Calling Conventions

Hakim Weatherspoon

CS 3410

Computer Science

Cornell University

The slides are the product of many rounds of teaching CS 3410 by Professors Weatherspoon, Bala, Bracy, McKee, and Sirer.
Big Picture: Where are we now?
Big Picture: Where are we going?

```c
int x = 10;
x = 2 * x + 15;
```

C

compiler

MIPS

assembly

assembler

machine

code

CPU

Circuits

Gates

Transistors

Silicon

```assembly
addiu r5, r0, 10
muli r5, r5, 2
addiu r5, r5, 15
```

r0 = 0

r5 = r0 + 10

r5 = r5 << 1 #r5 = r5 * 2

r5 = r15 + 15

```
001000000000010100000000000001010
0000000000000101001010100000000000000000000010101
001000010100101000000000000001111
```

```
000000000000010100101010000000000000000001111
```

```
00100000101001010000000000001111
```

```
0000000000000101001010100000000000000000010101
```

op = addiu r0         r5                                    10

op = addiu r5         r5 15

op = r-type               r5       r5 shamt=1     func=sll

r5 = r0 + 10

r5 = r5 * 2

r5 = r15 + 15
Goals for this week

Calling Convention for Procedure Calls
Enable code to be reused by allowing code snippets to be invoked

Will need a way to

• call the routine (i.e. transfer control to procedure)
• pass arguments
  – fixed length, variable length, recursively
• return to the caller
  – Putting results in a place where caller can find them
• Manage register
Calling Convention for Procedure Calls

Transfer Control
- Caller → Routine
- Routine → Caller

Pass Arguments to and from the routine
- fixed length, variable length, recursively
- Get return value back to the caller

Manage Registers
- Allow each routine to use registers
- Prevent routines from clobbering each others’ data

What is a Convention?
Warning: There is no one true MIPS calling convention.
lecture != book != gcc != spim != web
How do we share registers and use memory when making procedure calls?
Cheat Sheet and Mental Model for Today

- 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
- 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
- callee save regs are preserved
- caller save regs are not
- Global data accessed via $gp

<table>
<thead>
<tr>
<th>$sp</th>
<th>$fp</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>saved ra</td>
</tr>
<tr>
<td></td>
<td>saved fp</td>
</tr>
<tr>
<td></td>
<td>saved regs ($s0 ... $s7)</td>
</tr>
<tr>
<td>locals</td>
<td></td>
</tr>
<tr>
<td>outgoing args</td>
<td></td>
</tr>
</tbody>
</table>
MIPS Register

Return address: $31 (ra)
Stack pointer: $29 (sp)
Frame pointer: $30 (fp)
First four arguments: $4-$7 (a0-a3)
Return result: $2-$3 (v0-v1)
Callee-save free regs: $16-$23 (s0-s7)
Caller-save free regs: $8-$15,$24,$25 (t0-t9)
Reserved: $26, $27
Global pointer: $28 (gp)
Assembler temporary: $1 (at)
## MIPS Register Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>$zero</td>
</tr>
<tr>
<td>r1</td>
<td>$at</td>
</tr>
<tr>
<td>r2</td>
<td>$v0</td>
</tr>
<tr>
<td>r3</td>
<td>$v1</td>
</tr>
<tr>
<td>r4</td>
<td>$a0</td>
</tr>
<tr>
<td>r5</td>
<td>$a1</td>
</tr>
<tr>
<td>r6</td>
<td>$a2</td>
</tr>
<tr>
<td>r7</td>
<td>$a3</td>
</tr>
<tr>
<td>r8</td>
<td>$t0</td>
</tr>
<tr>
<td>r9</td>
<td>$t1</td>
</tr>
<tr>
<td>r10</td>
<td>$t2</td>
</tr>
<tr>
<td>r11</td>
<td>$t3</td>
</tr>
<tr>
<td>r12</td>
<td>$t4</td>
</tr>
<tr>
<td>r13</td>
<td>$t5</td>
</tr>
<tr>
<td>r14</td>
<td>$t6</td>
</tr>
<tr>
<td>r15</td>
<td>$t7</td>
</tr>
<tr>
<td>r16</td>
<td>$s0</td>
</tr>
<tr>
<td>r17</td>
<td>$s1</td>
</tr>
<tr>
<td>r18</td>
<td>$s2</td>
</tr>
<tr>
<td>r19</td>
<td>$s3</td>
</tr>
<tr>
<td>r20</td>
<td>$s4</td>
</tr>
<tr>
<td>r21</td>
<td>$s5</td>
</tr>
<tr>
<td>r22</td>
<td>$s6</td>
</tr>
<tr>
<td>r23</td>
<td>$s7</td>
</tr>
<tr>
<td>r24</td>
<td>$t8</td>
</tr>
<tr>
<td>r25</td>
<td>$t9</td>
</tr>
<tr>
<td>r26</td>
<td>$k0</td>
</tr>
<tr>
<td>r27</td>
<td>$k1</td>
</tr>
<tr>
<td>r28</td>
<td>$gp</td>
</tr>
<tr>
<td>r29</td>
<td>$sp</td>
</tr>
<tr>
<td>r30</td>
<td>$fp</td>
</tr>
<tr>
<td>r31</td>
<td>$ra</td>
</tr>
</tbody>
</table>

