

# Calling Conventions

**Hakim Weatherspoon**

**CS 3410, Spring 2011**

Computer Science

Cornell University

See P&H 2.8 and 2.12

# Announcements

---

PA1 due *this* Friday

Work in **pairs**

Use your resources

- FAQ, class notes, book, Sections, office hours, newsgroup, CSUGLab

PA2 will be available this Friday

- PA2 builds from PA1
- Work with **same** partner
- Due right before spring break

# Announcements

---

Prelims1: next Thursday, March 10<sup>th</sup> in class

- We will start at 1:25pm sharp, so come early
- Closed Book
  - Cannot use electronic device or outside material
- Practice prelims are online in CMS
- Material covered
  - Appendix C (logic, gates, FSMs, memory, ALUs)
  - Chapter 4 (pipelined [and non-pipeline] MIPS processor with hazards)
  - Chapters 2 and Appendix B (RISC/CISC, MIPS, and calling conventions)
  - Chapter 1 (Performance)
  - HW1, HW2, PA1, PA2

# Goals for Today

---

## Calling Conventions

- Anatomy of an executing program
- Register assignment conventions,
- Function arguments, return values
- Stack frame, Call stack, Stack growth
- Variable arguments

## Next time

- More on stack frames
- globals vs local accessible data
- callee vs caller saved registers

# Example program

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
```

# Anatomy of an executing program

0xfffffffffc

top

0x80000000

0x7fffffff

0x10000000

0x00400000

0x00000000

bottom

# math.s

---

math.c

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

tnorm:

# arg in r4, return address in r31

# leaves result in r4

abs:

# arg in r3, return address in r31

# leaves result in r3

# calc.s

dostuff:

calc.c

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

