &p2;
*p3 = &r;
```

	Address	Value
r	0xFFFF_FFFF	0x0A
s	0xFFFF_FFFE	0x0F
t	0xFFFF_FFFD	0x0D
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
	0xFFFF_FFFA	0xFF
p1	0xFFFF_FFF9	0xFE
	0xFFFF_FFF8	0xFF
	0xFFFF_FFF7	0xFF
	0xFFFF_FFF6	0xFF
p2	0xFFFF_FFF5	0xFD
	0xFFFF_FFF4	0xFF
	0xFFFF_FFF3	0xFF
	0xFFFF_FFF2	0xFF
p3	0xFFFF_FFF1	0xF9

Example

```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
```

```
*p1 = 20;     // s = 20;
*p2 = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;

p3 = &p2;
*p3 = &r;
```

	Address	Value
r	0xFFFF_FFFF	0x0A
s	0xFFFF_FFFE	0x14
t	0xFFFF_FFFD	0x0D
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
	0xFFFF_FFFA	0xFF
p1	0xFFFF_FFF9	0xFE
	0xFFFF_FFF8	0xFF
	0xFFFF_FFF7	0xFF
	0xFFFF_FFF6	0xFF
p2	0xFFFF_FFF5	0xFD
	0xFFFF_FFF4	0xFF
	0xFFFF_FFF3	0xFF
	0xFFFF_FFF2	0xFF
p3	0xFFFF_FFF1	0xF9

Example

```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
```

```
*p1 = 20;     // s = 20;
*p2 = 30;     // t = 30;
**p3 = 40;
*p3 = &t;
**p3 = 50;

p3 = &p2;
*p3 = &r;
```

	Address	Value
r	0xFFFF_FFFF	0x0A
s	0xFFFF_FFFE	0x14
t	0xFFFF_FFFD	0x1E
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
	0xFFFF_FFFA	0xFF
p1	0xFFFF_FFF9	0xFE
	0xFFFF_FFF8	0xFF
	0xFFFF_FFF7	0xFF
	0xFFFF_FFF6	0xFF
p2	0xFFFF_FFF5	0xFD
	0xFFFF_FFF4	0xFF
	0xFFFF_FFF3	0xFF
	0xFFFF_FFF2	0xFF
p3	0xFFFF_FFF1	0xF9

Example

```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
```

```
*p1 = 20;     // s = 20;
*p2 = 30;     // t = 30;
**p3 = 40;    // s = 40;
*p3 = &t;
**p3 = 50;

p3 = &p2;
*p3 = &r;
```

	Address	Value
r	0xFFFF_FFFF	0x0A
s	0xFFFF_FFFE	**0x28**
t	0xFFFF_FFFD	0x1E
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
	0xFFFF_FFFA	0xFF
p1	0xFFFF_FFF9	0xFE
	0xFFFF_FFF8	0xFF
	0xFFFF_FFF7	0xFF
	0xFFFF_FFF6	0xFF
p2	0xFFFF_FFF5	0xFD
	0xFFFF_FFF4	0xFF
	0xFFFF_FFF3	0xFF
	0xFFFF_FFF2	0xFF
p3	0xFFFF_FFF1	0xF9

Example

```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
```

```
*p1 = 20;     // s = 20;
*p2 = 30;     // t = 30;
**p3 = 40;    // s = 40;
*p3 = &t;     // p1 = &t;
**p3 = 50;

p3 = &p2;
*p3 = &r;
```

	Address	Value
r	0xFFFF_FFFF	0x0A
s	0xFFFF_FFFE	0x28
t	0xFFFF_FFFD	0x1E
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
	0xFFFF_FFFA	0xFF
p1	0xFFFF_FFF9	0xFD
	0xFFFF_FFF8	0xFF
	0xFFFF_FFF7	0xFF
	0xFFFF_FFF6	0xFF
p2	0xFFFF_FFF5	0xFD
	0xFFFF_FFF4	0xFF
	0xFFFF_FFF3	0xFF
	0xFFFF_FFF2	0xFF
p3	0xFFFF_FFF1	0xF9

Example

```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
```

```
*p1 = 20;     // s = 20;
*p2 = 30;     // t = 30;
**p3 = 40;    // s = 40;
*p3 = &t;     // p1 = &t;
**p3 = 50;    // t = 50;

p3 = &p2;
*p3 = &r;
```

	Address	Value
r	0xFFFF_FFFF	0x0A
s	0xFFFF_FFFE	0x28
t	0xFFFF_FFFD	0x32
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
	0xFFFF_FFFA	0xFF
p1	0xFFFF_FFF9	0xFD
	0xFFFF_FFF8	0xFF
	0xFFFF_FFF7	0xFF
	0xFFFF_FFF6	0xFF
p2	0xFFFF_FFF5	0xFD
	0xFFFF_FFF4	0xFF
	0xFFFF_FFF3	0xFF
	0xFFFF_FFF2	0xFF
p3	0xFFFF_FFF1	0xF9

Example

```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;

*p1 = 20;      // s = 20;
*p2 = 30;      // t = 30;
**p3 = 40;     // s = 40;
*p3 = &t;      // p1 = &t;
**p3 = 50;     // t = 50;

p3 = &p2;

*p3 = &r;
```

	Address	Value
r	0xFFFF_FFFF	0x0A
s	0xFFFF_FFFE	0x28
t	0xFFFF_FFFD	0x32
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
	0xFFFF_FFFA	0xFF
p1	0xFFFF_FFF9	0xFD
	0xFFFF_FFF8	0xFF
	0xFFFF_FFF7	0xFF
	0xFFFF_FFF6	0xFF
p2	0xFFFF_FFF5	0xFD
	0xFFFF_FFF4	0xFF
	0xFFFF_FFF3	0xFF
	0xFFFF_FFF2	0xFF
p3	0xFFFF_FFF1	0xF5

Example

```
char r = 10;
char s = 15;
char t = 13;
char *p1 = &s;
char *p2 = &t;
char **p3 = &p1;
```

```
*p1 = 20;     // s = 20;
*p2 = 30;     // t = 30;
**p3 = 40;    // s = 40;
*p3 = &t;     // p1 = &t;
**p3 = 50;    // t = 50;

p3 = &p2;

*p3 = &r;     // p2 = &r;
```

	Address	Value
r	0xFFFF_FFFF	0x0A
s	0xFFFF_FFFE	0x28
t	0xFFFF_FFFD	0x32
	0xFFFF_FFFC	0xFF
	0xFFFF_FFFB	0xFF
	0xFFFF_FFFA	0xFF
p1	0xFFFF_FFF9	0xFD
	0xFFFF_FFF8	0xFF
	0xFFFF_FFF7	0xFF
	0xFFFF_FFF6	0xFF
p2	0xFFFF_FFF5	0xFF
	0xFFFF_FFF4	0xFF
	0xFFFF_FFF3	0xFF
	0xFFFF_FFF2	0xFF
p3	0xFFFF_FFF1	0xF5

Exercise

In a typical stack memory, assume the variable addresses are assigned in stack order.

```
short x = 0x2050, y = 0x6633;
short*   p1   =   &x;
short*   p2   =   &y;
p2++;
```

After executing the above code:

x = ______________

y = ______________

p1 = ______________

p2 = ______________

Address	Value
0xFFFF_FFFF	
0xFFFF_FFFE	
0xFFFF_FFFD	
0xFFFF_FFFC	
0xFFFF_FFFB	
0xFFFF_FFFA	
0xFFFF_FFF9	
0xFFFF_FFF8	
0xFFFF_FFF7	
0xFFFF_FFF6	
0xFFFF_FFF5	
0xFFFF_FFF4	

What Happens if We Replace short with int?

In a typical stack memory, assume the variable addresses are assigned in stack order.

