Write an assembly language MIPS program that takes a value in register $1 and stores the sum of the digits in register $2.
Procedures

Let’s generalize the above. How do we write procedures/functions in MIPS? Some issues:

- How do we transfer control to and from a procedure?
- What if our procedures call other procedures?
- How do we pass parameters?
- How would recursion work?
- How would we return values?
- What if a procedure wants to use registers that have data already? Calling a function might clobber such data!
Discuss solutions

What if a procedure wants to use registers that have data already?
Discuss solutions

What if a procedure wants to use registers that have data already?

We could reserve registers. Clearly though we might run out. Should make sure procedures ensure that registers remain unchanged by the end of the procedure - but how?
Discuss solutions

What if a procedure wants to use registers that have data already?

We could reserve registers. Clearly though we might run out. Should make sure procedures ensure that registers remain unchanged by the end of the procedure - but how?
Well we have lots of memory in RAM that we can use! However we don’t want our procedures to use the same RAM. How can we guarantee that procedures don’t erase each other’s work?
Discuss solutions

What if a procedure wants to use registers that have data already?

We could reserve registers. Clearly though we might run out. Should make sure procedures ensure that registers remain unchanged by the end of the procedure - but how? Well we have lots of memory in RAM that we can use! However we don’t want our procedures to use the same RAM. How can we guarantee that procedures don’t erase each other’s work?

We would need to keep track of the free RAM. The loader helps us out here by letting us know where the start of free RAM is (see next slide).
Register $30$ points to the very bottom of the free RAM. It can be used as a bookmark to separate the used and unused free RAM if we allocate from the bottom and pop things off like a stack! (See next slide)
### RAM

<table>
<thead>
<tr>
<th>Your Program</th>
<th>Free RAM</th>
<th>Used RAM</th>
<th>$30</th>
</tr>
</thead>
</table>

Example

Suppose procedures $f$, $g$, and $h$ are such that:
- $f$ calls procedure $g$
- $g$ calls procedure $h$
- $h$ returns
- $g$ returns
- $f$ returns.

Then, your RAM with this model will look like it does on the right:

<table>
<thead>
<tr>
<th>Your Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free RAM</td>
</tr>
<tr>
<td>$h$ registers</td>
</tr>
<tr>
<td>$g$ registers</td>
</tr>
<tr>
<td>$f$ registers</td>
</tr>
</tbody>
</table>
In the previous example:

- Calling procedures pushes more registers onto the stack and returning pops them off.
- This is a stack and we call $30 our stack pointer.
- We can also use the stack for local storage if needed in procedures. Just reset $30 before procedures return.
- Note that the MIPS standard guideline often will use $29 for this purpose. (We use it differently)
Template for Procedures

Template for a procedure $f$ that modifies registers $\$1$ and $\$2$:
Template for Procedures

Template for a procedure $f$ that modifies registers $1$ and $2$:

$f$:

```
sw $1, -4($30)    ; Push registers we will modify
sw $2, -8($30)
lis $2            ; Decrement stack pointer
.word 8
sub $30, $30, $2
```

 Uh oh! How do we return?
Template for Procedures

Template for a procedure $f$ that modifies registers $\$1$ and $\$2$:

$f:$

\[
\begin{align*}
sw & \ \$1, -4(\$30) \quad ; \text{Push registers we will modify} \\
sw & \ \$2, -8(\$30) \\
lis & \ \$2 \quad ; \text{Decrement stack pointer} \\
.word & \ 8 \\
sub & \ \$30, \$30, \$2 \\
\quad ; \text{Insert procedure here}
\end{align*}
\]
Template for Procedures

Template for a procedure $f$ that modifies registers $1$ and $2$:

```
f:
    sw $1, -4($30) ; Push registers we will modify
    sw $2, -8($30)
    lis $2 ; Decrement stack pointer
    .word 8
    sub $30, $30, $2 ; Insert procedure here
    add $30, $30, $2 ; Assuming $2$ is still 8
    lw $2, -8($30) ; Pop registers to restore
    lw $1, -4($30)
    ; Uh oh! How do we return?
```
There is a problem with returning:

main:

lis $8

.word f ; Recall f is an address

jr $8 ; Jump to the line of f

(NEXT LINE)

Once f completes we really want to jump back to the line labelled above as (NEXT LINE), that is, we need to set the program counter to the line after the jr $8 command! How do we do that?
A New Command!

jalr $s: 0000 00ss sss0 0000 0000 0000 0000 1001

Jump and Link Register.
Sets $31 to be the PC and then sets the pc to be $s.
Accomplished by temp = $s then $31 = PC then PC = temp.

A new problem: jalr will overwrite register $31.
How do we return back to the loader from main?
What if procedures call each other?
A New Command!

jalr $s: 0000 00ss sss0 0000 0000 0000 0000 1001

Jump and Link Register.
Sets $31 to be the PC and then sets the pc to be $s.
Accomplished by temp = $s then $31 = PC then PC = temp.

A new problem: jalr will overwrite register $31.
How do we return back to the loader from main?
What if procedures call each other?

