Characters, Bits and Addresses

CSE 410, Spring 2007
Computer Systems

http://www.cs.washington.edu/410

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 (but not a google)

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)
  - » http://www.microsoft.com/globaldev/reference/iso.mspx
- Is that enough?
  - » For everyone on the planet?

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

Scalar Value	1st Byte	2nd Byte	3rd Byte	4th Byte
00000000 0xxxxxxx	0xxxxxxx			
00000yyy yyxxxxxx	110yyyyy	10xxxxxx		
zzzzyyyy yyxxxxxx	1110zzzz	10yyyyyy	10xxxxxx	
000uuuuu zzzzyyyy yyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

unicode scalar value:

a number N from 0 to $10FFFF_{16}$ ($1,114,111_{10}$)

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 10000_{16}, 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

Scalar Value	UTF-16
xxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxx
000uuuuuxxxxxxxxxxxxxxxx	110110wwwwxxxxxx 110111xxxxxxxxxx

Where wwww = uuuuu - 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 255_{10} or -128_{10} to 127_{10}
- 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

0000 0000	0000 0000	0000 0000	xxxx xxxx

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

0000 0000	0000 0000	0000 0000	0xxx xxxx

1111 1111	1111 1111	1111 1111	1xxx xxxx

Storing bytes

- No sign bit considerations
  - » the right most byte in the register is jammed into the byte address given
  - » **`sb $t0, 2($sp)`**

$t0 →	0000 0000	0000 0000	0000 0000	xxxx xxxx
$sp	3	2	1	0
0x7FFFEFFC →	0000 0000	xxxx xxxx	0000 0000	0000 0000

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
  - » Address of string is address of 1st character; first \0 byte signifies end of string

strcpy example

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

Software reliability issue: how safe is this?

strcpy compiled

```
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    $v0,$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 1111 1010 1111
1010 1111 0000 0000 0000 0000 0000 0000

srl  $t1,$t1,28

1010 1111 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 1010

ori  $t1,$t1,0x100

0000 0000 0000 0000 0000 0000 0000 1010
0000 0000 0000 0000 0000 0001 0000 1010
```


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

```
0x34080001  ori $8, $0, 1         ; 4: li $t0,1
0x3c01ffff  lui $1, -1            ; 5: li $t0,-1
0x3428ffff  ori $8, $1, -1
0x3408ffff  ori $8, $0, -1        ; 6: li $t0,0xFFFF
0x3c010001  lui $1, 1             ; 7: li $t0,0x1FFFF
0x3428ffff  ori $8, $1, -1
0x3c015555  lui $1, 21845         ; 8: li $t0,0x5555AAAA
0x3428aaaa  ori $8, $1, -21846
0x3c010040  lui $1, 64 [main]     ; 9: la $t0,main
0x34280020  ori $8, $1, 32 [main]
```


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