Readings and References

• Reading
  – Sections 3.7 through 3.10, A.1 through A.4, Patterson and Hennessy, Computer Organization & Design
    • note error in figure page 149, address 80012 repeated

• Other References
  – Sun demo of QuickSort vs BubbleSort
    <http://java.sun.com/applets/jdk1.1/demo/SortDemo/example1.html>

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

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)
  – see http://czyborra.com/charsets/iso8859.html
• How many languages are there?
  – 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)

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 does a million

Unicode scalar value:
a number N from 0 to 10FFFF (1,114,111)

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 or -128 to 127
• 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 0000 0000

• Signed: `lb $reg, a($reg)`
  – bit 7 is extended through bit 31
  0000 0000 0000 0000 0000 0000 0000 0000
  1111 1111 1111 1111 1111 1111 1111 1111

Storing bytes

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

Storing strings

• Counted strings (for example Pascal strings)
  – byte st[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**

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

• prototype matches libc
• pointers, not arrays
• better loop

**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 $v0,$v1          # return initial dst
j $ra              # return
```

**Manipulating the bits**

• Shift Logical
  – sll, srl, sltv, 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

**Example: bit manipulation**

```
       0000 0000 0000 0000 0000 1111 1010 1111
       1010 1111 0000 0000 0000 0000 0000 0000
       1010 1111 0000 0000 0000 0000 0000 0000
        0000 0000 0000 0000 0000 0000 0000 1010
        0000 0000 0000 0000 0000 000 1 0000 1010
```

```assembly
sll  $t1,$t1,24
srl  $t1,$t1,28
ori  $t1,$t1,0x100
```

**Example: C bit fields**

• Example in the book on page 229 is a typical application of bit fields

```
... unused ... received byte + +
```

• 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
  ```assembly
  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
  0x34000000 ori $8, $0, -1
  0x3c010000 lui $1, 0
  0x3c015555 lui $1, 21845 ; 6: li $t0,0xFFFF
  0x3428ffff ori $8, $1, -1
  0x3c010040 lui $1, 64 [main] ; 7: li $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
Starting a Program

- Two phases from source code to execution
- Build time
  - compiler creates assembly code
  - assembler creates machine code
  - linker creates an executable
- Run time
  - loader moves the executable into memory and starts the program

Build Time

- You’re experts on compiling from source to assembly and hand crafted assembly
- Two parts to translating from assembly to machine language:
  - Instruction encoding (including translating pseudoinstructions)
  - Translating labels to addresses
- Label translations go in the symbol table

Symbol Table

- Symbols are names of global variables or labels (including procedure entry points)
- Symbol table associates symbols with their addresses in the object file
- This allows files compiled separately to be linked

<table>
<thead>
<tr>
<th>Label</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>bigArray</td>
<td>0x10006000</td>
</tr>
<tr>
<td>LabelA</td>
<td>0x01031ff0</td>
</tr>
</tbody>
</table>

Modular Program Design

- Small projects might use only one file
  - Any time any one line changes, recompile and reassemble the whole thing (death of Pascal)
- For larger projects, recompilation time and complexity management is significant
- Solution: split project into modules
  - compile and assemble modules separately
  - link the object files

The Compiler + Assembler

- Translate source files to object files
  - Object files
    - Contain machine instructions (1’s & 0’s)
    - Bookkeeping information
      - Procedures and variables the object file defines
      - Procedures and variables the source files use but are undefined (unresolved references)
      - Debugging information associating machine instructions with lines of source code

The Linker

- The linker’s job is to “stitch together” the object files:
  1. Place the data modules in memory space
  2. Determine the addresses of data and labels
  3. Match up references between modules
- Creates an executable file
Determining Addresses

- Some addresses change during memory layout
- Modules were compiled in isolation
- Absolute addresses must be relocated
- Object file keeps track of instructions that use absolute addresses

Resolving References

- For example, in a word processing program, an input module calls a spell check module
- Module address is unresolved at compile time
- The linker matches unresolved symbols to locations in other modules at link time
- In SPIM, “main” is resolved when your program is loaded

Linker Example

- Code:
  - main:A=area(5.0)
  - static data:
    - PI = 3.1415
  - defined symbols:
    - main, PI
  - undefined symbols:
    - Area

- Code:
  - Area:return PI*r*r
  - static data:
    - PI = 3.1415
  - defined symbols:
    - Area
  - undefined symbols:
    - PI

- Defined symbols:
  - main, PI, Area

Libraries

- Some code is used so often, it is bundled into libraries for common access
- Libraries contain most of the code you use but didn’t write: e.g., printf()
- Library code is (often) merged with yours at link time

The Executable

- End result of compiling, assembling, and linking: the executable
  - Header, listing the lengths of the other segments
  - Text segment
  - Static data segment
  - Potentially other segments, depending on architecture & OS conventions

Run Time

- When a program is started ...
  - Some dynamic linking may occur
    - Some symbols aren’t defined until runtime
    - Windows’ dlls (dynamic link library)
  - The segments are loaded into memory
  - The OS transfers control to the program and it runs
- We’ll learn a lot more about this during the OS part of the course
QuickSort example

- QuickSort vs BubbleSort
  - don’t ever use a bubble sort, many better sort routines are available as source or library files
- The example QuickSort.c is taken from the Java example on the Sun demo page
- I converted it to C and compiled with gcc
- Helpful to review register usage, stack allocation, branching techniques