Calling Conventions

Hakim Weatherspoon
CS 3410, Spring 2011
Computer Science
Cornell University

See P&H 2.8 and 2.12

Announcements

PA2 due *next* Friday

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

Use your resources

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

Announcements

Prelims1: *this* Thursday, March 10th 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

Last time

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

Today

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

FAQ

Example program

0xFFFFFFFC

calc.c

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

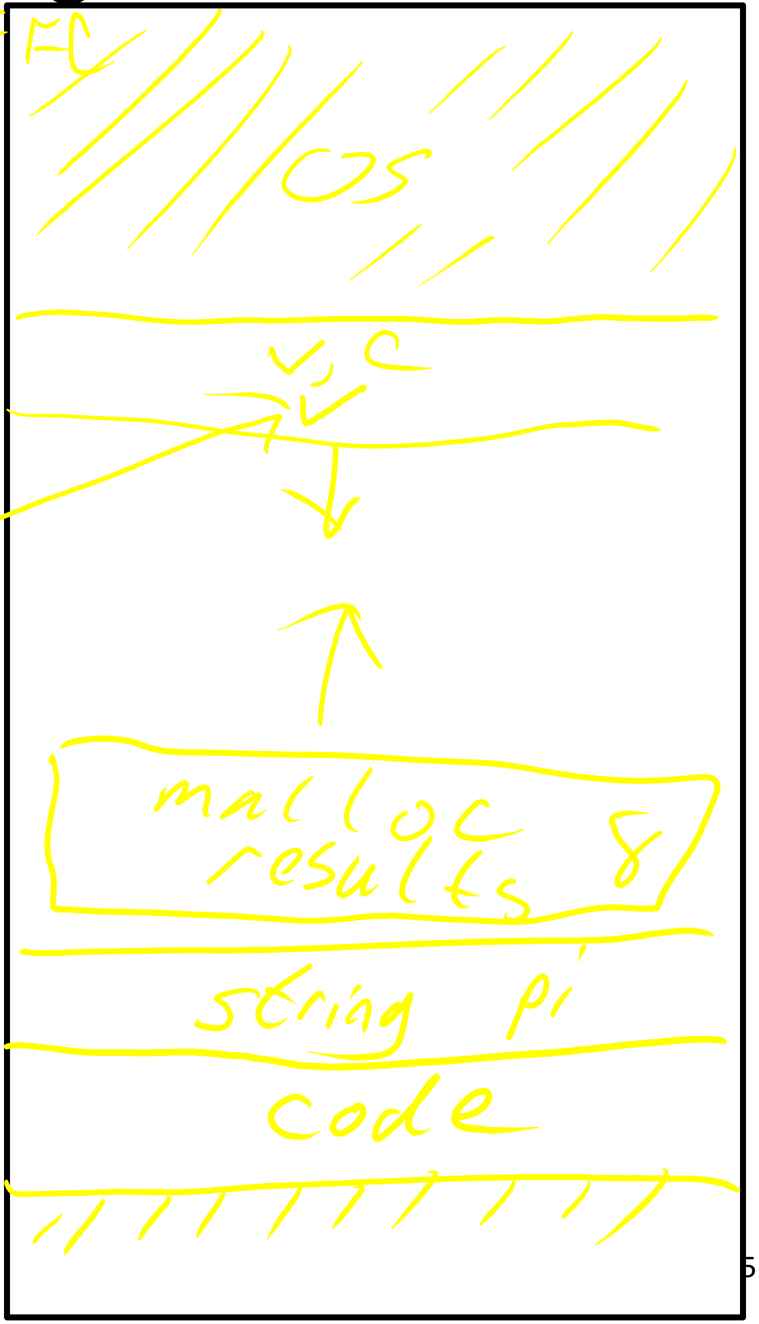
math.c

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

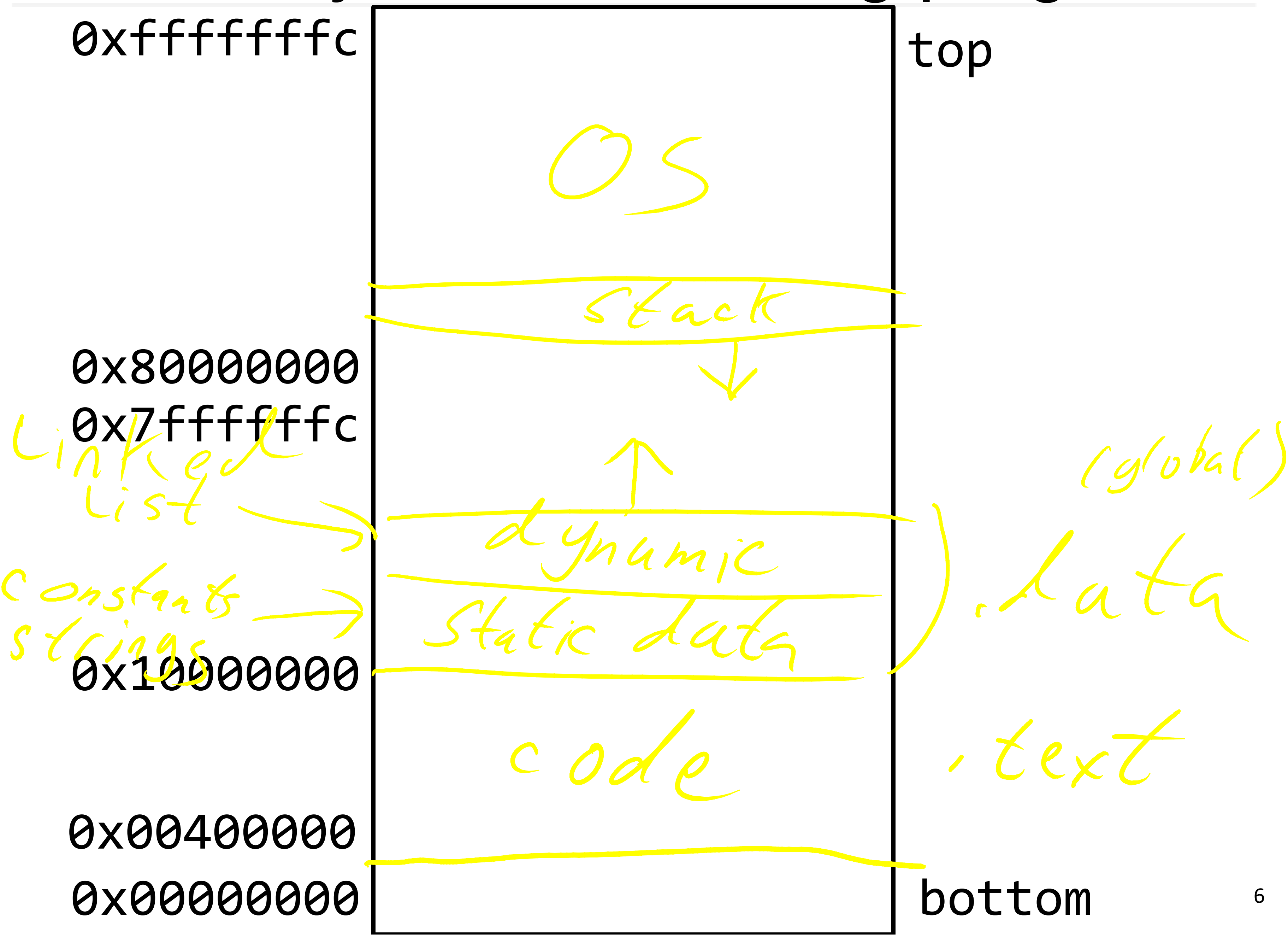
lib3410.o

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

0x0006000



Anatomy of an executing program



0xfffffffc

top

0x80000000

0x7ffffffc

0x10000000

0x00400000

0x00000000

bottom

math.s

math.c

r3

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

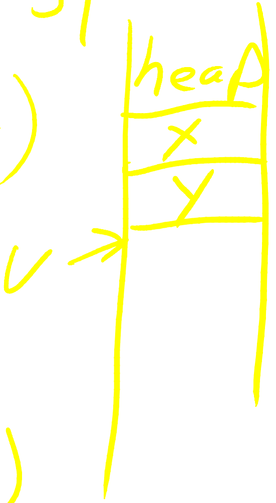
where to put args ret values

abs:

```
   # arg in r3, return address in r31
   # leaves result in r3
BLZ r3, neg
J   r31
neg:
   sub r3, r0, r3
   J r31
```

tnorm:

```
   # arg in r4, return address in r31
   # leaves result in r4
   Move r30, r31
   lw   r3, 0(r4)
   JAL  abs
   Move r6, r3
   lw   r3, 4(r4)
   JAL  abs        $ra = PC+4
   ADD  r4, r6, r3
   JR   r30
```



calc.s

calc.c

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

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

```
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)   ✓ # store x at vx
  LA r3, str2       # call prompt: arg in r3, ret in r3
  JAL prompt
  SW r3, 4(r6)   ✓ # store y at v+4
  MOVE r4, r6 # call tnorm: arg in r4, ret in r4
  JAL tnorm      ✓ r4 = abs(x) + abs(y)
  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
```

PROBLEM
clobbers r3, r30
r6

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;
}
```

A main calls
1 ask
2 test

B test calls
1 sqrt
2 print

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

Pseudo inst

BLZ = SLT $at, , ,

BNE $at

Example: Invoke

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

```
main:

LA $a0, strX
JAL ask # result in $v0
Move $r16, $v0
LA $a0, strY
JAL ask # result in $v0
Move $r17, $v0
Move $a0, r16 # x
Move $a1, r17 # y
JAL test
JR $ra
```

