inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture 11 – Introduction to MIPS Procedures I

2004-09-24

Lecturer PSOE Dan Garcia

www.cs.berkeley.edu/~ddgarcia

World's highest res! ⇒

2.2 inch VGA res LCD display @ 368 pixels per inch (ppi), and might be on cell phones soon…

www.i4u.com/article2157.html

Review

- In order to help the conditional branches make decisions concerning inequalities, we introduce a single instruction: "Set on Less Than"called `slt`, `slti`, `sltu`, `sltiu`
- One can store and load (signed and unsigned) bytes as well as words
- Unsigned add/sub don't cause overflow
- New MIPS Instructions:

```
sll, srl
slt, slti, sltu, sltiu
addu, addiu, subu
```

Example: The C Switch Statement (3/3)

- **Final compiled MIPS code:**

```
      bne $s5,$0,L1       # branch k!=0
      add  $s0,$s3,$s4 #k==0 so f=i+j
      j    Exit           # end of case so Exit
L1:   addi $t0,$s5,-1     # $t0=k-1
      bne  $t0,$0,L2      # branch k!=1
      add  $s0,$s1,$s2 #k==1 so f=g+h
      j    Exit           # end of case so Exit
L2:   addi $t0,$s5,-2     # $t0=k-2
      bne  $t0,$0,L3      # branch k!=2
      sub  $s0,$s1,$s2 #k==2 so f=g-h
      j    Exit           # end of case so Exit
L3:   addi $t0,$s5,-3     # $t0=k-3
      bne  $t0,$0,Exit # branch k!=3
      sub  $s0,$s3,$s4 #k==3 so f=i-j
Exit:
```

Removing breaks does NOT translate to removing jumps… (my bad)

C functions

```
main() {
 int i,j,k,m;
 ...
 i = mult(j,k); ...
 m = mult(i,i); ...

}

/* really dumb mult function */

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

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

What information must compiler/programmer keep track of?

What instructions can accomplish this?

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 (3/6)

C

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

MIPS

- Question: Why use `jr` here? Why not simply use `j`?
- Answer: `sum` might be called by many functions, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

```
0 sum: add $v0,$a0,$a1
4      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               #goto sum
```

- After:

```
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 (5/6)

- 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

Administrivia

- Newsgroup growing out of control…
  - Read postings before posting!
  - Read `Errata.txt` for each project/hw before posting
- Project 1 out (make sure to work on it this weekend), due next Friday
  - An easy HW4 will follow, due Wed after
- UCB Programming contest tomorrow from 1000 - 1530 in 306 Soda!
  - If you partake, EPA! Points! + 2 slip days
  - `www.cs/~hilfingr/programming-contest`
- Dan's videos:

`www.siggraph.org/publications/video-review/SVR.html`

Nested Procedures (1/2)

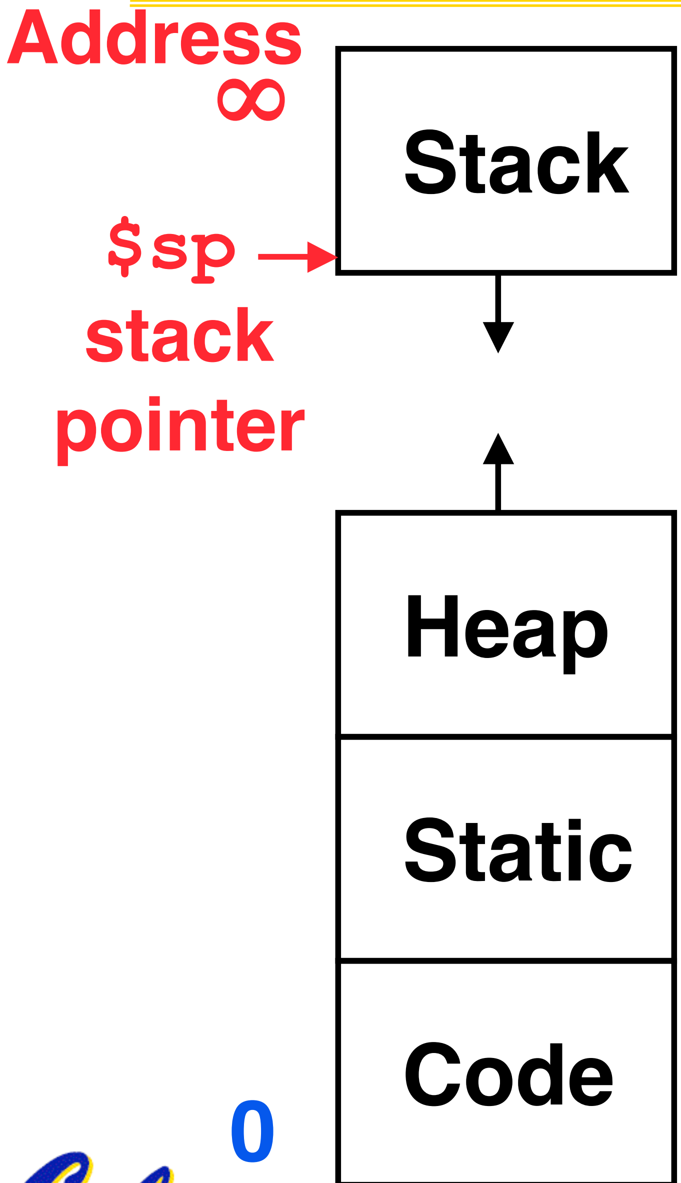
```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

- 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)

- In general, may need to save some other info in addition to `$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. E.g., C globals
  - Heap: Variables declared dynamically
  - Stack: Space to be used by procedure during execution; this is where we can save register values

C memory Allocation review



Space for saved procedure information

Explicitly created space, e.g., malloc(); C pointers

Variables declared once per program

Program

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

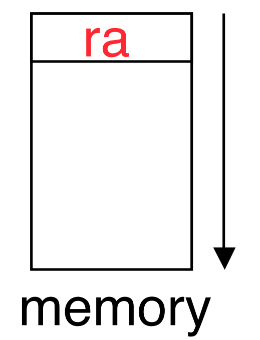
- Called with a `jal` instruction, returns with a `jr $ra`
- Accepts up to 4 arguments in `$a0`, `$a1`, `$a2` and `$a3`
- Return value is always in `$v0` (and if necessary in `$v1`)
- Must follow register conventions (even in functions that only you will call)! So what are they?

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 3rd Ed. green insert)

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.

Peer Instruction

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

When translating this to MIPS…

A. We COULD copy `$a0` to `$a1` (& then not store `$a0` or `$a1` on the stack) to store `n` across recursive calls.

B. We MUST save `$a0` on the stack since it gets changed.

C. We MUST save `$ra` on the stack since we need to know where to return to…

	ABC
1:	FFF
2:	FFT
3:	FTF
4:	FTT
5:	TFF
6:	TFT
7:	TTF
8:	TTT

"And in Conclusion..."

- **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!**