```
int x = 0x2050, y = 0x6633;
int* p1 = &x;
int* p2 = &y;
p2++;
```

After executing the above code:

x = ______________

y = ______________

p1 = ______________

p2 = ______________

Address	Value
0xFFFF_FFFF	
0xFFFF_FFFE	
0xFFFF_FFFD	
0xFFFF_FFFC	
0xFFFF_FFFB	
0xFFFF_FFFA	
0xFFFF_FFF9	
0xFFFF_FFF8	
0xFFFF_FFF7	
0xFFFF_FFF6	
0xFFFF_FFF5	
0xFFFF_FFF4	

Microprocessors/Microcontrollers

The Microprocessor Age: Early Microprocessors

- World's first microprocessor the Intel 4004 – designed in April 1971, released to market in 1971.
- A 4-bit microprocessor-programmable controller on a chip.
- Addressed 4096 (4KB), 4-bit-wide memory locations.
  - a **bit** is a binary digit with a value of one or zero
  - 4-bit-wide memory location often called a **nibble**
- The 4004 instruction set contained 45 instructions.

The Microprocessor Age: Early Microprocessors

- Main problems with early microprocessor were speed, word width, and memory size.
- Evolution of 4-bit microprocessor ended when Intel released the 4040, an updated 4004.
  - operated at a higher speed; lacked improvements in word width and memory size
- Texas Instruments and others also produced 4-bit microprocessors.
  - still survives in low-end applications such as microwave ovens and small control systems
  - Calculators still based on 4-bit BCD (**binary-coded decimal**) codes

The Microprocessor Age: Early Microprocessors

