Communicating with People (2.8)

• For communication
  – Use characters and strings

• Characters
  – 8-bit (one byte) data for ASCII
    \[
    \text{lb} \ $t0, \ 0($sp) \ ; \ \text{load byte}
    \]
    • Load a byte from memory, placing it in the rightmost 8-bits of registers
    \[
    \text{sb} \ $t0, \ 0($gp) \ ; \ \text{store byte}
    \]
    • Takes a byte from the rightmost 8-bits of a register and writes it to the memory
  – Unicode in Java (16-bits)
    \[
    \text{lh} \ $t0, \ 0($sp) \ ; \ \text{load halfword}
    \]
    • Load a byte from memory, placing it in the rightmost 16-bits of registers
    \[
    \text{sh} \ $t0, \ 0($gp) \ ; \ \text{store halfword}
    \]
    • Takes a byte from the rightmost 16-bits of a register and writes it to the memory
Q. Impact of Word Alignment to Byte/Halfword Storage

• MIPS software tries to keep the stack aligned to word address
  – A char variable will occupy four bytes, even though it requires less
  – Solution
    • Software will **pack** C string in 4 bytes per word, Java string in 2 halfwords per word
MIPS Addressing Mode (2.9)

- Addressing mode
  - A method that help you identify and find where the operand is
  - What you learned now
    - Register addressing
    - Immediate addressing
    - Base or displacement addressing

\[
\text{lw} \quad $t0, \ 32($s3)
\]
• We'd like to be able to load a 32 bit constant into a register
• Must use two instructions, new "load upper immediate" instruction
  `lui $t0, 1010101010101010`
  filled with zeros

  1010101010101010  0000000000000000

• Then must get the lower order bits right, i.e.,
  `ori $t0, $t0, 1010101010101010`

  1010101010101010  0000000000000000

  0000000000000000  1010101010101010

  1010101010101010  1010101010101010

Either compiler or assembler to break and then reassemble this
So $at is reserved for assembler
Addresses in Branches and Jumps

- Instructions:
  
  ```
  bne $s0,$s1,Exit
  j 1000
  ```

- Q. What’s the destination address of next instruction? And How far do you can jump (or branch)?
Addresses in Branches and Jumps

• Destination Address
  – MIPS uses **PC-relative address** (relative to PC+4, +/- 2^{15}) for all conditional branches
  \[
  \text{Next PC} = (\text{PC} + 4) + (16\text{-bit address } >> 2)
  \]
  – MIPS uses **long addresses (26-bits)** (pseduodirect addressing) for both jump and jump-and-link instructions
  \[
  \text{Next PC} = \{\text{PC}[31:28], (26\text{-bit address } >> 2)\}
  \]
  – Note. PC-relative addressing refer to the number of words to the next instruction instead of number of bytes (word address)
  – 16-bit field => 18-bit byte address displacement
  – 26-bit field => 28-bit byte address displacement
How Far Do You Can Jump or Branch?

- **Formats:**

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit address</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>16 bit address</td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td>26 bit address</td>
</tr>
</tbody>
</table>

- **Branch limitation:** $+/-2^{15}$, $(2^{18} = 256$KB address boundaries$)$
  - Is it enough: most branches are local (principle of locality)
  - How about larger space? Branch + Jump

  ```
  beq $s0, $s1, L1
  bne $s0, $s1, L2
  j L1
  L2:       .......
  ```

- **Jump limitation:** $+/-2^{25}$, $(2^{28} = 256$MB address boundaries$)$
  - How about larger space? Jump registers (32-bit value)

  ```
  j $s0
  ```
MIPS Addressing Mode

1. Immediate addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   \end{array}
   \quad \text{Immediate}
   \]

2. Register addressing
   \[
   \begin{array}{cccc}
   \text{op} & \text{rs} & \text{rt} & \text{rd} \ldots & \text{funct} \\
   \end{array}
   \quad \text{Registers}
   \]

