Characters, Bits and Addresses

CSE 410, Spring 2006
Computer Systems

http://www.cs.washington.edu/education/courses/410/06sp

Readings and References

• Reading
  » Section 2.8, Communicating with People
  » Section 2.9, MIPS addressing

Beyond Numbers

• Most computers today use 8-bit bytes to represent characters
• How many characters can you represent in an 8-bit byte?
  » 256
• How many characters are needed to represent all the languages in the world?
  » a gazillion, approximately

char

• American Standard Code for Information Interchange (ASCII)
  » published in 1968
  » defines 7-bit character codes ... 
  » which means only the first 128 characters
  » after that, it’s all “extensions” and “code pages”
• ISO 8859-x
  » codify the extensions to 8 bits (256 characters)
ISO 8859-x

- Each “language” defines the extended chars
  - Latin1 (West European), Latin2 (East European), Latin3 (South European), Latin4 (North European), Cyrillic, Arabic, Greek, Hebrew, Latin5 (Turkish), Latin6 (Nordic)
- How many languages are there?
  - a gazillion, approximately

Unicode

- Universal character encoding standard
  - http://www.unicode.org/
- 16 bits should cover just about everything ...
  - “original goal was to use a single 16-bit encoding that provides code points for more than 65,000 characters”
  - the Java char type is a 16-bit character
- How many characters are needed? ...

Unicode UTF-8

Table 3-5 specifies the bit distribution for the UTF-8 encoding form, showing the ranges of Unicode scalar values corresponding to one-, two-, three-, and four-byte sequences. For a discussion of the difference in the formulation of UTF-8 in ISO/IEC 10646, see Section C.3, UCS Transformation Formats.

Table 3-5. UTF-8 Bit Distribution

<table>
<thead>
<tr>
<th>Scalar Value</th>
<th>1st Byte</th>
<th>2nd Byte</th>
<th>3rd Byte</th>
<th>4th Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000-00001111</td>
<td>0xxxxxxx</td>
<td>0xxxxxxx</td>
<td>0xxxxxxx</td>
<td>0xxxxxxx</td>
</tr>
<tr>
<td>00001000-00001111</td>
<td>10yyyyy</td>
<td>10xxxxx</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00001111</td>
<td>1110zzz</td>
<td>10yyyyy</td>
<td>10xxxxx</td>
<td></td>
</tr>
<tr>
<td>00010000</td>
<td>11111111</td>
<td>10yyyyy</td>
<td>10xxxxx</td>
<td></td>
</tr>
</tbody>
</table>

unicode scalar value:

a number N from 0 to 10FFFF₁₆ (1,114,111₁₀)

Unicode UTF-16

Table 3-4 specifies the bit distribution for the UTF-16 encoding form. Note that for Unicode scalar values equal to or greater than U+10000, UTF-16 uses surrogate pairs. Calculation of the surrogate pair values involves subtraction of 10000₁₆ to account for the starting offset to the scalar value. ISO/IEC 10646 specifies an equivalent UTF-16 encoding form. For details, see Section C.3, UCS Transformation Formats.

Table 3-4. UTF-16 Bit Distribution

<table>
<thead>
<tr>
<th>Scalar Value</th>
<th>UTF-16</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000000000</td>
<td>xxxxxxxxxxxx</td>
</tr>
<tr>
<td>000100000000</td>
<td>110110000w0000000w0000</td>
</tr>
<tr>
<td>100111xxxxxxx</td>
<td>110111xxxxxxx</td>
</tr>
</tbody>
</table>

Where www = uu - 1.
Some character URLs

- ANSI X3.4 (ASCII)
  » http://czyborra.com/charsets/iso646.html
- ISO 8859 (International extensions)
  » http://czyborra.com/charsets/iso8859.html
- Unicode
  » http://www.unicode.org/
  » http://www.unicode.org/iuc/iuc10/x-utf8.html

Moving bytes

- A byte can contain an 8-bit character
- A byte can contain really small numbers
  0 to 255₁₀ or -128₁₀ to 127₁₀
- Sign extension desired effect:
  » sign bit not extended for characters
  » sign bit extended for numbers

Loading bytes

- Unsigned:  
  lbu $reg, a($reg)
  » the byte is 0-extended into the register

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

- Signed:  
  lb $reg, a($reg)
  » bit 7 is extended through bit 31

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

Storing bytes

- No sign bit considerations
  » the right most byte in the register is jammed into the byte address given

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccc}
3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{array}
\]
Storing strings