- With the microprocessor a commercially viable product, Intel released 8008 in 1971.
  - extended 8-bit version of 4004 microprocessor
- Addressed expanded memory of 16K bytes.
  - A **byte** is generally an 8-bit-wide binary number and a **K** is 1024.
  - memory size often specified in K bytes
- Contained additional instructions, 48 total.
- Provided opportunity for application in more advanced systems.
  - engineers developed demanding uses for 8008

The Microprocessor Age: Early Microprocessors

- Somewhat small memory size, slow speed, and instruction set limited 8008 usefulness.
- Intel introduced 8080 microprocessor in 1973.
  - first of the modern 8-bit microprocessors.
- Motorola Corporation introduced MC6800 microprocessor about six months later.
- 8080—and, to a lesser degree, the MC6800—ushered in the age of the microprocessor.
  - other companies soon introduced their own versions of the 8-bit microprocessor.

Early 8-Bit Microprocessors

Manufacturer	*Part Number*
Fairchild	F-8
Intel	8080
MOS Technology	6502
Motorola	MC6800
National Semiconductor	IMP-8
Rockwell International	PPS-8
Zilog	Z-8

Microprocessor Evolution

Data Bus width / Company	4 bit	8 bit	16 bit	32 bit	64 bit
Intel	4004 4040	8008 8080 8085	8088/6 80186 80286	80386 80486	80860 pentium
Zilog		Z80	Z8000 Z8001 Z8002		
Motorola		6800 6802 6809	68006 68008 68010	68020 68030 68040	

Microprocessors and Microcontrollers