- **zero**
- **assembler temp**
- **function return values**
- **function arguments**
- **temps (caller save)**
- **saved (callee save)**
- **more temps (caller save)**
- **reserved for kernel**
- **global data pointer**
- **stack pointer**
- **frame pointer**
- **return address**
Calling Convention for Procedure Calls

Transfer Control
- Caller → Routine
- Routine → Caller

Pass Arguments to and from the routine
- fixed length, variable length, recursively
- Get return value back to the caller

Manage Registers
- Allow each routine to use registers
- Prevent routines from clobbering each others’ data

What is a Convention?
Warning: There is no one true MIPS calling convention.
lecture != book != gcc != spim != web
How does a function call work?

```c
int main (int argc, char* argv[ ]) {
    int n = 9;
    int result = myfn(n);
}

int myfn(int n) {
    int f = 1;
    int i = 1;
    int j = n - 1;
    while(j >= 0) {
        f *= i;
        i++;
        j = n - i;
    }
    return f;
}
```
Jumps are not enough

Jumps to the callee
Jumps back
Jumps are not enough

Jumps to the callee
Jumps back
What about multiple sites?

??? Change target on the fly ???
Takeaway 1: Need Jump And Link

JAL (Jump And Link) instruction moves a new value into the PC, and simultaneously saves the old value in register $31 (aka $ra or return address).

Thus, can get back from the subroutine to the instruction immediately following the jump by transferring control back to PC in register $31.
Jump-and-Link / Jump Register

First call

main:
jal myfn

after1:
add $1,$2,$3
jal myfn

after2:
sub $3,$4,$5

myfn:
...
...
jr $31

JAL saves the PC in register $31
Subroutine returns by jumping to $31
JAL saves the PC in register $31
Subroutine returns by jumping to $31
What happens for recursive invocations?

main:
  jal myfn
after1:
  add $1,$2,$3
  jal myfn
after2:
  sub $3,$4,$5
myfn:
  ...
  ...
  jr $31

Second call
r31
JAL / JR for Recursion?

```c
int main (int argc, char* argv[ ]) {
    int n = 9;
    int result = myfn(n);
}

int myfn(int n) {
    int f = 1;
    int i = 1;
    int j = n - 1;
    while(j >= 0) {
        f *= i;
        i++;
        j = n - i;
    }
    return f;
}
```
JAL / JR for Recursion?

```c
int main (int argc, char* argv[ ]) {
    int n = 9;
    int result = myfn(n);
}

int myfn(int n) {
    if(n > 0) {
        return n * myfn(n - 1);  // highlighted
    } else {
        return 1;
    }
}
```
JAL / JR for Recursion?

First call

main:

jal myfn

after1:

add $1,$2,$3

myfn:

if (test)

jal myfn

after2:

jr $31

Problems with recursion:
Problems with recursion:
JAL / JR for Recursion?

