CS241 - Lecture 4

Graham Cooper

May 13th, 2015

Recall Labels

```
0 lis $2
4 .word 13
8 add $3, $0, $0
c top: add $3, $3, $2 ; top = 0x0c
10 lis $1
14 .word 1
18 sub $2, $2, $1
1c sub $2, $2, $1
1c bne $2, $0, top
20 jr $31
```

Procedures in MIPS

Two problems to solve

1. Call and return - how to transfer control into and out of a procedure. What if the proc calls another proc? - Parameters?

2. Registers - What if a procedure overwrites regs we were using? - Data is lost.

2)

We could reverse some registers for the procedure f, and some for the mainline - then they won't interfere.
- What if f calls g, g calls h,.... eventually run out of registers.
Instead:
Guarantee that procedures leave registers unchanged when complete.
How? - Use RAM. - Which RAM?
- How do we keep procedures from using the same RAM?

Diagram of RAM:

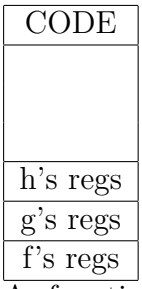
0	Your Program
	FREE RAM

Allocate RAM from either the top or the bottom of free RAM.
Keep Track of which RAM is in use and which isn't
MIPS Machine helps us out:
$30 initicalized by the loader to just past the last word of free RAM

Can use $30 as a bookmark to seperate used and unused RAM, if we allocate from the bottom of free RAM

Eg. Procedures f,g,h.
Say f calls g,
g calls h
h returns
g returns
f returns

Each procedure stores in RAM the registers it wants to use and restores the original values on return.



As functions are called, register $30 goes towards code, and as things returns the value goes away from the code

RAM is used in LIFO order - as a stack
$30

- the stack pointer
- contains the address of the top of the stack.

Templates for Procedures

(assume f uses $2 and $3)

```
0  f: sw $2, -4($30)
4  sw $3, -8($30)
8  lis $3
c  .word 8
10 sub $30, $30, $3
...
...
n  add $30, $30, $3
n+4 lw $3, -8($30)
n+8 lw $2, -4($30)
n+c ???
```

Usage:
0-4: Push registers f will modify onto the stack
8-10: Decrement $30 10-n: body of the function
n: assuming $3 is still 8, increment $30
n+4,n+8: popping the registers from the stack
n+c: return?

1)

Call and Return

call

```
main: ...
main: ...
main: ...
n  lis $5
n+4 .word f
n+8 jr $5
```

n+4: address of line labelled 'f'
n+8: jump to that line

return

- Need to set PC to the line after the jr
- How does f know where that is?

Solution

- jalr instruction ("Jump and link to reg")

- like jr but sets $31 to the address of the next instruction, before jumping. So replace jr $5 above with jalr $5 then return is jr $31

Q: jalr overwrites $31 - then how can we return to the loader? And what if f calls g?

A: need to save $31 on the stack first, and restore it when the call returns.

```
main: ...
main: ...
main: ...
n lis $5
n+4 .word f
n+8 sw $31, -4($30)
n+c lis $31
n+10 .word 4
n+14 sub $30, $30, $31
n+18 jalr $5
n+1c lis $31
n+20 .word 4
n+24 add $30, $30, $31
n+28 lw $31, -4($30)
...
...
n+i jr $31
```

n+8: push $31 onto the stack
n+8,n+14: Decrement stack pointer
n+1c,n+20: Increment stack pointer

f:

- push regs
- decrement $30
- body
- increment $30

- Pop Regs
- Return

Parameters and Results

- Generally use registers (DOCUMENT!!)
- if too many parameters use he stack

Eg: Sum 1 to N

```
; sum 1 to N - adds, 1,...N
; Registers:
; $1 - working - save this reg
; $2 - input - save this reg
; $3 - output - do not save
```

Sum 1 to N:

```
        sw $1, -4($30) ; Push $1 and $2
        sw $2, -8($30)
        lis $1
        .word 8
        sub $30, $30, $1
        add $3, $0, $0
        top: add $3, $3, $2
        lis $1
        .word 1
        sub $2, $2, $1
        bne $2, $0, top
        lis $1
        .word 8
        add $30, $30, $1
        lw $2, -8($30)
        lw $1, -4($30)
        jr $31
```

Recursion

- No extra machinery is needed!!!! YAY
- If registers, parameters and stack are managed properly, recursion will just work

Output:

Use sw to store a word at location 0xffff000c
- Least significant byte will be printed
-To print CS\n

```
lis $1
.word 0xffff000c
lis $2
.word 67 ; c
sw $2, 0($1)
lis $2
.word 83 ; s
sw $2, 0($1)
lis $2
.word 10 ; backslash n
sw $2, 0($1)
jr $31
```