Systems Architecture

Lecture 6: Branching and Procedures in MIPS

Jeremy R. Johnson
Anatole D. Ruslanov
William M. Mongan

Some or all figures from Computer Organization and Design: The Hardware/Software Approach, Third Edition, by David Patterson and John Hennessy, are copyrighted material (COPYRIGHT 2004 MORGAN KAUFMANN PUBLISHERS, INC. ALL RIGHTS RESERVED).
Introduction

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

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially

- $\text{beq } rs, rt, L1$
  - if ($rs == rt$) branch to instruction labeled $L1$;

- $\text{bne } rs, rt, L1$
  - if ($rs != rt$) branch to instruction labeled $L1$;

- $\text{j } L1$
  - unconditional jump to instruction labeled $L1$
Control

• Decision making instructions
  – Alter the control flow – change the "next" instruction to be executed

• MIPS conditional branch instructions:

  bne $t0, $t1, Label
  beq $t0, $t1, Label

• Example: if (i==j) h = i + j;
  bne $s0, $s1, Label
  add $s3, $s0, $s1
  Label: ....
If-Then Structure

- MIPS unconditional branch instructions:
  - `j label`

- Example:

```plaintext
if (i!=j)  
  \( h = i + j \);  
  beq $s4, $s5, Equal
  add $s3, $s4, $s5
else  
  \( h = i - j \);  
  j GoOn
  Equal: sub $s3, $s4, $s5
  GoOn: ...
```
Compiling If Statements

- **C code:**
  
  ```
  if (i==j) f = g+h;
  else f = g-h;
  ```
  
  - f, g, ... in $s0, $s1, ...

- **Compiled MIPS code:**
  
  ```
  bne $s3, $s4, Else
  add $s0, $s1, $s2
  j Exit
  Else: sub $s0, $s1, $s2
  Exit: ...
  ```

- Assembler calculates addresses
Compiling Loop Statements

- C code:
  
  ```c
  while (save[i] == k) i += 1;
  ```
  
  - i in $s3, k in $s5, address of save in $s6

- Compiled MIPS code:
  
  ```mips
  Loop: sll $t1, $s3, 2
       add $t1, $t1, $s6
       lw $t0, 0($t1)
       bne $t0, $s5, Exit
       addi $s3, $s3, 1
       j Loop
  Exit: ...
  ```
Branch Instruction Design

• Why not blt, bge, etc?
Branch Instruction Design

• Why not `blt`, `bge`, etc?
• Hardware for `<`, `≥`, … slower than `=`, `≠`
  – Combining with branch involves more work per instruction, requiring a slower clock
  – All instructions penalized!
• `beq` and `bne` are the common case
• This is a good design compromise
Less Than Test

• What about Branch-if-less-than?
• New instruction:

\[
\text{if } \ s1 < s2 \ \text{then} \\
\quad \ s0 = 1 \\
\quad \ \text{slt } s0, s1, s2 \ \text{else} \ \\
\quad \ s0 = 0
\]

• MIPS does not include blt, bgt, ble, bge, bgz, etc. instructions because it is considered too complicated.
• Assembler pseudoinstruction: "\text{blt } s1, s2, Label"
• Note that the assembler needs a register to do this, — there are important conventions for registers use
More Conditional Operations

• Set result to 1 if a condition is true
  – Otherwise, set to 0

• `slt rd, rs, rt`
  – if (rs < rt) rd = 1; else rd = 0;

• `slti rt, rs, constant`
  – if (rs < constant) rt = 1; else rt = 0;

• Use in combination with `beq`, `bne`

  `slt $t0, $s1, $s2  # if ($s1 < $s2)`
  `bne $t0, $zero, L  # branch to L`
## Quick review

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add $s1,$s2,$s3</code></td>
<td><code>$s1 = $s2 + $s3</code></td>
</tr>
<tr>
<td><code>sub $s1,$s2,$s3</code></td>
<td><code>$s1 = $s2 - $s3</code></td>
</tr>
<tr>
<td><code>lw $s1,100($s2)</code></td>
<td><code>$s1 = Memory[$s2+100]</code></td>
</tr>
<tr>
<td><code>sw $s1,100($s2)</code></td>
<td><code>Memory[$s2+100] = $s1</code></td>
</tr>
<tr>
<td><code>bne $s4,$s5,Label</code></td>
<td>Jump to Label if <code>$s4$ne$s5</code></td>
</tr>
<tr>
<td><code>beq $s4,$s5,Label</code></td>
<td>Jump to Label if <code>$s4=&lt;$s5</code></td>
</tr>
<tr>
<td><code>j Label</code></td>
<td>Next instr. is at Label</td>
</tr>
<tr>
<td><code>slt $t0, $s1, $s2</code></td>
<td>Set <code>$t0 = 1</code> if <code>$s1 &lt; $s2</code> else set <code>$t0 = 0</code></td>
</tr>
</tbody>
</table>
Quick review

- **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>op</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>16 bit address</td>
</tr>
<tr>
<td>J</td>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26 bit address</td>
</tr>
</tbody>
</table>
Loops

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

while (save[i] == k)
    i += 1;

Loop: sll $t1, $s3, 2  # $t1 = 4 * i
    add $t1, $t1, $s6  # address of save[i]
    lw $t0, 0($t1)     # get save[i]
    bne $t0, $s5, Exit # goto Exit if save[i] ≠ k
    add $s3, $s3, 1    # i = i + 1
    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, $t4 contains address of jump table - JT

    slt $t3, $s5, $zero # k < 0?
    bne $t3,$zero, Exit
    slti $t3, $s5, 4      # 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 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 used 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 $sp$. 
## Policy of Use Conventions

