Topic 6: MIPS Functions

Function Calls

Return from Functions

- main()
- foo()
- bar()
- baz()
What is a Function?

- How to call the function?
- How to get back when we return?
- How to pass parameters?
- How to return values?
- How to protect local variables?
- How to protect temporaries?
How to Call a Function?

- We already have \textit{j}
  - Jump to any label in the program\(^1\)

- Function names are \textbf{just labels}
  - But you should use \texttt{.globl} so that they are visible from other files

\(^1\) 26-bit limit, but you can go \texttt{almost} anywhere
How to Return from a Function?

- We **cannot** use $j$
  - Same function returns to many places
  - Often, in other files
  - Often, in code written later!

- Need **return address** as a parameter
- Need an instruction that can **jump anywhere**
How to Return from a Function?

- Two new instructions

- `jal` (jump and link)
  - Does `j`
  - Stores return address into $ra

- `jr` (jump register)
  - Jump to address in any register
  - Most often used for return-from-function
Calling & Returning

jal foo 2

foo() 3

main() 1

jr $ra 4

5
Oops!

$ra$ gets overwritten

jal bar
jal foo
jr $ra$

bar()
foo()
main()
Saving Registers

- We need to save $ra$ somehow
  - Keep it unchanged while we call another function

- Actually a general problem
  - Would be nice to save other registers, sometimes

- Solution: a stack!
Stack Frames

- A stack frame contains all of the local information for a function
  - Local variables
  - $ra
  - Anything else worth saving

- In theory, it's isolated

- But in assembly, all we have are loads and stores on main memory
Memory Layout

- stack
- .data
- .text
- code
- high address

- low address

- 0x7f ff fe 30
- $sp

- 0x10 01 00 00

- 0x00 40 00 00
- PC
Stack Frames

- In assembly, a stack frame is just an array of words
- $sp$ gives the base of the current frame
  - No bounds checking!
- Decrement $sp$ to allocate more space
- Increment $sp$ to free space
Hardware Doesn't Do Much

• Hardware provides:
  - jal
  - jr

• Software (conventions) provide:
  - Everything else
Calling Conventions (Overview)
Calling Conventions (4 pieces)

- Caller
  - Startup sequence
  - jal
  - Cleanup sequence

- Subroutine
  - Prologue
  - Body
  - Epilogue (ends with jr)
Stack Management

- Prologue
  - Allocate stack space
  - Save $ra
  - Save other regs …

- Epilogue
  - Restore $ra
  - Restore other regs
  - Free stack space
Standard Prologue

addiu $sp, $sp, -24
sw $fp, 0($sp)
sw $ra, 4($sp)
addiu $fp, $sp, 20

- Allocate 24 bytes (6 words)
- Save $fp
- Save $ra
- Change $fp
This is the stack state, when \texttt{foo()} is just about to call \texttt{bar()}.
main()

Stack frame

foo()
Stack frame

(no frame allocated yet)

bar():
addiu $sp, $sp, -24
sw $fp, 0($sp)
sw $ra, 4($sp)
addiu $fp, $sp, 20

$fp
$sp

stack grows down
main()

Stack frame

foo()

Stack frame

bar()

Stack frame

---

**bar()**:

- `addiu $sp, $sp, -24`
- `sw $fp, 0($sp)`
- `sw $ra, 4($sp)`
- `addiu $fp, $sp, 20`

$fp

$sp
main()
Stack frame

foo()
Stack frame

bar():
addiu $sp, $sp, -24
sw $fp, 0($sp)
sw $ra, 4($sp)
addiu $fp, $sp, 20

$sp
$fp
Old $fp
main()

Stack frame

foo()

Stack frame

bar():
addiu $sp, $sp, -24
sw $fp, 0($sp)
sw $ra, 4($sp)
addiu $fp, $sp, 20

$sp

Old $fp

$fp

$ra
bar():
addiu $sp, $sp, -24
sw $fp, 0($sp)
sw $ra, 4($sp)
addiu $fp, $sp, 20

main()

Stack frame

foo()

Stack frame

$sp
Old $fp

$fp
+20 bytes

$ra
In the Function Body...

