MIPS Assembly Procedures

Procedures

- Procedures let us reuse the functionality of a piece of code without duplicating it.
- Even the simplest programs will typically use a standard library of procedures for things like input and output.
- Most programs will also have many user-defined procedures.
- They are an essential tool for the organization and simplification of programs.
- **So how do we define them and use them in MIPS assembly language?**

Procedures and Labels

- Fundamentally, the idea behind procedures is to assign a *name* to a block of code, so we can reuse that code by referring to the name.
- Machine language is just a sequence of 32-bit instruction words, so it doesn’t really support this.
- However, in *assembly language* we have the concept of *labels.*
- A label lets us assign a name to a specific *location* in the code.
- Assembly doesn’t have a concept of “blocks” of code, like high-level languages do.
- But we can place a label at the *start* of a procedure, and imagine the “block” continues until the last point where the procedure can return.

Call and Return

- Figuring out how to assign names to blocks of code is only the beginning of our long nightmare.
- If we have a label called “procedureName”, how do we **call** the procedure? *(Jump to the label location and start executing the code)*
- Once the procedure is over, how do we **return** to the call site? *(Go back to point in the code that comes after the procedure call)*
- We will use a new instruction for this: **jalr** (Jump and Link Register).
- Before we look at this instruction, let’s look at two alternative methods that each have their own problems.

Attempt 1: Call with Branching

- We know the branch instructions beq and bne accept labels.

```
beq $0, $0, procedureName
```

- The above instruction will unconditionally branch to the location of the label procedureName, and start running the procedure!
- **…As long as the branch offset is in range.**
- Remember that labels need to be translated to a *16-bit two's complement branch offset* in the range -32768 to 32767.
- This means the calling code can be separated from the procedure code by at most 32767 instructions.
  - That's quite generous for a small hand-written assembly program, but might cause problems for large programs with compiler-generated code.

Attempt 2: Call with Jump Register

- Recall that we can use .word notation with labels:

```
.word procedureName
```

- This encodes the memory address corresponding to the label, i.e. the location of the labelled instruction, as a 32-bit word.
- We can then use **jr** (Jump Register) to jump to the procedure:

```
lis $3
.word procedureName
jr $3
```

- Since jr can jump to any 32-bit address, this solves the offset issue.
  - Technically there is still a limit on how far we can jump, but now it’s only limited by the amount of memory we have!

How Do We Return?

- The jr instruction works great for *calls*, but how do we *return?*
  - When the procedure ends, we expect control flow to resume at the instruction right after the jr call.
- We could put a “procedureNameReturn” label after the call, and jr to that label to return.
- But if the procedure is called multiple times, this gets *really* messy.
  - You would need separate return labels for every call site, and logic for deciding which return label to jump to…
- Aside from the offset issue, call with branching also has this problem (i.e., it is unclear how to easily return from a call).

Jump and Link Register

- **Assembly language notation:**

  `jalr $s`

- **Machine language encoding:**

  `000000 sssss 00000 00000 00000 001001`

- This instruction does two things:
  - Set PC to $s.
  - Set $31 to the *previous value of PC*.
- Thus, after using jalr, doing a “jr $31” will go to **the instruction after the jalr**. This lets you use “jr $31” to return from procedures!!

A Non-Working Example

```
; main program
lis $1
.word 1
lis $2
.word 240
lis $3
.word add
jalr $3
jr $31
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

Why doesn’t this work??

A Non-Working Example

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
 PC jalr $3
    jr $31
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

What happens when jalr executes?

A Non-Working Example

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    jalr $3
$31 jr $31
```

```
   ; procedure
   add:
PC add $3, $2, $1
   jr $31
```

jalr sets PC and $31 like this…

A Non-Working Example

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    jalr $3
$31 jr $31
```

```
   ; procedure
   add:
   add $3, $2, $1
PC jr $31
```

The procedure runs, then returns…

A Non-Working Example

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    jalr $3
$31 jr $31 PC
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

Now PC and $31 are *both* here!

A Non-Working Example

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    jalr $3
$31 jr $31 PC
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

So, we get stuck in an infinite loop.

A Working Example… (but not ideal)

```
; main program
lis $1
.word 1
lis $2
.word 240
lis $3
.word add
add $13, $31, $0
jalr $3
jr $13
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

