

Calling Conventions

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

See: P&H 2.8, 2.12

calc.c —

```
vector v = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int c = pi + tnorm(v);
print("result", c);
```

math.c —

```
int tnorm(vector v) {
    return abs(v->x)+abs(v->y);
}
```

lib3410.o —

```
global variable: pi
entry point: prompt
entry point: print
entry point: malloc
```

0xffffffffc

0x80000000
0x7fffffc

0x10000000

0x00400000
0x00000000

top

bottom

math.c —

```
int abs(x) {  
    return x < 0 ? -x : x;  
}  
int tnorm(vector v) {  
    return abs(v->x)+abs(v->y);  
}
```

abs:

```
# arg in r3, return address in r31  
# leaves result in r3
```

tnorm:

```
# arg in r4, return address in r31  
# leaves result in r4
```

calc.c

```

vector v = malloc(8);
v->x = prompt("enter x");
v->y = prompt("enter y");
int c = pi + tnorm(v);
print("result", c);

```

dostuff:

```

# no args, no return value, return addr in r31
MOVE r30, r31
LI r3, 8      # call malloc: arg in r3, ret in r3
JAL malloc
MOVE r6, r3 # r6 now holds v
LA r3, str1  # call prompt: arg in r3, ret in r3
JAL prompt
SW r3, 0(r6)
LA r3, str2  # call prompt: arg in r3, ret in r3
JAL prompt
SW r3, 4(r6)
MOVE r4, r6 # call tnorm: arg in r4, ret in r4
JAL tnorm
LA r5, pi
LW r5, 0(r5)
ADD r5, r4, r5
LA r3, str3  # call print: args in r3 and r4
MOVE r4, r5
JAL print
JR r30

```

```

.data
str1: .asciiz "enter x"
str2: .asciiz "enter y"
str3: .asciiz "result"
.text
.extern prompt
.extern print
.extern malloc
.extern tnorm
.global dostuff

```

Calling Conventions

- where to put function arguments
- where to put return value
- who saves and restores registers, and how
- stack discipline

Why?

- Enable code re-use (e.g. functions, libraries)
- Reduce chance for mistakes

Warning: There is no one true MIPS calling convention.
lecture != book != gcc != spim != web

```
void main() {  
    int x = ask("x?");  
    int y = ask("y?");  
    test(x, y);  
}
```

```
void test(int x, int y) {  
    int d = sqrt(x*x + y*y);  
    if (d == 1)  
        print("unit");  
    return d;  
}
```

r0	\$zero	zero	r16		
r1	\$at	assembler temp	r17		
r2	\$v0	function	r18		
r3	\$v1	return values	r19		
r4	\$a0		r20		
r5	\$a1	function	r21		
r6	\$a2	arguments	r22		
r7	\$a3		r23		
r8			r24		
r9			r25		
r10			r26	\$k0	reserved
r11			r27	\$k1	for OS kernel
r12			r28		
r13			r29		
r14			r30		
r15			r31	\$ra	return address

```
void main() {  
    int x = ask("x?");  
    int y = ask("y?");  
    test(x, y);  
}
```

main:
LA \$a0, strX
JAL ask # result in \$v0

LA \$a0, strY
JAL ask # result in \$v0

Call stack

- contains *activation records*
(aka *stack frames*)

One for each function invocation:

- saved return address
- local variables
- ... and more

Simplification:

- frame size & layout decided at compile time for each function

Convention:

- r29 is \$sp
(bottom elt
of call stack)