```
data
strX "x"
strY "y"
```

"Call stack"
assures
r16 ← $ra
do not change

Call Stack

Call stack

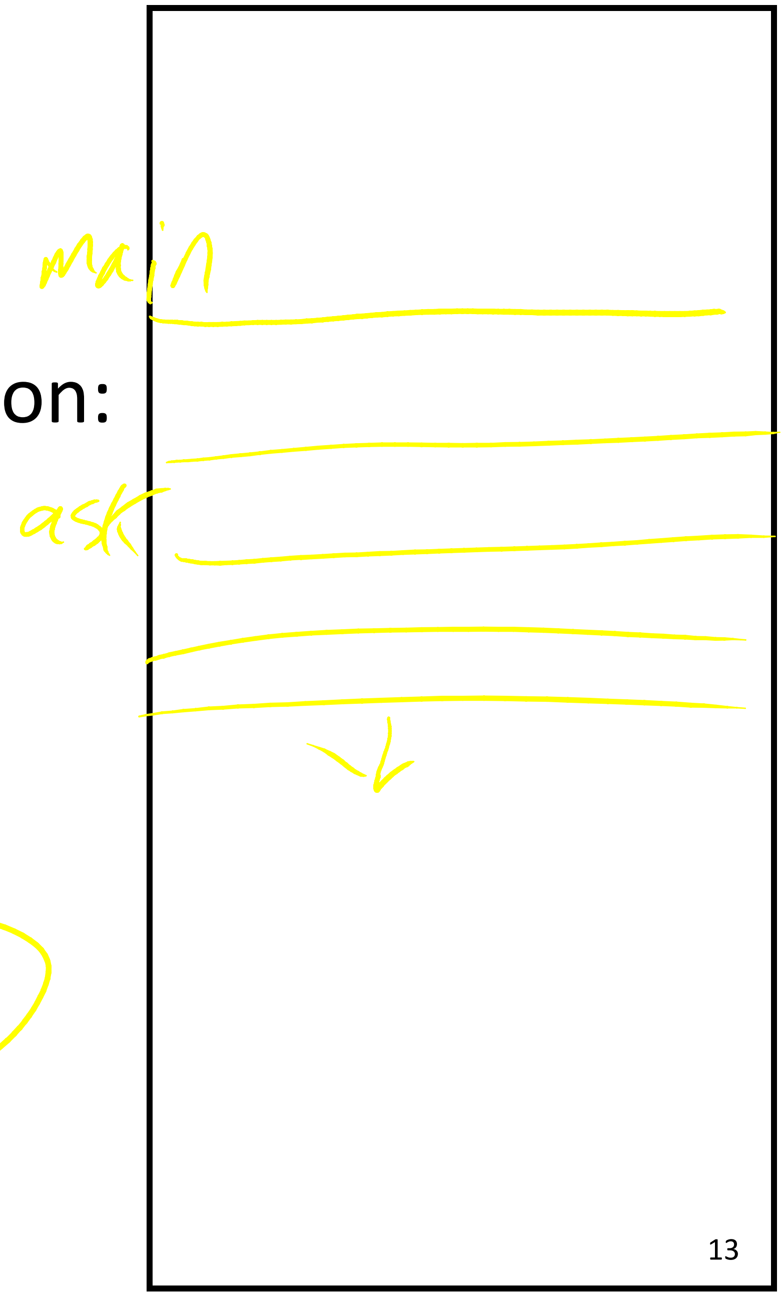
- contains *activation records* (aka *stack frames*)

One for each function invocation:

- saved return address
- local variables
- … and more

Simplification:

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



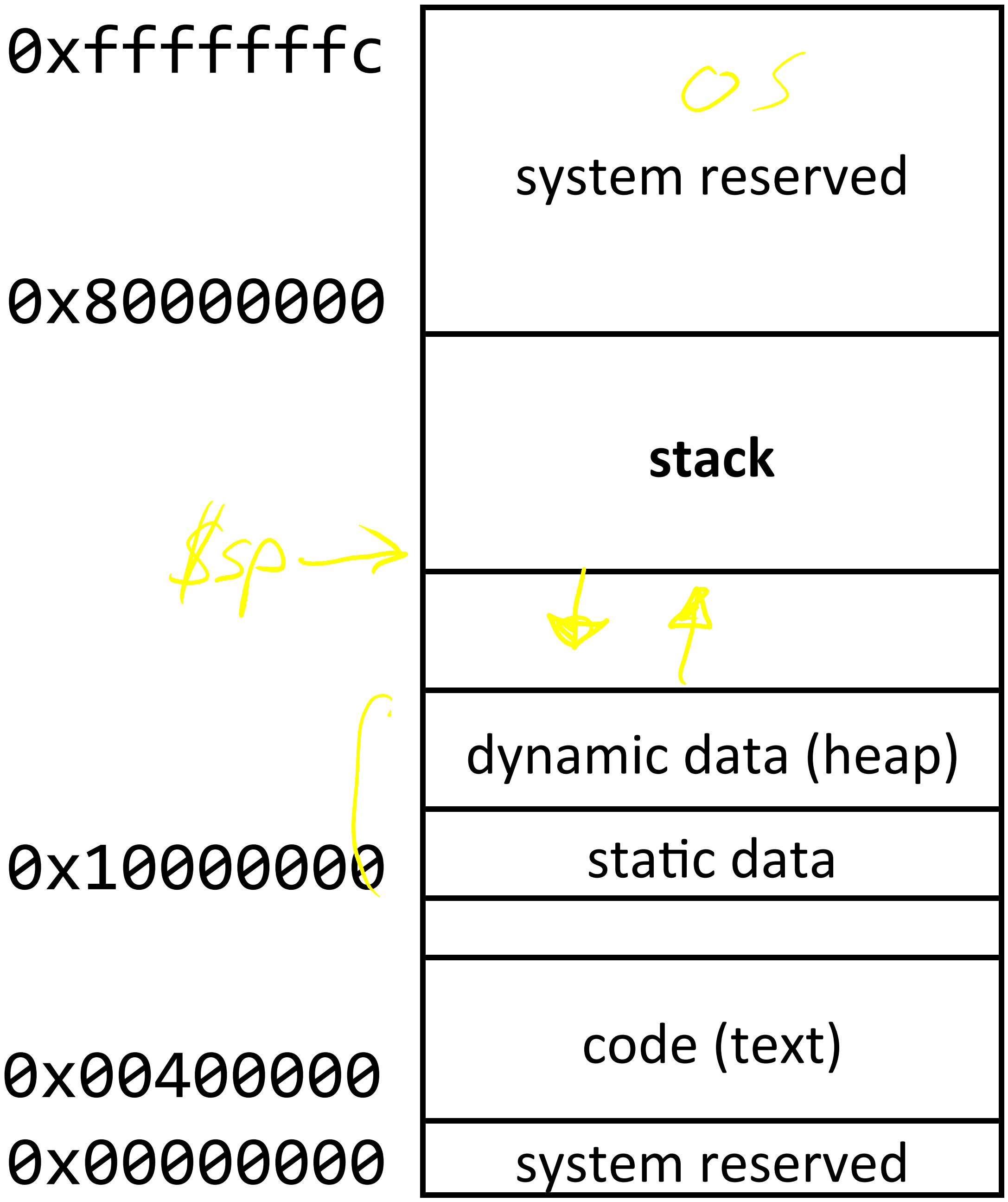
Stack Growth

Convention:

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

Stack grows **down**

Heap grows **up**



0xfffffffc

OS

system reserved

0x80000000

stack

$sp

dynamic data (heap)

0x10000000

static data

code (text)

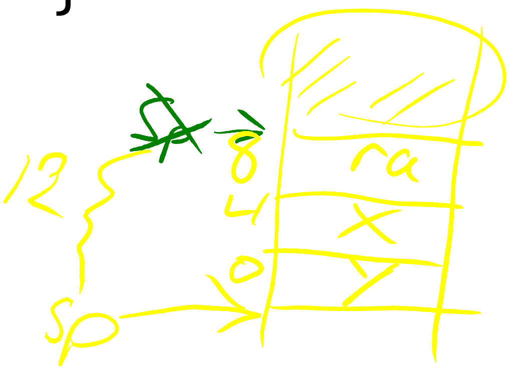
0x00400000

0x00000000

system reserved