A Working Example… (but not ideal)

```
; main program
lis $1
.word 1
lis $2
.word 240
lis $3
.word add
add $13, $31, $0
jalr $3
jr $13
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

- We backed up the original $31 in $13 so we can end the program correctly.
- What if the procedure happened to change $13? This would fail!
- What if the procedure calls another procedure? Then the procedure needs to back up *its own* return address.

The Chain of Return Addresses

- When a program calls a procedure, which calls another procedure, which calls another procedure, etc. a **call stack** is created.
- Each procedure *call* needs to keep track of its return address.
  - Different calls of the same procedure can have different return addresses.
- But the jalr instruction always puts the return address in $31, and $31 can only hold one return address at a time.
- Using register backups is not only messy, but *guaranteed to fail* if the call stack size exceeds the number of registers!
- Instead, we keep track of the call stack and the chain of return addresses by using *our own stack* in memory.

A Working Example… Using the Stack!

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    lis $4
    .word 4
PC  sw $31, -4($30)
    sub $30, $30, $4
    jalr $3
    add $30, $30, $4
    lw $31, -4($30)
    jr $31
```

```
    ; procedure
    add:
    add $3, $2, $1
    jr $31

    …



$30 ; out of bounds memory begins

    …

$31 ; return location for main
```

A Working Example… Using the Stack!

```
     ; main program
     lis $1
     .word 1
     lis $2
     .word 240
     lis $3
     .word add
     lis $4
     .word 4
     sw $31, -4($30)
PC   sub $30, $30, $4
     jalr $3
     add $30, $30, $4
     lw $31, -4($30)
     jr $31
```

```
    ; procedure
    add:
    add $3, $2, $1
    jr $31

    …
```

Push (Step 1): $31 is stored on the stack

```
    [return location for main]
$30 ; out of bounds memory begins

    …

$31 ; return location for main
```

A Working Example… Using the Stack!

```
     ; main program
     lis $1
     .word 1
     lis $2
     .word 240
     lis $3
     .word add
     lis $4
     .word 4
     sw $31, -4($30)
     sub $30, $30, $4
PC   jalr $3
     add $30, $30, $4
     lw $31, -4($30)
     jr $31
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

…

Push (Step 2): Stack pointer $30 updated

```
$30 [return location for main]
$30 ; out of bounds memory begins
```

…

```
$31 ; return location for main
```

A Working Example… Using the Stack!

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    lis $4
    .word 4
    sw $31, -4($30)
    sub $30, $30, $4
PC  jalr $3
    add $30, $30, $4
    lw $31, -4($30)
    jr $31
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

…

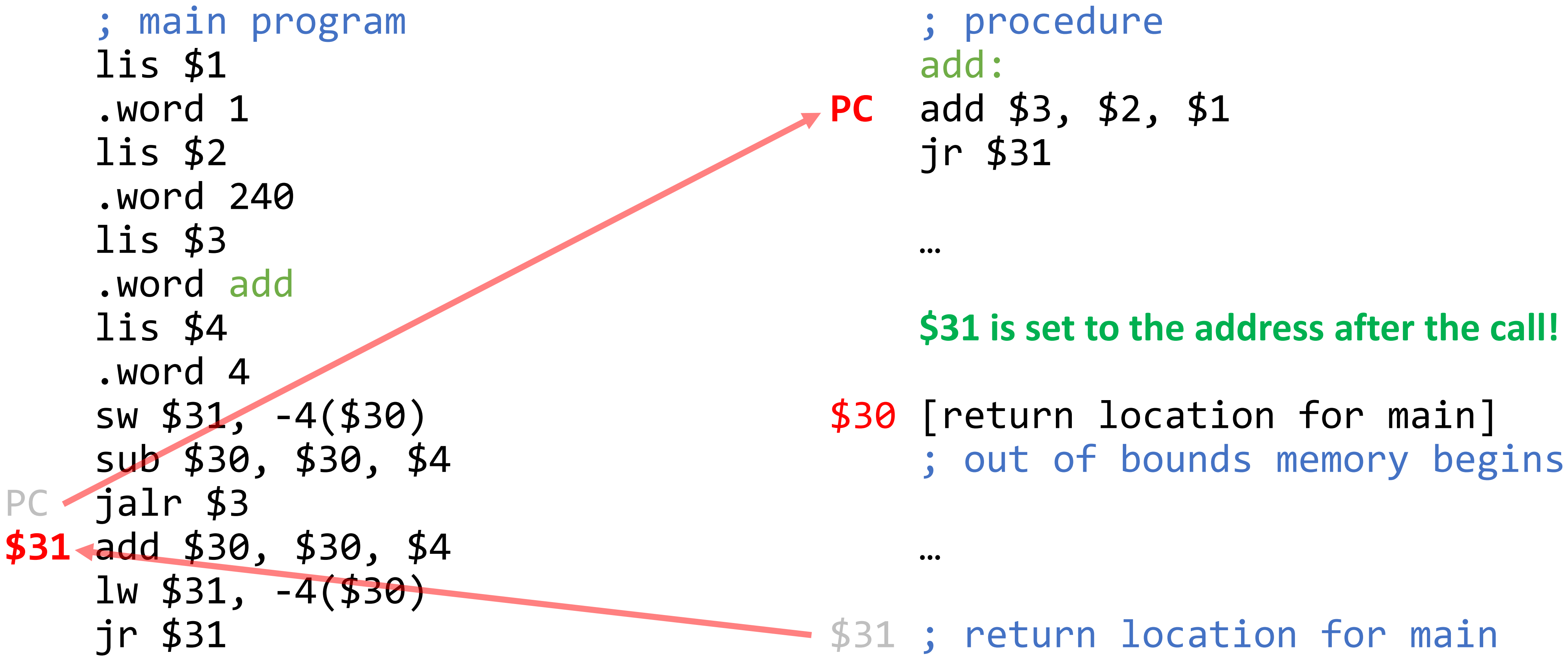
We're about to call the procedure…

```
$30 [return location for main]
    ; out of bounds memory begins
```

…

```
$31 ; return location for main
```

A Working Example… Using the Stack!



```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    lis $4
    .word 4
    sw $31, -4($30)
    sub $30, $30, $4
PC  jalr $3
$31 add $30, $30, $4
    lw $31, -4($30)
    jr $31
```

```
    ; procedure
    add:
PC  add $3, $2, $1
    jr $31

    …
```

$31 is set to the address after the call!

```
$30 [return location for main]
    ; out of bounds memory begins

    …

$31 ; return location for main
```

A Working Example… Using the Stack!

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    lis $4
    .word 4
    sw $31, -4($30)
    sub $30, $30, $4
    jalr $3
$31 add $30, $30, $4
    lw $31, -4($30)
    jr $31
```

```
    ; procedure
    add:
PC  add $3, $2, $1
    jr $31

    …
```

The procedure executes…

```
$30 [return location for main]
    ; out of bounds memory begins

    …

    ; return location for main
```

A Working Example… Using the Stack!

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    lis $4
    .word 4
    sw $31, -4($30)
    sub $30, $30, $4
    jalr $3
$31 add $30, $30, $4
    lw $31, -4($30)
    jr $31
```

```
    ; procedure
    add:
    add $3, $2, $1
PC  jr $31

    …
```

The procedure executes…

```
$30 [return location for main]
    ; out of bounds memory begins

    …

    ; return location for main
```

A Working Example… Using the Stack!

```
        ; main program
        lis $1
        .word 1
        lis $2
        .word 240
        lis $3
        .word add
        lis $4
        .word 4
        sw $31, -4($30)
        sub $30, $30, $4
        jalr $3
$31 PC  add $30, $30, $4
        lw $31, -4($30)
        jr $31
```

```
    ; procedure
    add:
    add $3, $2, $1
PC  jr $31

    …
```

Returns to the location in $31…

```
$30 [return location for main]
    ; out of bounds memory begins

    …

    ; return location for main
```

A Working Example… Using the Stack!

```
        ; main program
        lis $1
        .word 1
        lis $2
        .word 240
        lis $3
        .word add
        lis $4
        .word 4
        sw $31, -4($30)
        sub $30, $30, $4
        jalr $3
$31 PC  add $30, $30, $4
        lw $31, -4($30)
        jr $31
```

```
    ; procedure
    add:
    add $3, $2, $1
    jr $31

    …
```

Now we restore the old $31 from the stack!

```
$30 [return location for main]
    ; out of bounds memory begins

    …

    ; return location for main
```

A Working Example… Using the Stack!

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    lis $4
    .word 4
    sw $31, -4($30)
    sub $30, $30, $4
    jalr $3
$31 add $30, $30, $4
PC  lw $31, -4($30)
    jr $31
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

…

Pop (Step 1): Stack pointer $30 updated

```
$30 [return location for main]
$30 ; out of bounds memory begins
```

…

```
    ; return location for main
```

A Working Example… Using the Stack!

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    lis $4
    .word 4
    sw $31, -4($30)
    sub $30, $30, $4
    jalr $3
$31 add $30, $30, $4
PC  lw $31, -4($30)
    jr $31
```

```
; procedure
add:
add $3, $2, $1
jr $31

…
```

Pop (Step 2): Load old top of stack into $31

We copy this into $31 →

```
    [return location for main]
$30 ; out of bounds memory begins

    …

    ; return location for main
```

A Working Example… Using the Stack!

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    lis $4
    .word 4
    sw $31, -4($30)
    sub $30, $30, $4
    jalr $3
$31 add $30, $30, $4
    lw $31, -4($30)
PC  jr $31
```

```
; procedure
add:
add $3, $2, $1
jr $31
```

…

Pop (Step 2): Load old top of stack into $31

We copy this into $31

```
    [return location for main]
$30 ; out of bounds memory begins
```

…

```
$31 ; return location for main
```

A Working Example… Using the Stack!

```
    ; main program
    lis $1
    .word 1
    lis $2
    .word 240
    lis $3
    .word add
    lis $4
    .word 4
    sw $31, -4($30)
    sub $30, $30, $4
    jalr $3
    add $30, $30, $4
    lw $31, -4($30)
PC  jr $31
```

```
    ; procedure
    add:
    add $3, $2, $1
    jr $31

    …
```

Now we can properly return!

```
    [return location for main]
$30 ; out of bounds memory begins

    …

$31 ; return location for main
```

A Working Example… Using the Stack!

```
   ; main program
   lis $1
   .word 1
   lis $2
   .word 240
   lis $3
   .word add
   lis $4
   .word 4
   sw $31, -4($30)
   sub $30, $30, $4
   jalr $3
   add $30, $30, $4
   lw $31, -4($30)
PC jr $31
```

```
; procedure
add:
add $3, $2, $1
jr $31

…
```

Now we can properly return!

```
       [return location for main]
   $30 ; out of bounds memory begins

       …

PC $31 ; return location for main
```

A Working Example... Using the Stack!

```
; main program
lis $1
.word 1
lis $2
.word 240
lis $3
.word add
lis $4
.word 4
sw $31, -4($30)
sub $30, $30, $4
jalr $3
add $30, $30, $4
lw $31, -4($30)
jr $31
```

```
       ; procedure
       add:
       add $3, $2, $1
       jr $31

       …

       [return location for main]
   $30 ; out of bounds memory begins

       …

PC $31 ; return location for main
```

Preserving Other Registers

- Consider this procedure, which expects $1 to be a number from 1 to 26, and prints the corresponding letter (A = 1, B = 2, …, Z = 26).
- This code **has problems**, don't use it as an example. We'll fix it soon.

```
printAlpha:
lis $12
.word 0xFFFF000C ; storing to this address produces output
lis $4
.word 0x40       ; 0x41 = A, 0x42 = B, etc. in ASCII
add $1, $1, $4   ; convert to ASCII
sw $1, 0($12)
jr $31
```

Preserving Other Registers

- This code doesn’t preserve $31, but that’s okay, because it doesn’t call any other procedures.
- But what about *other registers?* This code clobbers $1, $4 and $12.

```
printAlpha:
lis $12
.word 0xFFFF000C ; storing to this address produces output
lis $4
.word 0x40       ; 0x41 = A, 0x42 = B, etc. in ASCII
add $1, $1, $4   ; convert to ASCII
sw $1, 0($12)
jr $31
```

Preserving Other Registers

- Imagine you wrote a screaming program that stores 1 in $1 and calls printAlpha 10 times, intending to print out the string AAAAAAAAAA.
- This would not work unless you manually reset $1 between each call!

```
printAlpha:
lis $12
.word 0xFFFF000C ; storing to this address produces output
lis $4
.word 0x40       ; 0x41 = A, 0x42 = B, etc. in ASCII
add $1, $1, $4   ; convert to ASCII
sw $1, 0($12)
jr $31
```

Preserving Other Registers

- Not to mention: If your code uses $4 or $12 for anything important, you need to back them up before calling this procedure.
- This procedure technically *works* but it’s a hassle to use.

```
printAlpha:
lis $12
.word 0xFFFF000C ; storing to this address produces output
lis $4
.word 0x40       ; 0x41 = A, 0x42 = B, etc. in ASCII
add $1, $1, $4   ; convert to ASCII
sw $1, 0($12)
jr $31
```

Preserving Other Registers

- **Convention:** Procedures should preserve the values of all registers that they modify, unless the modification is an intentional side effect of calling the procedure (e.g., storing a return value in $3).
- Aside from the return address, preserving modified registers generally isn't *necessary* to make a procedure work.
- However, procedures that freely mess up registers are *annoying to use* because you need to keep track of what it's going to mess up.
- Also, recursive procedures often outright won't work unless you properly preserve registers using the stack.

Batching Pushes and Pops

- Let’s say we want to push $1, $4, and $12 to the stack.
- Instead of this:

```
sw $1, -4($30)
lis $1
.word 4
sub $30, $30, $1
sw $4, -4($30)
lis $4
.word 4
sub $30, $30, $4
sw $12, -4($30)
lis $12
.word 4
sub $30, $30, $12
```

Batching Pushes and Pops

- Let’s say we want to push $1, $4, and $12 to the stack.
- We can do this:

```
sw $1,   -4($30)
sw $4,   -8($30)
sw $12, -12($30)
lis $12
.word 12
sub $30, $30, $12
```

- Put all the data on the stack, then decrement the stack pointer once.
- Decrement the stack pointer by 4x the number of things you pushed.

Batching Pushes and Pops

- Similarly, we can can batch pops.
- Increment first, then load the data.

```
lis $12
.word 12
add $30, $30, $12
lw $1,   -4($30)
lw $4,   -8($30)
lw $12, -12($30)
```

The Fixed printAlpha Procedure

```
printAlpha:
; save registers
sw $1,   -4($30)
sw $4,   -8($30)
sw $12, -12($30)
lis $12
.word 12
sub $30, $30, $12
```

```
; original body
lis $12
.word 0xFFFF000C
lis $4
.word 0x40
add $1, $1, $4
sw $1, 0($12)
```

```
; restore and return
lis $12
.word 12
add $30, $30, $12
lw $1,   -4($30)
lw $4,   -8($30)
lw $12, -12($30)
jr $31
```

Calling Code

- Here is another example of code that calls a procedure.

```
; save $31 on the stack
sw $31, -4($30)
lis $31
.word 4
sub $30, $30, $31
; load address of printAlpha into $5
lis $5
.word printAlpha
; C = 3
lis $1
.word 3
jalr $5 ; call printAlpha(3)
```

- This is a main program that uses printAlpha to print out “CS”.

```
; S = 19
lis $1
.word 19
jalr $5 ; call printAlpha(19)
; load $31 from stack
lis $31
.word 4
add $30, $30, $31
lw $31, -4($30)
; end the program
jr $31
```

Parameters/Arguments and Return Values

- We have not really talked about how to handle these.
- In short: “Decide on a convention and document it clearly.”
- For hand-written procedures, usually we say: “The procedure expects arguments in registers $X and $Y, and returns the result in $Z.”
  - Course convention: Return values are almost always in $3.
- Later (much later) when we are writing a compiler that *generates code* for procedures, we will store arguments on the stack.
  - This method is more general and doesn’t require keeping track of which procedures use which registers for arguments…
  - But using the stack for arguments when hand-writing procedures is a hassle.

Recursion

- If you follow our rules and conventions, recursion should “just work”.
    - Even mutual recursion!
- Recursive procedures, by definition, call a procedure as part of their code, so they must save and restore their return address ($31).
- They should save and restore other registers they modify, **except if** the register is used to hold the return value.
    - If you save and restore the return register, the return value gets lost!
- Sometimes you can get away with not saving and restoring certain registers if the value isn’t important outside of the procedure.
- But recursive procedures tend to *rely on the very same registers that they modify*. So not saving and restoring properly can really break things.

Stack Overflow!

- A recursive procedure (or collection of mutually recursive procedures) will push registers to the stack at the start of every call.
- If the recursion never terminates due to a coding error, the stack will keep growing and growing.
- A **stack overflow** is when the stack gets too large (usually, when it exceeds a preset size specified by the programming environment).
- The MIPS emulator does not have any kind of stack size limit, so the stack will keep growing until it *starts overwriting the program code.*
- This means the fetch-execute cycle will try to execute the contents of the stack and usually crash very quickly!

Recursive Procedure Example

- Compute the nth Fibonacci number F(n), where:
  - **F(0) = 0** and **F(1) = 1**.
  - **F(n) = F(n − 1) + F(n − 2)** for n ≥ 2.
- The parameter n should be placed in $1.
- The result F(n) should be returned in $3.
- The procedure should preserve all registers except for $3.
- The procedure will use the naïve recursive solution (exponential time).

The Fibonacci Procedure

```
; fib: computes the nth
;      fibonacci number
; fib(0) = 0, fib(1) = 1
; fib(n) = fib(n-1) + fib(n-2)
; parameters: $1 = n
; returns fib(n) in $3
fib:
; save registers
sw $1,   -4($30)
sw $4,   -8($30)
sw $5,  -12($30)
sw $11, -16($30)
sw $31, -20($30)
lis $31
.word 20
sub $30, $30, $31
```

```
; procedure body
lis $11         ; $11 = 1
.word 1
add $3, $0, $0  ; $3  = 0
; check base cases
beq $1, $0, end ; base case: n = 0
bne $1, $11, recurse
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
recurse:
lis $5
.word fib
sub $1, $1, $11 ; set $1 = n-1
jalr $5         ; $3 = fib(n-1)
add $4, $3, $0  ; copy $3 to $4
sub $1, $1, $11 ; set $1 = n-2
jalr $5         ; $3 = fib(n-2)
add $3, $4, $3  ; compute fib(n)
```

```
end:
; restore registers
lis $31
.word 20
add $30, $30, $31
lw $1,   -4($30)
lw $4,   -8($30)
lw $5,  -12($30)
lw $11, -16($30)
lw $31, -20($30)
; return to caller
jr $31
```

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 3	
$3 = ?	
$4 = ?	
$5 = ?	
$11 = ?	
$31 = return to caller	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 3	
$3 = ?	Caller's $31
$4 = ?	Caller's $11
$5 = ?	Caller's $5
$11 = ?	Caller's $4
$31 = return to caller	Caller's $1

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
➡ ; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 3	
$3 = ?	
$4 = ?	
$5 = ?	
$11 = ?	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to caller	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 3	
$3 = 0	
$4 = ?	
$5 = ?	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to caller	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 3	
$3 = 0	
$4 = ?	
$5 = ?	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to caller	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 3	
$3 = 0	
$4 = ?	
$5 = ?	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to caller	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 3	
$3 = 0	
$4 = ?	
$5 = ?	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to caller	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 3`	
`$3 = 0`	
`$4 = ?`	
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to caller`	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 2	
$3 = 0	
$4 = ?	
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to caller	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 2	
$3 = 0	
$4 = ?	
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 1	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 2	
$3 = 0	
$4 = ?	**[Call 1's Registers]** **$4, $5, $11** **$1 = 2** **$31 = return to call 1**
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 1	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 2	
$3 = 0	
$4 = ?	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 1	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	
$3 = 0	
$4 = ?	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 1	

Tracing the Fibonacci Procedure

Call 3: fib(1)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	
$3 = 0	
$4 = ?	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2	

Tracing the Fibonacci Procedure

Call 3: fib(1)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	**[Call 2's Registers]** **$4, $5, $11** **$1 = 1** **$31 = return to call 2**
$3 = 0	
$4 = ?	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2	

Tracing the Fibonacci Procedure

Call 3: fib(1)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	[Call 2's Registers] $4, $5, $11 $1 = 1 $31 = return to call 2
$3 = 0	
$4 = ?	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2	

Tracing the Fibonacci Procedure

Call 3: fib(1)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 1`	[Call 2's Registers] $4, $5, $11 $1 = 1 $31 = return to call 2
`$3 = 0`	
`$4 = ?`	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to call 2`	

Tracing the Fibonacci Procedure

Call 3: fib(1)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	[Call 2's Registers] $4, $5, $11 $1 = 1 $31 = return to call 2
$3 = 1	
$4 = ?	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2	

Tracing the Fibonacci Procedure

Call 3: fib(1)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	[Call 2's Registers] $4, $5, $11 $1 = 1 $31 = return to call 2
$3 = 1	
$4 = ?	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2	

Tracing the Fibonacci Procedure

Call 3: fib(1)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	
$3 = 1	
$4 = ?	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	
$3 = 1	
$4 = ?	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 1`	
`$3 = 1`	
`$4 = 1`	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to call 2`	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 0`	
`$3 = 1`	
`$4 = 1`	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to call 2`	

Tracing the Fibonacci Procedure

Call 4: fib(0)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 0`	
`$3 = 1`	
`$4 = 1`	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to call 2 (#2)`	

Tracing the Fibonacci Procedure

Call 4: fib(0)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 0`	**[Call 2's Registers]** **$5, $11** **$1 = 2, $4 = 1** **$31 = return to call 2, #2**
`$3 = 1`	
`$4 = 1`	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to call 2 (#2)`	

Tracing the Fibonacci Procedure

Call 4: fib(0)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 0`	[Call 2's Registers] $5, $11 $1 = 2, $4 = 1 $31 = return to call 2, #2
`$3 = 0`	
`$4 = 1`	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to call 2 (#2)`	

Tracing the Fibonacci Procedure

Call 4: fib(0)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 0	[Call 2's Registers] $5, $11 $1 = 2, $4 = 1 $31 = return to call 2, #2
$3 = 0	
$4 = 1	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2 (#2)	

Tracing the Fibonacci Procedure

Call 4: fib(0)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 0	
$3 = 0	
$4 = 1	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2 (#2)	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
➡ ; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 0	
$3 = 0	
$4 = 1	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 2 (#2)	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 0`	
`$3 = 1`	
`$4 = 1`	[Call 1's Registers] $4, $5, $11 $1 = 2 $31 = return to call 1
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to call 2 (#2)`	

Tracing the Fibonacci Procedure

Call 2: fib(2)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 2	
$3 = 1	
$4 = ?	
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 1	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 2	
$3 = 1	
$4 = ?	
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 1	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
➡ ; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 2	
$3 = 1	
$4 = 1	
$5 = address of fib proc.	
$11 = 1	[Caller’s Registers] $1, $4, $5, $11, $31
$31 = return to call 1	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 1`	
`$3 = 1`	
`$4 = 1`	
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to call 1`	

Tracing the Fibonacci Procedure

Call 5: fib(1)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	
$3 = 1	
$4 = 1	
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 1 (#2)	

Tracing the Fibonacci Procedure

Call 5: fib(1)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
➡ jr $31
```

Registers	Stack
`$1 = 1`	
`$3 = 1`	
`$4 = 1`	
`$5 = address of fib proc.`	
`$11 = 1`	[Caller's Registers] $1, $4, $5, $11, $31
`$31 = return to call 1 (#2)`	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	
$3 = 1	
$4 = 1	
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 1 (#2)	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 1	
$3 = 2	
$4 = 1	
$5 = address of fib proc.	
$11 = 1	[Caller's Registers] $1, $4, $5, $11, $31
$31 = return to call 1 (#2)	

Tracing the Fibonacci Procedure

Call 1: fib(3)

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
$1 = 3	
$3 = 2	
$4 = ? (caller’s)	
$5 = ? (caller’s)	
$11 = ? (caller’s)	
$31 = return to caller	

Tracing the Fibonacci Procedure

Result: fib(3) = 2

```
fib:
; save registers
…
; procedure body
; $11 = 1
lis $11
.word 1
; $3 = 0
add $3, $0, $0
; check base cases
; base case: n = 0
beq $1, $0, end
; go to recursive case
; if n != 1
bne $1, $11, recurse
; otherwise:
; base case: n = 1
add $3, $0, $11
beq $0, $0, end
```

```
recurse:
lis $5
.word fib
; $1 = n-1
sub $1, $1, $11
; $3 = fib(n-1)
jalr $5
; copy $3 into $4
add $4, $3, $0
; $1 = n-2
sub $1, $1, $11
; $3 = fib(n-2)
jalr $5
; compute fib(n)
add $3, $4, $3
end:
; restore registers
…
; return to caller
jr $31
```

Registers	Stack
`$1 = 3`	
`$3 = 2`	
`$4 = ? (caller’s)`	
`$5 = ? (caller’s)`	
`$11 = ? (caller’s)`	
`$31 = return to caller`	

Summary

- A lot of things that we take for granted when using procedures have to be implemented carefully in assembly.
- For example, we usually expect that calling a procedure will not overwrite important values in our code.
  - We need to save and restore registers to ensure this will not happen.
- We expect that procedures can call other procedures (including recursively) and the call stack will unwind naturally as they return.
  - We need to use the jalr instruction to ensure we return to the right place.
  - We need our own stack to keep track of the chain of return addresses.
- Even if you get the concepts, procedures are hard to debug. Beware!!