

Lecture #8 – MIPS Procedures

2005-09-26

There is one handout today at the front and back of the room!

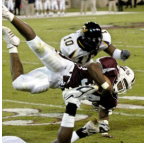


Lecturer PSOE, new dad Dan Garcia

www.cs.berkeley.edu/~ddgarcia

New Mexico St who? =>

No. 11 Cal runs over the Aggies 41-13 are are now 4-0 for 1st time since 1996. Justin Forsett ran for 235 yds, Joe Ayoob rushed for 3 TDs. Next Sat...Arizona @ home!



Review

- 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**.
- 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**
- Unsigned add/sub **don't cause overflow**
- New MIPS Instructions:
beq, bne, j, sll, srl
slt, slti, sltu, sltiu
addu, addiu, subu



C functions

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

What information must compiler/programmer keep track of?

```
/* 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 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;  
}
```

M
I
P
S
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;  
}
```

M
I
P
S
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;
}
```

- M**
- 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
```



CS61C L8 MIPS Procedures (7)

Garcia, Fall 2005 © UCB

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`.



CS61C L8 MIPS Procedures (8)

Garcia, Fall 2005 © UCB

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



CS61C L8 MIPS Procedures (9)

Garcia, Fall 2005 © UCB

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



CS61C L8 MIPS Procedures (10)

Garcia, Fall 2005 © UCB

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`.



CS61C L8 MIPS Procedures (11)

Garcia, Fall 2005 © UCB

Nested Procedures (2/2)

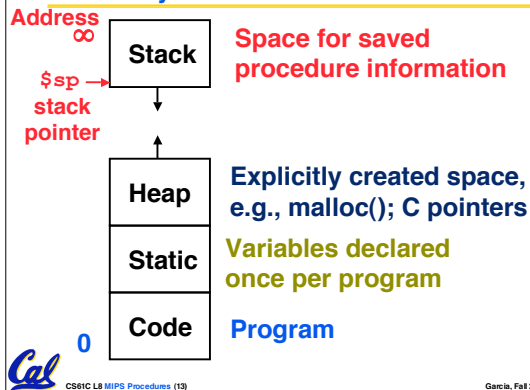
- In general, may need to save some other info in addition to `$ra`.
- When a C program is run, there are 3 important memory areas allocated:
 - **Static:** Variables declared once per program, cease to exist only after execution completes. E.g., C globals
 - **Heap:** Variables declared dynamically
 - **Stack:** Space to be used by procedure during execution; this is where we can save register values



CS61C L8 MIPS Procedures (12)

Garcia, Fall 2005 © UCB

C memory Allocation review



CS61C L8 MIPS Procedures (13)

Garcia, Fall 2005 © UCB

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;  
}
```



CS61C L8 MIPS Procedures (14)

Garcia, Fall 2005 © UCB

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:  ...
```



CS61C L8 MIPS Procedures (15)

Garcia, Fall 2005 © UCB

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.



CS61C L8 MIPS Procedures (16)

Garcia, Fall 2005 © UCB

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...



CS61C L8 MIPS Procedures (17)

Garcia, Fall 2005 © UCB

Basic Structure of a Function

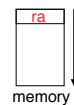
Prologue