Return from Recursive Call

Problems with recursion:
Need a “Call Stack”

Call stack
- contains activation records (aka stack frames)

Each activation record contains
- the return address for that invocation
- the local variables for that procedure

A stack pointer (sp) keeps track of the top of the stack
- dedicated register ($29) on the MIPS

Manipulated by push/pop operations
- push: move sp down, store
- pop: load, move sp up
Cheat Sheet and Mental Model for Today

- Write-Back Memory
- Instruction Fetch
- Instruction Decode
- Instruction Execution
- Instruction Write-Back

- PC
- Memory
- Register File
- Instruction
- New PC
- Control
- Extend
- Detect Hazard
- Instruction Decode
- Forward Unit
- Execute
- Memory
- Write-Back

- Compute jump/branch targets
- Control
- ALU
- Address
- d_in, d_out
- Memory

+4

Forward unit detects hazards.
Need a “Call Stack”

Call stack

• contains activation records (aka stack frames)

Each activation record contains

• the return address for that invocation
• the local variables for that procedure

A stack pointer (sp) keeps track of the top of the stack

• dedicated register ($29) on the MIPS

Manipulated by push/pop operations

• push: move sp down, store
• pop: load, move sp up

Push: ADDIU $sp, $sp, -4
SW $31, 0 ($sp)
Need a “Call Stack”

Call stack
- contains activation records (aka stack frames)

Each activation record contains
- the return address for that invocation
- the local variables for that procedure

A stack pointer (sp) keeps track of the top of the stack
- dedicated register ($29) on the MIPS

Manipulated by push/pop operations
- push: move sp down, store
- pop: load, move sp up

```
Push: ADDIU $sp, $sp, -4
SW $31, 0 ($sp)

Pop: LW $31, 0 ($sp)
ADDIU $sp, $sp, 4
JR $31
```
Need a “Call Stack”

Stack used to save and restore contents of $31
Need a “Call Stack”

Stack used to save and restore contents of $31
Stack Growth

(Call) Stacks start at a high address in memory

Stacks grow down as frames are pushed on

- Note: data region starts at a low address and grows up
- The growth potential of stacks and data region are not artificially limited
An executing program in memory

- **.text**
- **.data**

0x40000000

- Code (text)
- Static data
- Dynamic data (heap)
- Stack
- System reserved

0x00000000

- System reserved
An executing program in memory

- **0xfffffffffc**: System reserved
- **0x80000000**: Stack
- **0x7ffffffffc**: Dynamic data (heap)
- **0x10000000**: Static data
- **0x00400000**: Code (text)
- **0x00000000**: System reserved

*“Data Memory”*

*“Program Memory”*

Top

Bottom
Anatomy of an executing program

Instruction Fetch -> IF/ID

Instruction Decode -> ID/EX

Execute -> EX/MEM

Write-Back -> MEM/WB

Memory

Stack, Data, Code Stored in Memory

compute jump/branch targets

register file, $29 ($sp), $31 ($ra)

forward unit

detect hazard

system reserved

stack

dynamic data (heap)

static data

code (text)

system reserved

alu

control

extend

new pc

Instruction Fetch

PC

inst

+4

new pc

Instruction Decode

memory

PC

inst

+4
An executing program in memory

0xfffffffffc  
0x80000000  
0x7ffffffffc

system reserved

stack

dynamic data (heap)

static data

code (text)

system reserved

“Data Memory”

“Program Memory”

bottom
The Stack

Stack contains stack frames (aka “activation records”)
• 1 stack frame per dynamic function
• Exists only for the duration of function
• Grows down, “top” of stack is $sp, r29
• Example: lw $r1, 0($sp) puts word at top of stack into $r1

Each stack frame contains:
• Local variables, return address (later), register backups (later)

```c
int main(...) {
    ...
    myfn(x);
}
int myfn(int n) {
    ...
    myfn();
}
```
The Heap