We need to save this register first!
Main Changes

Modifications to main
Main Changes

Modifications to main

main:
  lis $8
  .word f
  sw $31, -4($30); Push $31 to stack
  lis $31; Use $31 since it has been saved
  .word 4
  sub $30, $30, $31
  jalr $8;
  lis $31; Use $31 since we will pop from stack
  .word 4
  add $30, $30, $31
  lw $31, -4($30); Pop $31 to stack
  jr $31; Return to loader
Procedure Changes

Modifications to $f$

```assembly
f:
    sw $1, -4($30) ; Push registers we will modify
    sw $2, -8($30)
    lis $2 ; Decrement stack pointer
    .word 8
    sub $30, $30, $2 ; Insert procedure here
    add $30, $30, $2 ; Assuming $2 is still 8
    lw $2, -8($30) ; Pop registers to restore
    lw $1, -4($30)
    jr $31; New line!
```
Unresolved Questions

Most of our original questions have been resolved except for one:

How to we pass parameters?
Unresolved Questions

Most of our original questions have been resolved except for one:

How to we pass parameters?

Typically we’ll just use registers. If we have too many, we could push parameters to the stack and then pop them from the stack. Documentation is vitally important here!

If we can do this correctly, then everything, including recursion, should just work properly.
Sum Evens 1 to N Slide 1 of 2

; sumEvens1ToN adds all even numbers from 1 to N
; requires N even
; Registers:
; $1 Scratch Register; Should Save!
; $2 Input Register; Should Save!
; $3 Output Register; Do NOT Save!
Sum Evens 1 to $N$ Slide 1 of 2

; sumEvens1ToN adds all even numbers from 1 to $N$
; requires $N$ even
; Registers:
; $1$ Scratch Register; Should Save!
; $2$ Input Register; Should Save!
; $3$ Output Register; Do NOT Save!

sumEvens1ToN:
  sw $1, -4($30) ; Save $1 and $2
  sw $2, -8($30)
  lis $1 ; Decrement stack pointer
  .word 8
  sub $30, $30, $1
  add $3, $0, $0 ; Don’t forget to initialize $3!
  lis $1
  .word 2

; ....continued on next slide!
top:
    add $3, $3, $2 ; 0x14
    sub $2, $2, $1
    bne $2, $0, top
lis $1
    .word 8
add $30, $30, $1
lw $2, -8($30)
lw $1, -4($30) ; Reload $1 and $2
jr $31 ; Back to caller
;End sumEvens1ToN
Another outstanding problem: How to we print to the screen or read input?
Another outstanding problem: How to we print to the screen or read input?

We do this one byte at a time!

- **Output**: Use `sw` to store words in location `0xffffffff000c`. Least significant byte will be printed.
- **Input**: Use `lw` to load words in location `0xffffffff0004`. Least significant byte will be the next character from `stdin`. 
Example

Printing CS241 to the screen followed by a newline character:

```assembly
lis $1
.word 0xffff000c
lis $2
.word 67 ; C
sw $2, 0($1)
lis $2
.word 83 ; S
sw $2, 0($1)
lis $2
.word 50 ; 2
sw $2, 0($1)
lis $2
.word 52 ; 4
sw $2, 0($1)
lis $2
.word 49 ; 1
sw $2, 0($1)
lis $2
.word 10 ; \n
sw $2, 0($1)
jr $31
```
The Assembler

Recall part of our longterm goal is to convert assembly code (our MIPS language) into machine code (zeroes and ones).

- Input: Assembly code
- Output: Machine code

Any such translation process involves two phases: **Analysis** and **Synthesis**.

- Analysis: Understand what is meant by the input source
- Synthesis: Output the equivalent target code in the new format
Assembly File

- Think of it as a string of characters.
- We want to first break it down into meaningful tokens such as labels, numbers, .word, MIPS instructions and so on.
- This is done for you in asm.rkt and asm.cc.

Your job (should you choose to accept it):
- Analysis: Group tokens into instructions if possible
- Synthesis: Output equivalent machine code.

If the tokens are not valid instructions, output ERROR to stderr.
Assignment Advice

- There are many more incorrect tokens than correct ones.
- Focus on finding correct ones! (More on this in upcoming weeks)
- Later we will discuss parsing, a formal way of grouping tokens.
The Biggest Assembler Problem

How do we assemble this code:

```
beq $0, $1, myLabel
myLabel: add $1, $1, $1
```

The problem is when we see `myLabel` for the first time, we don't know what the correct corresponding address is!

What is the best fix to this?
Standard Solution:

Make two passes:

- **Pass 1**: Group tokens into instructions and record address of all labelled instructions. This can be done in a symbol table, that is, a list of label address pairs.

  **Note**: multiple labels are possible for the same line! For example, \( f: \quad g: \quad \text{add } \$1, \$1, \$1. \)

- **Pass 2**: translate each instructions into machine code. If it refers to a label, look up the associated address in the aforementioned symbol table and perform the computation to compute the correct corresponding needed value.
When writing your assembler, you will do two things:

- Output the machine code coming from the assembled MIPS code to stdout.
- Output the symbol table to stderr.