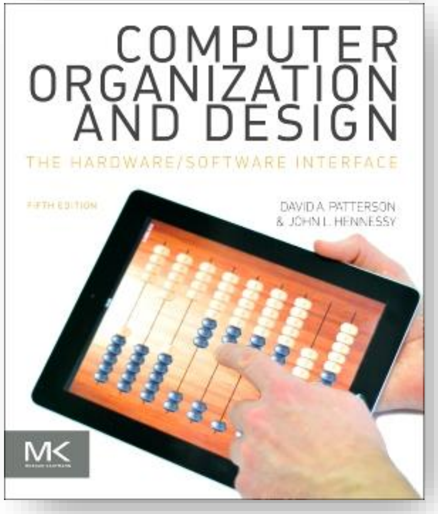


ESE 345 Computer Architecture

MIPS Functions

Function Call Bookkeeping

What are the properties of a function?

1. Put *arguments* in a place where the function can access them
2. Transfer control to the function
3. The function will acquire any (local) storage resources it needs
4. The function performs its desired task
5. The function puts *return value* in an accessible place and “cleans up”
6. Returns control
7. Black-box operation/scoping
8. Re-entrancy

MIPS Registers for Function Calls

- Registers play a major role in keeping track of information for function calls.
- `$a0-$a3`: four *argument* registers to pass parameters
- `$v0-$v1`: two *value* registers to return values
- `$s0-$s7`: local variables registers
- `$ra`: *return address* register that saves

where a function is called from

- The stack is also used; more later.

Function Call Example

```
... sum(a,b); ...          /* a→$s0,b→$s1 */

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

C

MIPS

address (decimal)

```
1000  addi $a0,$s0,0        # x = a
1004  addi $a1,$s1,0        # y = b
1008  addi $ra,$zero,1016   # $ra=1016
1012  j    sum              # jump to sum
1016
...
2000  sum: add $v0,$a0,$a1
2004  jr   $ra              # new instruction
```

Instruction Support for Functions

- Single instruction to jump and save return address: jump and link (jal)
- **Before**:

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum #go to sum
```

- **After**:

```
1008 jal sum # $ra=1012,go to sum
```

- Why have a `jal`? Make the common case fast: function calls are very common. Also, you don't have to know where the code is loaded into memory with `jal`.

MIPS Instructions for Function Calls

- Jump and Link (**`jal`**)
  - **`jal label`**
  - Saves the location of *following* instruction in register **`$ra`** and then jumps to **`label`** (function address)
  - Used to invoke a function
- Jump Register (**`jr`**)
  - **`jr src`**
  - Unconditional jump to the address specified in **`src`** (almost always used with **`$ra`**)
  - Used to return from a function

Function Call Example

```
... sum(a,b); ...           /* a→$s0,b→$s1 */

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

C

MIPS

address (decimal)

```
1000  addi $a0,$s0,0     # x = a
1004  addi $a1,$s1,0     # y = b
1008  jal  sum           # $ra=1012, goto sum
1012
...

2000  sum: add $v0,$a0,$a1
2004  jr   $ra           # return
```

Nested Procedures (1/2)

- `int sumSquare(int x, int y) {`
  `return mult(x,x)+ y;`
  `}`
- Something called **`sumSquare`**, now **`sumSquare`** is calling **`mult`**.
- So there's a value in `$ra` that **`sumSquare`** wants to jump back to, but this will be overwritten by the call to **`mult`**.
- Need to save **`sumSquare`** return address before call to **`mult`**.

Nested Procedures (1/2)

- Also need to save any registers that are needed across the procedure call:

```
int fact(int a) {                  fact: addi $v0 $0 1
 if (a == 0) {                           beq $a0 $0 done
  return 1;          ------>             add $s0 $a0 $0
  } else {                               addi $a0 $a0 -1
  return a * fact(a-1);                  jal fact
  }                                      mul* $v0 $s0 $v0
}                                  done: jr $ra
```

Why won't our factorial work?

Six Steps of Calling a Function

1. Put *arguments* in a place where the function can access them `$a0-$a3`
2. Transfer control to the function `jal`
3. The function will acquire any (local) storage resources it needs
4. The function performs its desired task
5. The function puts *return value* in an accessible place and "cleans up" `$v0-$v1`
6. Control is returned to you `jr`

Using the Stack (1/2)

- Where should we save registers? **The Stack**
- So we have a stack pointer register `$sp` which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

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

Example: sumSquare (1/2)

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

- What do we need to save?
  - Call to **`mult`** will overwrite `$ra`, so save it
  - Reusing **`$a1`** to pass 2nd argument to **`mult`**, but need current value (`y`) later, so save **`$a1`**
- To save something to the Stack, move **`$sp`** *down* the required amount and fill the "created" space

Example: sumSquare (2/2)

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