Heap holds dynamically allocated memory

- Program must maintain pointers to anything allocated
  - Example: if $r3 holds x
  - lw $r1, 0($r3) gets first word x points to
- Data exists from malloc() to free()

```
void some_function() {
    int *x = malloc(1000);
    int *y = malloc(2000);
    free(y);
    int *z = malloc(3000);
}
```
Data Segment

Data segment contains global variables
• Exist for all time, accessible to all routines
• Accessed w/global pointer
  • $gp, r28, points to middle of segment
  • Example: lw $r1, 0($gp) gets middle-most word
    (here, max_players)

```c
int max_players = 4;

int main(...) {
    ...
}
```
```c
int n = 100;

int main (int argc, char* argv[ ]) {
    int i, m = n, sum = 0;
    int* A = malloc(4*m + 4);
    for (i = 1; i <= m; i++) {
        sum += i; A[i] = sum;
    }
    printf ("Sum 1 to %d is %d\n", n, sum);
}
```
Takeaway2: Need a Call Stack

JAL (Jump And Link) instruction moves a new value into the PC, and simultaneously saves the old value in register $31 (aka $ra or return address). Thus, can get back from the subroutine to the instruction immediately following the jump by transferring control back to PC in register $31.

Need a Call Stack to return to correct calling procedure. To maintain a stack, need to store an activation record (aka a “stack frame”) in memory. Stacks keep track of the correct return address by storing the contents of $31 in memory (the stack).
Calling Convention for Procedure Calls

Transfer Control

• Caller $\rightarrow$ Routine
• Routine $\rightarrow$ Caller

Pass Arguments to and from the routine

• fixed length, variable length, recursively
• Get return value back to the caller

Manage Registers

• Allow each routine to use registers
• Prevent routines from clobbering each others’ data
Next Goal
Need consistent way of passing arguments and getting the result of a subroutine invocation
Arguments & Return Values

Need consistent way of passing arguments and getting the result of a subroutine invocation

Given a procedure signature, need to know where arguments should be placed

- int min(int a, int b);
- int subf(int a, int b, int c, int d, int e);
- int isalpha(char c);
- int treesort(struct Tree *root);
- struct Node *createNode();
- struct Node mynode();

Too many combinations of char, short, int, void *, struct, etc.
- MIPS treats char, short, int and void * identically
Simple Argument Passing (1-4 args)

```c
main() {
    int x = myfn(6, 7);
    x = x + 2;
}
```

First four arguments:
- passed in registers $4$-$7$
  - aka $a0$, $a1$, $a2$, $a3$

Returned result:
- passed back in a register
  - Specifically, $2$, aka $v0$

```c
main:
    li $a0, 6
    li $a1, 7
    jal myfn
    addiu $r1, $v0, 2
```

Note: This is *not* the entire story for 1-4 arguments. Please see *the Full Story* slides.
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)

Q: What about argument lists?
Many Arguments (5+ args)

main() {
    myfn(0,1,2,3,4,5);
    ...
}

main:
    li $a0, 0
    li $a1, 1
    li $a2, 2
    li $a3, 3
    addiu $sp,$sp,-8
    li $8, 4
    sw $8, 0($sp)
    li $8, 5
    sw $8, 4($sp)
    jal myfn

First four arguments:
  passed in $4-$7
  • aka $a0-$a3

Subsequent arguments:
  "spill" onto the stack

Note: This is not the entire story for 5+ arguments.
Please see the Full Story slides.
Arguments Passing: the Full Story

```plaintext
main()
{
  myfn(0,1,2,3,4,5);
  ...
}
```

Arguments 1-4:
passed in $4-$7
room on stack

Arguments 5+:
placed on stack

Stack decremented by
max(16, #args x 4)
Here: max (16, 24) = 24
Pros of Argument Passing Convention

• Consistent way of passing arguments to and from subroutines
• Creates single location for all arguments
  • Caller makes room for $a0-$a3 on stack
  • Callee must copy values from $a0-$a3 to stack
    → callee may treat all args as an array in memory
  • Particularly helpful for functions w/ variable length inputs:
    printf("Scores: %d %d %d\n", 1, 2, 3);
• Aside: not a bad place to store inputs if callee needs to call a function (your input cannot stay in $a0 if you need to call another function!)
Frame Layout & the Frame Pointer

```
blue() {
    pink(0,1,2,3,4,5);
}
```
Frame Layout & the Frame Pointer