3. Base addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   \end{array}
   \quad \text{Address}
   \]

4. PC-relative addressing
   \[
   \begin{array}{ccc}
   \text{op} & \text{rs} & \text{rt} \\
   \end{array}
   \quad \text{Address}
   \]

5. Pseudodirect addressing
   \[
   \begin{array}{c}
   \text{op} \\
   \end{array}
   \quad \text{Address}
   \]
To summarize:

### MIPS operands

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 registers</td>
<td>$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at</td>
<td>Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.</td>
</tr>
<tr>
<td>$2^{30}$ memory words</td>
<td>Memory[0], Memory[4], ... Memory[4294967292]</td>
<td>Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.</td>
</tr>
</tbody>
</table>

### MIPS assembly language

<table>
<thead>
<tr>
<th>Category</th>
<th>Instruction</th>
<th>Example</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
<td>Three operands; data in registers</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 100</td>
<td>$s1 = $s2 + 100</td>
<td>Used to add constants</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = Memory[$s2 + 100]</td>
<td>Word from memory to register</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Word from register to memory</td>
</tr>
<tr>
<td></td>
<td>load byte</td>
<td>lb $s1, 100($s2)</td>
<td>Memory[$s2 + 100]</td>
<td>Byte from memory to register</td>
</tr>
<tr>
<td></td>
<td>store byte</td>
<td>sb $s1, 100($s2)</td>
<td>Memory[$s2 + 100] = $s1</td>
<td>Byte from register to memory</td>
</tr>
<tr>
<td></td>
<td>load upper immediate</td>
<td>lui $s1, 100</td>
<td>$s1 = 100 * 2^{16}</td>
<td>Loads constant in upper 16 bits</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>branch on equal</td>
<td>beq $s1, $s2, 25</td>
<td>if ($s1 == $s2) go to PC + 4 + 100</td>
<td>Equal test; PC-relative branch</td>
</tr>
<tr>
<td></td>
<td>branch on not equal</td>
<td>bne $s1, $s2, 25</td>
<td>if ($s1 != $s2) go to PC + 4 + 100</td>
<td>Not equal test; PC-relative</td>
</tr>
<tr>
<td></td>
<td>set on less than</td>
<td>slt $s1, $s2, $s3</td>
<td>if ($s2 &lt; $s3) $s1 = 1; else $s1 = 0</td>
<td>Compare less than; for beq, bne</td>
</tr>
<tr>
<td></td>
<td>set less than immediate</td>
<td>slti $s1, $s2, 100</td>
<td>if ($s2 &lt; 100) $s1 = 1; else $s1 = 0</td>
<td>Compare less than constant</td>
</tr>
<tr>
<td>Unconditional jump</td>
<td>jump</td>
<td>j 2500</td>
<td>go to 10000</td>
<td>Jump to target address</td>
</tr>
<tr>
<td></td>
<td>jump register</td>
<td>jr $ra</td>
<td>go to $ra</td>
<td>For switch, procedure return</td>
</tr>
<tr>
<td></td>
<td>jump and link</td>
<td>jal 2500</td>
<td>$ra = PC + 4; go to 10000</td>
<td>For procedure call</td>
</tr>
</tbody>
</table>
Overview of MIPS

- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- rely on compiler to achieve performance — what are the compiler's goals?
- help compiler where we can
If you designed a new processor, what is the basic system software tool set you need?
Tool Chain

A translation hierarchy

C program

Compiler

Assembly language program

Assembler

Object: Machine language module

Linker

Object: Library routine (machine language)

Executable: Machine language program

Loader

Debugger

Simulator

Memory
Tool Chain

• Two good examples
  – SimpleScalar
    www.simplescalar.com
  – GNUPro
    www.intel.com (Search GNUPro)
Tool Chain

• Basic Set:
  – Simulator (functional / cycle-accurate)
  – Binary utilities: assembler, linker, objdump, ar, nm
  – Compilers: C, C++, Fortran, and etc.
  – Debugger
  – Others: performance monitor (VTune of Intel)
Tool Chain

