CS61C – Machine Structures

Lecture 12 - MIPS Procedures II & Logical Ops

2/13/2006

John Wawrzynek

(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/

Review

- Functions called with `jal`, return with `jr $ra`.
- The stack is your friend: Use it to save anything you need. Just be sure to leave it the way you found it.
- Instructions we know so far
  - Arithmetic: `add, addi, sub, addu, addiu, subu`
  - Memory: `lw, sw`
  - Decision: `beq, bne, slt, slti, sltu, sltiu`
  - Unconditional Branches (Jumps): `j, jal, jr`
- Registers we know so far
  - All of them!
  - There are CONVENTIONS when calling procedures!

Register Conventions (1/4)

- **CalleR: the calling function**
- **CalleE: the function being called**
- **When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged.**
- **Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (`jal`) and which may be changed.**

Register Conventions (2/4) - saved

- **`$0`: No Change. Always 0.**
- **`$s0`-`$s7`: Restore if you change. Very important, that's why they're called saved registers. If the callee changes these in any way, it must restore the original values before returning.**
- **`$sp`: Restore if you change. The stack pointer must point to the same place before and after the `jal` call, or else the caller won't be able to restore values from the stack.**
- **HINT -- All saved registers start with S!**

Register Conventions (3/4) - volatile

- `$ra`: Can Change. The `jal` call itself will change this register. Caller needs to save on stack if nested call.
- `$v0`-`$v1`: Can Change. These will contain the new returned values.
- `$a0`-`$a3`: Can change. These are volatile argument registers. Caller needs to save if they'll need them after the call.
- `$t0`-`$t9`: Can change. That's why they're called temporary: any procedure may change them at any time. Caller needs to save if they'll need them afterwards.

Register Conventions (4/4)

- What do these conventions mean?
  - If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a `jal` call.
  - Function E must save any S (saved) registers it intends to use before garbling up their values
  - Remember: Caller/callee need to save only temporary/saved registers they are using, not all registers.

Administrivia

- Midterm Exam I
  - Friday 2/24 6-8pm, 1 Pimentel

    (2 weeks from today)
  - Review Session TBA
- Project 2 due earlier that week
  - Tuesday 2/21 11:59pm
- HW4 due next Wednesday
- No HW due 2/22

Example: Fibonacci Numbers 1/8

- The Fibonacci numbers are defined as follows: F(n) = F(n – 1) + F(n – 2), F(0) and F(1) are defined to be 1
- In scheme, this could be written:

```
(define (Fib n)
 (cond   ((= n 0) 1)
         ((= n 1) 1)
         (else (+ (Fib (- n 1))
                  (Fib (- n 2)))))
```

Example: Fibonacci Numbers 2/8

- Rewriting this in C we have:

```
int fib(int n) {
  if(n == 0) { return 1; }
  if(n == 1) { return 1; }
  return (fib(n - 1) + fib(n - 2));
}
```


Example: Fibonacci Numbers 3/8

- Now, let’s translate this to MIPS!
- You will need space for three words on the stack
- The function will use one $s register, $s0
- Write the Prologue:

```
fib:
addi $sp, $sp, -12   # Space for three words
sw $ra, 8($sp)       # Save return address
sw $s0, 4($sp)       # Save s0
```

Example: Fibonacci Numbers 4/8

° Now write the Epilogue:

```
fin:
lw $s0, 4($sp)          # Restore $s0
lw $ra, 8($sp)          # Restore return address
addi $sp, $sp, 12       # Pop the stack frame
jr $ra                  # Return to caller
```


Example: Fibonacci Numbers 5/8

° Finally, write the body. The C code is below. Start by translating the lines indicated in the comments

```
int fib(int n) {
 if(n == 0) { return 1; } /*Translate Me!*/
 if(n == 1) { return 1; } /*Translate Me!*/
 return (fib(n - 1) + fib(n - 2));
}
  addi $v0, $zero, 1     # $v0 = 1
  beq  $a0, $zero, fin   #
  addi $t0, $zero, 1     # $t0 = 1
  beq  $a0, $t0, fin     #
  Continued on next slide. . .
```