Notice
- Pink’s arguments are on blue’s stack
- \( sp \) changes as functions call other functions, complicates accesses

→ Convenient to keep pointer to bottom of stack == frame pointer $30$, aka $fp$

\[ fp \] can be used to restore $sp$ on exit

\[
\text{blue}() \{
    \text{pink}(0,1,2,3,4,5);
\}
\]

\[
\text{pink}(\text{int } a, \text{int } b, \text{int } c, \text{int } d, \text{int } e, \text{int } f) \{
    \ldots
\}
\]
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, $v1
• stack frame ($fp to $sp) contains:
  – $ra (clobbered on JAL to sub-functions)
  – space for 4 arguments to Callee
  – arguments 5+ to Callee
## MIPS Register Conventions so far:

<table>
<thead>
<tr>
<th>r0</th>
<th>$zero</th>
<th>zero</th>
<th>r16</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r1</td>
<td>$at</td>
<td>assembler temp</td>
<td>r17</td>
<td></td>
</tr>
<tr>
<td>r2</td>
<td>$v0</td>
<td>function</td>
<td>r18</td>
<td></td>
</tr>
<tr>
<td>r3</td>
<td>$v1</td>
<td>return values</td>
<td>r19</td>
<td>Pseudo-Instructions</td>
</tr>
<tr>
<td>r4</td>
<td>$a0</td>
<td></td>
<td>r20</td>
<td>e.g. BLZ</td>
</tr>
<tr>
<td>r5</td>
<td>$a1</td>
<td>function</td>
<td>r21</td>
<td>SLT $at</td>
</tr>
<tr>
<td>r6</td>
<td>$a2</td>
<td>arguments</td>
<td>r22</td>
<td>BNE $at, 0, L</td>
</tr>
<tr>
<td>r7</td>
<td>$a3</td>
<td></td>
<td>r23</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td></td>
<td></td>
<td>r24</td>
<td></td>
</tr>
<tr>
<td>r9</td>
<td></td>
<td></td>
<td>r25</td>
<td></td>
</tr>
<tr>
<td>r10</td>
<td></td>
<td></td>
<td>r26</td>
<td>$k0 reserved for OS kernel</td>
</tr>
<tr>
<td>r11</td>
<td></td>
<td></td>
<td>r27</td>
<td>$k1</td>
</tr>
<tr>
<td>r12</td>
<td></td>
<td></td>
<td>r28</td>
<td></td>
</tr>
<tr>
<td>r13</td>
<td></td>
<td></td>
<td>r29</td>
<td></td>
</tr>
<tr>
<td>r14</td>
<td></td>
<td></td>
<td>r30</td>
<td></td>
</tr>
<tr>
<td>r15</td>
<td></td>
<td></td>
<td>r31</td>
<td>$ra return address</td>
</tr>
</tbody>
</table>
C & MIPS: the fine print

C allows passing whole structs
- int dist(struct Point p1, struct Point p2);
- Treated as collection of consecutive 32-bit arguments
  – Registers for first 4 words, stack for rest
- Better: int dist(struct Point *p1, struct Point *p2);

Where are the arguments to:
void sub(int a, int b, int c, int d, int e);
void isalpha(char c);
void treesort(struct Tree *root);

Where are the return values from:
struct Node *createNode();
struct Node mynode();

Many combinations of char, short, int, void *, struct, etc.
- MIPS treats char, short, int and void * identically
Globals and Locals

**Global variables** are allocated in the “data” region of the program

- Exist for all time, accessible to all routines

**Local variables** are allocated within the stack frame

- Exist solely for the duration of the stack frame

Dangling pointers are pointers into a destroyed stack frame

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

How does a function load global data?