• Preprocessor
  – Functionality
    • Header files, Definitions, Conditional compilation, pragma

• Tool Chain - Compiler
  – Transform a program from high level language to assembly language (or machine language)
    • Higher productivity
  – Optimizations
    • As good as hand coded assembly code, especially for large program
Tool Chain-Assembler

• Tool Chain - Assembler
  – Transform assembly code into binary (machine code)
  – **Pseudoinstruction**
    • Instruction that is not implement in the hardware but appears in assembly language for simple translation and programming
    • **Cost:** reserving one register $at for use by the assembler
    • **Advantages:** rich set of assembly language instructions

Move $t0, $t1  =>  add $t0, $zero, $t1
Blt          =>  slt + bne
Far branch and jump
32-bit constant load
Tool Chain-Assembler

- **Object file**
  - Output of assembler

  - **Object file header**
    - describes the size and position of other pieces of object file
  - **Text segment**
    - Machine language code
  - **Static data segment**
    - Data allocated for the life of the program
  - **Relocation information**
    - Identify instructions and data words that depend on absolute addresses when the program is loaded into memory
  - **Symbol table**
    - Labels that are not defined, e.g. external ref.
  - **Debugging information**
    - Description so that debugger can associate machine instructions with C source files and make data structures readable
Tool Chain

• Tool Chain - Linker
  – Linking – resolve symbols
  – Relocation – assign memory address

• Tool Chain - loader
  – Cannot see by user
  – Done by Shell and OS kernel

• Tool Chain – Library
  – Static or dynamic library
How Compiler Optimize

- High level optimization
  - Procedure inlining
  - Loop transformations
    - Loop unrolling to reduce loop overhead
    - Loop interchanged for better memory behaviors
- Local and global optimization
  - Common subexpression elimination
  - Strength reduction
  - Constant propagation
  - Copy propagation
  - Dead store elimination
Optimization Summary

• High level (O3)
  – Procedure integration
    • At or near the source level; processor independent
    • Replace procedure call by procedure body

• Local (O1): with straight-line code
  – Common subexpression elimination
  – Constant propagation
  – Stack height reduction

• Global (O2): across a branch
  – Global common subexpression elimination
  – Copy propagation
  – Code motion
  – Induction variable elimination

• Processor dependent (O1)
  – Strength reduction
    • Replace multiply by a constant with shifts
  – Pipeline scheduling
    • Reorder instructions to improve pipeline performance
  – Branch offset optimization