MICROPROCESSOR: CPU built on a single chip

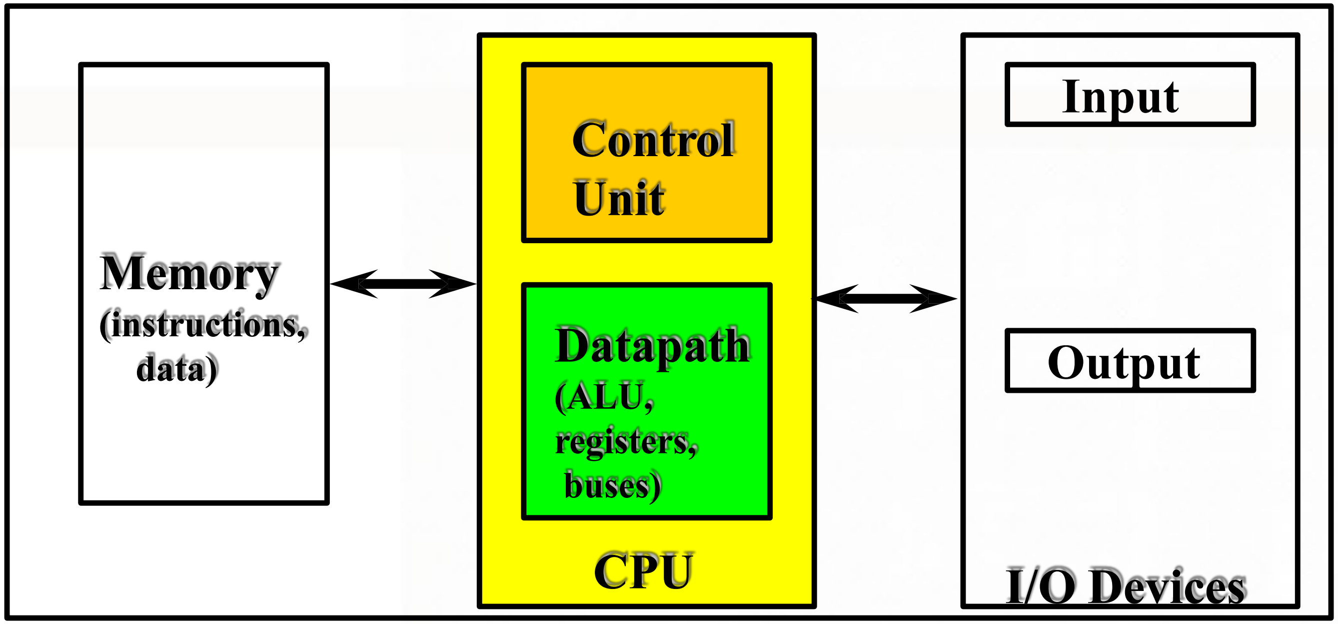
MICROCONTROLLER: Whole microprocessor system/microcomputer manufactured on a single chip - **“one chip solution”** (small, cheap, flexible, powerful)

CISC and RISC Processors

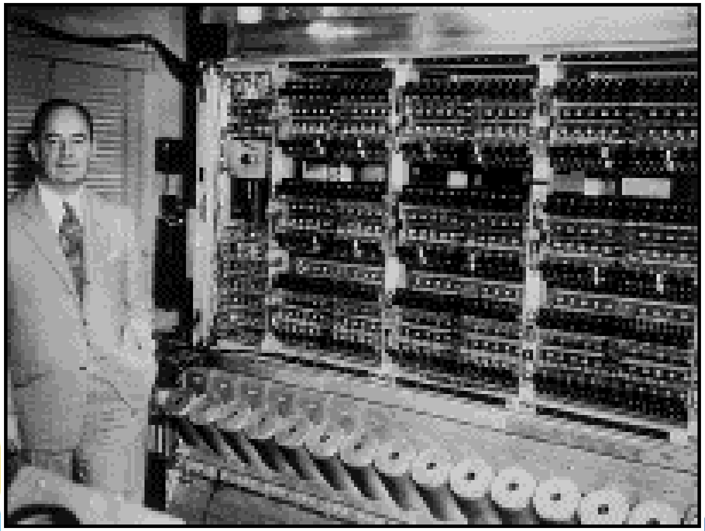
- **CISC** - microprocessors with large number of different instructions (100-250)

  more complex processor hardware, easier to program, slower, smaller and more compact programs, less memory required (Z80, Intel 8080/8085, Motorola 6800/6802/6809/68000)
- **RISC** processors (reduced instruction set) - **smaller set of instructions**, each executes **faster**, **larger memory** requirements and more **complex programs**

Generic Processor Model



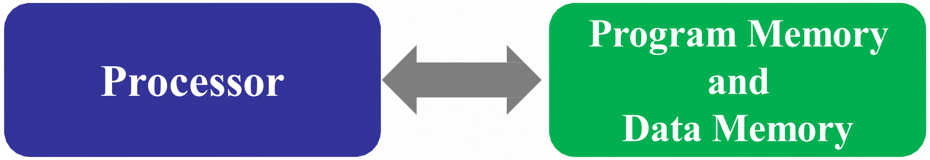
Simple Computer System utilizing CISC Processors

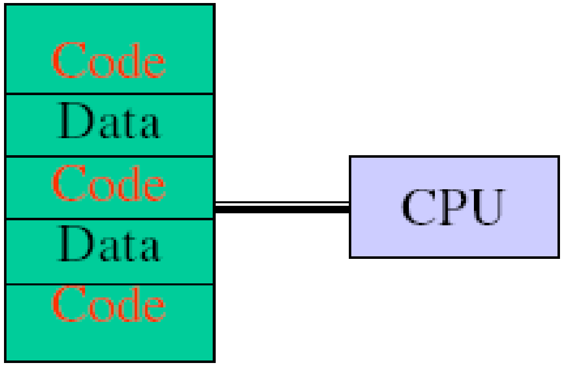


John Louis von Neumann (1903-1957)

Generic Processor Model

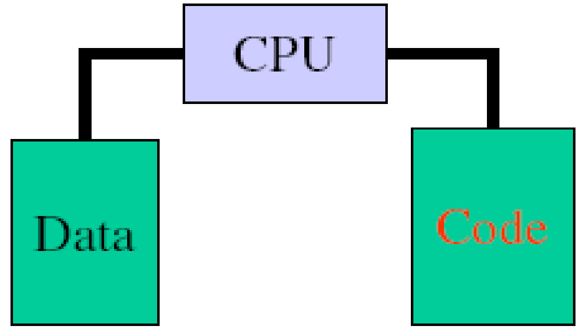
CPU: The Von-Neumann (Princeton) Computer Model utilizing CISC





Generic Processor Model

CPU: The Harvard Computer Model utilizing RISC



Central Processing Unit (CPU)

- is the heart of a computer
- performs the following tasks:
  - **Fetches instruction(s) and/or data from memory**
  - **Decodes the instruction(s)**
  - **Executes the indicated sequence of operations**