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)

  sw $v0, 4($sp)

  sw $v0, 0($sp)

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

callee push stack frame

pop stack frame

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); $a0, a1

int paint(char c, short d, struct point p); $a0, $a1

int treesort(struct Tree *root, int[] A); $a2, $3

struct Tree *createTree(); 32bits $a0, $a1

int max(int a, int b, int c, int d, int e);

Conventions: a, b, c, d in regs

- align everything to multiples of 4 bytes e on stack
- first 4 words in $a0...$a3, "spill" rest to stack a0-a3

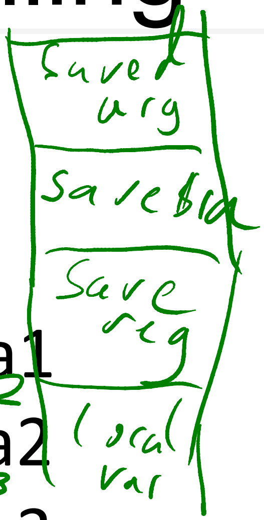
Argument Spilling

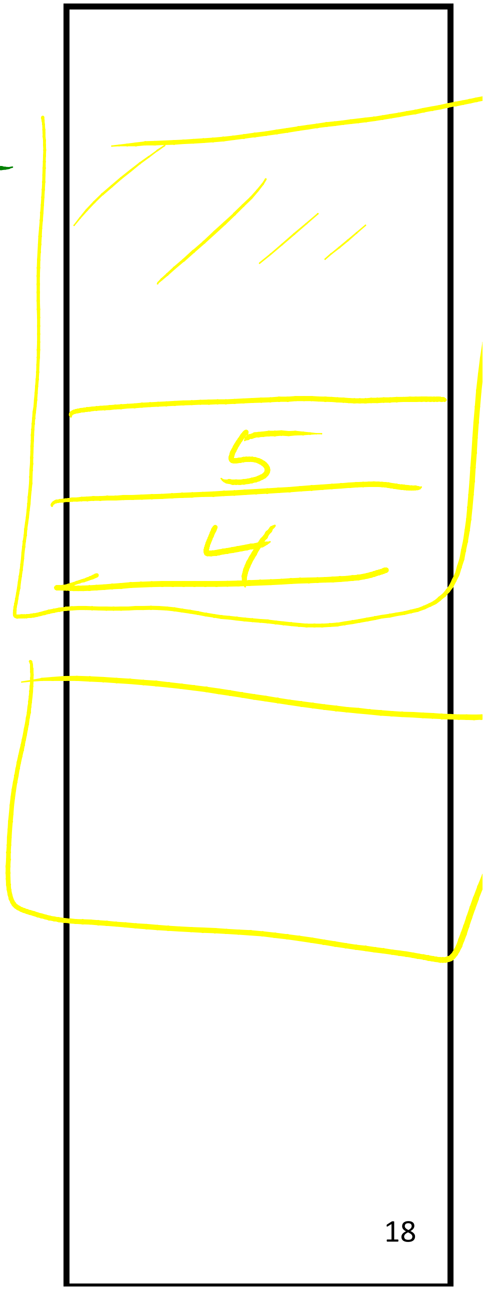
invoke sum(0, 1, 2, 3, 4, 5);