- Stack frame can be extended during the body, to save temporary values
  - Decrement $sp$ to extend
  - Increment $sp$ to free space when done

![Diagram of stack frame with $fp$, $ra$, and $sp$](image-url)
Standard Epilogue

lw $ra, 4($sp)
lw $fp, 0($sp)
addiu $sp, $sp, 24
jr $ra
main()

Stack frame

foo()

Stack frame

bar():

```
lw   $ra, 4($sp)
lw   $fp, 0($sp)
addiu $sp, $sp, 24
jr    $ra
```

Copy back into registers
bar():

lw   $ra, 4($sp)
lw   $fp, 0($sp)
addiu $sp, $sp, 24
jr    $ra
main()

Stack frame

foo()

Stack frame

(frame has been freed)

bar()

lw $ra, 4($sp)

lw $fp, 0($sp)

addiu $sp, $sp, 24

jr $ra
main()
Stack frame
foo()
Stack frame

(bar():
lw $ra, 4($sp)
lw $fp, 0($sp)
addiu $sp, $sp, 24
jr $ra

(sp)
(fp)

Code returns to foo()
Startup/Cleanup Code

- Startup code runs before the `jal` instruction
- Cleanup code runs after the `jal` instruction
- Both run in the **caller**.
- Sometimes NOP
What Does the Startup Code Do?

- No standard startup code
- Do only what is necessary
  - Pass parameters
  - Save temporary values (restore in cleanup code)
What Gets Saved Where?

- We have \( t_X \), \( s_X \) registers:
  - \( t_X \) registers can be destroyed by any function
    - Save in startup code (in the caller) if you care
  - \( s_X \) registers must never be destroyed
    - Save in prologue if you plan to overwrite them
Saving τX Registers

- To save τX registers over a function call
  - Must be performed by caller (called function is free to destroy τX values)
  - Push onto stack in startup code
  - Pop off stack in cleanup code
Saving tX Registers

addiu $sp, $sp, -8
sw $t3, 4($sp)
sw $t8, 0($sp)
jal otherFunc
lw $t8, 0($sp)
lw $t3, 4($sp)
addiu $sp, $sp, 8
main()

Stack frame

foo()

Stack frame

This is the stack state, when foo() is just about to call bar().

But first, foo() wants to save two tX registers.

foo():
addiu $sp, $sp, -8
sw $t3, 4($sp)
sw $t8, 0($sp)
jal bar
lw $t8, 0($sp)
lw $t3, 4($sp)
addiu $sp, $sp, 8
main()

Stack frame

foo()

Stack frame

Two new 1-word slots

foo():
addiu $sp, $sp, -8
sw $t3, 4($sp)
sw $t8, 0($sp)
jal bar
lw $t8, 0($sp)
lw $t3, 4($sp)
addiu $sp, $sp, 8
main():

Stack frame

foo():

Stack frame

$fp

$sp

addiu $sp, $sp, -8
sw $t3, 4($sp)
sw $t8, 0($sp)
jal bar
lw $t8, 0($sp)
lw $t3, 4($sp)
addiu $sp, $sp, 8
main()

Stack frame

foo()

Stack frame

foo():
addiu $sp, $sp, -8
sw $t3, 4($sp)
sw $t8, 0($sp)
jal bar
lw $t8, 0($sp)
lw $t3, 4($sp)
addiu $sp, $sp, 8
main()

Stack frame

foo()

Stack frame

bar() runs now...

foo():
addiu $sp, $sp, -8
sw $t3, 4($sp)
sw $t8, 0($sp)
jal bar
lw $t8, 0($sp)
lw $t3, 4($sp)
addiu $sp, $sp, 8

bar() runs now...
$fp$

$sp$

Restore $t8$ into the registers

```
main()

Stack frame

foo()

Stack frame

foo():
addiu $sp, $sp, -8
sw $t3, 4($sp)
sw $t8, 0($sp)
jal bar
lw $t8, 0($sp)
lw $t3, 4($sp)
addiu $sp, $sp, 8
```
main()

Stack frame

foo()

Stack frame

$fp

$sp

$t3

$t8

Restore $t3 into the registers

foo():

addiu $sp, $sp, -8
sw $t3, 4($sp)
sw $t8, 0($sp)
jal bar

lw $t8, 0($sp)
lw $t3, 4($sp)
addiu $sp, $sp, 8
main()

Stack frame

foo()

Stack frame

(frame size has been reduced)

foo():

```
addiu $sp, $sp, -8
sw $t3, 4($sp)
sw $t8, 0($sp)
jal bar
lw $t8, 0($sp)
lw $t3, 4($sp)
addiu $sp, $sp, 8
```
Saving $s_X$ Registers

- All functions are required to **not modify** any $s_X$ registers
  - Also: $fp$, $ra$

- It's OK to change them if you **restore them at the end**
  - Function chooses what $s_X$ it wants to modify
  - Save them right after prologue
  - Restore them right before epilogue
Saving $sX$ Registers

... standard prologue ...

```
addiu $sp, $sp, -4
sw $s2, 0($sp)
```

... body ...

```
lw $s2, 0($sp)
addiu $sp, $sp, 4
```

... standard epilogue ...
Review
Parameters and Return Values

• Parameters
  – Passed using registers $a0, $a1, $a2, $a3
  – Additional parameters (if any) on stack

• Return values
  – Returned using registers $v0
  – If you need a second: $v1
    • 64-bit return values
    • Multiple return values (value, error status)
Passing Parameters

- Set parameter values in the **startup code**
- Read parameter values anywhere in the function body

- Sometimes useful to save parameters for later
  - Example: You are going to call another function
  - Standard stack frame has 4 slots for saving aX registers
The 4 aX Slots

- You are **not required** to save the parameters to the aX slots
- You **must not expect** that the caller saved the parameters to the aX slots
- You **may** use the slots if you wish, but
- **Only** to save the original parameters
- Cannot save space, cannot use for other purposes
The 4 `ax` Slots

- Why the limitations?
- I'm not 100% sure
- Educated guess:

  These slots can be used by the caller to save values which it might restore later.
More Than 4 Parameters

- If more than 4 parameters:
  - First 4 in ax registers
  - Rest on the stack
    - Caller writes to stack but does not allocate stack space
    - Called function allocates a larger starting frame than normal
More Than 4 Parameters

<table>
<thead>
<tr>
<th>foo()</th>
<th>bar()</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi  $a0, $zero, 10</td>
<td>addiu $sp, $sp, -32</td>
</tr>
<tr>
<td>addi  $a1, $zero, 20</td>
<td>sw     $fp, 0($sp)</td>
</tr>
<tr>
<td>addi  $a2, $zero, 30</td>
<td>sw     $ra, 4($sp)</td>
</tr>
<tr>
<td>addi  $a3, $zero, 40</td>
<td>addiu $fp, $sp, 28</td>
</tr>
<tr>
<td>addi  $t0, $zero, 50</td>
<td></td>
</tr>
<tr>
<td>sw     $t0, -8($sp)</td>
<td></td>
</tr>
<tr>
<td>addi  $t0, $zero, 60</td>
<td></td>
</tr>
<tr>
<td>sw     $t0, -4($sp)</td>
<td></td>
</tr>
<tr>
<td>jal    bar</td>
<td></td>
</tr>
</tbody>
</table>
This is the stack state, when `foo()` is just about to call `bar()`.

```assembly
main():

Stack frame

foo()

Stack frame

$fp

$sp

```
The first four parameters don't affect the stack at all.
The 5th parameter goes below foo()'s stack frame.
main()

Stack frame

foo()

Stack frame

arg6 = 60
arg5 = 50

(no frame allocated yet)

**foo()**:  
addi $a0, $zero, 10  
addi $a1, $zero, 20  
addi $a2, $zero, 30  
addi $a3, $zero, 40  
addi $t0, $zero, 50  
sw $t0, -8($sp)  
addi $t0, $zero, 60  
sw $t0, -4($sp)  
jal bar

The 6th parameter goes **above** the 5th.
main()  

Stack frame  

Stack frame  

foo()  

Stack frame  

```
$fp
$sp

arg6 = 60
arg5 = 50
(no frame allocated yet)
```

```
foo():
addi   $a0, $zero, 10
addi   $a1, $zero, 20
addi   $a2, $zero, 30
addi   $a3, $zero, 40
addi   $t0, $zero, 50
sw     $t0, -8($sp)
addi   $t0, $zero, 60
sw     $t0, -4($sp)
jal    bar
```
When `bar()` runs, it allocates space for the 2 arguments already there, **plus** the standard 6-word frame.
bar():
addiu $sp, $sp, -32
sw $fp, 0($sp)
sw $ra, 4($sp)
addiu $fp, $sp, 28
Other Uses for the Stack

- Storing local variables
  - Common when registers are scarce (x86, others)
  - Also, if a function has many locals
  - Only update stack when necessary, keep in regs

- Local arrays
  - Arrays in registers don't work!
  - Assembly only: variable-length arrays possible

- Local structs
  - One variable, many fields