- consists of Control Unit (instruction decode, sequencing of operations), Datapath (registers, arithmetic and logic unit, buses).
- mainframe CPUs sometimes consist of several linked microchips, each performing a separate task, but most other computers require only a single microprocessor as a CPU.

CPU functional sections

(1) the **control unit** (CU)

- times and regulates all elements of the computer system;
- decodes the program instructions (such as instructions to add, move, or compare data);
- has a program counter which contains the location of the next instruction to be executed;
- has a status register which monitors the execution of instructions and keeps track of overflows, carries, borrows, etc.

CPU functional sections

(2) the **arithmetic/logic unit (ALU)**

- performs arithmetic operations (such as addition and subtraction)
- performs logic operations (such as testing a value to see if it is true or false)

as required by the instructions which are decoded by the control unit

CPU functional sections

(3) Registers

- A number of general-purpose registers accessed by instructions to store addresses of data, instruction operands (i.e., data), or the results of arithmetic calculations/logic operations (i.e., ALU results)

CPU functional sections

(4) the **internal bus**

- is a network of communication lines that links internal CPU elements;
- offers several different data paths for input from and output to other elements of the computer system.

Bus System

- The key elements of any micro computer system are connected together by a BUS. A bus consists of a set of wires carrying addresses, data and control information. Three types of Bus usually exist :-

    - **Address Bus**
    - **Data Bus**
    - **Control Bus**

Address bus

- The contents of this bus specify the address of the memory location or I/O device with which the processor wishes to communicate.

- Since address always originate from the processor the address bus is **Uni-Directional**.

Data bus

- The data between the processor and external units are transferred via the data bus.
- The data bus is **bi-directional** in nature, and its size typically lies in the range of 8 to 32 bits.

Control bus

- The control Bus carries necessary commands and other signals. The signals are primarily used to synchronise the I/O Activities.

- The control bus is **uni-directional** for an 8-bit processor, but **bi-directional** for a 16-bit processor onwards.

Memory

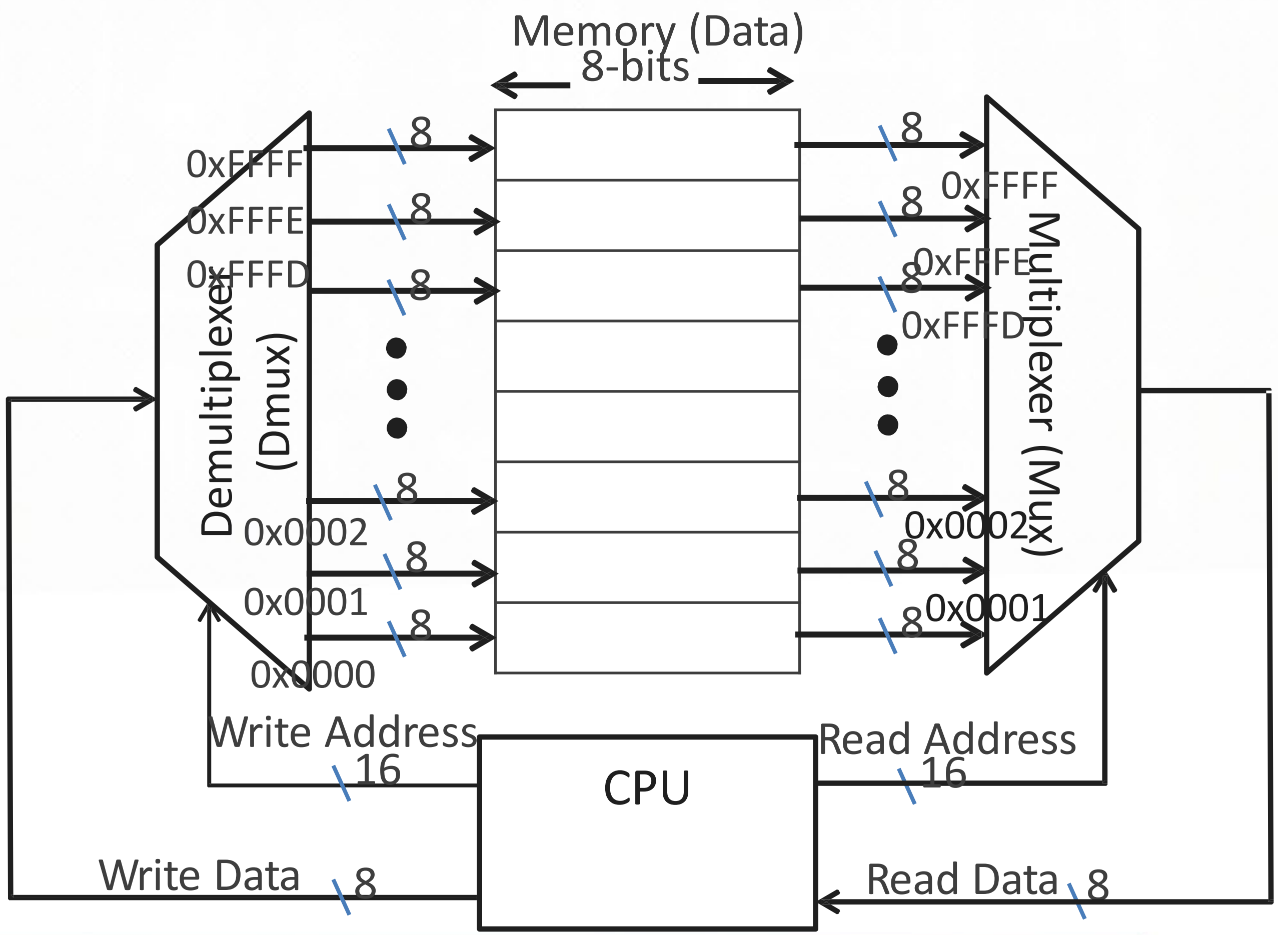
- is simply a mechanism in which information can be stored
- operations that can be performed on memory are <u>reading</u> information from it, or <u>writing</u> information to it
- Memory is used to store everything that has to be accessible to the CPU - that is:

  * instructions which are binary coded tell the computer to do something useful (e.g., <u>**add**</u> two numbers together).