Example: Fibonacci Numbers 6/8

- Almost there, but be careful, this part is tricky!

```
int fib(int n) {
 . . .
 return (fib(n - 1) + fib(n - 2));
}
```

```
 addi $a0, $a0, -1      # $a0 = n - 1
 sw $a0, 0($sp)         # Need $a0 after jal
 jal fib                # fib(n - 1)
 lw $a0, 0($sp)         # restore $a0
 addi $a0, $a0, -1      # $a0 = n - 2
```


Example: Fibonacci Numbers 7/8

- Remember that $v0 is caller saved!

```
int fib(int n) {
 . . .
 return (fib(n - 1) + fib(n - 2));
}
```

```
add $s0, $v0, $zero   # Place fib(n – 1)
                      # somewhere it won't get
                      # clobbered
jal fib               # fib(n - 2)
add $v0, $v0, $s0     # $v0 = fib(n-1) + fib(n-2)
To the epilogue and beyond. . .
```

Example: Fibonacci Numbers 8/8

° Here's the complete code for reference:

```
fib:  addi $sp, $sp, -12
      sw $ra, 8($sp)
      sw $s0, 4($sp)
      addi $v0, $zero, 1
      beq $a0, $zero, fin
      addi $t0, $zero, 1
      beq $a0, $t0, fin
      addi $a0, $a0, -1
      sw $a0, 0($sp)
      jal fib
      lw $a0, 0($sp)
      addi $a0, $a0, -1
      add $s0, $v0, $zero
      jal fib
      add $v0, $v0, $s0
fin:  lw $s0, 4($sp)
      lw $ra, 8($sp)
      addi $sp, $sp, 12
      jr $ra
```


Bitwise Operations

° Up until now, we've done arithmetic (`add`, `sub`,`addi`), memory access (`lw` and `sw`), and branches and jumps.

° All of these instructions view contents of register as a single quantity (such as a signed or unsigned integer)

° New Perspective: View register as 32 raw bits rather than as a single 32-bit number

° Since registers are composed of 32 bits, we may want to access individual bits (or groups of bits) rather than the whole.

° Introduce two new classes of instructions:
- Logical & Shift Ops

Logical Operators (1/3)

- Two basic logical operators:
  - AND: outputs 1 only if both inputs are 1
  - OR: outputs 1 if at least one input is 1
- Truth Table: standard table listing all possible combinations of inputs and resultant output for each. E.g.,

A	B	A AND B	A OR B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Logical Operators (2/3)

- Logical Instruction Syntax:

  1 2,3,4

  - where

    1) operation name

    2) register that will receive value

    3) first operand (register)

    4) second operand (register) or immediate (numerical constant)
- In general, can define them to accept > 2 inputs, but in the case of MIPS assembly, these accept exactly 2 inputs and produce 1 output
  - Again, rigid syntax, simpler hardware

Logical Operators (3/3)

- Instruction Names:
  - `and`, `or`: Both of these expect the third argument to be a register
  - `andi`, `ori`: Both of these expect the third argument to be an immediate
- MIPS Logical Operators are all bitwise, meaning that bit 0 of the output is produced by the respective bit 0's of the inputs, bit 1 by the bit 1's, etc.
  - C: Bitwise AND is `&` (e.g., `z = x & y;`)
  - C: Bitwise OR is `|` (e.g., `z = x | y;`)

Uses for Logical Operators (1/3)

- Note that `and`ing a bit with 0 produces a 0 at the output while `and`ing a bit with 1 produces the original bit.
- This can be used to create a mask.
  - Example:

      1011 0110 1010 0100 0011 | 1101 1001 1010

  mask: 0000 0000 0000 0000 0000 | 1111 1111 1111

  - The result of `and`ing these:

      0000 0000 0000 0000 0000 | 1101 1001 1010

  mask last 12 bits

Uses for Logical Operators (2/3)

- **The second bitstring in the example is called a mask. It is used to isolate the rightmost 12 bits of the first bitstring by masking out the rest of the string (e.g. setting it to all 0s).**
- **Thus, the `and` operator can be used to set certain portions of a bitstring to 0s, while leaving the rest alone.**
  - **In particular, if the first bitstring in the above example were in `$t0`, then the following instruction would mask it:**

```
andi   $t0,$t0,0xFFF
```


