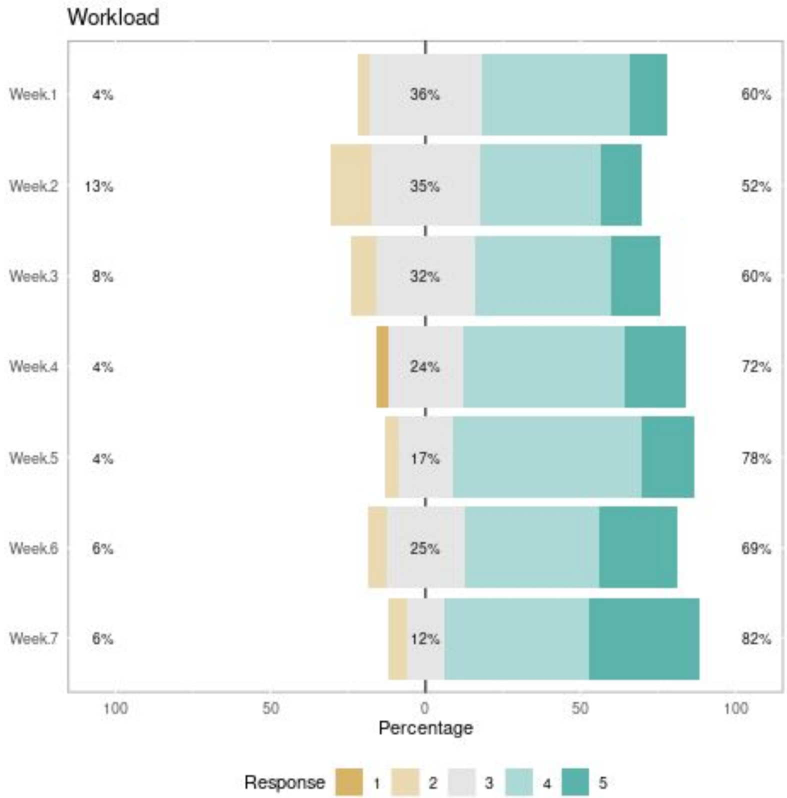
0x0D - Procedure/Function Calls

ENGR 3410: Computer Architecture

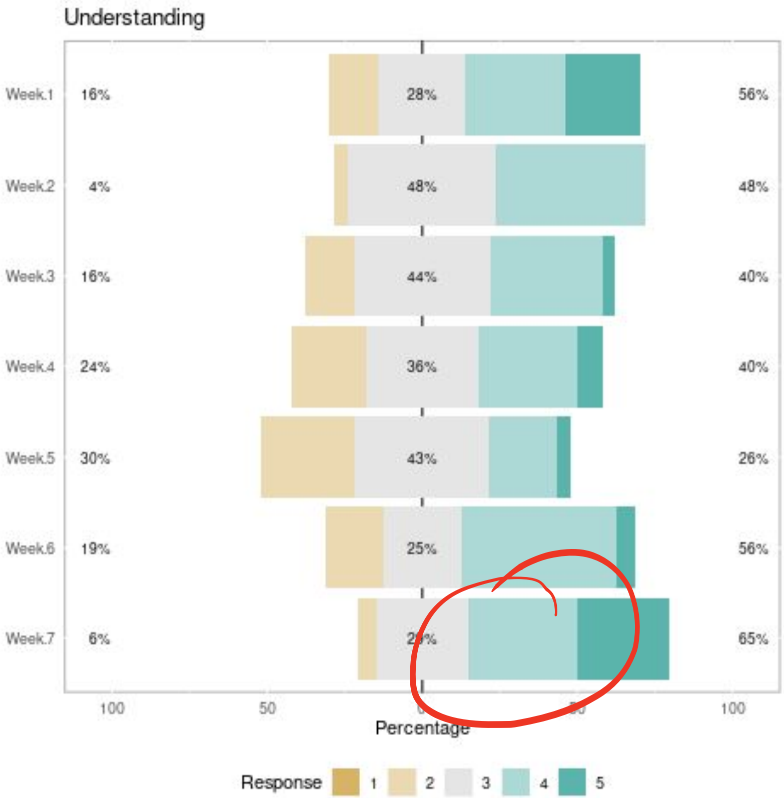
Jon Tse

Fall 2020

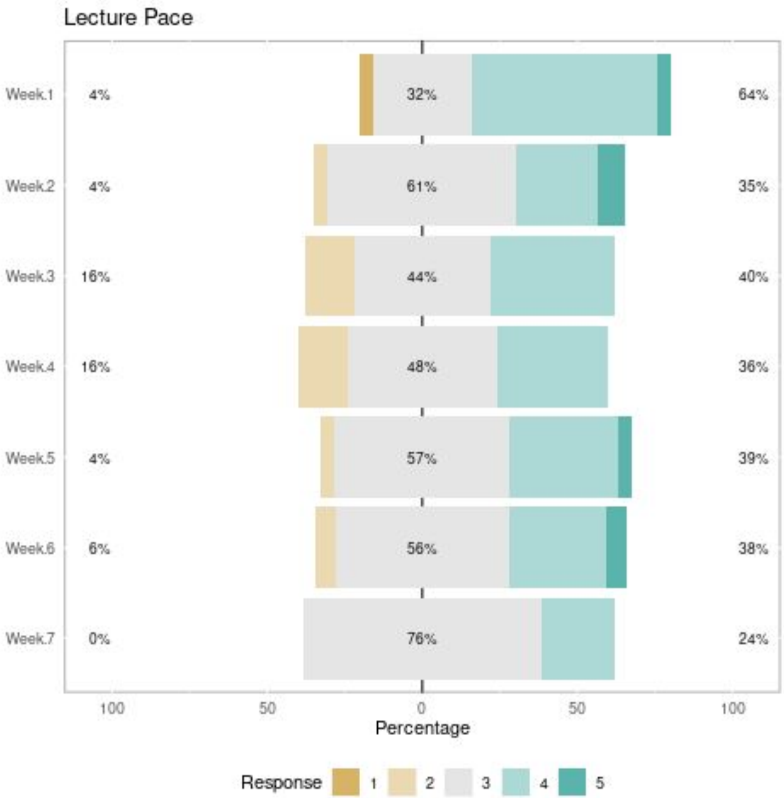
Feedback - Workload



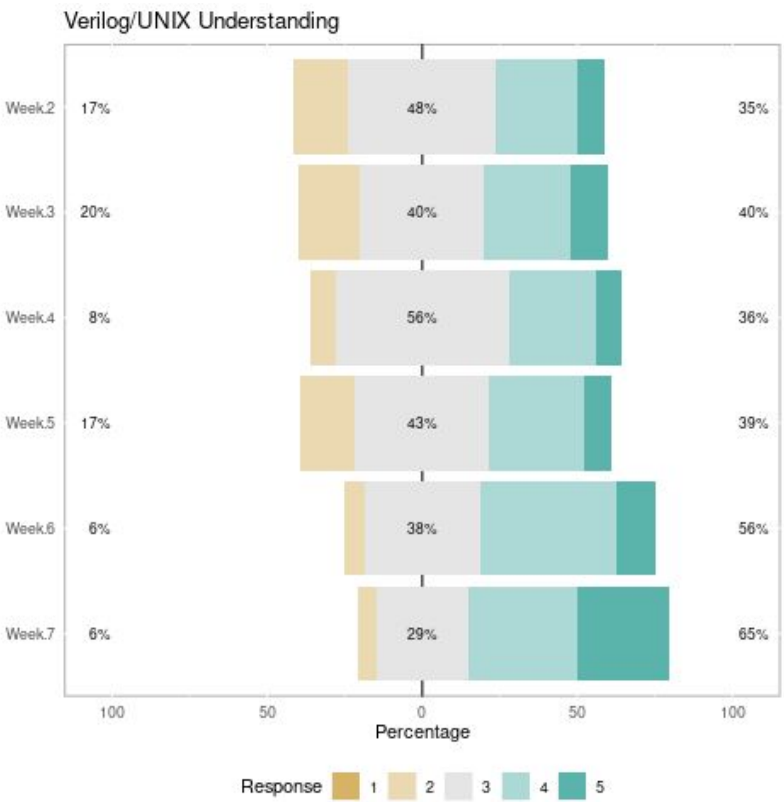
Feedback - Understanding



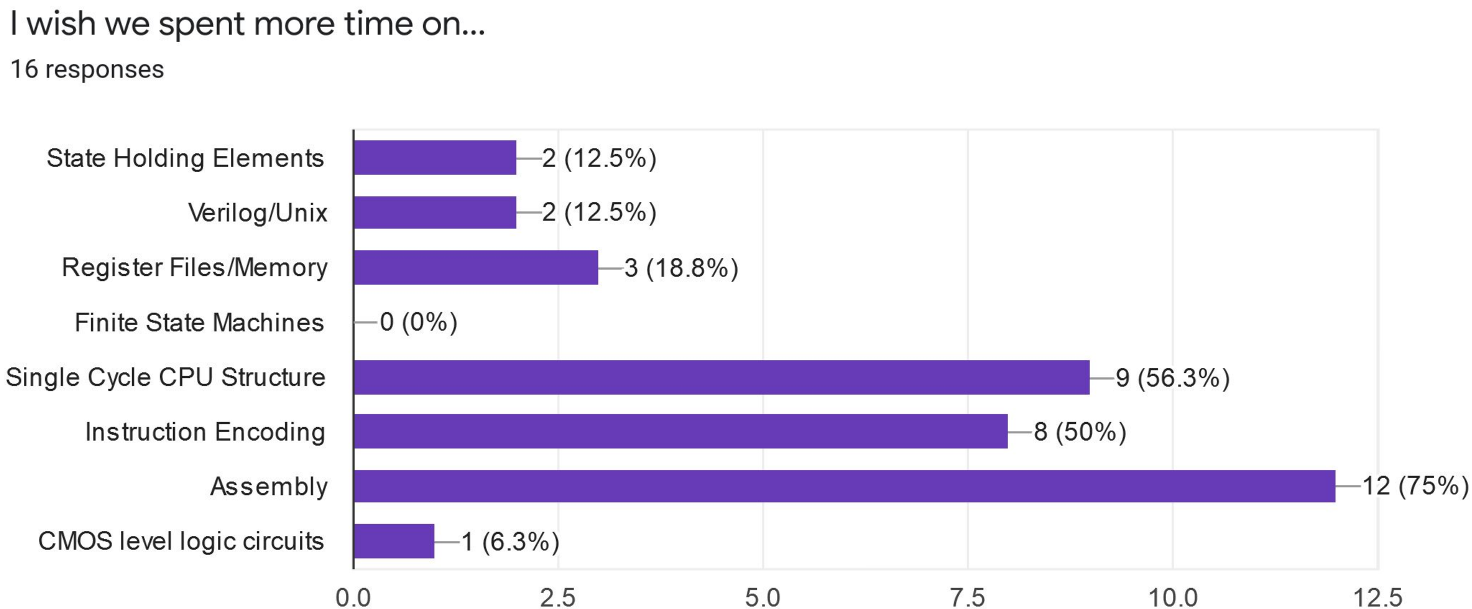
Feedback - Pace



Feedback - Tools



Feedback - Topics



Feedback - Thoughts

- “HW6 was a firehose amount of work :(and took me 10 hours at least compared to usually 5-6 hours max for other homework...”
- Will we be talking about virtual memory?

Housekeeping

- Midterm was due, unless you had extension
- Lab 4 is here! - START EARLY
  - 3 Parts
  - Don’t just do what’s due and stop.
- SEU, Control Bits, and Scary Implications

Today

- Calling Simple/“Leaf” Functions
- The Stack
- Calling Generic Functions
- Calling Conventions

MIPS Assembly vs High Level Languages

- All operations on registers, except load/store
- Simple instructions with 1-2 operands
- No scope, nesting, variables, loops, functions
- Explicit flow control via branch & jump
  - Labels represent an address in data or instruction memory

What is a function?

- A procedure is a stored subroutine that performs a specific task based on the parameters it is provided.
  - <Pattersen & Hennessy>

- a = f(x,y,z); // You have seen this before

What is a function?

- A block of code formatted in a way that allows it to be *called* by other code, and then returns execution to the *caller*

- How is this similar to / different from one of our “GOTO” targets?

The Life of a Called Function

- The *caller* stores parameters “somewhere”
- Control (Execution) is passed to the *callee*
- The callee does some stuff
- The callee stores the results “somewhere”
- Control is passed back to the caller

Calling Conventions

- A *Calling Convention* standardizes where those “somewheres” are.
  - Registers? Data Memory?
  - Return values?
  - How do we return control back to the caller?

- It is just another type of contract

Execution Flow

- Lets work through a simple example

```
void main()
{
    // do some stuff
    MyFunction();
    // do other stuff
}
```

Execution Flow

- Idea 1:
- Jump to subroutine
- Jump back when done

main()

```
#Caller (main)
#do stuff
j MyFunction
returnToMain:
#do other stuff
```

Myfunc()

```
#callee
MyFunction:
#code code code
j returnToMain
```

Execution Flow

- Idea 1:
- Jump to subroutine
- Jump back when done
- What if I want to use the function again elsewhere?

```
#Caller (main)
#do stuff
j MyFunction
returnToMain:
#do other stuff

#callee
MyFunction:
#code code code
j returnToMain
```

Execution Flow

- Idea 2:
- Store return address
- Jump to subroutine
- Jump back when done

```
#Caller (main)
#do stuff
ReturnAddr = PC+???
j MyFunction
returnToMain:
#do other stuff

#callee
MyFunction:
#code code code
jr ReturnAddr
```

Execution Flow

- In MIPS, this is called Jump And Link (jal)
- jal stores the PC+4 to $31 and then jumps to the target address
- $31 is also known as $ra
  - Return Address
- Jump Register (jr) to get back

```
#Caller (main)
#do stuff
jal MyFunction
#do other stuff

#callee
MyFunction:
#code code code
jr $ra
```

PC

PC+4

Recall: Register File Allocation

Register	Name	Function	Comment
$0	$zero	Always 0	No-op on write
$1	$at	Reserved for assembler	Don't use it!
$2-3	$v0-v1	Function return	
$4-7	$a0-a3	Function call parameters	
$8-15	$t0-t7	Volatile temporaries	Not saved on call
$16-23	$s0-s7	Temporaries (saved across calls)	Saved on call
$24-25	$t8-t9	Volatile temporaries	Not saved on call
$26-27	$k0-k1	Reserved kernel/OS	Don't use them
$28	$gp	Pointer to global data area	
$29	$sp	Stack pointer	
$30	$fp	Frame pointer	
$31	$ra	Function return address	

i = 7
func()
k = i + 1

Execution Flow

- Storing the Return Address in $ra is great!
  - but only for Leaf Functions

- What if a function calls another function?
  Idea: save return address in Data Memory
  - Every function has a dedicated space to store $ra

main() → my func() → other func()

Execution Flow

func1

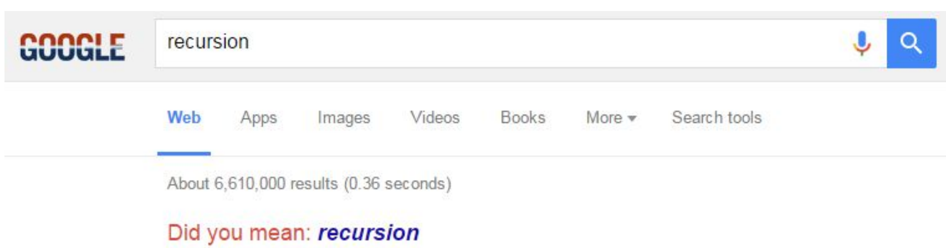
func2

func3

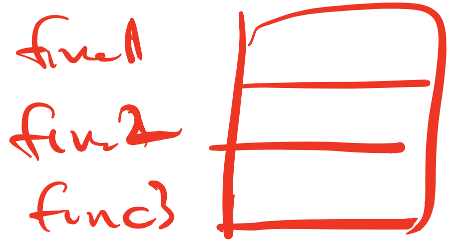
- Storing the Return Address in $ra is great!
  - but only for Leaf Functions

- What if a function calls another function? Idea: save return address in Data Memory
  - Every function has a dedicated space to store $ra

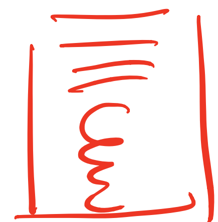
- What if a function calls itself?



Stack



- Dedicating memory per function is limiting
  - Wastes space when it isn't active
  - Can't recurse
- Instead, allocate data memory as needed
- We use the "Call Stack"
  - Last In, First Out
  - **Push** stuff onto the head of the stack
  - **Pop** stuff back off of it



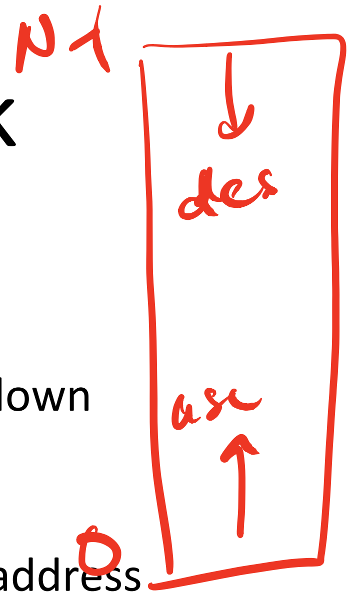
The Call Stack

- Is a Stack of “Frames”
- Each active function instance has a Frame
- Frames hold the instance memory
  - Like the return address

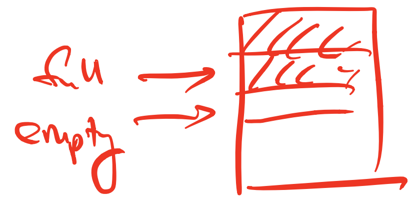
Mechanics of the Call Stack

- Architecture Decisions:
  - Where is the Stack?
  - How do we address it / point to it?
    - Linked List? Raw Pointer?
- Calling Convention Decisions:
  - What stuff is put on it?
  - In what order? (Frame Definition)
  - By whom? (Caller? Callee?)

Mechanics of the Stack



- Ascending/Descending
  - Ascending stacks start at address 0 and grow up
  - Descending stacks start at the other end and grow down
- Full / Empty
  - Empty stack pointers point to the next unallocated address
  - Full stack pointers point to the top item in the stack
- MIPs uses a "full descending" stack
  - With $29 ($sp) as the stack pointer



- Other choices common, e.g. ARM supports all 4 but usually uses full descending

malloc
mem allocate

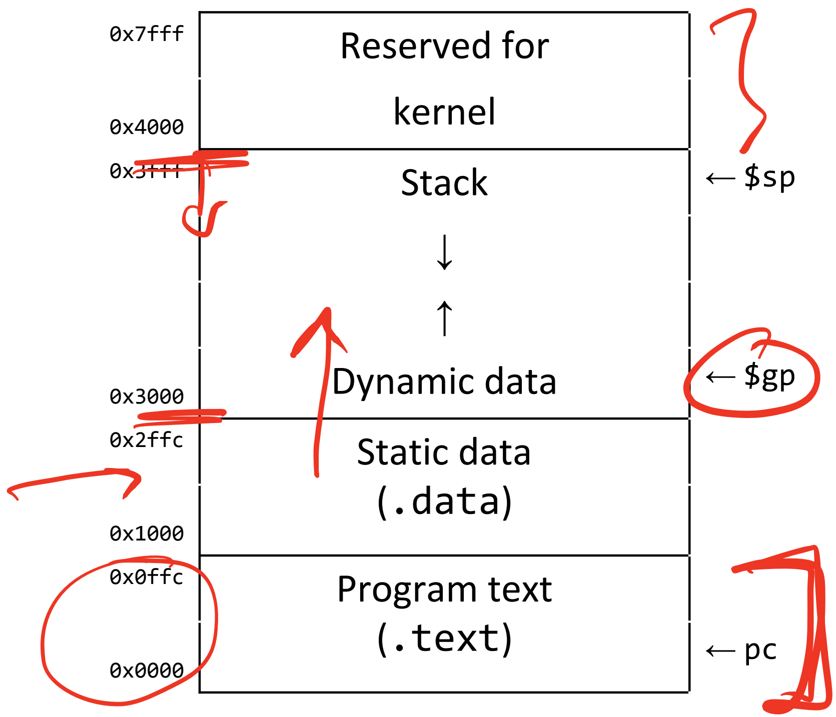
Why Descending?

- There are two dynamic memory spaces

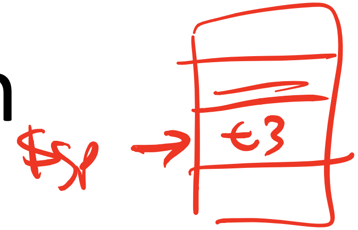
- The "Heap" handles memory dynamically allocated at run time
  - malloc /free
  - new / delete

- Traditionally:
  - Heap ascends
  - Stack descends

PC



Stack Example: Push



- PUSH $t3 onto the stack

```
addi $sp, $sp, -4     #move sp one word down
sw $t3, 0($sp)        #push $t3
```

Mem[$sp + 0] = $t3

- What does the stack look like after this push?

- Does order of operations matter here?

sw $t3, -4($sp)
addi $sp, $sp, -4

Stack Example: Pop

- POP $t3 from the stack

```
lw $t3, 0($sp)      #pop $t3
addi $sp, $sp, 4      #move sp one word up
```

- What does the stack look like after this pop?
  - Same as when we started

Stack Example: Multiple Push

- Two consecutive PUSHes, $t3 and $t4

```
addi $sp, $sp, -4
sw $t3, 0($sp)       #push $t3
addi $sp, $sp, -4
sw $t4, 0($sp)       #push $t4
```

- As one fused operation

```
addi $sp, $sp, -8      #allocate 2 words
sw $t3, 4($sp)       #push $t3
sw $t4, 0($sp)       #push $t4
```

2x update

Mem[$sp +4] = $t3

- Does order of operations matter here?

Stack Example: Multiple Pop

Two consecutive POPes, $t4 and $t3 (in reverse order)

```
lw $t4, 0($sp)      #pop $t4
addi $sp, $sp, 4
lw $t3, 0($sp)      #pop $t3
addi $sp, $sp, 4
```

- As one fused operation

```
lw $t3, 4($sp)      #pop $t3
lw $t4, 0($sp)      #pop $t4
addi $sp, $sp, 8       #delete 2 words
```

- Does order of operations matter here?

Factorial Function

```
int Fact(int n){
if(n>1)
  return n* Fact(n-1)
else
  return 1
```

Factorial Function

```
int Fact(int n){
if(n<=1) goto end:
  return n* Fact(n-1)
end:
  return 1
```

Factorial Function

```
$v0 Fact(int n){
if(n<=1) goto end:
   $v0 =n* Fact(n-1)
   jr $ra
end:
   $v0 = 1
   jr $ra
```

Factorial Function

```
$v0 Fact ($a0)
ble $a0, 1, end:
$v0 =n* Fact(n-1)
jr $ra
end:
$v0 = 1
jr $ra
```

- We have most of what we need:
  - Goto flow control for if
  - jr $ra for return
  - Registers assigned
- Now we need to call Fact
  - What do we save?
  - What order?
- Lets focus on the call site

Factorial Function Call Site

- To Call Fact:
  - Push registers I need to save
    - \$ra
    - \$a0 = n-1
  - Setup Arguments
    - N-1: \$a0 = \$a0-1
  - Jump and Link to Fact:
  - Restore registers

n

fact(n-1)

Factorial Function Call Site

```
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
addi $a0, $a0, -1
jal fact
lw $ra, 4($sp)
lw $a0, 0($sp)
addi $sp, $sp, 8
```

- To Call Fact:
  - **Push $ra, $a0**
  - Setup $a0
  - Jump and Link Fact:
  - Restore $ra, $a0

Factorial Function Call Site

```
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
addi $a0, $a0, -1
jal fact
lw $ra, 4($sp)
lw $a0, 0($sp)
addi $sp, $sp, 8
```

- To Call Fact:
  - Push $ra, $a0
  - **Setup $a0**
  - Jump and Link Fact:
  - Restore $ra, $a0

Factorial Function Call Site

```
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
addi $a0, $a0, -1
jal fact
lw $ra, 4($sp)
lw $a0, 0($sp)
addi $sp, $sp, 8
```

- To Call Fact:
  - Push $ra, $a0
  - Setup $a0
  - **Jump and Link Fact:**
  - Restore $ra, $a0

Factorial Function Call Site

```
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)
addi $a0, $a0, -1
jal fact
lw $ra, 4($sp)
lw $a0, 0($sp)
addi $sp, $sp, 8
```

- To Call Fact:
  - Push $ra, $a0
  - Setup $a0
  - Jump and Link Fact:
  - **Restore $ra, $a0**

fact(n)
if (n > 1)
return n * fact(n-1)
else
return 1

Factorial Function

$ra
$a0

```
fact:
;if(N<=1) return 1
ble $a0, 1, end

;Push $ra, $a0
addi $sp, $sp, -8
sw $ra, 4($sp)
sw $a0, 0($sp)

;Argument N-1
addi $a0, $a0, -1

jal fact
```

PC +4

```
;Pop $ra, $a0
lw $ra, 4($sp)
lw $a0, 0($sp)
addi $sp, $sp, 8

;Return N*Fact(N-1)
mul $v0, $v0, $a0
jr $ra

end:
;Return 1
$v0 = 1
jr $ra
```

2

1 * 2

6 = 2 * 3

Calling Function

```
li $a0, 4
jal factorial
move $s0, $v0

li $a0, 2
jal factorial
move $s1, $v0

li $a0, 7
jal factorial
move $s2, $v0

li $v0, 10
syscall
```

- Calls Factorial several times
- Stores results in $sN
- li, move are *pseudoinstructions*
  - What do they assemble to?
- The final two lines call a special simulator function to end execution
  - 10 means exit
  - Look up other syscalls in help

Calling Convention

- We have decisions to make for registers:
  - Are they used as part of the call?
  - Are they preserved across the call?
  - Are they reserved for other uses?

- … and about passing arguments around
  - In registers?
  - On the stack?
  - In generic data memory?

Where is the stack frame constructed?

- By the Caller
  - Right before and after the “jal”
  - Preserve own register state “just in case”

- By the Callee
  - Push after “functionname:”, Pop before “jr”
  - Preserve register state so they are usable

- Which reduces unnecessary push/pops?

MIPs Specifics

- Uses $vN to store results

- Uses $aN to store first 4 arguments
  - A is for Argument
  - Extra are pushed to the stack

MIPs Specifics

- $sN are Saved Temporaries
  - The callee is responsible for saving these **if used**
  - The caller can assume they are unchanged
- $tN are Volatile Temporaries
  - The callee can do whatever it wants with these
  - The caller can't rely on these across a call
- Advantages/Disadvantages to these?

Frame Pointer

- $sp can move around during a procedure
  - Makes frame contents shift around relative to the stack pointer
  - Can make debugging hard

$fp

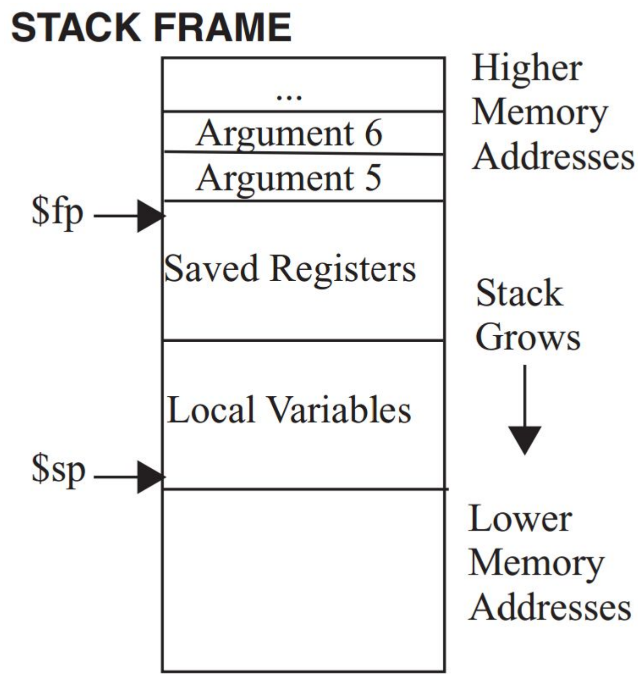
- A **Frame Pointer** stays put during a procedure
  - Value of Stack Pointer just before function was called
  - Makes debugging (and compiling) easier!

Frame Pointer

- Not strictly necessary
  - Can use Stack Pointer only
  - But it can help with debugging
- Not all implementations use it
  - GNU MIPS C Compiler does
  - MIPS MIPS C compiler does not

A MIPS calling convention's frame

- Arguments at top of previous frame
- $ra
- Saved nonvolatiles
- Locals



Vocab

- Stack Overflow / Stack Smashing
  - Running out of stack space
  - Writes over other data! (Heap)
  - Possible security vulnerability
- Unwind the Stack
  - Popping “frames” off the stack
  - Usually in the context of exception handling
- Walking the Stack
  - Looking at the Stack and figuring out where you are
  - Usually in the context of Debugging
  - Need to be able to “see” where the frames are

Vocab

- Misaligning or Unbalancing the stack
  - Pushing or popping a different number of times
  - Catastrophic Error!
- Stack Dump
  - Produced when program crashes
  - Helps you understand where you were when stuff went wrong

Exercise

- Write the “Fibonacci” function recursively:
- You will have to invent parts of your own calling convention here. Take notes on what you invented.
  - Where do you store parameters? Return Values? Return Addresses?
- Easier if you write in pseudocode first
  - Drawing your stacks helps too!

Recursive Fibonacci with test case

```
int main() {
   int fib4 = Fibonacci(4);
   int fib10 = Fibonacci(10);
   return fib4+fib10;
}

int Fibonacci(int x) {
   if (x == 0) return 0;  // Stopping conditions
   if (x == 1) return 1;
   int fib_1 = Fibonacci(x - 1);
   int fib_2 = Fibonacci(x - 2);
   return fib_1+fib_2;
}
```

Stack Dump Example

Error Message:
Value cannot be null.
Parameter name: value
Exception:
ArgumentNullException
Suggestion:
Contact National Instruments to report this error
Stack Trace:
at System.BitConverter.ToInt32(Byte[] value, Int32 startIndex)
at NationalInstruments.LabVIEW.VI.VirtualMachine.Target.ExecutionHighlighting.ProcessUpdate(Byte[] buffer)
at NationalInstruments.X3.Model.DeviceModel.OnExHighlightReadCompletedEventHandler(Object sender, ReadCompletedEventArgs e)
at NationalInstruments.X3.Model.DeviceModel.<>c__DisplayClass37.<ReadTargetMemory>b__36(Object theSender, RequestCompletedEventArgs theE)