Systems Architecture I

Topics
MIPS Instruction Set*
Branching and Procedures in MIPS**

*This lecture was derived from material in the text (sec. 3.1-3.5).
**This lecture was derived from material in the text (sec. 3.5-3.6).

Notes Courtesy of Jeremy R. Johnson
Systems Architecture I

Topic 1: MIPS Instruction Set
Introduction

- Objective: To introduce the MIPS instruction set and to show how MIPS instructions are represented in the computer.

- The stored-program concept:
  - Instructions are represented as numbers.
  - Programs can be stored in memory to be read or written just like data.
Instructions

• Language of the machine

• More primitive than higher level languages (e.g. C, C++, Java)
  – e.g. no sophisticated control flow, primitive data structures

• MIPS (SGI, NEC, Nintendo) developed in the early 80’s (RISC)
  – Regular (32 bit instructions, small number of different formats)
  – Relatively small number of instructions
  – Register (all instructions operate on registers)
  – Load/Store (memory accessed only with load/store instructions, with few addressing modes)

• Design goals: maximize performance and minimize cost, reduce design time
MIPS Arithmetic

- All instructions have 3 operands
- Operand order is fixed (destination first)

- Example

  C code: \[ A = B + C \]

  MIPS code: `add $s0, $s1, $s2`

  (associated with variables by compiler)
Temporary Variables

- Regularity of instruction format requires that expressions get mapped to a sequence of binary operations with temporary results being stored in temporary variables.

- Example

  C code: 
  
  \[ f = (g + h) - (i + j); \]

  Assume: \( f, g, h, i, j \) in \( $s0 \) through \( $s4 \) respectively

  MIPS code: 
  
  ```
  add $t0, $s1, $s2 # $t0 = g+h
  add $t1, $s3, $s4 # $t1 = i+j
  sub $s0, $t0, $t1 # f = $t0 - $t1
  ```
Registers vs. Memory

- Operands for arithmetic instructions must be registers, — only 32 registers provided

- Compiler associates variables with registers

- When too many registers are used to simultaneously fit in 32 registers, the compiler must load and store temporary results to/from memory. The compiler tries to put the most frequently occurring variables in registers. Extra temporary variables must be “spilled” to memory.
Memory Organization

• Viewed as a large, single-dimension array, where a memory address is an index into the array

• In MIPS memory is "Byte addressing", which means that the index points to a byte of memory.

<table>
<thead>
<tr>
<th></th>
<th>8 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

...
Memory Organization

- Most data items are grouped into words (a MIPS word is 4 bytes)

  \[
  \begin{array}{|c|}
  \hline
  0 & 32 \text{ bits of data} \\
  4 & 32 \text{ bits of data} \\
  8 & 32 \text{ bits of data} \\
  12 & 32 \text{ bits of data} \\
  \vdots & \text{Registers hold 32 bits of data} \\
  \hline
  \end{array}
  \]

- \(2^{32}\) bytes with byte addresses from 0 to \(2^{32}-1\)
- \(2^{30}\) words with byte addresses 0, 4, 8, ... \(2^{32}-4\)
- Words are aligned (alignment restriction)
- bytes can be accessed from left to right (big endian) or right to left (little endian)
Load and Store

- All arithmetic instructions operate on registers
- Memory is accessed through load and store instructions

Example:


Assume that $s3$ contains the base address of $A$

MIPS code:  
\[
\begin{align*}
1w & \$t0, 32($s3) \\
add & \$t0, \$t0, \$s2 \\
sw & \$t0, 48($s3)
\end{align*}
\]

(Store word has destination last.)
Representing Instructions in the Computer

- MIPS instructions are 32 bits
  - opcode, operands

Example:

```
add $t0, $s0, $s1
```

$t0 = $8, $s0 = $16, $s1 = $17

```
000000 10000 10001 01000 00000 100000
```

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
</table>

```
Representing Instructions in the Computer

- Instruction formats:
- R format (register format - add, sub, ...)

```
| op | rs | rt | rd | shamt | func |
```

- I format (immediate format - lw, sw, ...)

```
| op | rs | rt | address |
```

- Example: lw $s1, 100($s2)

```
35  18  17  100
```
# Addressing in Branches and Jumps

- **J format format** (jump format – j, jal)

<table>
<thead>
<tr>
<th>op</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bits</td>
<td>26-bits</td>
</tr>
</tbody>
</table>

- **Example: j 10000**

<table>
<thead>
<tr>
<th>2</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-bits</td>
<td>26-bits</td>
</tr>
</tbody>
</table>
Design Principles

• Simplicity favors regularity
  – All instructions 32 bits
  – All instructions have 3 operands

• Smaller is faster
  – Only 32 registers

• Good design demands good compromises
  – All instructions are the same length
  – Limited number of instruction formats: R, I, J

• Make common cases fast
  – 16-bit immediate constant
  – Only two branch instructions
Systems Architecture I

Topic 2: Branching and Procedures in MIPS
Introduction

- Objective: To illustrate how programming constructs such as conditionals, loops and procedures can be translated into MIPS instructions.
Control Flow

- if statements, if-else statements
- Compiling an if-else statement

\[
\text{if (i == j) } f = g + h; \text{ else } f = g - h; \\
\]

Assume \( f \) through \( j \) in \( $s0 \) through \( $s4 \)

\[
\text{ bne } $s3, $s4, \text{ Else} \\
\text{ add } $s0, $s1, $s2 \\
\text{ j Exit} \\
\text{ Else: } \text{ sub } $s0, $s1, $s2 \\
\text{ Exit: } \\
\]
Compiling a Less Than Test

• Use slt instruction (only branch on equal or not equal)

if (a < b)
    ...

    slt $t0, $s0, $s1           # $t0 gets 1 if $s0 < $s1 0 otherwise
    bne $t0, $zero, Less       # goto Less if $t0 ≠ 0

• MIPS does not include a blt since it is “too complicated” and would increase the complexity of its implementation.
Loops

• Compiling a while loop (Assume i, j, k correspond to $s3, $s4, and $s5 and that the base address of save is in $s6)

while (save[i] == k)
    i = i + j;

Loop:
    add $t1, $s3, $s3  # $t1 = 2 \times i
    add $t1, $t1, $t1  # $t1 = 4 \times i
    add $t1, $t1, $s6  # address of save[i]
    lw $t0, 0($t1)     # save[i]
    bne $t0, $s5, Exit # goto Exit if save[i] ≠ k
    add $s3, $s3, $s4  # i = i + j
    j Loop

Exit:
Switch Statement

- Use a “jump table”

