

lecture 11

MIPS assembly language 4

- functions
- MIPS stack

February 15, 2016

MIPS registers

● already mentioned
● new today

Name	Register Number	Usage	Preserved on call
● \$zero	0	the constant value 0	n.a.
● \$at	1	reserved for the assembler	n.a.
● \$v0-\$v1	2-3	value for results and expressions	no
● \$a0-\$a3	4-7	arguments (procedures/functions)	yes
● \$t0-\$t7	8-15	temporaries	no
● \$s0-\$s7	16-23	saved	yes
● \$t8-\$t9	24-25	more temporaries	no
● \$k0-\$k1	26-27	reserved for the operating system	n.a.
● \$gp	28	global pointer	yes
● \$sp	29	stack pointer	yes
● \$fp	30	frame pointer	yes
● \$ra	31	return address	yes

functions in C

```

main() {
    int m, n;
    m = myfunction(n);
}

int myfunction(int j) {
    int k;
    return k;
}
    
```

parent/ Caller

child/ callee

parent must:

- save in Memory any registers it will need later (after return)
- provide arguments to child
- provide return address to child (so child can jump back when done)
- branch to child

child must:

- allocate space for local variables (registers, Memory) and not write over parent's data (registers, Memory)
- compute value and return it to parent
- branch back to parent

parent must:

- branch to child

child must:

- branch back to parent

How?

Jump and Link, Jump register

```

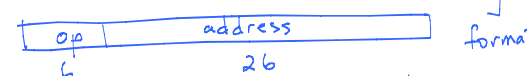
main:
    jal myfunction
    jr $ra
    
```

writes address of next instruction into
\$ra = \$31

```

myfunction:
    jr $ra
    
```

jal



J format

offset address of instruction to jump to
(defined a few lectures from now)

jr



R format

Provide argument(s), return value(s)

How?

\$a0, \$a1, \$a2, \$a3 argument registers
\$4, \$5, \$6, \$7

\$v0, \$v1 return value registers
\$2, \$3

provide argument(s)

```

main() {
    int m, n;
    m = myfunction(n);
}

```

move jal \$a0, \$s0
move \$s3, \$v0

return value(s)

```

int myfunction(int j) {
    int k;
    return k;
}

```

move \$v0, \$s1
jr \$ra

Encapsulation Problem

- author of parent function might not know which \$s and \$t registers the child uses
- author of child function might not know which \$s and \$t registers the parent uses



Kitchen Policies

- By default, all dishes are clean. Use dishes, then clean them.
- By default, all dishes are dirty. Wash what you need and leave them dirty.

Problems arise when two house mates have different policies.

MIPS register conventions (policies)

parent (caller)

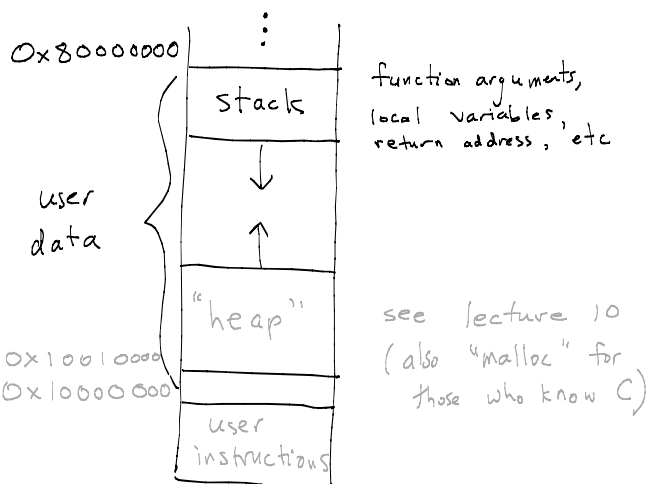
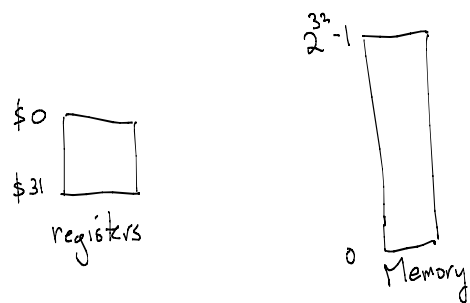
- assume that \$s0, -- \$s7 will contain same values before and after call
- don't assume that \$t0, -- \$t7 will contain same values after call
(If parent will need values in \$t0, -- \$t7 after call, then these values must be stored in Memory prior to call, and loaded after call.)