• global variables are just above 0x10000000

Convention: *global pointer*

• $28$ 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)
Anatomy of an executing program

<table>
<thead>
<tr>
<th>Address</th>
<th>Memory Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xfffffffffc</td>
<td>system reserved</td>
</tr>
<tr>
<td>0x80000000</td>
<td>stack</td>
</tr>
<tr>
<td>0x7fffffff</td>
<td>dynamic data (heap)</td>
</tr>
<tr>
<td>0x00000000</td>
<td>static data</td>
</tr>
<tr>
<td>0x00400000</td>
<td>code (text)</td>
</tr>
<tr>
<td>0x00000000</td>
<td>system reserved</td>
</tr>
</tbody>
</table>

$gp

$gp points to 0x10000000
Frame Pointer

It is often cumbersome to keep track of location of data on the stack

• The offsets change as new values are pushed onto and popped off of the stack

Keep a pointer to the bottom of the top stack frame

• Simplifies the task of referring to items on the stack

A frame pointer, $30, aka $fp

• Value of $sp upon procedure entry
• Can be used to restore $sp on exit
Conventions so far

• first four arg words passed in $a0-$a3
• remaining args passed in parent’s stack frame
• return value (if any) in $v0, $v1
• stack frame ($fp to $sp) contains:
  • $ra (clobbered on JALs)
  • space for 4 arguments to Callees
  • arguments 5+ to Callees
• global data accessed via $gp
Calling Convention for Procedure Calls

Transfer Control
- Caller → Routine
- Routine → Caller

Pass Arguments to and from the routine
- fixed length, variable length, recursively
- Get return value back to the caller

Manage Registers
- Allow each routine to use registers
- Prevent routines from clobbering each others’ data
Next Goal

What convention should we use to share use of registers across procedure calls?
Register Management

Functions:
• Are compiled in isolation
• Make use of general purpose registers
• Call other functions in the middle of their execution
  • These functions also use general purpose registers!
  • No way to coordinate between caller & callee

→ Need a convention for register management
Register Usage

Suppose a routine would like to store a value in a register
Two options: callee-save and caller-save

Callee-save:
  • Assume that one of the callers is already using that register to hold a value of interest
  • Save the previous contents of the register on procedure entry, restore just before procedure return
  • E.g. $31

Caller-save:
  • Assume that a caller can clobber any one of the registers
  • Save the previous contents of the register before proc call
  • Restore after the call

MIPS calling convention supports both
Caller-saved

Registers that the caller cares about: $t0... $t9

About to call a function?
• Need value in a t-register after function returns?
  → save it to the stack before fn call
  → restore it from the stack after fn returns
• Don’t need value? → do nothing

Functions
• Can freely use these registers
• Must assume that their contents are destroyed by other functions

Suppose:
- $t0$ holds $x$
- $t1$ holds $y$
- $t2$ holds $z$

Where do we save and restore?

```c
void myfn(int a) {
    int x = 10;
    int y = max(x, a);
    int z = some_fn(y);
    return (z + y);
}
```
Callee-saved

Registers a function intends to use: $s0... $s9

About to use an s-register? You **MUST**:

- Save the current value on the stack *before* using
- Restore the old value from the stack before fn returns

Functions

- Must save these registers before using them
- May assume that their contents are preserved even across fn calls

```c
void myfn(int a) {
    int x = 10;
    int y = max(x, a);
    int z = some_fn(y);
    return (z + y);
}
```

Suppose:

- $s0$ holds $x$
- $s1$ holds $y$
- $s2$ holds $z$

Where do we save and restore?
Caller-Saved Registers in Practice

Assume the registers are free for the taking, use with no overhead

Since subroutines will do the same, must protect values needed later:
  Save before fn call
  Restore after fn call

Notice: Good registers to use if you don’t call too many functions or if the values don’t matter later on anyway.

```plaintext
main:
  ...
  [use $8 & $9]
  ...
  addiu $sp,$sp,-8
  sw $9, 4($sp)
  sw $8, 0($sp)
  jal mult
  lw $9, 4($sp)
  lw $8, 0($sp)
  addiu $sp,$sp,8
  ...
  [use $8 & $9]
```
Caller-Saved Registers in Practice

Assume the registers are free for the taking, use with no overhead

Since subroutines will do the same, must protect values needed later:
- Save before fn call
- Restore after fn call

Notice: Good registers to use if you don’t call too many functions or if the values don’t matter later on anyway.
Callee-Saved Registers in Practice

Assume caller is using the registers

Save on entry

Restore on exit

Notice: Good registers to use if you make a lot of function calls and need values that are preserved across all of them.