```
# 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

---

## 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

# Example

---

```
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;  
}
```

# MIPS Register Conventions

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

# Example: Invoke

---

```
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

---

## *Call stack*

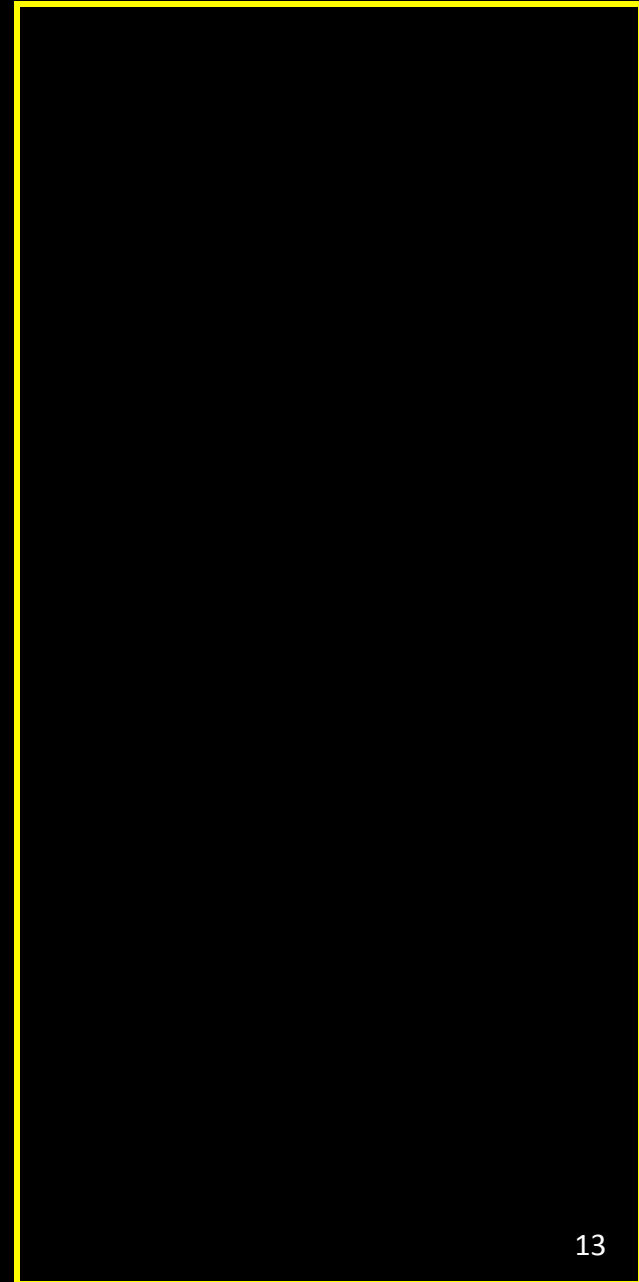
- contains *activation records*  
(aka *stack frames*)

One for each function invocation:

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

Simplification:

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



# Stack Growth

Convention:

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

Stack grows **down**

Heap grows **up**

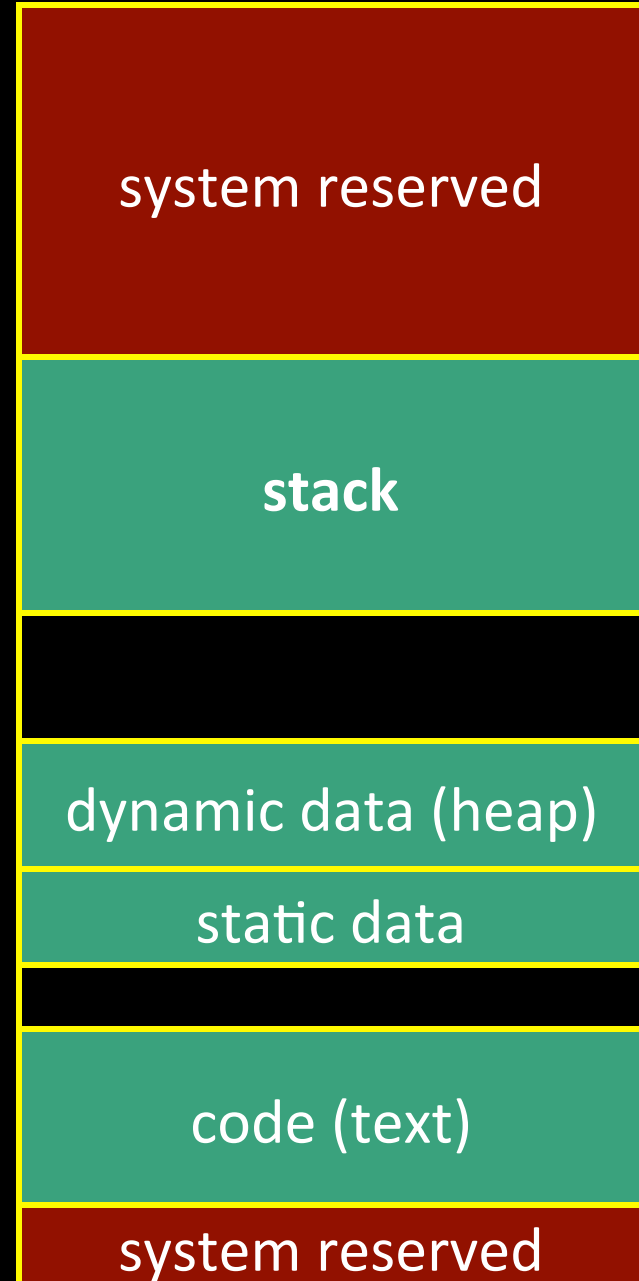
0xfffffffffc

0x80000000

0x10000000

0x00400000

0x00000000



# Example: Stack frame push / pop

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

```
main:  
    # allocate frame  
    ADDUI $sp, $sp, -12 # $ra, x, y  
    # save return address in frame  
    SW $ra, 8($sp)
```

```
    # restore return address  
    LW $ra, 8($sp)  
    # deallocate frame  
    ADDUI $sp, $sp, 12
```

# Recap

---

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?



# Arguments & Return Values

---

```
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  $\$a0\ldots\$a3$ , “spill” rest to stack

# Argument Spilling

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

# Argument Spilling

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

...

# VarArgs

---

## 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

# Recap

---

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