MIPS register conventions

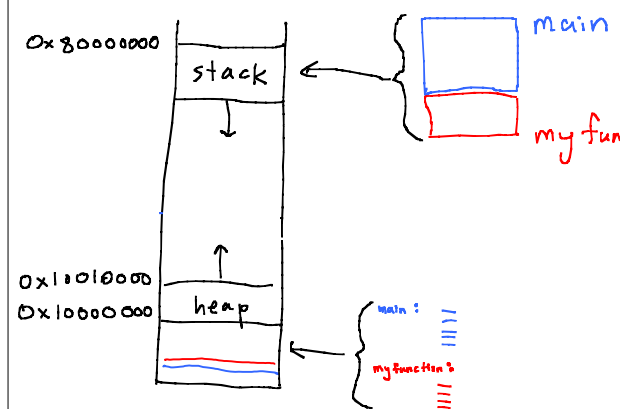
child (callee)

- assume that \$t0, -- \$t7 are not being used by parent
- assume that \$s0, -- \$s7 are being used by parent.
(If child needs to use any of \$s0, -- \$s7, then it first needs to store the current values in Memory, and re-load the values prior to returning to parent.)

How and where do functions store register values in Memory?



Stack frames

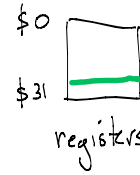


\$t0
\$t1

```
main() {
    int m, n;
    m = myfunction(n);
    ...
}

int myfunction(int j) {
    int k;
    ...
    return k;
}
```

Stack pointer (register \$sp = \$29)
contains the lowest address of the stack.



```

    addi    $sp, $sp, -8
    sw      $t1, 0($sp)
    sw      $t0, 4($sp)
    jal     child
    lw      $t1, 0($sp)
    lw      $t0, 4($sp)
    addi    $sp, $sp, 8

```

allowed

```

sw    $t0, -4($sp)
sw    $t1, -8($sp)
addi  $sp, $sp, -8
jal   child
addi  $sp, $sp, 8($sp)
lw    $t1, -8($sp)
lw    $t0, -4($sp)

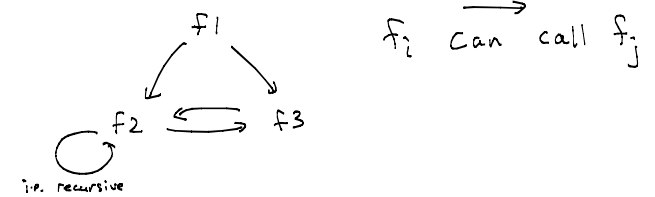
```

do work
here
using
register \$50

```

addi    $sp, $sp, -4
sw      $s0, 0($sp)
        :
        :
        :
→ lw     $s0, 0($sp)
addi    $sp, $sp, 4
jr      $ra

```

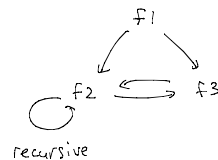


When functions call another, we need to ensure that data is safe (not written over).

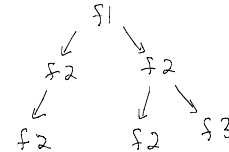
When a function is both a child and a parent, it must store the return address (to its own parent) before it calls its child. It also needs to store $\$$ registers (from parent) that it modifies.

(See conventions on slide 2).