Also, good if caller is actually using the registers, otherwise the save and restores are wasted. But hard to know this.
Callee-Saved Registers in Practice

main:
  addiu $sp,$sp,-32
  sw $ra,28($sp)
  sw $fp, 24($sp)
  sw $s1, 20($sp)
  sw $s0, 16($sp)
  addiu $fp, $sp, 28
  ...
  [use $s0 and $s1]
  ...
  lw $ra,28($sp)
  lw $fp,24($sp)
  lw $s1, 20($sp)
  lw $s0, 16($sp)
  addiu $sp,$sp,32
  jr $ra

Assume caller is using the registers
Save on entry
Restore on exit

Notice: Good registers to use if you make a lot of function calls and need values that are preserved across all of them.
Also, good if caller is actually using the registers, otherwise the save and restores are wasted. But hard to know this.
### Frame Layout on Stack

<table>
<thead>
<tr>
<th>fp</th>
<th>saved ra</th>
<th>saved fp</th>
<th>saved regs ($s0 ... $s7)</th>
<th>locals</th>
<th>outgoing args</th>
</tr>
</thead>
</table>

Assume a function uses two callee-save registers.

How do we allocate a stack frame?

How large is the stack frame?

What should be stored in the stack frame?

Where should everything be stored?
Frame Layout on Stack

fp →
- saved ra
- saved fp
- saved regs ($s0 ... $s7)
- locals
- outgoing args

sp →

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADDIU $sp, $sp, -32  # allocate frame</td>
</tr>
<tr>
<td></td>
<td>SW $ra, 28($sp)  # save $ra</td>
</tr>
<tr>
<td></td>
<td>SW $fp, 24($sp)  # save old $fp</td>
</tr>
</tbody>
</table>
|   | SW $s1, 20($sp)  # save ...
|   | SW $s0, 16($sp)  # save ...
|   | ADDIU $fp, $sp, 28  # set new frame ptr |
|   | ...  # BODY ...
|   | LW $s0, 16($sp)  # restore ...
|   | LW $s1, 20($sp)  # restore ...
|   | LW $fp, 24($sp)  # restore old $fp
|   | LW $ra, 28($sp)  # restore $ra
|   | ADDIU $sp,$sp, 32  # dealloc frame
|   | JR $ra  # dealloc frame |
Frame Layout on Stack

blue's ra
saved fp
saved regs
args for pink

pink's ra
blue's fp
saved regs
x
args for orange

fp→ orange's ra
pink's fp
saved regs
buf[100]

sp→

blue() {
    pink(0,1,2,3,4,5);
}
pink(int a, int b, int c, int d, int e, int f) {
    int x;
    orange(10,11,12,13,14);
}
orange(int a, int b, int c, int, d, int e) {
    char buf[100];
    gets(buf);  // no bounds check!
}

What happens if more than 100 bytes is written to buf?
MIPS Register Recap

Return address: $31 (ra)
Stack pointer: $29 (sp)
Frame pointer: $30 (fp)
First four arguments: $4-$7 (a0-a3)
Return result: $2-$3 (v0-v1)
Callee-save free regs: $16-$23 (s0-s7)
Caller-save free regs: $8-$15,$24,$25 (t0-t9)
Reserved: $26, $27
Global pointer: $28 (gp)
Assembler temporary: $1 (at)
# MIPS Register Conventions

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

- first four arg words passed in $a0-$a3
- remaining args passed in parent’s stack frame
- return value (if any) in $v0, $v1
- stack frame ($fp to $sp) contains:
  - $ra (clobbered on JALs)
  - local variables
  - space for 4 arguments to Callees
  - arguments 5+ to Callees
- callee save regs: preserved
- caller save regs: not preserved
- global data accessed via $gp

```
| saved ra | $fp |
| saved fp |
| saved regs ($s0 ... $s7) |
| locals |
| outgoing args |
```

```
$sp →
```

```
$fp →
```
Activity #1: Calling Convention Example

```c
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;
}
```