<table>
<thead>
<tr>
<th>Name</th>
<th>Register number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>the constant value 0</td>
</tr>
<tr>
<td>$v0-$v1</td>
<td>2-3</td>
<td>values for results and expression evaluation</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>arguments</td>
</tr>
<tr>
<td>$t0-$t7</td>
<td>8-15</td>
<td>temporaries</td>
</tr>
<tr>
<td>$s0-$s7</td>
<td>16-23</td>
<td>saved</td>
</tr>
<tr>
<td>$t8-$t9</td>
<td>24-25</td>
<td>more temporaries</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address</td>
</tr>
</tbody>
</table>

Register 1 ($at) reserved for assembler, 26-27 for the operating system.
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.
Procedure 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.

<table>
<thead>
<tr>
<th>Low address</th>
<th>a.</th>
<th>b.</th>
<th>c.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fp$ arrow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$sp$ arrow</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- $fp$ points to the current frame.
- Saved argument registers (if any).
- Saved return address.
- Saved saved registers (if any).
- Local arrays and structures (if any).
Leaf Procedure

/* Example from page 134 */

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

Leaf Procedure

func: 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              # return to calling program
Memory Layout

• **Text:** program code
• **Static data:** global variables
  – e.g., static variables in C, constant arrays and strings
  – $gp$ initialized to address allowing ±offsets into this segment
• **Dynamic data:** heap
  – E.g., malloc in C, new in Java
• **Stack:** automatic storage

![Memory Layout Diagram]

- $sp \rightarrow 7fff \text{ fff}_{\text{hex}}$
- $gp \rightarrow 1000 \text{ 8000}_{\text{hex}}$
- $1000 \text{ 0000}_{\text{hex}}$
- $pc \rightarrow 0040 \text{ 0000}_{\text{hex}}$
- 0

Reserved

Static data

Dynamic data

Text

Stack
Character Data

• **Byte-encoded character sets**
  – ASCII: 128 characters
    • 95 graphic, 33 control
  – Latin-1: 256 characters
    • ASCII, +96 more graphic characters

• **Unicode: 32-bit character set**
  – Used in Java, C++ wide characters, ...
  – Most of the world’s alphabets, plus symbols
  – UTF-8, UTF-16: variable-length encodings
Byte/Halfword Operations

- Could use bitwise operations
- MIPS byte/halfword load/store
  - String processing is a common case

\[
\begin{align*}
\text{lb} & \ rt, \ offset(rs) \\
\text{lh} & \ rt, \ offset(rs) \\
\text{lbu} & \ rt, \ offset(rs) \\
\text{lhu} & \ rt, \ offset(rs) \\
\text{sb} & \ rt, \ offset(rs) \\
\text{sh} & \ rt, \ offset(rs)
\end{align*}
\]
  - Sign extend to 32 bits in rt
  - Zero extend to 32 bits in rt
  - Store just rightmost byte/halfword
String Copy Example

- C code (naïve):
  - Null-terminated string

```c
void strcpy (char x[], char y[])
{
    int i;
    i = 0;
    while ((x[i]=y[i])!='\0')
        i += 1;
}
```
  - Addresses of x, y in $a0, $a1
  - i in $s0
String Copy Example

- **MIPS code:**

```mips
strcpy:
    addi $sp, $sp, -4      # adjust stack for 1 item
    sw $s0, 0($sp)         # save $s0
    add $s0, $zero, $zero  # i = 0

L1: add $t1, $s0, $a1     # addr of y[i] in $t1
    lbu $t2, 0($t1)        # $t2 = y[i]
    add $t3, $s0, $a0     # addr of x[i] in $t3
    sb $t2, 0($t3)         # x[i] = y[i]
    beq $t2, $zero, L2    # exit loop if y[i] == 0
    addi $s0, $s0, 1       # i = i + 1
    j L1                   # next iteration of loop

L2: lw $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4       # pop 1 item from stack
    jr $ra                 # and return
```
Recursive Procedure

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

int fact(int n)
{
    if (n < 1) return(1);
    else return(n * fact(n-1));
}
fact:
    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
Arrays vs. Pointers

- **Array indexing involves**
  - Multiplying index by element size
  - Adding to array base address

- **Pointers correspond directly to memory addresses**
  - Can avoid indexing complexity
### Example: Clearing and Array

**clear1** (int array[], int size) {
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}

```assembly
move $t0,$zero  # i = 0
loop1: sll $t1,$t0,2  # $t1 = i * 4
    add $t2,$a0,$t1  # $t2 =
        # &array[i]
sw $zero, 0($t2)  # array[i] = 0
    addi $t0,$t0,1  # i = i + 1
    slt $t3,$t0,$a1  # $t3 =
        # (i < size)
    bne $t3,$zero,loop1  # if (...)
    # goto loop1
```

**clear2** (int *array, int size) {
    int *p;
    for (p = &array[0]; p < &array[size];
        p = p + 1)
        *p = 0;
}

```assembly
move $t0,$a0  # p = & array[0]
    sll $t1,$a1,2  # $t1 = size * 4
    add $t2,$a0,$t1  # $t2 =
        # &array[size]
loop2: sw $zero,0($t0)  # Memory[p] = 0
    addi $t0,$t0,4  # p = p + 4
    slt $t3,$t0,$t2  # $t3 =
        #(p<&array[size])
    bne $t3,$zero,loop2  # if (...)
    # goto loop2
```
Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer