CS61C : Machine Structures

Lecture 2.2.2

MIPS Part III: Procedures

2004-07-01

Kurt Meinz

`inst.eecs.berkeley.edu/~cs61c`

Topic Outline

- Functions
- More Logical Operations

Function Call Bookkeeping

- What are the properties of a function?
  - Function call transfers control somewhere else and then returns.
  - Arguments
  - Return Value
  - Black-box operation/scoping
  - Re-entrancy

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.

Instruction Support for Functions (1/6)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
   return x+y;
}
```

MIPS

```
address
 1000
 1004
 1008
 1012
 1016

 2000
 2004
```

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.

Instruction Support for Functions (2/6)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
   return x+y;
}
```

MIPS

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

 2000 sum: add $v0,$a0,$a1
 2004 jr   $ra # new instruction
```

Instruction Support for Functions (4/6)

- Single instruction to jump and save return address: jump and link (`jal`)
- **Before**:

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum               #go to sum
```

- **After**:

```
1008 jal sum  # $ra=1012,go to 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 (5/6)

- Syntax for `jal` (jump and link) is same as for `j` (jump):

```
jal 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

Instruction Support for Functions (6/6)

- Syntax for `jr` (jump register):

```
jr register
```

- Instead of providing a label to jump to, the `jr` instruction provides a register which contains an address to jump to.
- Only useful if we know exact address to jump to.
- Very useful for function calls:
  - `jal` stores return address in register (`$ra`)
  - `jr $ra` jumps back to that address

Nested Procedures (1/2)

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

- Something called `sumSquare`, now `sumSquare` is calling `mult`.
- So there's a value in `$ra` that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`.
- Need to save `sumSquare` return address before call to `mult`.

Nested Procedures (2/2)

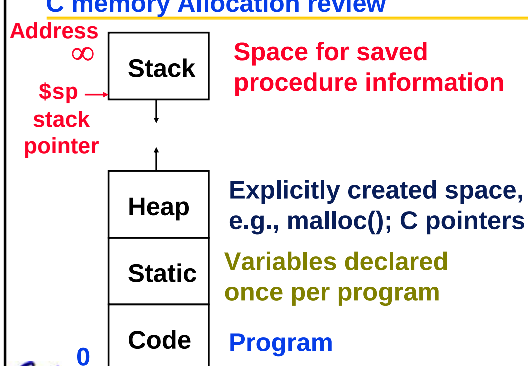
- Also need to save any registers that are needed across the procedure call:

```
int fact(int a) {
 if (a == 0) {
  return 1;
 } else {
  return a * fact(a-1);
 }
}
```

```
fact:  addi $v0 $0 1
       beq $a0 $0 done
       add $s0 $a0 $0
       addi $a0 $a0 -1
       jal fact
       mul* $v0 $s0 $v0
done:     jr $ra
```

Why won't our factorial work?

C memory Allocation review



Using the Stack (1/2)

- So we have a register `$sp` which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```
int sumSquare(int x, int y) {
 return mult(x,x)+ y;
}
```


Using the Stack (2/2)

- Hand-compile `int sumSquare(int x, int y) { return mult(x,x)+ y; }`

```
sumSquare:
"push"  addi $sp,$sp,-8  # space on stack
        sw $ra, 4($sp)   # save ret addr
        sw $a1, 0($sp)   # save y

        add $a1,$a0,$zero # mult(x,x)
        jal mult          # call mult

        lw $a1, 0($sp)   # restore y
        add $v0,$v0,$a1  # mult()+y
        lw $ra, 4($sp)   # get ret addr
"pop"   addi $sp,$sp,8   # restore stack
        jr $ra
mult: ...
```


Steps for Making a Procedure Call

1) Save necessary values onto stack.

2) Assign argument(s), if any.

3) `jal` call

4) Restore values from stack.

Rules for Procedures

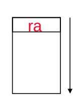
- Called with a `jal` instruction, returns with a `jr $ra`
- Accepts up to 4 arguments in $a0, $a1, $a2 and $a3
- Return value is always in $v0 (and if necessary in $v1)
- Must follow register conventions (even in functions that only you will call)! So what are they?

Basic Structure of a Function

Prologue