**Correct Order:**

1. Body First
2. Determine stack frame size
3. Complete Prologue/Epilogue
Activity #2: Calling Convention Example:
Prologue, Epilogue

test:

# allocate frame
# save $ra
# save old $fp
# callee save ...
# callee save ...
# callee save ...
# set new frame ptr
    ...
    ...
# restore ...
# restore ...
# restore ...
# restore old $fp
# restore $ra
# dealloc frame
Next Goal

Can we optimize the assembly code at all?
Activity #3: Calling Convention Example

```c
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;
}
```

How can we optimize the assembly code?
Activity #3: Calling Convention Example: Prologue, Epilogue

test:

    # allocate frame
    # save $ra
    # save old $fp
    # callee save ...
    # callee save ...
    # callee save ...
    # set new frame ptr
        ...
        ...
    # restore ...
    # restore ...
    # restore ...
    # restore old $fp
    # restore $ra
    # dealloc frame
Minimum stack size for a standard function?
Minimum stack size for a standard function?

- $fp \rightarrow$
  - saved ra
  - saved fp
  - saved regs ($s0 \ldots s7$)
  - locals
  - outgoing
  - args

- $sp \rightarrow$
Leaf Functions

*Leaf function* does not invoke any other functions

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

Optimizations?
Next Goal
Given a running program (a process), how do we know what is going on (what function is executing, what arguments were passed to where, where is the stack and current stack frame, where is the code and data, etc)?
Anatomy of an executing program

- 0xfffffffffc: system reserved
- 0x80000000: stack
- 0x7fffffff: dynamic data (heap)
- 0x10000000: static data
- 0x00400000: code (text)
- 0x00000000: system reserved

Top:
- .data
- PC

Bottom:
### Activity #4: Debugging

<table>
<thead>
<tr>
<th>Function</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>init()</td>
<td>0x400000</td>
</tr>
<tr>
<td>printf(s, ...)</td>
<td>0x4002B4</td>
</tr>
<tr>
<td>vnorm(a,b)</td>
<td>0x40107C</td>
</tr>
<tr>
<td>main(a,b)</td>
<td>0x4010A0</td>
</tr>
<tr>
<td>pi</td>
<td>0x10000000</td>
</tr>
<tr>
<td>str1</td>
<td>0x10000004</td>
</tr>
</tbody>
</table>

#### CPU Status

<table>
<thead>
<tr>
<th>CPU:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$pc=0x004003C0</td>
</tr>
<tr>
<td>$sp=0x7FFFFFAC</td>
</tr>
<tr>
<td>$ra=0x00401090</td>
</tr>
</tbody>
</table>

#### Call Trace

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x4003C0</td>
</tr>
<tr>
<td>0x7FFFFF4</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>0x7FFFFFDC</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0x004010c4</td>
</tr>
<tr>
<td>0x7FFFFFDC</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
<tr>
<td>0x00000015</td>
</tr>
<tr>
<td>0x7FFFFFFB0</td>
</tr>
<tr>
<td>0x10000004</td>
</tr>
<tr>
<td>0x00401090</td>
</tr>
</tbody>
</table>

- **What func is running?**
- **Who called it?**
- **Has it called anything?**
- **Will it?**
- **Args?**
- **Stack depth?**
- **Call trace?**
Convention Summary

• How to write and Debug a MIPS program using calling convention
• 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, $v1
• stack frame ($fp to $sp) contains:
  – $ra (clobbered on JAL to sub-functions)
  – $fp
  – local vars (possibly clobbered by sub-functions)
  – contains extra arguments to sub-functions (i.e. argument “spilling”)
  – contains space for first 4 arguments to sub-functions
• callee save regs are preserved
• caller save regs are not
• Global data accessed via $gp

$fp → $sp →