

inst.eecs.berkeley.edu/~cs61c

UCB CS61C : Machine Structures

Lecture 10 – Introduction to MIPS Procedures I

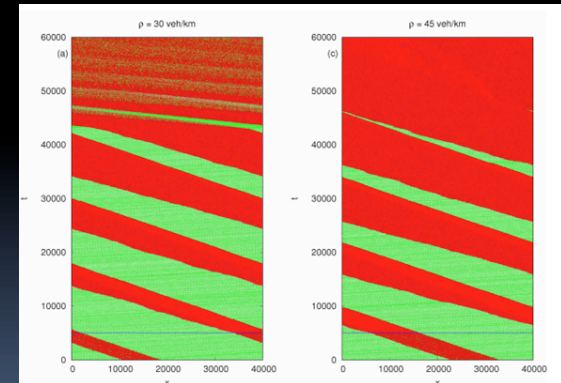
2014-02-14



Sr Lecturer SOE
Dan Garcia

IN-CAR ALGORITHM COULD DISSOLVE TRAFFIC!

“If cars broadcast their speeds to other vehicles” ... (and the speeds of cars were automatically controlled – you could still steer) ... “a simple in-car algorithm could help dissolve traffic jams as soon as they occur!”. Key idea – be optimistic leaving the jam and defensive leading into it.



www.technologyreview.com/blog/arxiv/27166/

Review

- **MIPS Machine Language Instruction:**
32 bits representing a single instruction

| | | | | | | |
|---|--------|----------------|----|-----------|-------|-------|
| R | opcode | rs | rt | rd | shamt | funct |
| I | opcode | rs | rt | immediate | | |
| J | opcode | target address | | | | |

- Branches use PC-relative addressing, Jumps use absolute addressing.
- Disassembly is simple and starts by decoding opcode field. (more next lecture)



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 = 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)

```
... sum(a,b);... /* a,b:$s0,$s1 */  
}
```

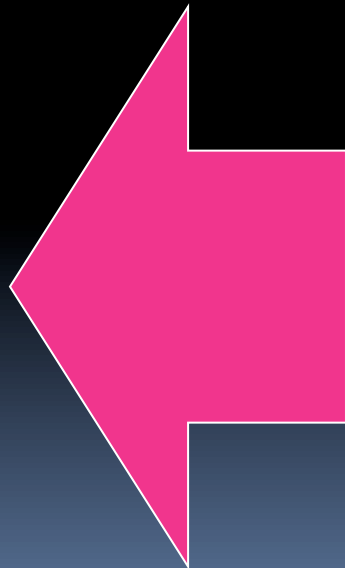
C

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

address (shown in decimal)

M
I
P
S

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)

```
... sum(a,b);... /* a,b:$s0,$s1 */  
}
```

C

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

address (shown in decimal)

**M
I
P
S**

```
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)

```
... sum(a,b);... /* a,b:$s0,$s1 */  
}
```

C

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

- Question: Why use **jr** here? Why not use **j**?

M

- Answer: **sum** might be called by many places, so we can't return to a fixed place. The calling proc to **sum** must be able to say "return here" somehow.

P
S



```
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              #goto sum
```

- **After:**

```
1008 jal sum  # $ra=1012,goto sum
```

- Why have a **jal**?

- Make the common case fast: function calls very common.
- Don't have to know where code is in 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.
- 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`.



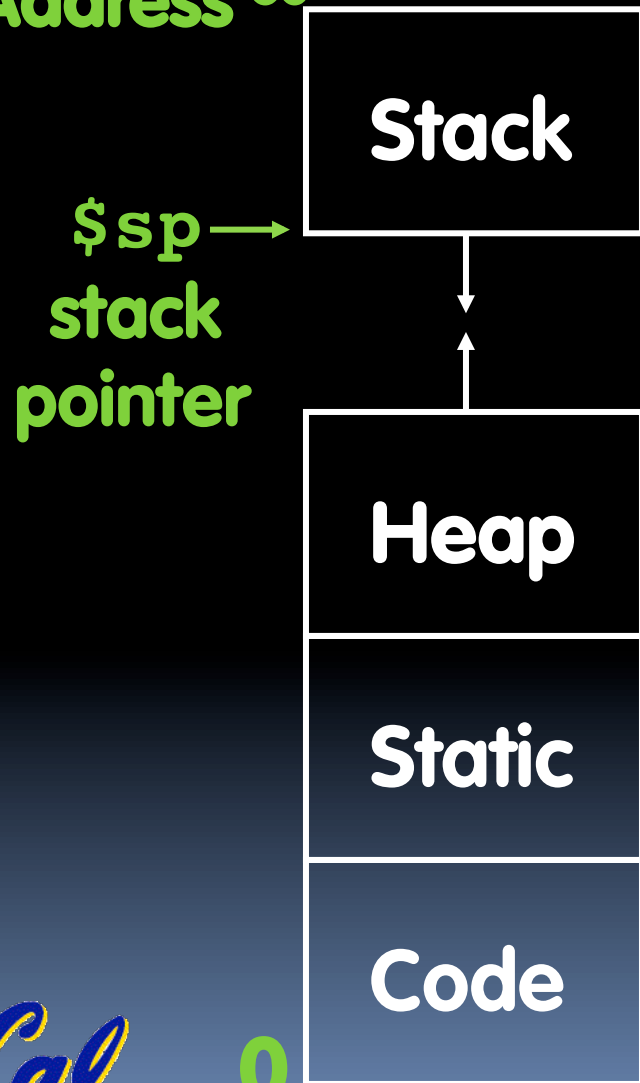
Nested Procedures (2/2)

- 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 via **malloc**
 - **Stack**: Space to be used by procedure during execution; this is where we can save register values



C Memory Allocation

Address ∞



Space for local vars, saved procedure information

Explicitly created space, i.e., `malloc()`

Variables declared once per program; e.g., globals (doesn't change size)

Program (doesn't change size)



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  
addi $sp,$sp,8 # restore stack  
jr $ra
```

“pop”

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**

So what are they?



Basic Structure of a Function

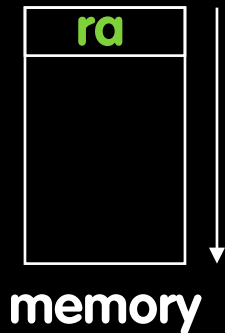
Prologue

```
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 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...

- 1) We COULD copy \$a0 to \$a1 (& then not store \$a0 or \$a1 on the stack) to store n across recursive calls.
- 2) We MUST save \$a0 on the stack since it gets changed.
- 3) We MUST save \$ra on the stack since we need to know where to return to...

- | | 1 | 2 | 3 |
|----|---|---|---|
| a) | F | F | F |
| b) | F | F | T |
| c) | F | T | F |
| c) | F | T | T |
| d) | T | F | F |
| d) | T | F | T |
| e) | T | T | F |
| e) | T | T | T |



“And in Conclusion...”

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