Stack grows **down**

Heap grows **up**

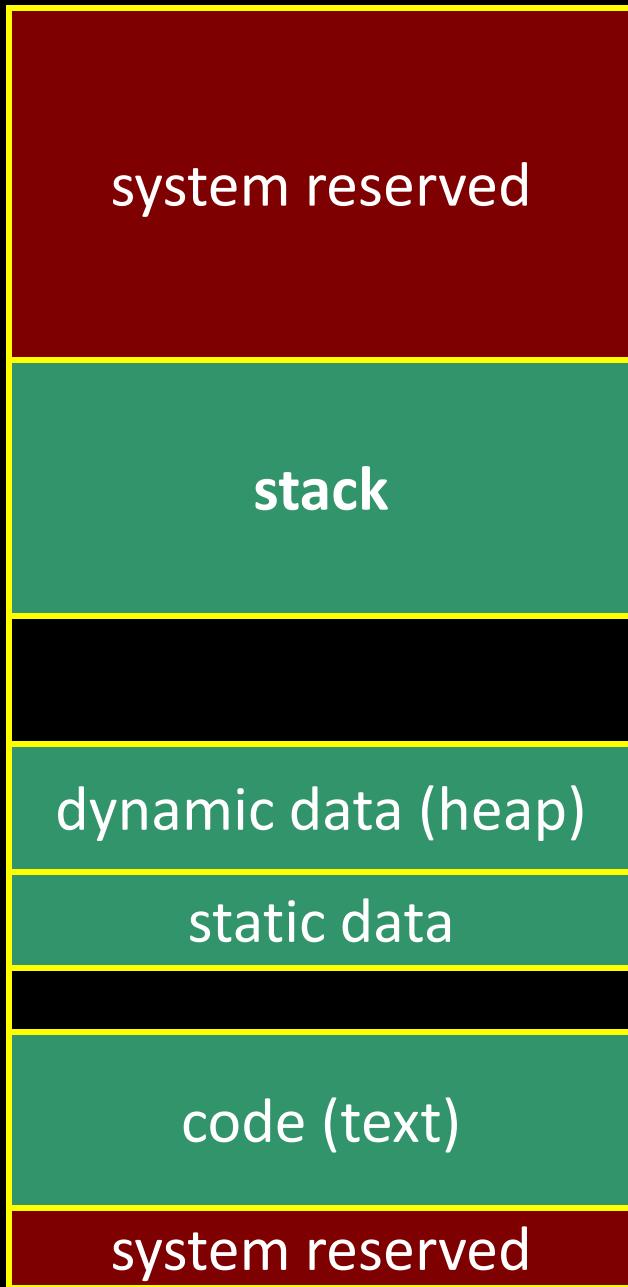
0xffffffffc

0x80000000

0x10000000

0x00400000

0x00000000



```
void main() {           main:  
    int x = ask("x?");   # allocate frame  
    int y = ask("y?");   ADDUI $sp, $sp, -12 # $ra, x, y  
    test(x, y);         # save return address in frame  
}  
                           SW $ra, 8($sp)
```

```
# restore return address  
LW $ra, 4($sp)  
# deallocate frame  
ADDUI $sp, $sp, 12
```

Conventions so far:

- args passed in \$a0, \$a1, \$a2, \$a3
- return value (if any) in \$v0, \$v1
- stack frame at \$sp
 - contains \$ra (clobbered on JAL to sub-functions)
 - contains local vars (possibly clobbered by sub-functions)

Q: What about real argument lists?

```
int min(int a, int b);  
int paint(char c, short d, struct point p);  
int treesort(struct Tree *root, int[] A);  
struct Tree *createTree();  
int max(int a, int b, int c, int d, int e);
```

Conventions:

- align everything to multiples of 4 bytes
- first 4 words in \$a0...\$a3, “spill” rest to stack

```
invoke sum(0, 1, 2, 3, 4, 5);  
main:  
...  
LI $a0, 0  
LI $a1, 1  
LI $a2, 2  
LI $a3, 3  
ADDI $sp, $sp, -8  
LI r8, 4  
SW r8, 0($sp)  
LI r8, 5  
SW r8, 4($sp)  
JAL sum  
ADDI $sp, $sp, 8
```

```
sum:  
...  
ADD $v0, $a0, $a1  
ADD $v0, $v0, $a2  
ADD $v0, $v0, $a3  
LW $v1, 0($sp)  
ADD $v0, $v0, $v1  
LW $v1, 4($sp)  
ADD $v0, $v0, $v1  
...  
JR $ra
```

```
printf(fmt, ...)
```

```
main:
```

```
...
```

```
LI $a0, str0
```

```
LI $a1, 1
```

```
LI $a2, 2
```

```
LI $a3, 3
```

```
# 2 slots on stack
```

```
LI r8, 4
```

```
SW r8, 0($sp)
```

```
LI r8, 5
```

```
SW r8, 4($sp)
```

```
JAL sum
```

```
printf:
```

```
...
```

```
if (argno == 0)
```

```
    use $a0
```

```
else if (argno == 1)
```

```
    use $a1
```

```
else if (argno == 2)
```

```
    use $a2
```

```
else if (argno == 3)
```

```
    use $a3
```

```
else
```

```
    use $sp+4*argno
```

```
...
```

Variable Length Arguments

Initially confusing but ultimately simpler approach:

- Pass the first four arguments in registers, as usual
- Pass the rest on the stack (in order)
- Reserve space on the stack for all arguments, including the first four

