inst.eecs.berkeley.edu/~cs61c

CS61C : Machine Structures

Lecture #8 – Decisions (cont), Procedures

2008-07-03

Albert Chae, Instructor

Review

- Memory transfer: `lw, sw, lb, sb`
- A Decision allows us to decide what to execute at run-time rather than compile-time.
- C Decisions are made using conditional statements within `if`, `while`, `do while`, `for`.
- MIPS Decision making instructions are the conditional branches: `beq` and `bne`.
- Unsigned add/sub don't cause overflow
- New MIPS Instructions:
  `beq, bne, j, sll, srl`
  `addu, addiu, subu,`

Loops Review

- Key Concept: Though there are multiple ways of writing a loop in MIPS, the key to decision making is conditional branch

```
do {g= g + A[i];
 i= i + j;
 } while (i != h);
```

```
Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
```

Inequalities in MIPS (1/3)

- Until now, we've only tested equalities (== and != in C). General programs need to test < and > as well.
- Create a MIPS Inequality Instruction:
  - "Set on Less Than"
  - Syntax: `slt reg1,reg2,reg3`
  - Meaning: `reg1 = (reg2 < reg3);`

```
if (reg2 < reg3)
     reg1 = 1;
else reg1 = 0;
```

Same thing…

  - In computereeze, "set" means "set to 1", "reset" means "set to 0".

Inequalities in MIPS (2/3)

- How do we use this? Compile by hand:

```
if (g < h) goto Less; #g:$s0, h:$s1
```

- Answer: compiled MIPS code…

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
bne $t0,$0,Less # goto Less
                # if $t0!=0
                # (if (g<h)) Less:
```

- Branch if `$t0 != 0` ➔ `(g < h)`
- Register `$0` always contains the value `0`, so `bne` and `beq` often use it for comparison after an `slt` instruction.
- A `slt` ➔ `bne` pair means `if(… < …)goto…`

Inequalities in MIPS (3/3)

- Now, we can implement <, but how do we implement >, ≤ and ≥ ?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can we implement ≤ in one or more instructions using just `slt` and the branches?
- What about >?
- What about ≥?

Immediates in Inequalities

- There is also an immediate version of `slt` to test against constants: `slti`
  - Helpful in `for` loops

C

```
    if (g >= 1) goto Loop
```

MIPS

```
Loop: . . .

slti $t0,$s0,1      # $t0 = 1 if
                    # $s0<1 (g<1)
beq  $t0,$0,Loop    # goto Loop
                    # if $t0==0
                    # (if (g>=1))
```

What about unsigned numbers?

- Also unsigned inequality instructions:

  `sltu`, `sltiu`

…which sets result to 1 or 0 depending on unsigned comparisons

- What is value of `$t0`, `$t1`?

(`$s0 = FFFF FFFA`hex, `$s1 = 0000 FFFA`hex)

```
slt $t0, $s0, $s1
sltu $t1, $s0, $s1
```

MIPS Signed vs. Unsigned – diff meanings!

- MIPS Signed v. Unsigned is an “overloaded” term
  - Do/Don't sign extend (`lb`, `lbu`)
  - Don't overflow (`addu`, `addiu`, `subu`, `multu`, `divu`)
  - Do signed/unsigned compare (`slt`, `slti`/`sltu`, `sltiu`)

Example: The C Switch Statement (1/3)

- Choose among four alternatives depending on whether `k` has the value 0, 1, 2 or 3. Compile this C code:

```
switch (k) {
  case 0: f=i+j; break; /* k=0 */
  case 1: f=g+h; break; /* k=1 */
  case 2: f=g-h; break; /* k=2 */
  case 3: f=i-j; break; /* k=3 */
}
```

Example: The C Switch Statement (2/3)

- This is complicated, so simplify.
- Rewrite it as a chain of if-else statements, which we already know how to compile:

```
if(k==0) f=i+j;
  else if(k==1) f=g+h;
    else if(k==2) f=g-h;
      else if(k==3) f=i-j;
```

- Use this mapping:

```
f:$s0, g:$s1, h:$s2,
i:$s3, j:$s4, k:$s5
```

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:
```

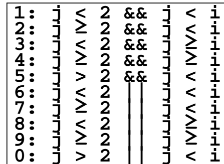
Peer Instruction

```
Loop:addi $s0,$s0,-1      # i = i - 1
     slti $t0,$s1,2       # $t0 = (j < 2)
     beq  $t0,$0 ,Loop    # goto Loop if $t0 == 0
     slt  $t0,$s1,$s0     # $t0 = (j < i)
     bne  $t0,$0 ,Loop    # goto Loop if $t0 != 0
```

($s0=i, $s1=j)

What C code properly fills in the blank in loop below?

`do {i--;} while(__);`



```
1: j < 2 && j < i
2: j ≥ 2 && j < i
3: j < 2 && j ≥ i
4: j ≥ 2 && j ≥ i
5: j > 2 && j < i
6: j < 2 || j < i
7: j ≥ 2 || j < i
8: j < 2 || j ≥ i
9: j ≥ 2 || j ≥ i
0: j > 2 || j < i
```

Chae, Summer 2008 © UCB

Administrivia

- Extra OH today (12-2pm in 329 Soda)
- Don’t unplug things or eat in lab. Inst will get angry.
- Assignments
  - Quiz 3 due 7/5 @ 11:59pm
  - Quiz 3 due 7/8 @ 11:59pm
  - HW2 due 7/7 @ 11:59pm
  - Proj1 due 7/11 @ 11:59pm

Chae, Summer 2008 © UCB

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?

Chae, Summer 2008 © UCB

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.

Chae, Summer 2008 © UCB

Instruction Support for Functions (1/6)

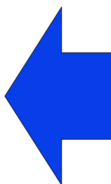
C

```
... 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.

Chae, Summer 2008 © UCB

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
```

Chae, Summer 2008 © UCB

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

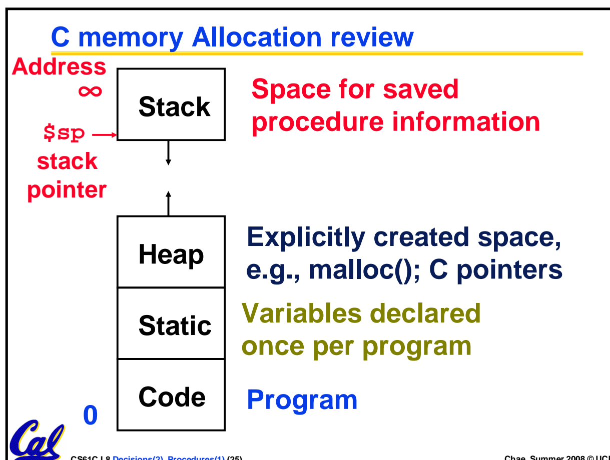
Nested Procedures (1/2)

```
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



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 # prep args
        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?
  - We'll see these in a few slides…

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…)

ra

memory

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, lb, sb, lbu`
  - Decision: `beq, bne, slt, slti, sltu, sltiu`
  - Unconditional Branches (Jumps): `j, jal, jr`
- Registers we know so far
  - All of them!