```
switch (k) {
    case 0:    f = i + j;   break;
    case 1:    f = g + h;   break;
    case 2:    f = g - h;   break;
    case 3:    f = i - j;   break;
}
```

- Variables f - k in $s0 - $s5, $t2 contains 4, $t4 contains address of jump table - JT

```
slt $t3, $s5, $zero # k < 0?
beq $t3, $zero, Exit
slt $t3, $s5, $t2 # k < 4?
beq $t3, $zero, Exit
```

```
add $t1, $s5, $s5
add $t1, $t1, $t1
add $t1, $t1, $t4 # addr JT[k]
lw $t0, 0($t1) # JT[k]
```

```
jr $t0
```

```
L0:  add $s0, $s3, $s4 j Exit
L1:  add $s0, $s1, $s2 j Exit
...  
```
Procedures

• In the execution of a procedure, the program must follow these steps:
  – Place parameters in a place where the procedure can access them
  – Transfer control to the procedure
  – Acquire the storage resources needed for the procedure
  – Perform the desired task
  – Place the result in a place where the calling program can access it
  – Return control to the point of origin
Registers for Procedure Calling and the jal Instruction

- $a0 - $a3: four argument registers in which to pass parameters
- $v0 - $v1: two value registers in which to return values
- $ra: one return address register to return to the point of origin
- jal ProcedureAddress: instruction to transfer control to a procedure and store the return address in $ra ($ra is set to PC + 4, address of the next instruction after procedure call)
- jr $ra - used to transfer control back to the calling program
Saving Registers using a Stack

• Additional registers used by the called procedure must be saved prior to use, or the values used by the calling procedure will be corrupted.

• The old values can be saved on a stack (call stack). After the called procedure completes, the old values can be popped off the stack and restored.

• $sp: stack pointer register contains the address of the top of the stack. By convention, address on the stack grows from higher addresses to lower address, which implies that a push subtracts from $sp and a pop adds to $sp.
Register Conventions

• The 8 “saved” registers $s0 - $s7 must be preserved on a procedure call, i.e. the called procedure must save these before using them.

• The 10 “temporary” registers $t0 - $t9 are not preserved by the called procedure. The calling procedure can not assume they will not change after a procedure call and, hence, must save them prior to the call if the values are needed after the call.

• Saved registers should be used for long lived variables, while temporary registers should be used for short lived variables
Nested Procedures and Automatic Variables

- A new call frame or activation record must be created for each nested procedure call.

- Argument registers and the return address register must be saved in addition to saved registers since new values will be put in them for the nested procedure call.

- Automatic variables (i.e. variables that are local to a procedure and are discarded when the procedure completes) are also allocated on the call stack. They are popped when the call completes.
Activation Records (Frames) and the Call Stack

- An activation record (frame) is a segment on the stack containing a procedure’s saved registers and local variables.
- Each time a procedure is called a frame ($fp$: frame pointer register points to the current frame) is placed on the stack.
Leaf Procedure

/* Example from page 134 */

int leaf_example (int g, int h, int l, int j)
{
    int f;
    f = (g+h) - (i+j);
    return f;
}
Leaf Procedure

```assembly
sub $sp,$sp,4  # push stack and save registers
sw $s0,0($sp)
add $t0,$a0,$a1  # g + h
add $t1,$a2,$a3  # i + j
sub $s0,$t0,$t1  # (g+h) - (i+j)
add $v0,$s0,$zero  # return f = (g+h)-(i+j)
lw $s0,0($sp)  # restore registers and pop stack
add $sp,$sp,4
jr $ra  # return to calling program
```
Recursive Procedure

/* Factorial example from pp. 136-137 */

int fact(int n)
{
    if (n < 1) return(1);
    else return(n * fact(n-1));
}
sub $sp,$sp,8     # push stack
sw $ra,4($sp)     # save return address
sw $a0,0($sp)    # save n

slt $t0,$a0,1     # test n < 1
beq $t0,$zero,L1 # branch if n >= 1
add $v0,$zero,1  # return 1
add $sp,$sp,8    # pop stack
jr $ra            # return to calling procedure

L1:
sub $a0,$a0,1     # set parameter to n-1
jal fact          # call fact(n-1)
lw $a0,0($sp)     # restore previous value of n
lw $ra,4($sp)     # restore previous return address
mul $v0,$a0,$v0   # return n * fact(n-1)

add $sp,$sp,8     # pop stack
jr $ra            # return to calling procedure