```
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
```

1 = 0+1
3 = 1+2
6 = 3+3
10 = 6+4
5
15 = 10+5





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
```

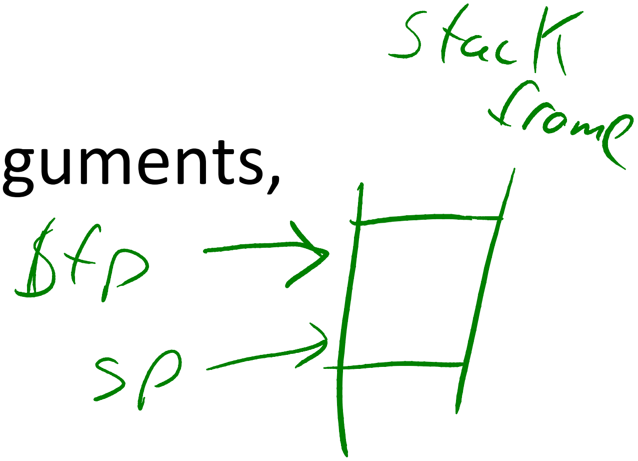
```
printf:
…
if (argno == 0)
      use $a0
else if (argno == 1)
      use $a1
else if (argno == 2)
      use $a2
else if (argno == 3)
      use $a3
else
      use $sp+(4*argno)
…
```

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

Debugging

0(sp) in P

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

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

Prolog ↑

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

higher ↑

ra for vnorm

0x7FFFFFB0

$sp → AC

What func is running? printf

Who called it? vnorm

Has it called anything? No

Will it? b/c no space for 4 args

Args? str1, 0x15

Stack depth? 0(sp) in printf
4+8(sp) args str1, 0x15
20(sp) ra for main

Call trace? printf, vnorm, main, init

Frame Pointer

Frame pointer marks boundaries

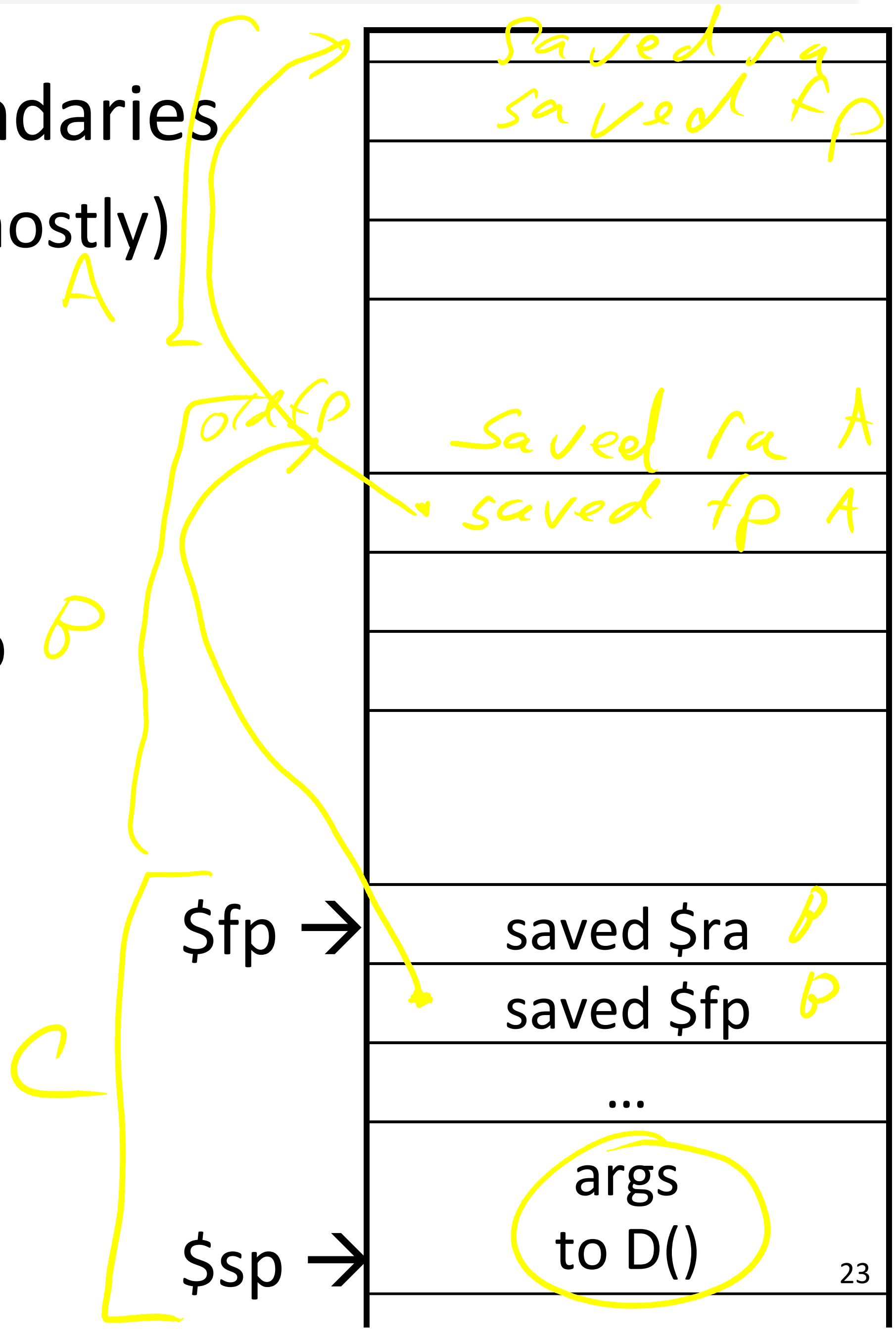
- Optional (for debugging, mostly)