  * data are binary coded information to be used by the instructions (e.g., <u>**two numbers**</u> to be added together).

CPU, Memory, and Addresses

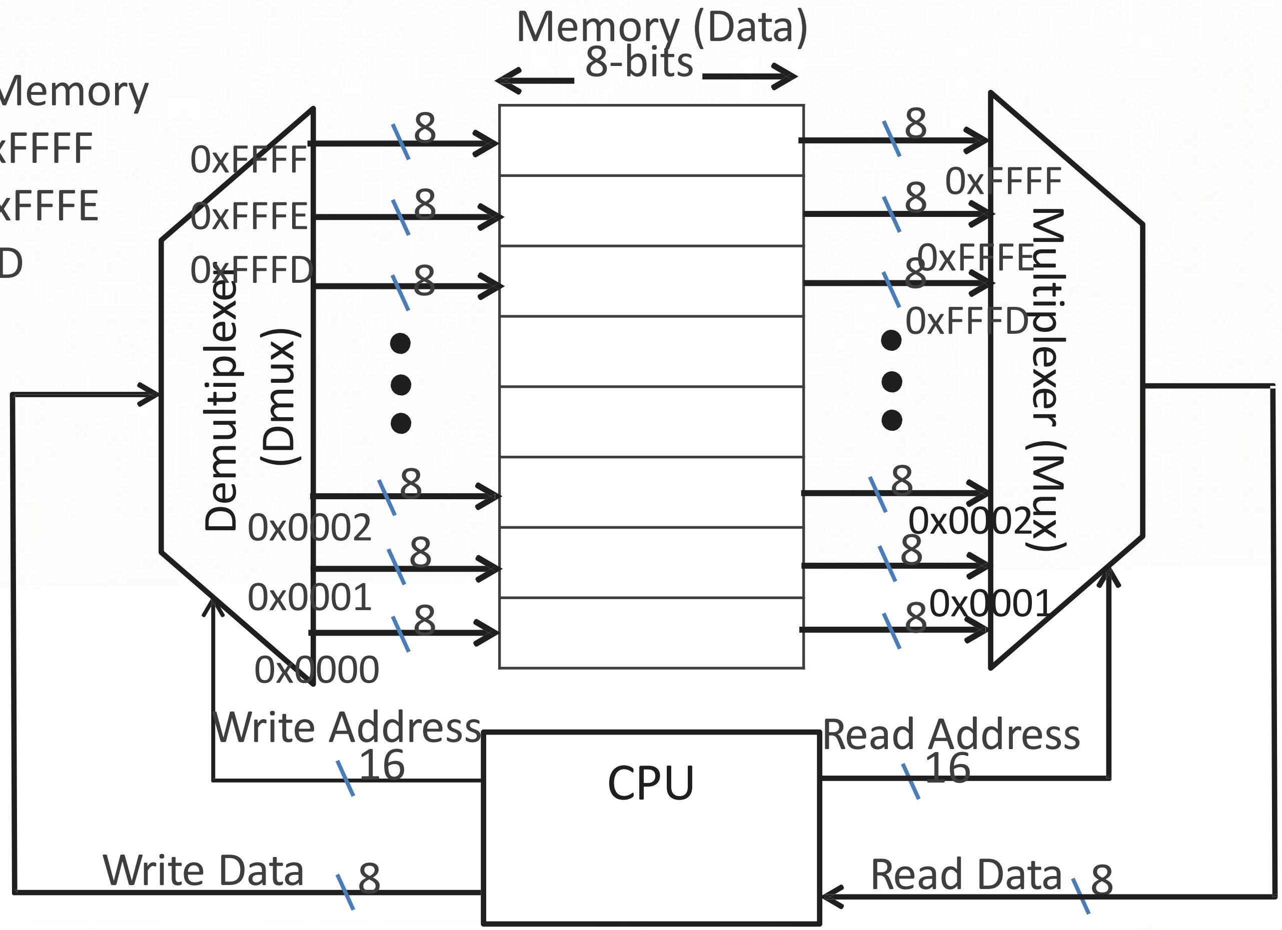
- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

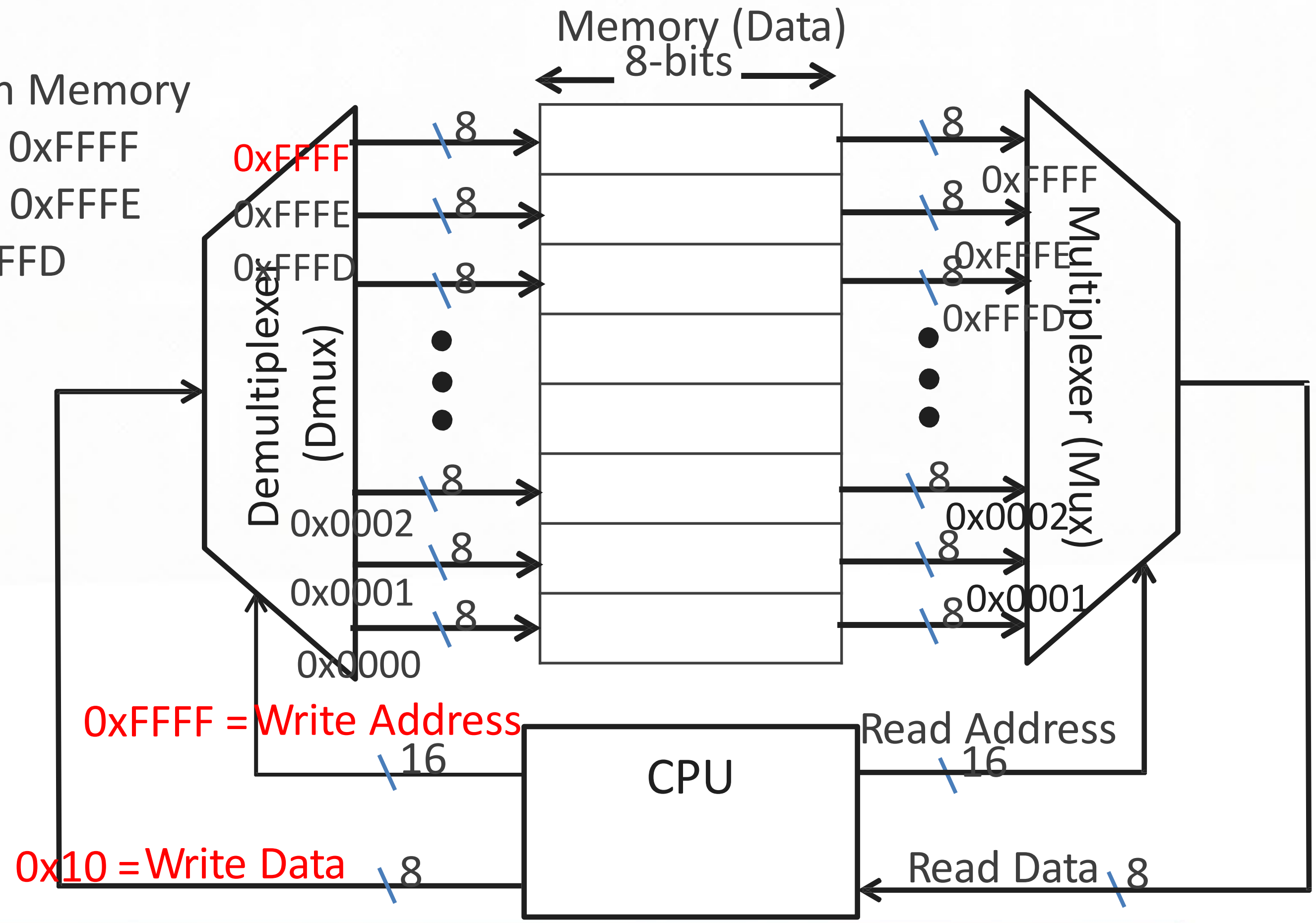
Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

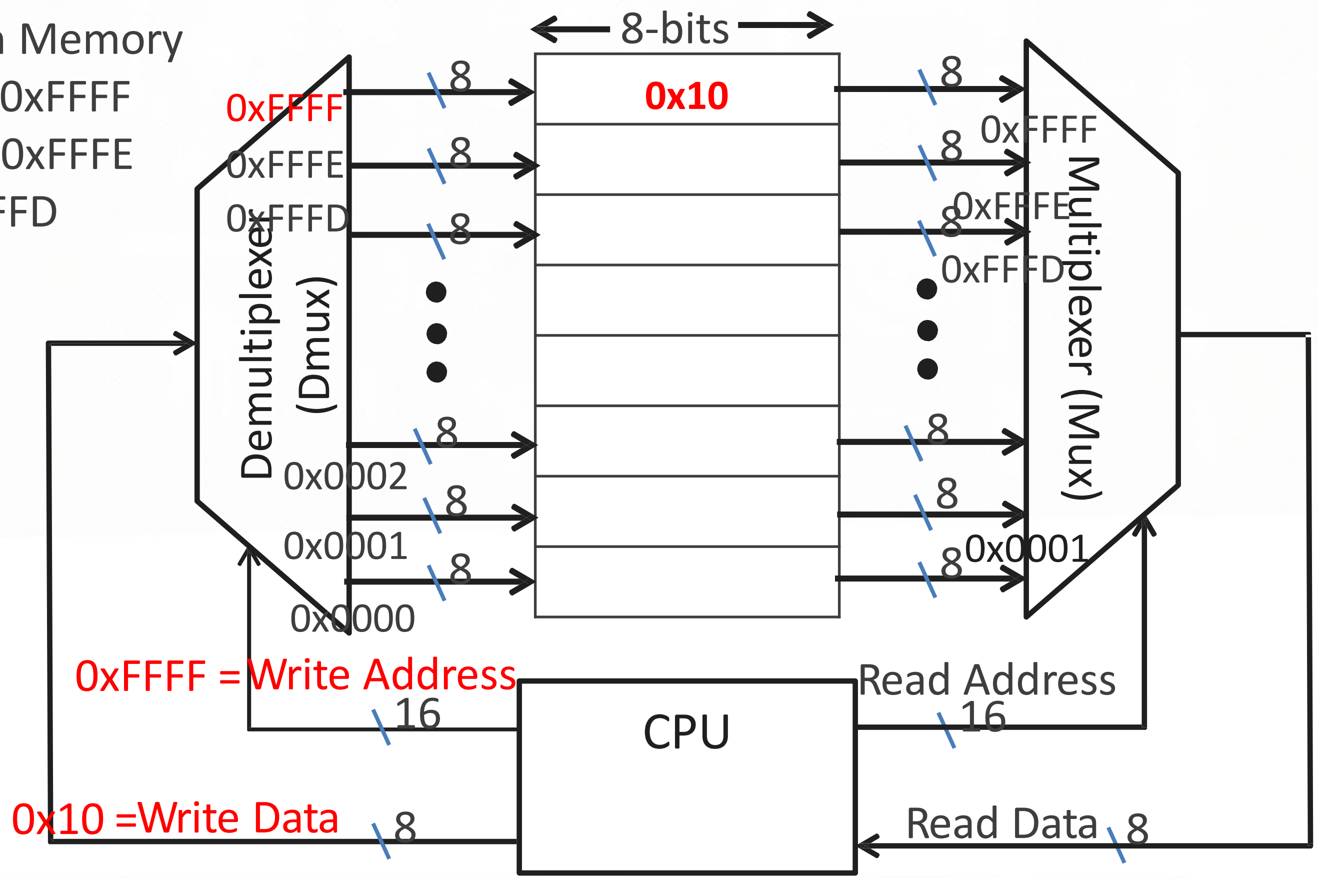
Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

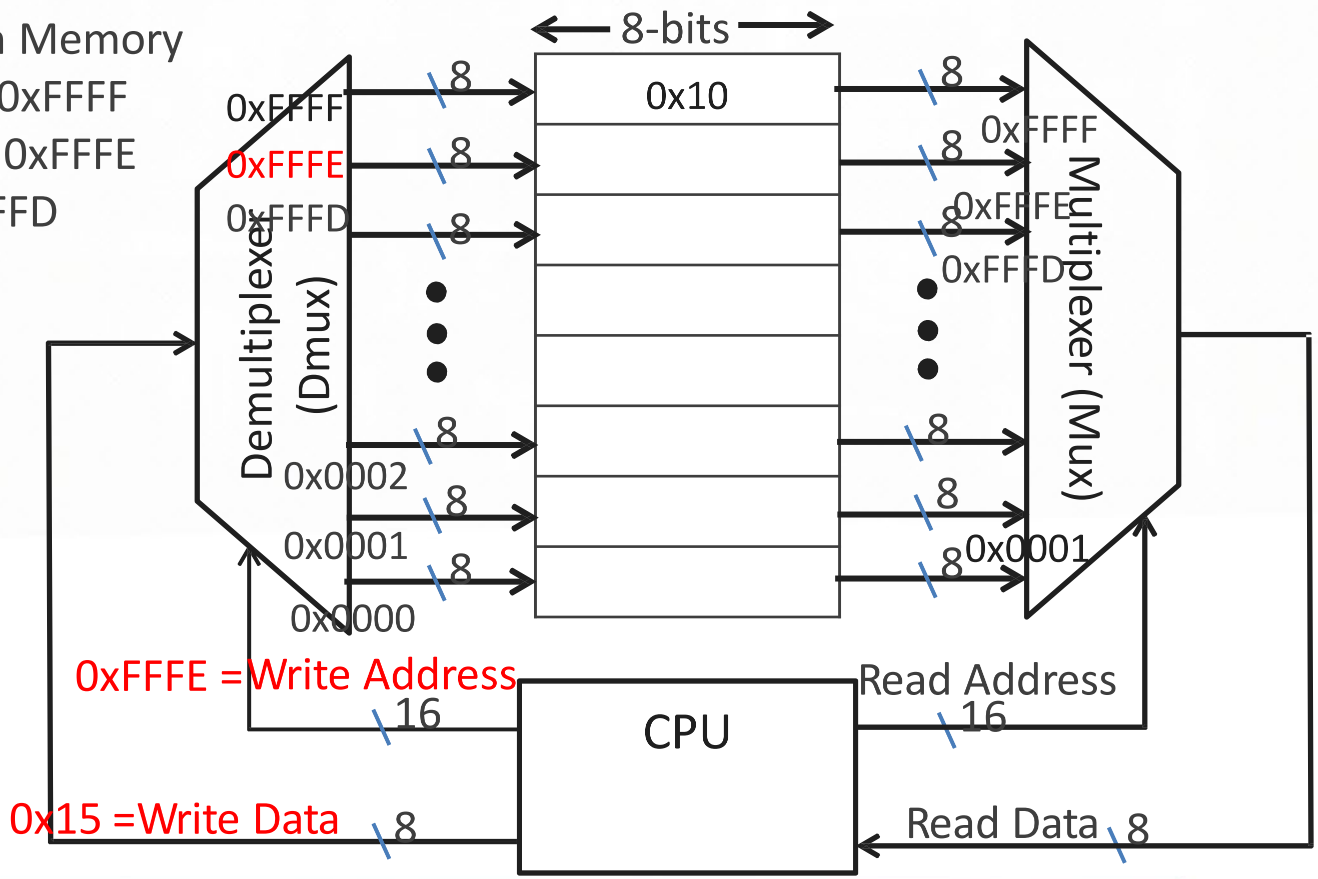
Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

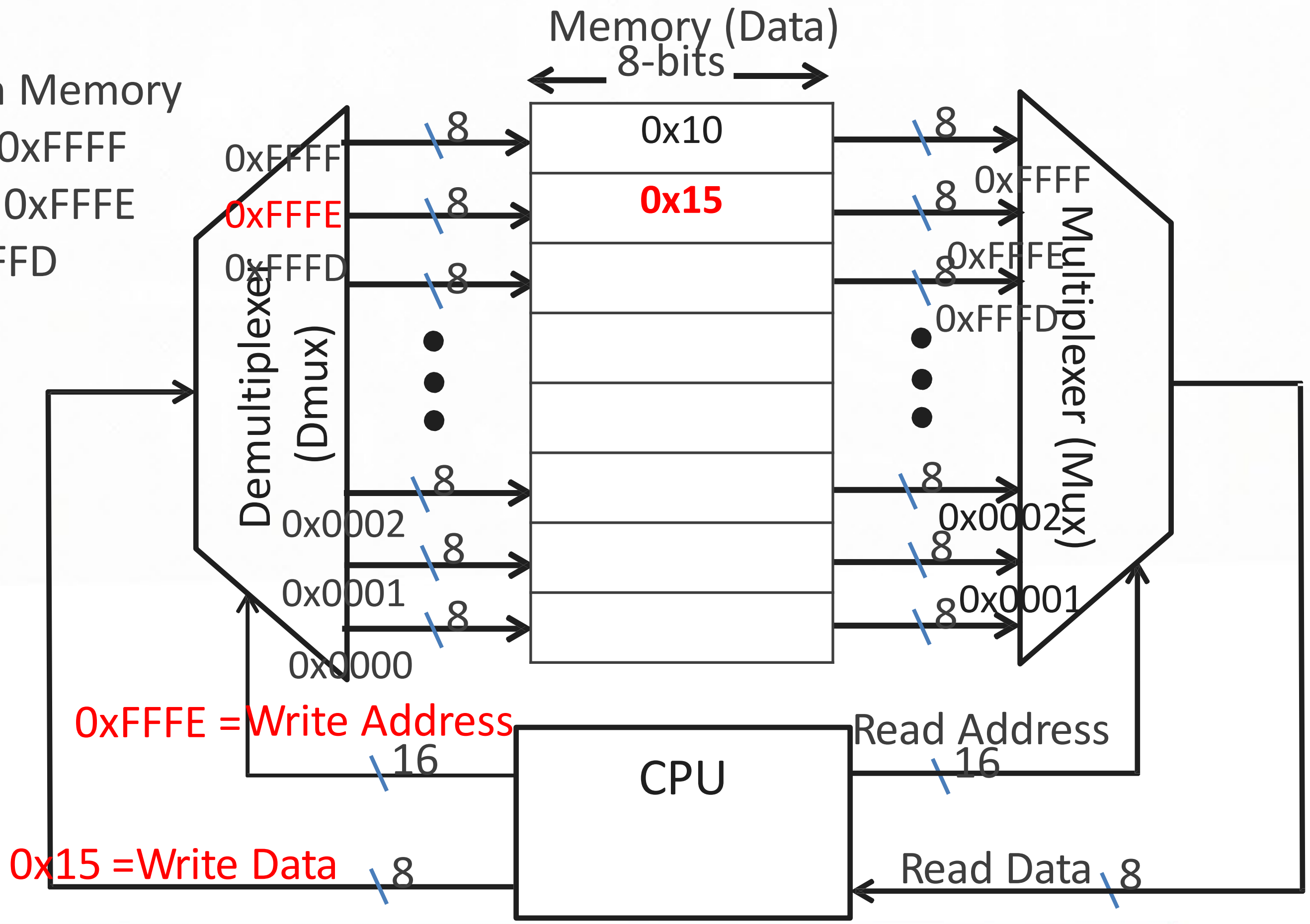
Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

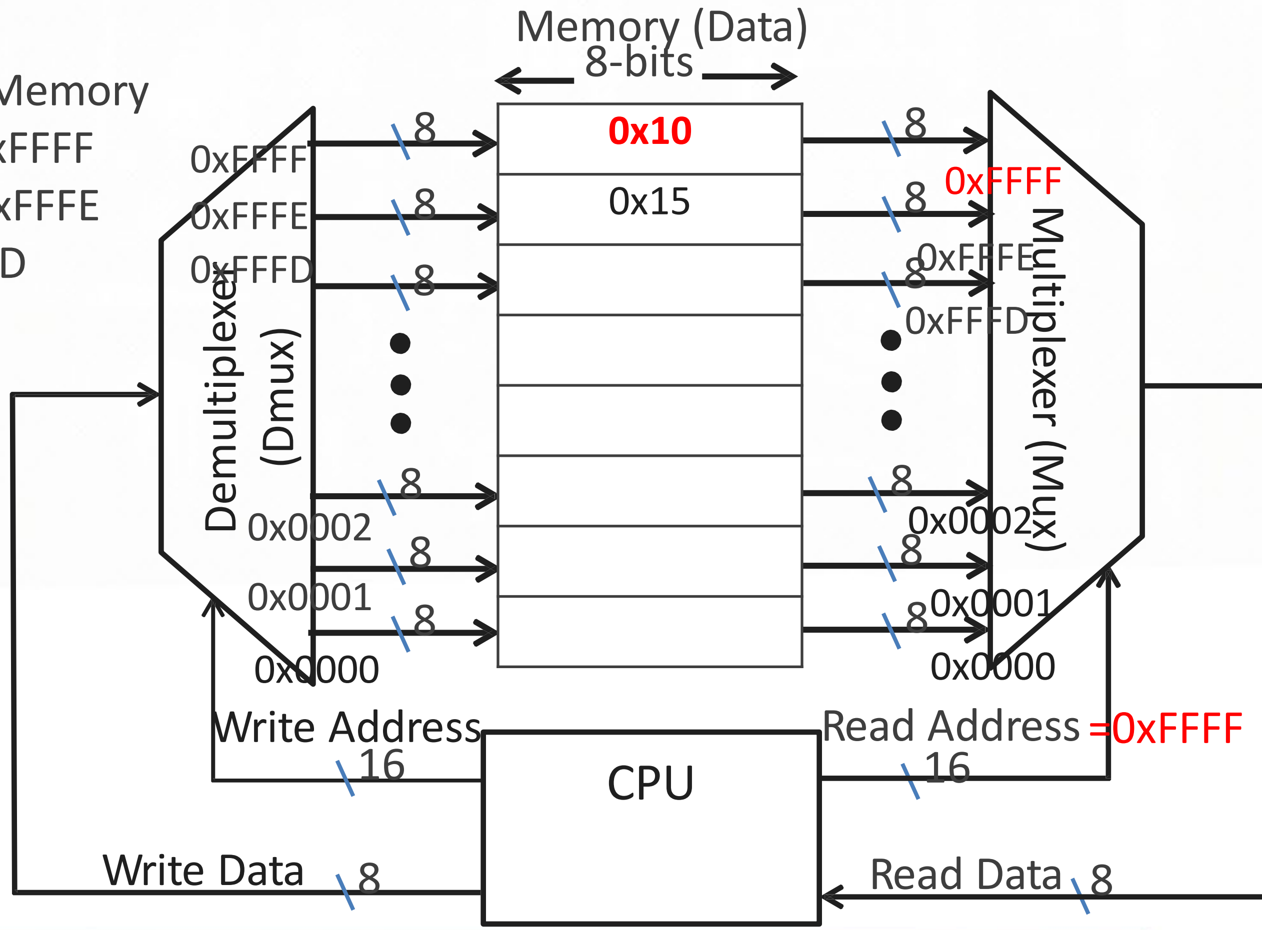
Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

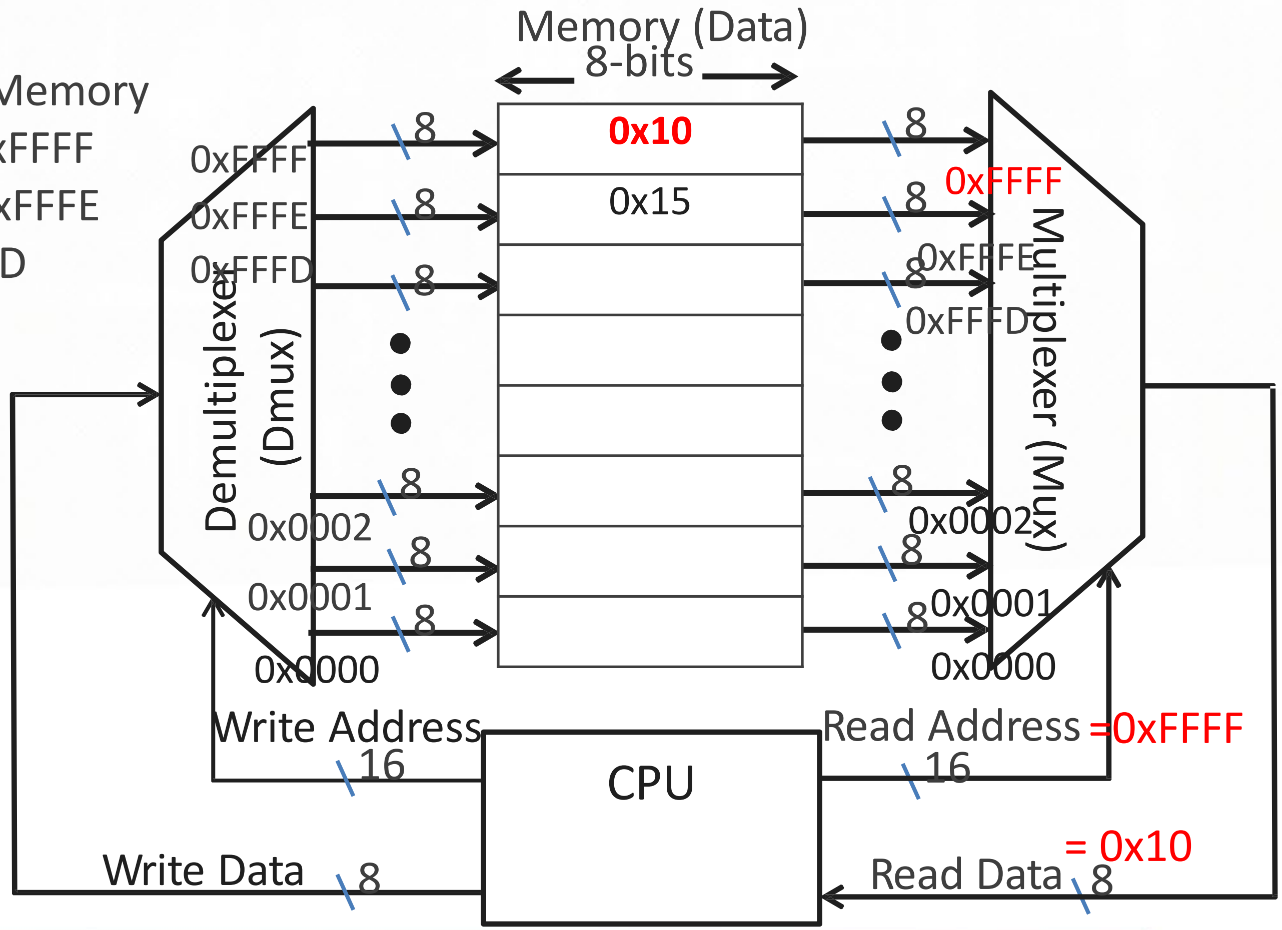
Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

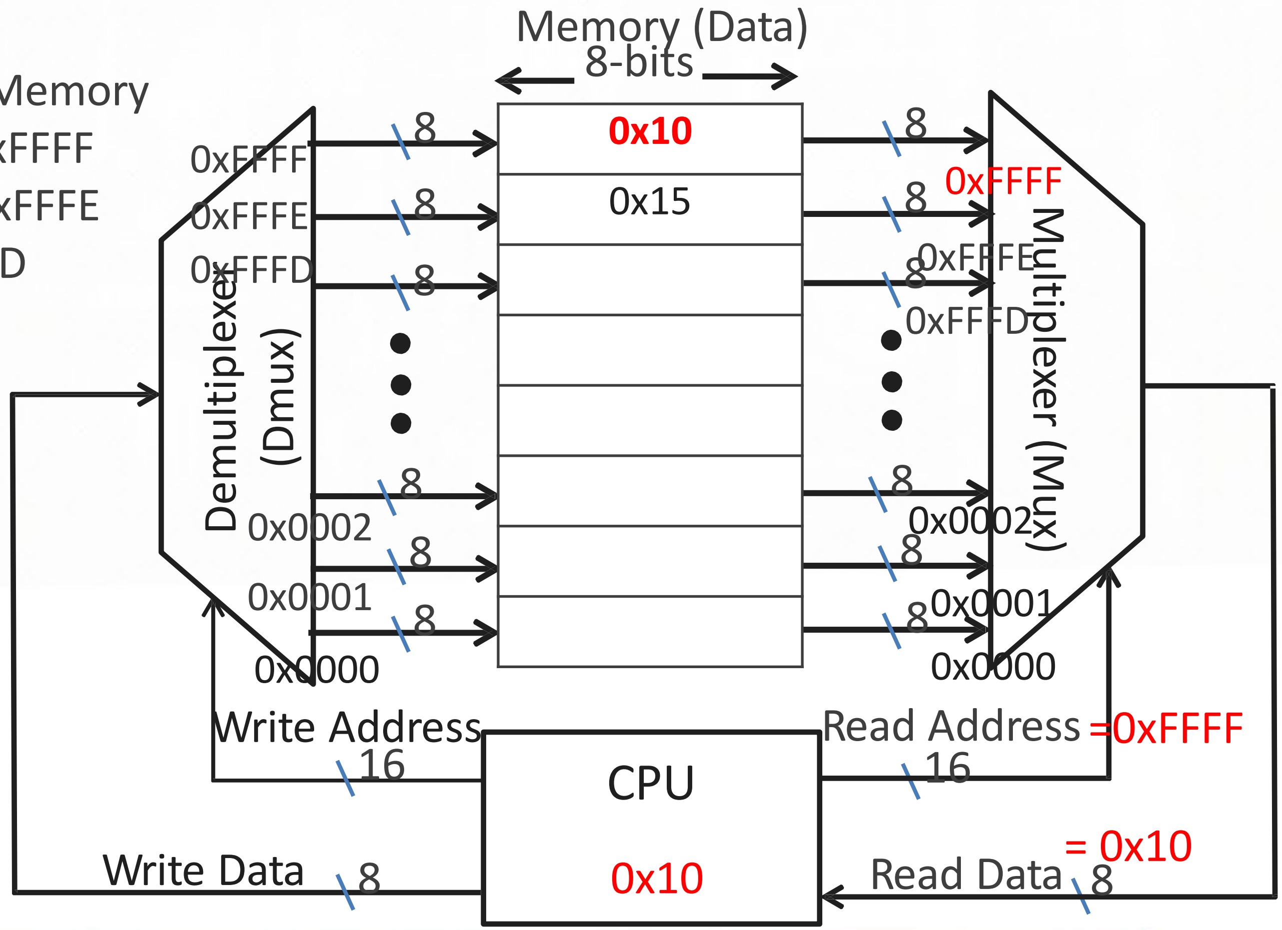
Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

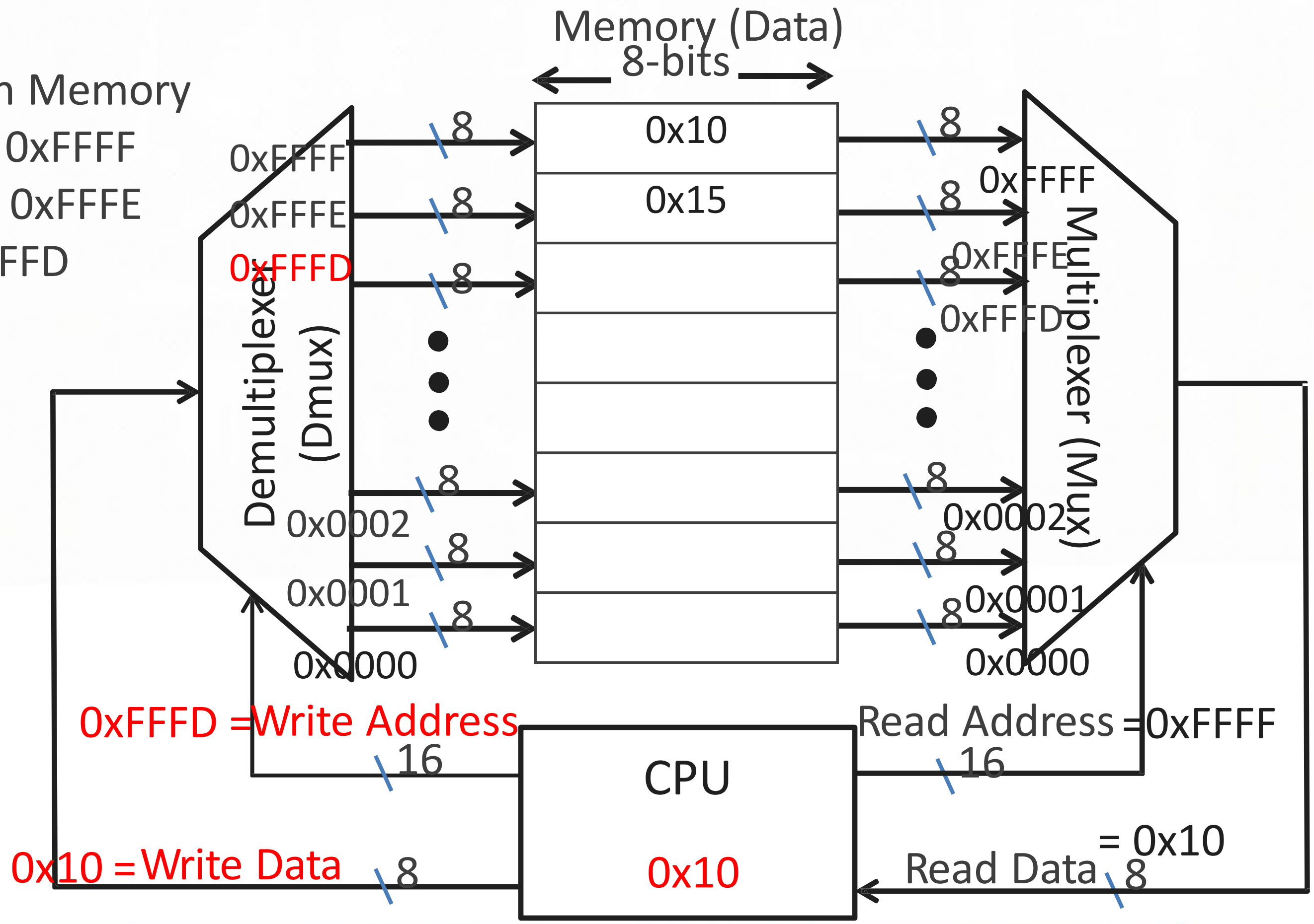
Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

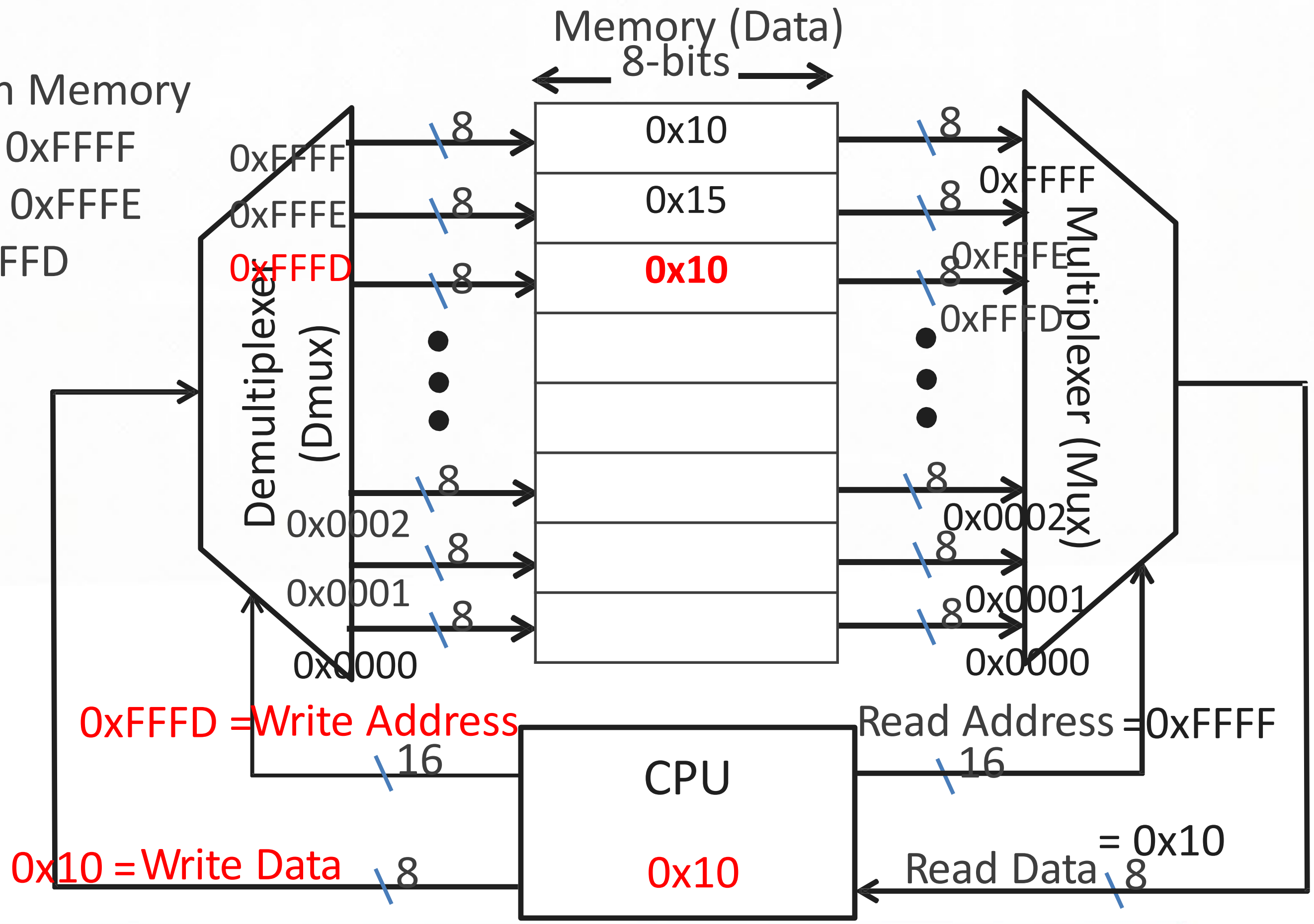
Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



CPU, Memory, and Addresses

- Simplified Hardware picture showing how a CPU, Memory, and Addresses (16-bit) relate to each other

Assuming Stack order in Memory
char r = 0x10; // r is at 0xFFFF
char s = 0x15; // s is at 0xFFFE
char t = r; // t is at 0xFFFD



Exercise

