Memory Alignment & Ordering

CIT 595
Spring 2010

Memory Alignment

- If memory is byte addressable and ISA word is greater than 1 byte
- E.g. For LC3 2-byte pieces of data (words)
  - Aligned accesses are those that are at even addresses i.e. 0, 2, 4…

Handling Unalignment

- No Support
  - Attempting to access an unaligned address can cause bus error, which terminates the program altogether

- Hardware Support
  - ISAs specify special instructions to align on word boundary or be able to handle unaligned access
    - Intel: load/store byte

- Software Support
  - Exception routine (TRAP) then services the problem
    - This slows things down because now the OS is involved

Additional Support by Compilers: Padding

- 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
Additional Support by Compilers: Align Directive

- This directive is used to make sure a piece of data is aligned.
- Simply place "ALIGN n" on a line immediately before the data you want aligned.
  - Where you replace n with the size of the data you want aligned.
- Adds enough "padding" so that the following data begins at a multiple of n.

Byte Ordering

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

<table>
<thead>
<tr>
<th>Byte 1</th>
<th>Byte 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>34</td>
</tr>
</tbody>
</table>

- Two kinds byte ordering:
  - Little Endian
    - 0: 34
    - 1: 12
  - Big Endian
    - 0: 12
    - 1: 34

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.

Working with Endianess

- 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 generated object code they is going to be stored.
  - TCP/IP networking protocols have big-endian format for data.
Determining Endianness

```c
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]);
    p = (int *) ((char*)p + 1);
    printf("p before increment:%x\n", p);
    p = (int *) ((char*)p + 1);
    printf("p after increment: %x\n", *p);
    return 0;
}
```

- Output on eniac
  - d[0]-> loc:bffff298 data:3
  - d[1]-> loc:bffff29c data:4
  - p before increment: bffff298
  - *p = 4000000
- Result indicates little endian byte ordering