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()



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	$sp	**stack pointer**
r14			r30	$fp	**frame pointer**
r15			r31	$ra	return address

Global Pointer

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)

data ranges 0x1000 0000

0x1000 FFFF

middle

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	$gp	**global pointer**
r13			r29	$sp	stack pointer
r14			r30	$fp	frame pointer
r15			r31	$ra	return address

Callee and Caller Saved Registers

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

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

Callee-save: Always…

- save before modifying
- restore before returning

Caller-save: If necessary…

- save before calling anything
- restore after it returns

1 caller-save temporary stuff

2 callee-save stuff needs to be preserved

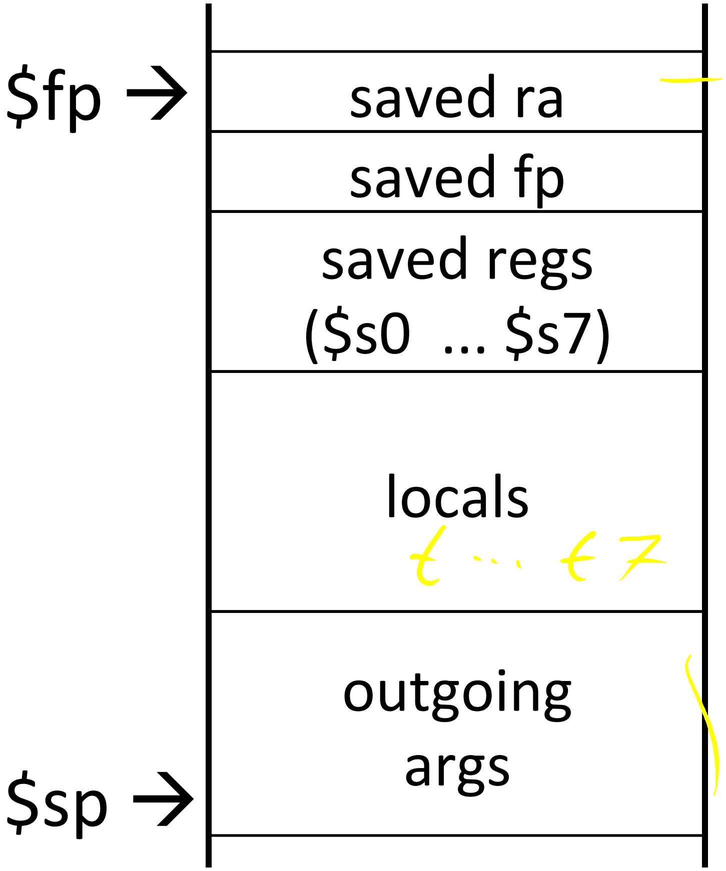
MIPS Register Conventions

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

Recap

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



$fp →

saved ra

saved fp

saved regs ($s0 ... $s7)

