Implementing Procedures in MIPS

By now you should know why programmers use procedures!

Flashback to 1st year:

- Where do you place the code you write for a function?
- What happens when you call a function?
- Where are parameters and local variables stored?
- Where does control go when function returns?
- How do values get returned from a function?
- What's the difference between a procedure and a function?

Implementing Procedures in MIPS

All those little things you take for granted in a high-level language, you will need to implement yourself!

Procedures in MIPS are a bit different:

- All procedures share the same set of registers
- Procedures do not return values

Logistics: How to call and return from a procedure?

Problem: How to share registers?

- A caller may have critical data stored in a register that the callee should not overwrite!

Sharing Registers

Strategy: guarantee that when a procedure ends, the values in the registers are the same as when the procedure was called.

Where can we save register data?

Sharing Registers

Strategy: guarantee that when a procedure ends, the values in the registers are the same as when the procedure was called.

Where can we save register data?

- Two places we typically store data is in memory (RAM) or registers (CPU).
- It would be nice to save everything in registers (fast access, etc) but space is very limited.
- If we use registers, we may run out.

Where does C/C++ store data related to function calls?

Sharing Registers

Strategy: guarantee that when a procedure ends, the values in the registers are the same as when the procedure was called.

Where can we save register data?

- Two places we typically store data is in memory (RAM) or registers (CPU).
- It would be nice to save everything in registers (fast access, etc) but space is very limited.
- If we use registers, we may run out.

Where does C/C++ store data related to function calls? **Call stack**

Storing on the Stack

Recall: a loader allocates a block of RAM (larger than program) and loads our program at the top of the block, then sets the PC $\leftarrow 0$.

It also sets $30 to the memory address immediately following the allocated block.

- Use $30 to store address of the top of stack
- Grow stack from high memory addresses to low

For example: if procedure `f` calls `g` and `g` calls `h` then

- `f` stores register data at the bottom of stack
  `g` stores register data above `f`
  `h` stores register data at top of stack

Strategy: each time a procedure is called, it will save the current value stored in the registers it wants to use on the stack and restore the original values when it ends.

- Only need to save registers that the procedure will overwrite. If in doubt, save everything.
- Remember registers are 32 bits or 4 bytes.
- Remember to increment (decrement) $30 when you push (pop).
- Remember the order you placed items on the stack.
- Careful of “off by 1 errors".

Template for Procedures

Suppose procedure `f` modifies registers $1 and $2:

```
f: sw $1, -4($30)    ; Push registers f modifies
   sw $2, -8($30)
   lis $2            ; Decrement stack pointer
   .word 8
   sub $30, $30, $2
   ; Body of your procedure goes here
   lis $2            ; Increment stack pointer
   .word 8
   add $30, $30, $2
   lw $2, -8($30)    ; Pop registers to restore
   lw $1, -4($30)
   ; How do we return?
```

Calling and Returning

Label `f` represents the memory address of procedure `f`.

```
main:
      lis $5
      .word f
      jr $5
      ; RETURN HERE
      ...

f: ...
```

How do we know the memory address where `f` returns to?

Returning - `jalr`

MIPS Reference Sheet: `jalr $s`

Last instruction: Jump and Link Registers

Copies PC into $31 then jumps to address stored in $s.

I'm suppose to remember something about $31?

Returning - `jalr`

MIPS Reference Sheet: `jalr $s`

Last instruction: Jump and Link Registers

Copies PC into $31 then jumps to address stored in $s.

I'm suppose to remember something about $31?

- $31 is special - it stores the memory address (in the loader program) we jump back to when our program ends.

Who saves $31? Procedure `f`?

Returning - `jalr`

MIPS Reference Sheet: `jalr $s`

Last instruction: Jump and Link Registers

Copies PC into $31 then jumps to address stored in $s.

I'm suppose to remember something about $31?

- $31 is special - it stores the memory address (in the loader program) we jump back to when our program ends.

Who saves $31? Procedure `f`?

- We need to save it before we jump to `f`.
- The caller saves $31 first, then calls the procedure.

```
main:
      lis $5
      .word f
      sw $31, -4($30)   ; Push $31
      lis $31
      .word -4
      add $30, $30, $31
      jalr $5           ; Jump to f
      lis $31           ; Pop to restore $31
      .word 4
      add $30, $30, $31
      lw $31, -4($30)
      jr $31            ; Return to loader
```

```
f:
   sw $1, -4($30)    ; Push registers f modifies
   sw $2, -8($30)
   lis $2            ; Decrement stack pointer
   .word 8
   sub $30, $30, $2
   ; Body of your procedure goes here
   lis $2            ; Increment stack pointer
   .word 8
   add $30, $30, $2
   lw $2, -8($30)    ; Pop registers to restore
   lw $1, -4($30)
   jr $31            ; *NEW* Return to caller
```

Parameters and Result Passing

- Simple approach: use registers (Document!)
- If too many parameters, can use memory (stack)

Example: Procedure `sumEvens2ToN`

```
; sumEvens2ToN: adds all even numbers from 2 .. N
; Requires: N is even
; Registers:
;   $1 - Temporary work
;   $2 - Parameter N
;   $3 - Sum to return
```

Which registers should be saved?

Parameters and Result Passing

- Simple approach: use registers (Document!)
- If too many parameters, can use memory (stack)

Example: Procedure `sumEvens2ToN`

```
; sumEvens2ToN: adds all even numbers from 2 .. N
; Requires: N is even
; Registers:
;   $1 - Temporary work   Must Save This!
;   $2 - Parameter N      Should Save This!
;   $3 - Sum to return    Do NOT Save!
```

Which registers should be saved?

```
sumEvens2ToN:
  sw $1, -4($30)     ; Save $1 and $2 on stack
  sw $2, -8($30)
  lis $1             ; Use Temporary work register
  .word 8
  sub $30, $30, $1   ; Decrement stack pointer
  add $3, $0, $0     ; Initialize sum <- 0
  lis $1             ; Use Temporary work register
  .word 2
 topLoop:
         add $3, $3, $2
         sub $2, $2, $1
         bne $2, $0, top
; ... continued on next slide
```

```
lis $1              ; Restore $1 and $2
.word 8
add $30, $30, $1
lw $2, -8($30)
lw $1, -4($30)
jr $31              ; Jump back to caller
```

Printing to Stdout

Use `sw` to store a word in address `0xffff000c`.

Least significant byte will be printed to `stdout`.

Example: Write a program that prints "CS`\n`" followed by newline.

```
lis $1
.word 0xffff000c
lis $2
.word 67  ; ASCII 'C'
sw $2, 0($1)
lis $2
.word 83  ; ASCII 'S'
sw $2, 0($1)
lis $2
.word 10  ; ASCII '\n'
sw $2, 0($1)
jr $31
```

Reading from Stdin

Use `lw` to read a word from address `0xffff0004`.

Least significant byte will be read from `stdin`.