Simplifies varargs functions

- Store a0-a3 in the slots allocated in parent's frame
- Refer to all arguments through the stack

Conventions so far:

- first four arg words passed in \$a0, \$a1, \$a2, \$a3
- remaining arg words passed on the stack
- return value (if any) in \$v0, \$v1
- stack frame at \$sp
 - contains \$ra (clobbered on JAL to sub-functions)
 - contains local vars (possibly clobbered by sub-functions)
 - contains extra arguments to sub-functions
 - contains space for first 4 arguments to sub-functions

init(): 0x400000
printf(s, ...): 0x4002B4
vnorm(a,b): 0x40107C
main(a,b): 0x4010A0
pi: 0x10000000
str1: 0x10000004

CPU:
\$pc=0x004003C0
\$sp=0x7FFFFFAC
\$ra=0x00401090

What func is running?

Who called it?

Has it called anything?

Will it?

Args?

Stack depth?

Call trace?

0x7FFFFFB0

0x00000000
0x0040010c
0x0040010a
0x00000000
0x00000000
0x00000000
0x00000000
0x00000000
0x004010c4
0x00000000
0x00000000
0x00000000
0x00000015
0x10000004
0x00401090

Frame pointer marks boundaries

- Optional (for debugging, mostly)

Convention:

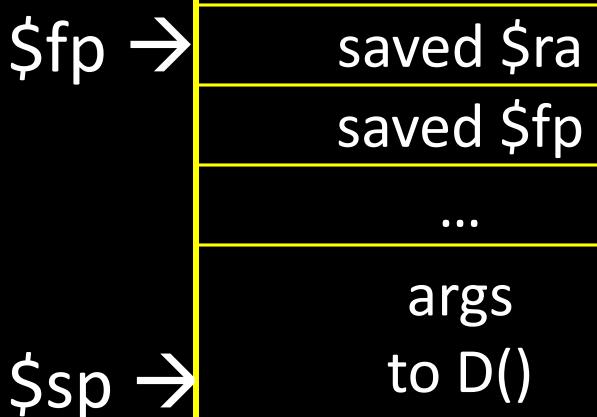
- r30 is \$fp
(top elt of current frame)
- Callee: always push old \$fp
on stack

E.g.:

A() called B()

B() called C()

C() about to call D()



r0	\$zero	zero	r16		
r1	\$at	assembler temp	r17		
r2	\$v0	function	r18		
r3	\$v1	return values	r19		
r4	\$a0		r20		
r5	\$a1	function	r21		
r6	\$a2	arguments	r22		
r7	\$a3		r23		
r8			r24		
r9			r25		
r10			r26	\$k0	reserved
r11			r27	\$k1	for OS kernel
r12			r28		
r13			r29	\$sp	stack pointer
r14			r30	\$fp	frame pointer
r15			r31	\$ra	return address

How does a function load global data?

- global variables are just above 0x10000000

Convention: *global pointer*

- r28 is \$gp (pointer into *middle* of global data section)
 $\$gp = 0x10008000$
- Access most global data using LW at \$gp +/- offset
LW \$v0, 0x8000(\$gp)
LW \$v1, 0x7FFF(\$gp)

r0	\$zero	zero	r16		
r1	\$at	assembler temp	r17		
r2	\$v0	function	r18		
r3	\$v1	return values	r19		
r4	\$a0		r20		
r5	\$a1	function	r21		
r6	\$a2	arguments	r22		
r7	\$a3		r23		
r8			r24		
r9			r25		
r10			r26	\$k0	reserved
r11			r27	\$k1	for OS kernel
r12			r28	\$gp	global pointer
r13			r29	\$sp	stack pointer
r14			r30	\$fp	frame pointer
r15			r31	\$ra	return address

Q: Remainder of registers?

A: Any function can use for any purpose

- places to put extra local variables, local arrays, ...
- places to put callee-save

Callee-save: Always...