locals

t ... t7

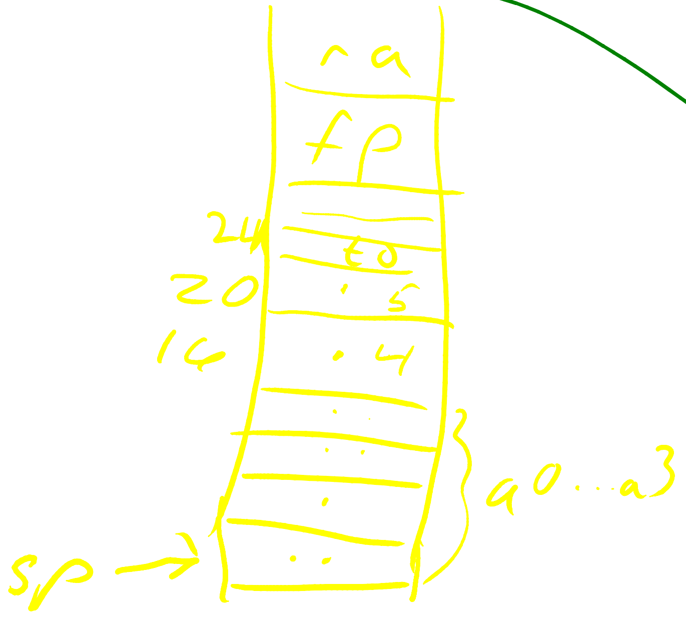
outgoing args

$sp →

Example

a0 a1

```
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;
}
```



Prolog

```
Move s0, a0
Move s1, a1
AND t0, a0, a1
OR t1, a0, a1
ADD t0, t0, t1
Move a0, t0
LI a1, 1
LI a2, 2
LI a3, 3
LI t1, 4
sw t1, 16(sp)
LI t1, 5
Sw t1, 20(sp)
(# tmp) sw t0, 24(sp)
JAL Sum
nop
```

```
LW t0, 24(sp)
   (tmp in t0)
Move a0, v0
 (s is 1st arg of sum)
Move a1, t0
Move a2, s1
Move a3, s0
sw s1, 16(sp)
Sw s0, 20(sp)
JAL sum
nop
# u is in v0
ADD v0, v0, s0
 (u+a)
ADD v0, v0, s1
```

Epilog

Prolog, Epilog

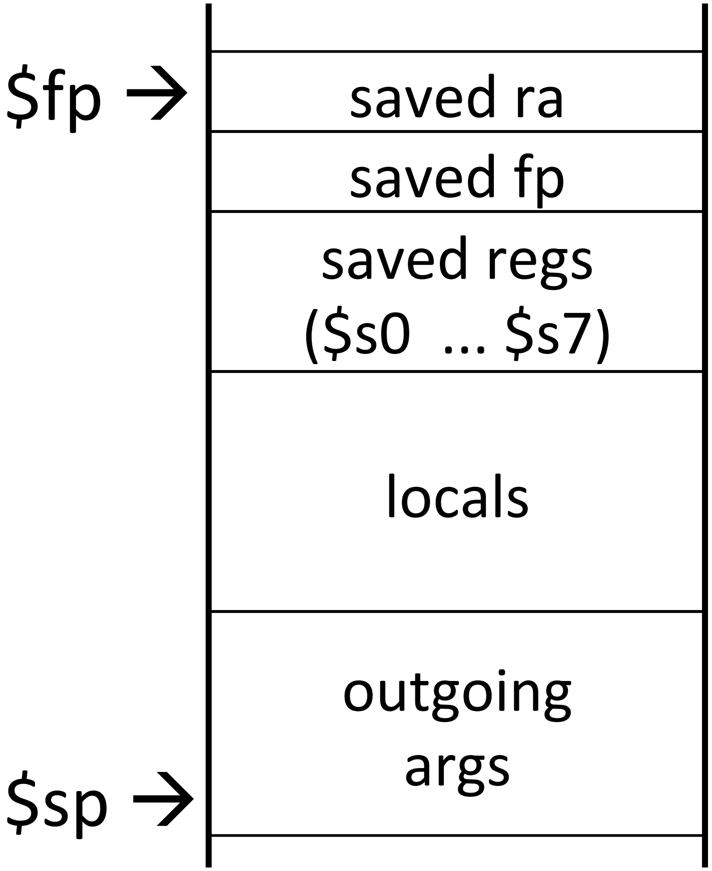
```
test:                          # uses...
ADDIU sp, sp, -44              # allocate frame
sw ra, 40(sp)                  # save $ra
sw fp, 36(sp)                  # save old $fp
sw s0, 32(sp)                  # save ...
sw s1, 28(sp)                  # save ...
ADDIU fp, sp, 40               # set new frame pointer
body                              ...
                                  ...
lw s1, 28(sp)                  # restore ...
lw s0, 32(sp)                  # restore ...
lw fp, 36(sp)                  # restore old $fp
lw ra, 40(sp)                  # restore $ra
ADDIU sp, sp, +44              # dealloc frame
JR ra
```

t will be at 24(sp)

Recap

Minimum stack size for a standard function?

