

Memory Alignment and Byte Ordering

CIT 595
Spring 2008

Word vs. Byte Addressable Memory

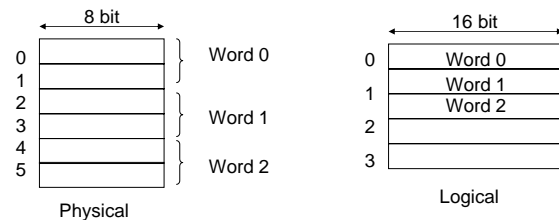
- Byte Addressable
 - Individual byte has a unique address
- Word Addressable
 - Word has a unique address
 - Word not necessary a byte
 - E.g. 2 byte in LC3 Logical Memory
 - Usually what the machine can operate on at once

CIT595

2

Memory Alignment

- Issue of Alignment
 - If memory is byte addressable and ISA word is greater than 1 byte
- E.g: LC3 needs a 16-bit (2 byte) for processing and memory is byte addressable



CIT595

3

Memory Alignment (contd..)

- If accessing an n -byte piece of data, the beginning address must be a multiple of n in order for the access to be aligned
- All other accesses are unaligned accesses
- E.g. For LC3 2-byte pieces of data (words)
 - Aligned accesses are those that are at even addresses i.e. 0,2,4...

CIT595

4

Handling Unalignment (Misalignment)

- Hardware Support
 - Intel: load/store byte
 - Dec Alpha: load/store word but other instructions that can manipulate memory word
- Software Support
 - Exception routine (TRAP) then services the problem
 - This slows things down because now the OS is involved
- On some Unix systems, an attempt to use misaligned data results in a *bus error*, which terminates the program altogether

CIT595

5

Alignment Support by Compilers

- E.g. Size that structs occupy is often larger than the sum of their members' size

```
struct Employee{  
    int ID;  
    char state[3];  
    int salary;  
};
```

- On 32-bit machine, Employee struct should occupy 11 bytes (4+3+4)
 - Assuming char is 1 byte and int is 4 bytes
- However, most compilers add an unused padding byte so that it aligns on a 4 byte boundary
 - Consequently, Employee occupies 12 bytes rather than 11

CIT595

6

Byte Ordering

- The way computer stores bytes of multiple byte data element in byte addressable memory
- Consider 16-bit integer value 1234 (hex)

Byte 1	Byte 0
12	34

Word Addressable

- Two kinds byte ordering

	Little Endian	Big Endian
0	34	12
1	12	34

CIT595

7

Little vs. Big Endian

- Why different ordering?
 - Developed independently with different reasoning's
- For certain test look at only base address
 - Big: High Order byte come first, test for +ve or -ve number
 - Little: Low Order byte comes first, test for odd or even number
- Some operations can be faster in Little Endian
 - e.g. Least significant bytes remain untouched and new digits can be added to the right at a higher address

CIT595

8

Working with Endianness

- Byte order must be reversed when switching endianness
 - If you try to read data files that were created on a machine that is of a different endian nature from your machine
- Examples
 - Graphics Applications
 - > Window BMP uses Little Endian
 - > Adobe Photoshop uses Big Endian
 - Language compilers have to know which way the object code they develop is going to be stored
 - TC/IP networking protocols have big-endian format for data

CIT595

9

Unaligned Access & Byte Ordering Example

```
int main(){  
  
    int d[2];  
    d[0] = 3;  
    d[1] = 4;  
    int * p = d;  
  
    printf("d[0]-> loc:%x data:%x\n",&d[0],  
          d[0]);  
    printf("d[1]-> loc:%x data:%x\n",&d[1],  
          d[1]);  
  
    printf("p before increment:%x\n",p);  
  
    p = (int *) ((char*)p + 1);  
    printf("p after increment: %x\n",p);  
    printf("*p = %x\n", *p);  
  
    return 0;  
  
}
```

■ Output

```
d[0]-> loc:bffff298 data:3  
d[1]-> loc:bffff29c data:4  
p before increment:bffff298  
p after increment: bffff299  
*p = 4000000
```

- Result indicates little endian byte ordering

CIT595

10