C335
Computer Structures

MIPS Instructions (Part #6 - 1)

Dr. David R. Surma
Department of Computer and Information Sciences
Adapted from Morgan Kaufmann, Dr. L. Zhang and others
Programming Styles

- Procedures (subroutines, functions) allow the programmer to structure programs making them easier to understand and debug and allowing code to be reused.

- Procedures allow the programmer to concentrate on one portion of the code at a time.
  - Parameters act as barriers between the procedure and the rest of the program and data, allowing the procedure to accept passed values (arguments) and to return values (results).
Six Steps in Execution of a Procedure

- Main routine (caller) places parameters in a place where the procedure (callee) can access them
  - $a0 - a3$: four argument registers
- Caller transfers control to the callee
- Callee acquires the storage resources needed
- Callee performs the desired task
- Callee places the result value in a place where the caller can access it
  - $v0 - v1$: two value registers for result values
- Callee returns control to the caller
  - $ra$: one return address register to return to the point of origin
Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls.

**Register conventions:**

- Return address: $ra
- Arguments: $a0, $a1, $a2, $a3
- Return value: $v0, $v1
- Local variables: $s0, $s1, … , $s7

- The **stack** is also used; more later.
In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}

address
1000  add  $a0,$s0,$zero  # x = a; x in $a0
1004  add  $a1,$s1,$zero  # y = b; y in $a1
1008  addi $ra,$zero,1016  #$ra=1016
1012  j  sum  # jump to sum
1016  ...

2000  sum:  add  $v0,$a0,$a1
2004  jr  $ra  # return to the caller
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}

Question: Why use \texttt{jr} here? Why not simply use \texttt{j}?

Answer: \texttt{sum} might be called by many places, so we can’t return to a fixed place. The calling proc to \texttt{sum} must be able to say “return here” somehow.

\begin{verbatim}
2000  sum: add $v0,$a0,$a1
2004  jr   $ra  # return to the caller
\end{verbatim}
Instruction Support for Functions (4/5)

- Single instruction to jump and save return address: “jump and link” (jal)

- Before:

  1008 addi $ra,$zero,1016 #$ra=1016
  1012 j sum #goto sum

- After (using jal):

  1008 jal sum # $ra=1012,goto sum

- Why have a jal?

  - Make the common case fast: function calls are very common. (Also, you don’t have to know where the code is loaded into memory with jal.)
Instruction Support for Functions (5/5)

- Syntax for `jal` (jump and link) is same as for `j` (jump):
  
  \[
  \text{jal} \quad \text{label}
  \]

- `jal` *should really be called* `laj` for “link and jump”:
  - Step 1 (link): Save address of *next* instruction into \$ra (Why next instruction? Why not current one?)
  - Step 2 (jump): Jump to the given label
Spilling Registers

- What if the callee needs to use more registers than allocated to argument and return values?
  - It uses a stack – a “last-in-first-out” data structure

- One of the general registers, $sp ($29), is used to address the stack (which “grows” from high address to low address)

- Add data onto the stack – push
  \[ $sp = sp - 4 \]
  Store the data on stack at new $sp (a pre-decrement operation)

- Remove data from the stack – pop
  \[ $sp = sp + 4 \]
  Load the data from stack at $sp (a post-increment operation)
Compiling a C Leaf Procedure

- **Leaf** procedures are ones that do not call other procedures. Give the MIPS assembler code for

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

where \( g, h, i, \) and \( j \) are in \( $a0, $a1, $a2, $a3 \)

```mips
leaf_ex:    addi $sp,$sp,-8   #make stack room
            sw $s0,4($sp)    #save $s0 on stack
            sw $s1,0($sp)   #save $s1 on stack
            add $s0,$a0,$a1
            add $s1,$a2,$a3
            sub $s1,$s0,$s1
            add $v0, $s1,$zero  #$v0=$s1
            lw $s1,0($sp)     #restore $s1
            lw $s0,4($sp)     #restore $s0
            addi $sp,$sp,8    #adjust stack ptr
            jr $ra
```
Compiling a C Leaf Procedure

- **Key items:**

  ```
  leaf_ex:  addi $sp,$sp,-8  #make stack room
           sw $s0,4($sp)  #save $s0 on stack
           sw $s1,0($sp)  #save $s1 on stack
           add $s0,$a0,$a1
           add $s1,$a2,$a3
           sub $s1,$s0,$s1
           add $v0, $s1,$zero  #$v0=$s1
           lw $s1,0($sp)  #restore $s1
           lw $s0,4($sp)  #restore $s0
           addi $sp,$sp,8  #adjust stack ptr
           jr $ra
  ```

- Important to “leave things as we found them”
- $s0 and $s1 are being used as local variables. Original contents must be restored - why?
- sp must be restored to original contents
int sumSquare(int x, int y) {
    return mult(x, x) + y;
}

- Something called \textit{sumSquare}, now \textit{sumSquare} is calling procedure \textit{mult}.

- So there’s a value in $\text{ra}$ that \textit{sumSquare} wants to jump back to, but this will be overwritten by the call to \textit{mult}.

- Need to save \textit{sumSquare} return address before call to \textit{mult}. 
Nested Procedures (2/2)

- In general, may need to save some other info in addition to $\text{ra}$.

- When a C program is run, there are 3 important memory areas allocated:
  - **Static**: Variables declared once per program, cease to exist only after execution completes. (C global variables).
  - **Heap**: Variables declared dynamically
  - **Stack**: Space to be used by procedure during execution; this is where we can save register values
MIPS Memory Allocation for Program and Data

- **Static data**
  - 0x 1000 8000
  - 0x 1000 0000

- **Dynamic data (heap)**
  - 0x 1000 0000

- **Stack**
  - 0x 7f f f f f c

- **Text (Your code)**
  - 0x 0040 0000

- **Reserved**
  - 0x 0000 0000

- **PC**
- **$gp**
- **$sp**
Procedure Call and Stack

Stacking of Subroutine Calls & Returns and Environments:

A: CALL B
  B: CALL C
     C: RET
          RET

A B
A B C
A B
A