```
entry_label:
addi $sp,$sp, -framesize
sw $ra, framesize-4($sp)  # save $ra
save other regs if need be
```

Body ··· (call other functions…)



Epilogue

```
restore other regs if need be
lw $ra, framesize-4($sp)  # restore $ra
addi $sp,$sp, framesize
jr $ra
```


MIPS Registers

The constant 0	$0	$zero
Reserved for Assembler	$1	$at
Return Values	$2-$3	$v0-$v1
Arguments	$4-$7	$a0-$a3
Temporary	$8-$15	$t0-$t7
Saved	$16-$23	$s0-$s7
More Temporary	$24-$25	$t8-$t9
Used by Kernel	$26-27	$k0-$k1
Global Pointer	$28	$gp
Stack Pointer	$29	$sp
Frame Pointer	$30	$fp
Return Address	$31	$ra

(From COD 2nd Ed. p. A-23)
Use names for registers -- code is clearer!

Other Registers

- $at: may be used by the assembler at any time; unsafe to use
- $k0-$k1: may be used by the OS at any time; unsafe to use
- $gp, $fp: don't worry about them
- Note: Feel free to read up on $gp and $fp in Appendix A, but you can write perfectly good MIPS code without them.

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.

Break Time!

```
r: ...     # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   ...     ### PUSH REGISTER(S) TO STACK?
   jal e   # Call e
   ...     # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   jr $ra  # Return to caller of r

e: ...     # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   jr $ra  # Return to r
```

What does `r` have to push on the stack before "`jal e`"?

s0? $sp? $v0? $t0? $a0? $ra?

Topic Outline

- Functions
- More Logical Operations

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 contents of 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 anding a bit with 0 produces a 0 at the output while anding 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 anding 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 (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
  - remember to use `sra`

Bonus Example: Compile This (1/5)

```
main() {
 int i,j,k,m; /* i-m:$s0-$s3 */
 ...
 i = mult(j,k); ...
 m = mult(i,i); ...
}

int mult (int mcand, int mlier){
 int product;

 product = 0;
 while (mlier > 0)  {
  product += mcand;
  mlier -= 1; }
 return product;
 }
```


Bonus Example: Compile This (2/5)

```
__start:

 add $a0,$s1,$0     # arg0 = j
 add $a1,$s2,$0     # arg1 = k
 jal mult           # call mult
 add $s0,$v0,$0     # i = mult()
 ...
 add $a0,$s0,$0     # arg0 = i
 add $a1,$s0,$0     # arg1 = i
 jal mult           # call mult
 add $s3,$v0,$0     # m = mult()
 ...
done
```

```
main() {
int i,j,k,m; /* i-m:$s0-$s3 */
...
i = mult(j,k); ...
m = mult(i,i); ... }
```


Bonus Example: Compile This (3/5)

- Notes:
  - `main` function ends with done, not `jr $ra`, so there's no need to save $ra onto stack
  - all variables used in `main` function are saved registers, so there's no need to save these onto stack

Bonus Example: Compile This (4/5)

```
mult:
     add  $t0,$0,$0      # prod=0
Loop:
     slt  $t1,$0,$a1     # mlr > 0?
     beq  $t1,$0,Fin     # no=>Fin
     add  $t0,$t0,$a0    # prod+=mc
     addi $a1,$a1,-1     # mlr-=1
     j    Loop           # goto Loop
Fin:
     add  $v0,$t0,$0     # $v0=prod
     jr   $ra            # return
```

```
int mult (int mcand, int mlier){
int product = 0;
while (mlier > 0)  {
 product += mcand;
 mlier -= 1; }
return product;
}
```


Bonus Example: Compile This (5/5)

- Notes:
  - no `jal` calls are made from `mult` and we don't use any saved registers, so we don't need to save anything onto stack
  - temp registers are used for intermediate calculations (could have used s registers, but would have to save the caller's on the stack.)
  - `$a1` is modified directly (instead of copying into a temp register) since we are free to change it
  - result is put into $v0 before returning (could also have modified `$v0` directly)

"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`