```c
typedef struct coord{
  char x;
  char y;
} coord_t;

coord_t  *coord_ptr;
int *num_ptr;
int **p_ptr = &num_ptr;
char a = 0x07;
coord_t my_coord[2];
int num_array[2]={1,4};

int main(){
  coord_ptr = my_coord;
  num_ptr = num_array;
  my_coord[1].x = 0x33;
  coord_ptr++;
  coord_ptr->y = 0x44;
  **p_ptr = 9
  num_ptr = num_ptr + 2;
  *num_ptr = 0x5040;
  p_ptr++;
  *p_ptr = 0xFEC0;
```

Address	Variable Name	Value
0xFFFF_FFFF	coord_ptr	
0xFFFF_FFFE		
0xFFFF_FFFD		
0xFFFF_FFFC		
0xFFFF_FFFB	num_ptr	
0xFFFF_FFFA		
0xFFFF_FFF9		
0xFFFF_FFF8		
0xFFFF_FFF7	p_ptr	
0xFFFF_FFF6		
0xFFFF_FFF5		
0xFFFF_FFF4		
0xFFFF_FFF3	a	
0xFFFF_FFF2	my_coord[1].y	
0xFFFF_FFF1	my_coord[1].x	
0xFFFF_FFF0	my_coord[0].y	
0xFFFF_FFEF	my_coord[0].x	
0xFFFF_FFEE	num_array[1]	
0xFFFF_FFED		
0xFFFF_FFEC		
0xFFFF_FFEB		
0xFFFF_FFEA	num_array[0]	
0xFFFF_FFE9		
0xFFFF_FFE8		
0xFFFF_FFE7		