Uses for Logical Operators (3/3)

- **Similarly, note that `or`ing a bit with 1 produces a 1 at the output while `or`ing a bit with 0 produces the original bit.**
- **This can be used to force certain bits of a string to 1s.**
  - **For example, if `$t0` contains `0x12345678`, then after this instruction:**

```
ori  $t0, $t0, 0xFFFF
```

  - **… `$t0` contains `0x1234FFFF` (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).**

Shift Instructions (review) (1/4)

- Move (shift) all the bits in a word to the left or right by a number of bits.
  - Example: shift right by 8 bits

    0001 0010 0011 0100 0101 0110 0111 1000

    0000 0000 0001 0010 0011 0100 0101 0110
  - Example: shift left by 8 bits

    0001 0010 0011 0100 0101 0110 0111 1000

    0011 0100 0101 0110 0111 1000 0000 0000

Shift Instructions (2/4)

- Shift Instruction Syntax:

  1   2,3,4
  - where

    1) operation name

    2) register that will receive value

    3) first operand (register)

    4) shift amount (constant < 32)
- MIPS shift instructions:
  1. `sll` (shift left logical): shifts left and fills emptied bits with 0s
  2. `srl` (shift right logical): shifts right and fills emptied bits with 0s
  3. `sra` (shift right arithmetic): shifts right and fills emptied bits by sign extending

Shift Instructions (3/4)

- Example: shift right arith by 8 bits

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110

- Example: shift right arith by 8 bits

1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

Shift Instructions (4/4)

- Since shifting may be faster than multiplication, a good compiler usually notices when C code multiplies by a power of 2 and compiles it to a shift instruction:

  `a *= 8;` (in C)

  would compile to:

  `sll   $s0,$s0,3` (in MIPS)

- Likewise, shift right to divide by powers of 2 (rounds towards -∞)
  - remember to use `sra`

Uses for Shift Instructions (1/4)

- Suppose we want to isolate byte 0 (rightmost 8 bits) of a word in $t0. Simply use:

```
andi   $t0,$t0,0xFF
```

- Suppose we want to isolate byte 1 (bit 15 to bit 8) of a word in $t0. We can use:

```
andi   $t0,$t0,0xFF00
```

but then we still need to shift to the right by 8 bits...

Uses for Shift Instructions (2/4)

- Could use instead:

```
sll   $t0,$t0,16
srl   $t0,$t0,24
```

0001 0010 0011 0100 0101 0110 0111 1000

0101 0110 0111 1000 0000 0000 0000 0000

0000 0000 0000 0000 0000 0000 0101 0110

Uses for Shift Instructions (3/4)

- In decimal:
  - Multiplying by 10 is same as shifting left by 1:
    - $714_{10} \times 10_{10} = 7140_{10}$
    - $56_{10} \times 10_{10} = 560_{10}$
  - Multiplying by 100 is same as shifting left by 2:
    - $714_{10} \times 100_{10} = 71400_{10}$
    - $56_{10} \times 100_{10} = 5600_{10}$
  - Multiplying by 10^n is same as shifting left by n

Uses for Shift Instructions (4/4)

- In binary:
  - Multiplying by 2 is same as shifting left by 1:
    - $11_2 \times 10_2 = 110_2$
    - $1010_2 \times 10_2 = 10100_2$
  - Multiplying by 4 is same as shifting left by 2:
    - $11_2 \times 100_2 = 1100_2$
    - $1010_2 \times 100_2 = 101000_2$
  - Multiplying by 2^n is same as shifting left by n

“And in Conclusion…”

- **Register Conventions: Each register has a purpose and limits to its usage. Learn these and follow them, even if you’re writing all the code yourself.**
- **Logical and Shift Instructions**
    - **Operate on bits individually, unlike arithmetic, which operate on entire word.**
    - **Use to isolate fields, either by masking or by shifting back and forth.**
    - **Use shift left logical, `sll`, for multiplication by powers of 2**
    - **Use shift right arithmetic, `sra`, for division by powers of 2.**
- **New Instructions:**
  `and,andi, or,ori, sll,srl,sra`