- Counted strings (for example Pascal strings)
  » byte str[0] holds length: max 255 char
- Counted strings (for example Java strings)
  » int variable holds length: max 2B char
- Terminated strings (for example C strings)
  » no length variable, must count: max n/a

strcpy example

```c
char *strcpy(char *dst, const char *src) {
    char *s = dst;
    while (*dst++ = *src++) != '\0')
    return s;
}
```

strcpy compiled

```assembly
strcpy:
    move $v1,$a0 # remember initial dst
loop:
    lbu $v0,0($a1) # load a byte
    sb $v0,0($a0) # store it
    sll $v0,$v0,24 # toss the extra bytes
    addu $a1,$a1,1 # src++
    addu $a0,$a0,1 # dst++
    bne $v0,$zero,loop # loop if not done
    move $v0,$v0,$v1 # return initial dst
    j $ra # return
```

Manipulating the bits

- Shift Logical
  » sll, srl, sllv, srlv - shift bits in word, 0-extend
  » use these to isolate bits in a word
  » shift amount in instruction or in register
- Bit by bit
  » and, andi - clear bits in destination
  » or, ori - set bits in destination

Shift to the left, shift to the right, push down, pop up, byte, byte, byte!
Example: bit manipulation

```
sll $t1,$t1,24
0000 0000 0000 0000 0000 0000 0000 0000 1111 1010 1010

srl $t1,$t1,28
1010 1111 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010

ori $t1,$t1,0x100
0000 0000 0000 0000 0000 0000 0000 0001 0000 0010 0010
```

Example: C bit fields

- Example of typical application of bit fields
  - unused
  - received byte
  - e
  - r

- But, note poor choice of field locations
  - the received byte is not aligned
  - the byte must be shifted before it can be used

- To: EE designers of interfaces
  - please consider alignment when selecting fields

Multiply and Divide

- There is a separate integer multiply unit
- Use pseudo-instructions to access
  - `mul $t0,$t1,$t2` # $t0 = $t1 * $t2
  - `div $t0,$t1,$t2` # $t0 = $t1 / $t2

- These are relatively slow
  - multiply 5-12 clock cycles
  - divide 35-80 clock cycles

Addressing modes

- Register
  - `jr $ra`
- Offset + Register
  - `lw $t0,0($sp)`
- Immediate
  - `addi $t0,17`
- PC relative
  - `bnez $t0,loop`
- Pseudodirect
  - `jal proc`
Register only

- Use the 32 bits of the specified register as the desired address
- Can specify anywhere in the program address space, without limitation
- `jr $ra`
  » return to caller after procedure completes

Offset + Register

- Specify 16-bit signed offset to add to the base register
- Transfer (lw, sw) base register is specified
  » `lw $t0,4($sp)`
  » `sw $t0,40($gp)`

Immediate

- The 16-bit field holds the constant value

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ori $8, $0, 1</td>
<td>0x34080001</td>
<td>4: li $t0,1</td>
</tr>
<tr>
<td>lui $1, -1</td>
<td>0x3c01ffff</td>
<td>5: li $t0,-1</td>
</tr>
<tr>
<td>ori $8, $1, -1</td>
<td>0x3428ffff</td>
<td>6: li $t0,0xFFFF</td>
</tr>
<tr>
<td>ori $8, $0, -1</td>
<td>0x3408ffff</td>
<td>7: li $t0,0x1FFFF</td>
</tr>
<tr>
<td>lui $1, 21845</td>
<td>0x3c015555</td>
<td>8: li $t0,0x5555AAAA</td>
</tr>
<tr>
<td>ori $8, $1, -1</td>
<td>0x3428aaaa</td>
<td>9: li $t0,$t1,skip</td>
</tr>
<tr>
<td>lui $1, 64 [main]</td>
<td>0x3c010040</td>
<td>9: li $t0,main</td>
</tr>
<tr>
<td>ori $8, $1, 32 [main]</td>
<td>0x34280020</td>
<td>9: li $t0,main</td>
</tr>
</tbody>
</table>

PC relative

- Branch (beq, bne) base register is PC
  » `beq $t0,$t1,skip`
- The 16-bit value stored in the instruction is considered to be a word offset
  » multiplied by 4 before adding to PC
  » can branch over ± 32 K instruction range
Pseudodirect

- The specified offset is 26 bits long
  - Considered to be a word offset
  - multiplied by 4 before use
- The top 4 bits of the PC are concatenated with the new 28 bit offset to give a 32-bit address
- Can jump within 256 MB segment