(Static, compile time)



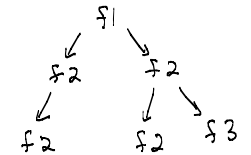
(dynamic, runtime)



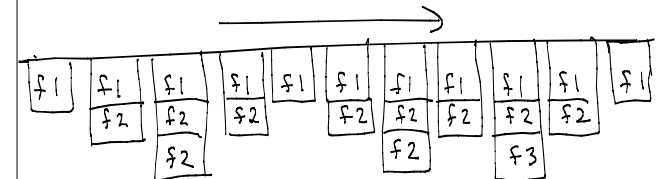
"Call tree"

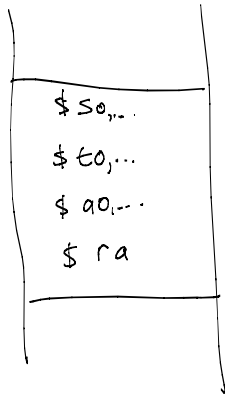
("pre-order traversal" -
COMP 250)

Call tree



Evolution of stack over time





stack
frame in
general
case

Example: recursion

```
int sumton (int n) { // n > 0
    if (n == 0)
        return 0;
    else
        return n + sumton(n-1);
}
```

sumton:

```
# if n == 0 then branch to base case
# else push the two items onto the stack:
# - return address ($ra)
# - argument n ($a0)
#
# compute (n-1) argument for next call
# jump and link to sumton (recursive)

# load the return address (pop)
# load argument from the stack (pop)
# change the stack pointer

# register $v0 contains result of sumton(n-1)
# add argument n to $v0
# return to parent
```

basecase:

```
# assign 0 as the value in $v0
# return to parent
```

sumton:

```
beq $a0,$zero, basecase # if n == 0 then branch to base case
addi $sp,$sp,-8          # else, make space for 2 items on the stack
sw $ra, 4($sp)           # store return address on stack
sw $a0, 0($sp)           # store argument n on stack
                           # (will need it to calculate returned value)
addi $a0,$a0,-1          # compute argument for next call: n = n-1
jal sumton               # jump and link to sumton (recursive)

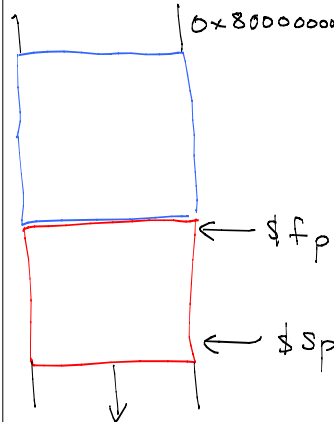
lw $ra, 4($sp)           # load the return address
lw $a0, 0($sp)           # load n from the stack
addi $sp,$sp,8           # change the stack pointer
```

```
# register $v0 contains result of sumton(n-1)
add $v0,$a0,$v0          # add n to $v0
jr $ra                  # return to parent
```

basecase:

```
addi $v0,$zero,0         # assign 0 as the value in $v0
jr $ra                  # return to parent
```

Frame Pointer (\$30)



Sometimes the
size of stack
frame is not
fixed.

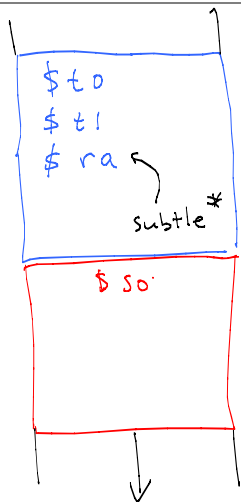
It may be useful
to keep track
of where frame
starts.

0x80000000

```
main() {
    int m,n;
    m = myfunction(n);
}
```

```
int myfunction(int j) {
    int k;
    ...
    return k;
}
```

* kernel calls main



Announcements

A2 is due next Sunday at midnight.

A3 will be posted soon and will be due on the last day of Reading Week (~3 weeks from now)

A4 (last one) will be posted in mid-March and due at end of March.

Last lecture is April 13. Final exam is April 26 (tent.)