```
sumSquare:
          addi $sp,$sp,-8       # make space on stack
          sw $ra, 4($sp)        # save ret addr
"push"    sw $a1, 0($sp)        # save y
          add $a1,$a0,$zero     # set 2nd mult arg
          jal mult              # call mult
          lw $a1, 0($sp)        # restore y
          add $v0,$v0,$a1       # ret val = mult(x,x)+y
"pop"     lw $ra, 4($sp)        # get ret addr
          addi $sp,$sp,8        # restore stack
          jr $ra
mult:     ...
```

Basic Structure of a Function

Prologue

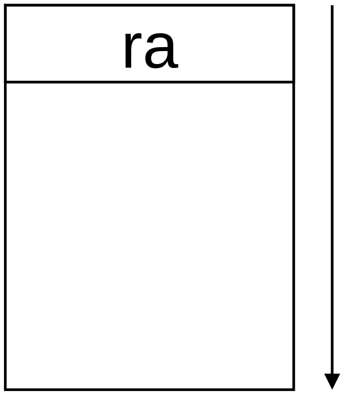
```
func_label:
addi $sp,$sp, -framesize
sw $ra, <framesize-4>($sp)
save other regs if need be
```

Body **(call other functions…)**

```
    ...
```

Epilogue

```
restore other regs if need be
lw $ra, <framesize-4>($sp)
addi $sp,$sp, framesize
jr $ra
```

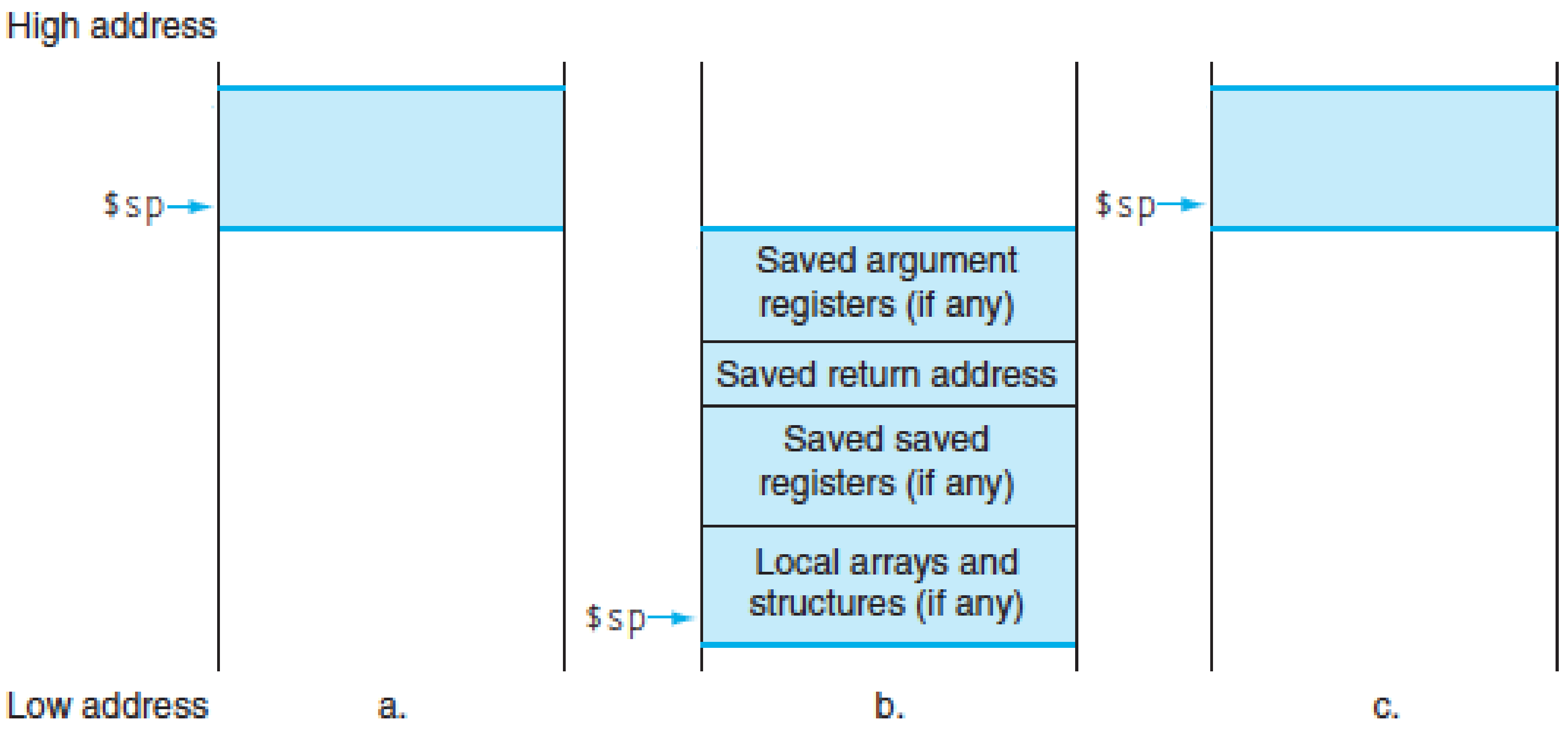


stack

Local Variables and Arrays

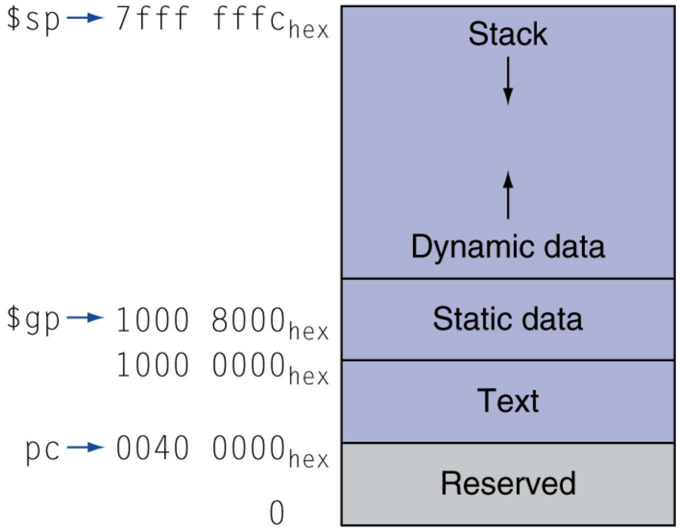
- Any local variables the compiler cannot assign to registers will be allocated as part of the stack frame (**Recall:** spilling to memory)
- Locally declared arrays and structs are also allocated as part of the stack frame
- Stack manipulation is same as before
  - Move `$sp` down an extra amount and use the space it created as storage

Stack Before, During, After Call



Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Register Conventions

- **CalleR:** the calling function
- **CalleE:** the function being called
- Register Conventions: A set of generally accepted rules as to which registers will be unchanged after a procedure call (**`jal`**) and which may have changed

Saved Registers

- These registers are expected to be the same before and after a function call
  - If **calle**E uses them, it must restore values before returning
  - This means save the old values, use the registers, then reload the old values back into the registers
- **`$s0-$s7`** (*saved* registers)
- **`$sp`** (stack pointer)
  - If not in same place, the caller won't be able to properly restore values from the stack
- **`$ra`** (return address)

Volatile Registers

- These registers can be freely changed by the **calleE**
  - If **calleR** needs them after a function call, it must save those values before making the function call
- `$t0-$t9` (*temporary* registers)
- `$v0-$v1` (return values)
  - These will contain the new returned values
- `$a0-$a3` (return address and arguments)
  - These will change if **calleE** invokes another function (nested function means **calleE** is also a **calleR**)

Register Conventions Summary

- One more time for luck:
  - **CalleR** must save any volatile registers it is using onto the stack before making a procedure call
  - **CalleE** must save any saved registers it intends to use before garbling up their values
- Notes:
  - **CalleR** and **calleE** only need to save the appropriate registers *they are using* (not all!)
  - Don't forget to restore the values later

Example: Using Saved Registers

```
myFunc: # Uses $s0 and $s1
    addiu     $sp,$sp,-12  # This is the Prologue
    sw        $ra,8($sp)   # Save saved registers $s0 and $s1
    sw        $s0,4($sp)
    sw        $s1,0($sp)
    ...                    # Do stuff with myFunc $s0 and $s1
    jal       func1        # $s0 and $s1 unchanged by
    ...                    #   function calls, so can keep
    jal       func2        #   using them normally
    ...                    # Do stuff with $s0 and $s1
    lw        $s1,0($sp)   ## This is the Epilogue
    lw        $s0,4($sp)   # Restore saved registers $s0 and $s1
    lw        $ra,8($sp)
    addiu     $sp,$sp,12
    jr        $ra          # return
```

Example: Using Volatile Registers

```
myFunc: # Uses $t0 after func1 call
   addiu      $sp,$sp,-4  # This is the Prologue
   sw         $ra,0($sp)  # Save saved registers
   ...                    # Do stuff with $t0
   addiu      $sp,$sp,-4  # Save volatile registers
   sw         $t0,0($sp)  #   before calling a function
   jal        func1       # Function func1 may change $t0
   lw         $t0,0($sp)  # Restore volatile registers
   addiu      $sp,$sp,4   #   before you use them again
   ...                    # Do stuff with $t0
   lw         $ra,0($sp)  # This is the Epilogue
   addiu      $sp,$sp,4   # Restore saved registers
   jr         $ra         # return
```

Choosing Your Registers

- Minimize register footprint
  - Optimize to reduce number of registers you need to save by choosing which registers to use in a function
  - Only save when you absolutely have to
- Function does NOT call another function
  - Use only `$t0-$t9` and there is nothing to save!
- Function calls other function(s)
  - Values you need throughout go in `$s0-$s7`, others go in `$t0-$t9`
  - At each function call, check number arguments and return values for whether you or not you need to save

MIPS Registers

The constant 0	\$0	\$zero
Reserved for Assembler	\$1	\$at
Return Values	\$2-\$3	\$v0-\$v1
Arguments	\$4-\$7	\$a0-\$a3
Temporary	\$8-\$15	\$t0-\$t7
Saved	\$16-\$23	\$s0-\$s7
More Temporary	\$24-\$25	\$t8-\$t9
Used by Kernel	\$26-27	\$k0-\$k1
Global Pointer	\$28	\$gp
Stack Pointer	\$29	\$sp
Frame Pointer	\$30	\$fp
Return Address	\$31	\$ra

The Remaining Registers

- $at (assembler)
  - Used for intermediate calculations by the assembler (pseudo-code); *unsafe to use*
- $k0-$k1 (kernal)
  - May be used by the OS at any time; *unsafe to use*
- $gp (global pointer)
  - Points to global variables in Static Data; *rarely used*
- $fp (frame pointer)
  - Points to top of current frame in Stack; *rarely used*

Let’s Try!

```
r: # r uses R/W $s0,$v0,$t0,$a0,$sp,$ra, mem
   ...# first save $s0 and $ra at the start
   .. # r works R/W with $s0,$v0,$t0,$a0,mem
 ### PUSH MORE REGISTERS ON STACK before “jal e”?
   jal e # Call e
..# r uses R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
    # restore $s0,$ra,$sp before return from r
     jr $ra # Return to caller of r
e: ... # R/W $s0,$v0,$t0,$a0,$sp,$ra,mem
   jr $ra # Return to r
```

What does r have to push on the stack before and restore after “jal e”? s0? $sp? $v0? $t0? $a0? $ra?

Back to Factorial Procedure Example

- C code:

```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

  - Argument n in $a0
  - Result in $v0

Factorial Procedure: MIPS Code

```
fact:
     addi $sp, $sp, -8      # adjust stack for 2 items
     sw   $ra, 4($sp)       # save return address
     sw   $a0, 0($sp)       # save argument
     slti $t0, $a0, 1       # test for n < 1
     beq  $t0, $zero, L1
     addi $v0, $zero, 1     # if so, result is 1
     addi $sp, $sp, 8       #   pop 2 items from stack
     jr   $ra               #   and return
  L1: addi $a0, $a0, -1      # else decrement n
     jal  fact               # recursive call
     lw   $a0, 0($sp)       # restore original n
     lw   $ra, 4($sp)       #   and return address
     addi $sp, $sp, 8       # pop 2 items from stack
     mul  $a0, $v0          # multiply to get result
     mflo $v0
     jr   $ra               # and return
```

Summary

- MIPS function implementation:
  - Jump and link (**`jal`**) invokes, jump register (**`jr $ra`**) returns
  - Registers **`$a0-$a3`** for arguments, **`$v0-$v1`** for return values
- Register conventions preserves values of registers between function calls
  - Different responsibilities for **calleR** and **calleE**
  - Registers classified as saved and volatile
- Use the Stack for spilling registers, saving return address, and local variables

Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - David Patterson (UCB)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - Justin Hsia (UCB)