Safe first, then speed
A Full Example to Translate C Code

• General steps for translating C to assembly language
  – Allocate registers to program variables
  – Produce code for the body of the procedure
  – Preserve registers across the procedure invocation

```c
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```
Translation

- Register allocation
  - Parameter passing: $a0, $a1
  - All temporary registers can be used since swap is a leaf procedure, $t0, $t1, $t2

- Produce code
  - Get the address of v[k] by multiplying k by 4
    Swap:   sll $t1, $a1, 2
            add $t1, $a0, $t1
  - Load v[k] and v[k+1]
    lw $t0, 0($t1)
    lw $t2, 4($t1)
  - Store v[k] and v[k+1]
    sw $t2, 0($t1)
    sw $t0, 4($t1)

- Return
  jr $ra
Alternative Architectures

• Design alternative:
  – provide more powerful operations
  – goal is to reduce number of instructions executed
  – danger is a slower cycle time and/or a higher CPI

  —“The path toward operation complexity is thus fraught with peril.
  To avoid these problems, designers have moved toward simpler instructions”

• Sometimes referred to as “RISC vs. CISC”
  – virtually all new instruction sets since 1982 have been RISC
  – VAX: minimize code size, make assembly language easy
    instructions from 1 to 54 bytes long!

• We’ll look at PowerPC and 80x86
PowerPC

• Indexed addressing
  – example: $t1 = \text{Mem}ary[a0+s3]$
  – What do we have to do in MIPS?

• Update addressing
  – update a register as part of load (for marching through arrays)
  – example: $t0 = \text{Mem}ory[s3+4]; s3 = s3 + 4$
  – What do we have to do in MIPS?

• Others:
  – load multiple/store multiple
  – a special counter register “bc Loop”
    
    \textit{decrement counter, if not 0 goto loop}
80x86

- 1978: The Intel 8086 is announced (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added

“This history illustrates the impact of the “golden handcuffs” of compatibility

“adding new features as someone might add clothing to a packed bag”

“an architecture that is difficult to explain and impossible to love”
A dominant architecture: 80x86

• See your textbook for a more detailed description
• Complexity:
  – Instructions from 1 to 17 bytes long
  – one operand must act as both a source and destination
  – one operand can come from memory
  – complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
• Saving grace:
  – the most frequently used instructions are not too difficult to build
  – compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”
• Registers in the 32-bit subset that originated with 80386
IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Register restrictions</th>
<th>MIPS equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register Indirect</td>
<td>Address Is in a register.</td>
<td>not ESP or EBP</td>
<td>lw $s0,0($s1)</td>
</tr>
<tr>
<td>Based mode with 8- or 32-bit</td>
<td>Address is contents of base register plus displacement.</td>
<td>not ESP or EBP</td>
<td>lw $s0,100($s1)#≤16-bit #displacement</td>
</tr>
<tr>
<td>displacement</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Base plus scaled index</td>
<td>The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.</td>
<td>Base: any GPR Index: not ESP</td>
<td>mil $t0,$s2,4</td>
</tr>
<tr>
<td>Base plus scaled index with</td>
<td></td>
<td></td>
<td>add $t0,$t0,$s1</td>
</tr>
<tr>
<td>8- or 32-bit displacement</td>
<td></td>
<td></td>
<td>lw $s0,0($t0)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE 2.42** IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register $s1$. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)
IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE name</td>
<td>if equal(condition code)(EIP=name): EIP-128 ≤ name ≤ EIP+128</td>
</tr>
<tr>
<td>JMP name</td>
<td>EIP=name</td>
</tr>
<tr>
<td>CALL name</td>
<td>SP=SP-4; M[SP]=EIP+5; EIP=name;</td>
</tr>
<tr>
<td>MOV EAX,[EDI+45]</td>
<td>EAX=MEEDI+45</td>
</tr>
<tr>
<td>PUSH ESI</td>
<td>SP=SP-4; M[SP]=ESI</td>
</tr>
<tr>
<td>POP EDI</td>
<td>EDI=M[SP]; SP=SP+4</td>
</tr>
<tr>
<td>ADD EAX,#6765</td>
<td>EAX=EAX+6765</td>
</tr>
<tr>
<td>TEST EDX,#42</td>
<td>Set condition code (flags) with EDX and 42</td>
</tr>
<tr>
<td>MOVSL</td>
<td>MEIDI=M[ESI]; EDI=EDI+4; ESi=ESI+4</td>
</tr>
</tbody>
</table>

**FIGURE 2.43 Some typical IA-32 instructions and their functions.** A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)
IA-32 instruction Formats

- Typical formats: (notice the different lengths)

- **a. JE EIP + displacement**
  
  JE | Condition | Displacement

- **b. CALL**
  
  CALL | Offset

- **c. MOV EBX, [EDI + 45]**
  
  MOV | d w r/m Postbyte | Displacement

- **d. PUSH ESI**
  
  PUSH | Reg

- **e. ADD EAX, #6765**
  
  ADD | Reg w | Immediate

- **f. TEST EDX, #42**
  
  TEST | w Postbyte | Immediate
Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate

- **Design Principles:**
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast

- Instruction set architecture
  - a very important abstraction indeed!
References

- Part of the slides of this lecture are adapted from http://www.capsl.udel.edu/courses/cpeg323/2004/
- Part of the slides of this lecture are adapted from the supplementary slides of the textbook