- save before modifying
- restore before returning

```
int main() {  
    int x = prompt("x?");  
    int y = prompt("y?");  
    int v = tnorm(x, y)  
    printf("result is %d", v);  
}
```

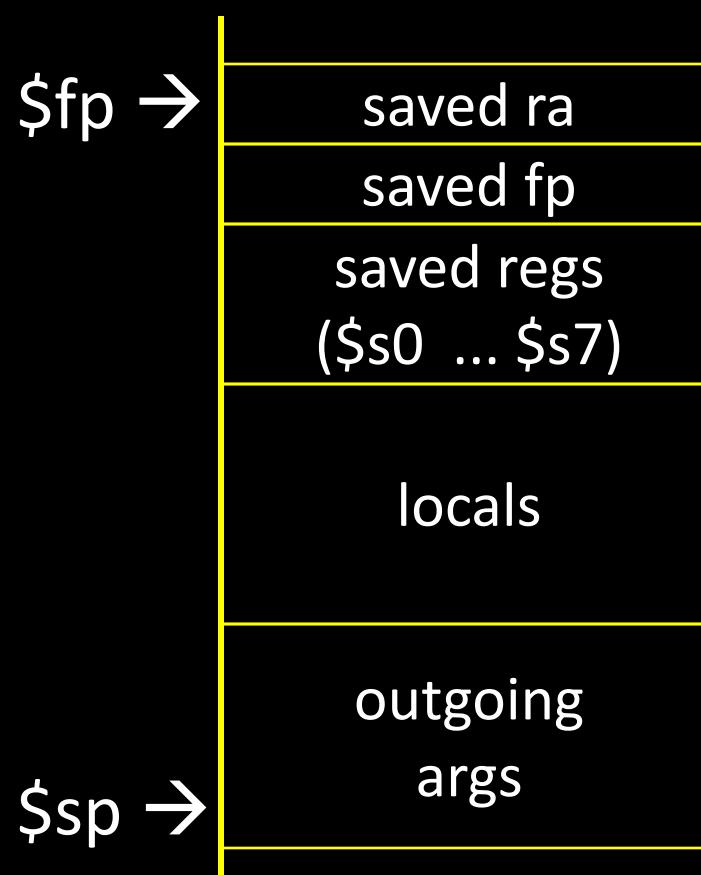
Caller-save: If necessary...

- save before calling anything
- restore after it returns

r0	\$zero	zero	r16	\$s0	
r1	\$at	assembler temp	r17	\$s1	
r2	\$v0	function	r18	\$s2	
r3	\$v1	return values	r19	\$s3	
r4	\$a0		r20	\$s4	saved (callee save)
r5	\$a1	function	r21	\$s5	
r6	\$a2	arguments	r22	\$s6	
r7	\$a3		r23	\$s7	
r8	\$t0		r24	\$t8	more temps
r9	\$t1		r25	\$t9	(caller save)
r10	\$t2		r26	\$k0	reserved for
r11	\$t3	temps	r27	\$k1	kernel
r12	\$t4	(caller save)	r28	\$gp	global data pointer
r13	\$t5		r29	\$sp	stack pointer
r14	\$t6		r30	\$fp	frame pointer
r15	\$t7		r31	\$ra	return address

Conventions so far:

- first four arg words passed in \$a0, \$a1, \$a2, \$a3
- remaining arg words passed in parent's stack frame
- return value (if any) in \$v0, \$v1
- globals accessed via \$gp
- callee save regs are preserved
- caller save regs are not



```
int test(int a, int b) {  
    int tmp = (a&b)+(a|b);  
    int s = sum(tmp,1,2,3,4,5);  
    int u = sum(s,tmp,b,a,b,a);  
    return u + a + b;  
}
```

test:

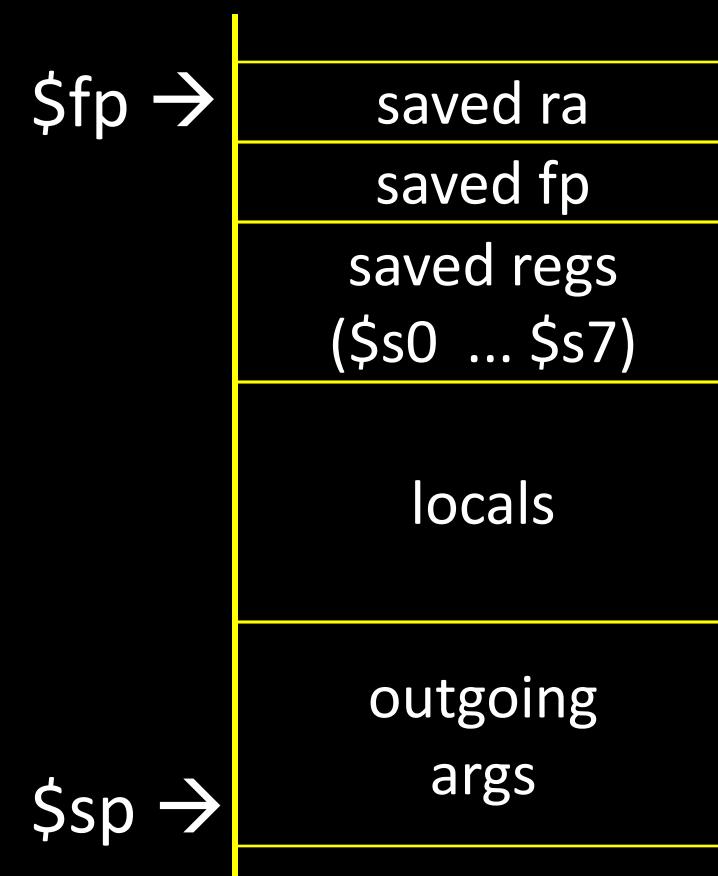
```
# uses...
# allocate frame
# save $ra
# save old $fp
# save ...
# save ...
# set new frame pointer
```