4 × 6 (ra + fp + 4 arg)



Leaf Functions

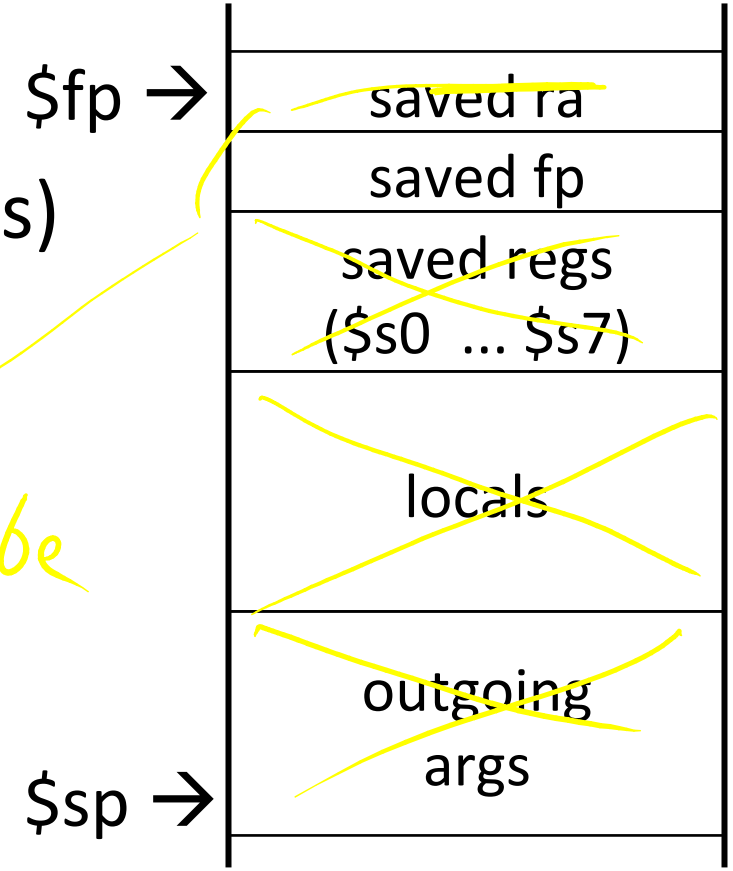
Leaf function does not invoke any other functions

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

2×4 = 8 by

Optimizations?

- No saved regs (or locals)
- No outgoing args
- Don't push $ra
- No frame at all? maybe



Globals and Locals

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; } — death

FAQ

FAQ

- caller/callee saved registers
- CPI
- writing assembling
- reading assembly

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

Caller-save registers are responsibility of the caller

- Caller-save register values saved only if used after call/return
- The callee function can use caller-saved registers

Callee-save register are the responsibility of the callee

- Values must be saved by callee before they can be used
- Caller can assume that these registers will be restored

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

eax, ecx, and edx are caller-save…

- … a function can freely modify these registers
- … but must assume that their contents have been destroyed if it in turns calls a function.

ebx, esi, edi, ebp, esp are callee-save

- A function may call another function and know that the callee-save registers have not been modified
- However, if it modifies these registers itself, it must restore them to their original values before returning.

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

A caller-save register must be saved and restored around any call to a subprogram.
In contrast, for a callee-save register, a caller need do no extra work at a call site (the callee saves and restores the register if it is used).

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

CALLER SAVED: MIPS calls these temporary registers, $t0-t9

- the calling program saves the registers that it does not want a called procedure to overwrite
- register values are NOT preserved across procedure calls

CALLEE SAVED: MIPS calls these saved registers, $s0-s8

- register values are preserved across procedure calls
- the called procedure saves register values in its AR, uses the registers for local variables, restores register values before it returns.

Caller-saved vs. Callee-saved

Caller-save: If necessary… ($t0 .. $t9)

- save before calling anything; restore after it returns

Callee-save: Always… ($s0 .. $s7)

- save before modifying; restore before returning

Registers $t0-$t9 are caller-saved registers

- … that are used to hold temporary quantities
- … that need not be preserved across calls

Registers $s0-s8 are callee-saved registers

- … that hold long-lived values
- … that should be preserved across calls

~~caller-saved register~~

- ~~A register saved by the routine being called~~

~~callee-saved register~~

- ~~A register saved by the routine making a procedure call.~~

What is it?

CPI

Cycles Per Instruction

~~A measure of latency (delay)?~~

~~“ADD takes 5 cycles to finish”~~

or

A measure of throughput?

“N ADDs are completed in N cycles”

CPI = weighted average *throughput* over all instructions *in a given workload*

CPI = 1.0 means that on average…

… an instruction is completed every 1 cycle

CPI = 2.0 means that on average…

… an instruction is completed every 2 cycles

CPI = 5.0 means that on average…

… an instruction is completed every 5 cycles

Example CPI = 1.0

CPI = 1.0 means that on average…

… an instruction is completed every 1 cycle

Example CPI = 2.0

CPI = 2.0 means that on average…

… an instruction is completed every 2 cycles

Example CPI = 0.5

CPI = 0.5 means that on average…

… an instruction is completed every 0.5 cycles

CPI Calculation

Suppose 10 stage pipeline and…

- 1 instruction zapped on every taken jump or branch
- 3 stalls for every memory operation

Q: What is CPI?

… for pure arithmetic workload? 1

… for pure memory workload? 4

… for pure jump workload? 2

… for 50/50 arithmetic/jump workload?

.5×1 + .5×2 = 1.5

… for 50%/25%/25% arith/mem/branch?

.5×1 + .25×4 + .25×2 = 2

… if one fifth of the branches are taken?

.5 ×1 + .25×4 + .20×1 + .5×2 = 1.8