```
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
```



CS61C L8 MIPS Procedures (18)

Garcia, Fall 2005 © UCB

Administrivia

- We have a midterm & review time & date
 - Review: Sun 2005-10-16 @ 2pm in 10 Evans
 - Midterm: Mon 2005-10-17, 5:30-8:30pm [here!](#)
 - DSP or Conflicts? Email Jeremy
- Dan's before-class graphics videos:
www.siggraph.org/publications/video-review/SVR.html
- Project 1 due tonight @ 11:59pm
 - An easy HW3 follows, due Friday



CS61C L8 MIPS Procedures (19)

Garcia, Fall 2005 © UCB

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.



CS61C L8 MIPS Procedures (21)

Garcia, Fall 2005 © UCB

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!



CS61C L8 MIPS Procedures (22)

Garcia, Fall 2005 © UCB

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.



CS61C L8 MIPS Procedures (23)

Garcia, Fall 2005 © UCB

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.



CS61C L8 MIPS Procedures (24)

Garcia, Fall 2005 © UCB

Parents leaving for weekend analogy (1/5)

- Parents (main) leaving for weekend
- They (caller) give keys to the house to kid (callee) with the rules (calling conventions):
 - You can trash the temporary room(s), like the den and basement (registers) if you want, we don't care about it
 - BUT you'd better leave the rooms (registers) that we want to save for the guests untouched. "these rooms better look the same when we return!"



CS61C L8 MIPS Procedures (25)

Garcia, Fall 2005 © UCB

Who hasn't heard this in their life?

Parents leaving for weekend analogy (2/5)

- Kid now “owns” rooms (**registers**)
- Kid wants to use the **saved** rooms for a wild, wild party (**computation**)
- What does kid (**callee**) do?
 - Kid takes what was in these rooms and puts them in the garage (**memory**)
 - Kid throws the party, **trashes everything** (except garage, who goes there?)
 - Kid restores the rooms the parents wanted **saved after the party** by **replacing the items from the garage (memory) back into those saved rooms**



CS61C L8 MIPS Procedures (26)

Garcia, Fall 2005 © UCB

Parents leaving for weekend analogy (3/5)

- Same scenario, except **before** parents return and kid replaces **saved** rooms...
- Kid (**callee**) has left valuable stuff (**data**) all over.
 - Kid’s friend (**another callee**) wants the house for a party when the **kid** is away
 - Kid knows that friend might **trash the place** destroying valuable stuff!
 - Kid remembers rule parents taught and now becomes the “heavy” (**caller**), instructing friend (**callee**) on good rules (**conventions**) of house.



CS61C L8 MIPS Procedures (27)

Garcia, Fall 2005 © UCB

Parents leaving for weekend analogy (4/5)

- If kid had data in **temporary rooms** (which were going to be trashed), there are three options:
 - Move items directly to garage (**memory**)
 - Move items to **saved rooms** whose contents have already been moved to the garage (**memory**)
 - Optimize lifestyle (**code**) so that the amount you’ve got to shlep stuff back and forth from garage (**memory**) is minimized
- Otherwise: “Dude, where’s my data?!”



CS61C L8 MIPS Procedures (28)

Garcia, Fall 2005 © UCB

Parents leaving for weekend analogy (5/5)

- **Friend** now “owns” rooms (**registers**)
- Friend wants to use the **saved** rooms for a wild, wild party (**computation**)
- What does friend (**callee**) do?
 - Friend takes what was in these rooms and puts them in the garage (**memory**)
 - Friend throws the party, **trashes everything** (except garage)
 - Friend restores the rooms the kid wanted **saved after the party** by **replacing the items from the garage (memory) back into those saved rooms**



CS61C L8 MIPS Procedures (29)

Garcia, Fall 2005 © UCB

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;
}
```



CS61C L8 MIPS Procedures (31)

Garcia, Fall 2005 © UCB

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); ... }
                      }
```



CS61C L8 MIPS Procedures (32)

Garcia, Fall 2005 © UCB

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



CS61C L8 MIPS Procedures (3)

Garcia, Fall 2005 © UCB

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



CS61C L8 MIPS Procedures (4)

Garcia, Fall 2005 © UCB

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)



CS61C L8 MIPS Procedures (5)

Garcia, Fall 2005 © UCB

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!



CS61C L8 MIPS Procedures (6)

Garcia, Fall 2005 © UCB

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.



CS61C L8 MIPS Procedures (7)

Garcia, Fall 2005 © UCB

"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!
 - There are CONVENTIONS when calling procedures!



CS61C L8 MIPS Procedures (8)

Garcia, Fall 2005 © UCB