...

...

```
# restore ...
# restore ...
# restore old $fp
# restore $ra
# dealloc frame
```

Minimum stack size for a standard function?



Leaf function does not invoke any other functions

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

Optimizations?

\$fp →

No saved regs (or locals)

No outgoing args

Don't push \$ra

No frame at all?

\$sp →

saved ra

saved fp

saved regs
(\$s0 ... \$s7)

locals

outgoing
args

Global variables in data segment

- Exist for all time, accessible to all routines

Dynamic variables in heap segment

- Exist between malloc() and free()

Local variables in stack frame

- Exist solely for the duration of the stack frame

Dangling pointers into freed heap mem are bad

Dangling pointers into old stack frames are bad

- C lets you create these, Java does not
- `int *foo() { int a; return &a; }`

```
#include <stdio.h>
int n = 100;
int main (int argc, char* argv[]) {
    int i, m = n, count = 0;
    for (i = 1; i <= m; i++) { count += i; }
    printf ("Sum 1 to %d is %d\n", n, count);
}
```

```
[csug01] mipsel-linux-gcc -S add1To100.c
```

	.data	\$L2:	lw	\$2,24(\$fp)
	.globl n		lw	\$3,28(\$fp)
	.align 2		slt	\$2,\$3,\$2
n:	.word 100		bne	\$2,\$0,\$L3
	.rdata		lw	\$3,32(\$fp)
	.align 2		lw	\$2,24(\$fp)
\$str0: .asciiz	"Sum 1 to %d is %d\n"		addu	\$2,\$3,\$2
	.text		sw	\$2,32(\$fp)
	.align 2		lw	\$2,24(\$fp)
	.globl main		addiu	\$2,\$2,1
main: addiu	\$sp,\$sp,-48		sw	\$2,24(\$fp)
sw	\$31,44(\$sp)		b	\$L2
sw	\$fp,40(\$sp)	\$L3:	la	\$4,\$str0
move	\$fp,\$sp		lw	\$5,28(\$fp)
sw	\$4,48(\$fp)		lw	\$6,32(\$fp)
sw	\$5,52(\$fp)		jal	printf
la	\$2,n		move	\$sp,\$fp
lw	\$2,0(\$2)		lw	\$31,44(\$sp)
sw	\$2,28(\$fp)		lw	\$fp,40(\$sp)
sw	\$0,32(\$fp)		addiu	\$sp,\$sp,48
li	\$2,1		j	\$31
sw	\$2,24(\$fp)			

main: addiu \$sp, \$sp, -64

sw \$fp, 56(\$sp)

addu \$fp, \$0, \$sp

sw \$0, 48(\$fp)

sw \$0, 52(\$fp)

\$L2: lw \$2, 52(\$fp)

nop

slt \$4, \$2, 7

beq \$4, \$0, \$L3

nop

sll \$4, \$2, 2

...

addiu \$2, \$2, 1

sw \$2, 52(\$fp)

J \$L2

```
int strlen(char *a) {  
    int n;  
    while (a[n] != 0) n++;  
    return n;  
}  
  
strlen:    addiu   sp, sp, -16  
           sw      fp, 8(sp)  
           addiu   fp, sp, 16  
           lw      v0, 0(fp)  
           nop  
  
strlen_top:   addu    t0, a0, v0  
           lb     t1, 0(t0)  
           nop  
           beqz   t1, strlen_done  
           nop  
           addiu   v0, v0, 1  
           b      strlen_top  
           nop  
  
strlen_done:  
           lw      fp, 8(sp)  
           addiu   sp, sp, 16  
           jr      ra  
           nop
```