CS 2110 Lab 13
Recursive Assembly

The TAs :)

10/10/2018

1 Recitation Outline

1. Helpful Tips and Tricks for Recursive Assembly
2. Example file (fibonacci.asm)

2 Helpful Tips and Tricks for Recursive Assembly

2.1 Important Registers

R7 - holds the current return address
R6 - holds the stack pointer/top of the stack, increment/decrement this when you want to pop/push things onto the stack
R5 - holds the current frame pointer. This is useful when you want to obtain values on the stack

2.2 The STACK

The stack is our calling convention for recursive assembly. The stack grows downwards (into lower memory), but we draw the stack such that we consider the top of the stack to be the lowest memory address on the stack.

TOP (LOWER MEMORY)			
	saved registers	<-R6 (top of stack)	
	local	<-R5 (1 above oldFP)	
	old FP (R5)		
	RA (old R7)		
	RV		
	1st param		
xF000			
BOTTOM (HIGHER MEMORY)			

3 Example file - fibonacci

Note

This example file is an example of how you can code a recursive assembly function. Please feel free to use this as a reference, but there is no substitute for logically thinking your way through a recursive assembly problem. When in doubt, draw the stack and walk through your code in complx.

1. To begin our coding, take a look at the code snippet below. Most of it is recognizable - the .orig x3000 and corresponding .end which represent where our instructions will lie in memory. Two of these lines are not recognizable - fibonacci and STACK .fill xF000. Fibonacci is the beginning of our subroutine and all of our code will begin **underneath this line**. To run this code in complx, go to Debug -> simulate subroutine call and choose the fibonacci subroutine call. The STACK .fill xF000 line of code indicates that our stack will begin at memory address xF000 and is needed for the subroutine call.

```
.orig x3000

halt

fibonacci

STACK .fill xF000
.end
```

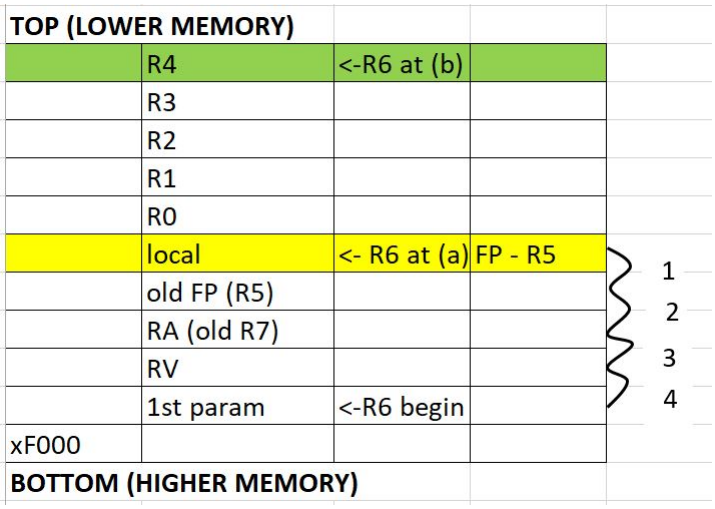
2. First, we need to write the initial build up of the stack. Remember R6 holds the stack pointer, R5 holds the current frame pointer, and R7 holds the return address.

```
ADD R6, R6, -4 ;place stack pointer to build up first portion of the stack (a)
STR R7, R6, 2  ;save old return address
STR R5, R6, 1  ;save old frame pointer
ADD R5, R6, 0  ;set current frame pointer
ADD R6, R6, -5 ;place stack pointer to save all of the registers on the stack (b)
STR R0, R6, 4  ;save R0
STR R1, R6, 3  ;save R1
STR R2, R6, 2  ;save R2
STR R3, R6, 1  ;save R3
STR R4, R6, 0  ;save R4
```

We can see a more visual picture of the stack in the image below. The first thing to note is that the stack grows from higher memory to lower memory. Secondly, I built my stack by moving my stack pointer twice - once to create room for the return value and save the return address and old frame pointer. The other time to create room and save all registers and one local on the stack. At the end of this operation - our stack pointer (R6) is pointed to the very bottom of the stack (highlighted in green), while our frame pointer (R5) points to the memory address one above where the old FP was stored (highlighted in yellow).

TOP (LOWER MEMORY)			
	R4	<-R6 at (b)	
	R3		
	R2		
	R1		
	R0		
	local	<- R6 at (a)	FP - R5
	old FP (R5)		
	RA (old R7)		
	RV		
	1st param	<-R6 begin	
xF000			
BOTTOM (HIGHER MEMORY)			

3. Next we need to grab 'n', our parameter. We know from our calling convention that our first parameter is located directly underneath the spot for the return value, with the second parameter located underneath that. Knowing this, we grab 'n' by grabbing the element located at 4 + FP



4. After grabbing the parameter, (using the line `LDR R0, R5, 4`), we'll write the logic for the base case. First, based on the value of the parameter, we will branch to the BASE label, where all of our base case logic will go. All we need to do in this base case is place the return value on the stack (located at 3 + FP), then unconditionally branch to stack tear-down

```
LDR R0, R5, 4  ;R0 <= n

ADD R2, R0, -1
BRnz BASE

BASE
STR R0, R5, 3  ;place the return value in the return value place of the stack
BR STACK_BREAKDOWN

STACK_BREAKDOWN
```

5. The breakdown of the stack is the exact opposite of the build-up. We use the frame pointer to restore the registers, then bring the stack pointer down to the frame pointer. From there, we can restore the old frame pointer and the return address. Now that the correct return address is in R7, we have our R6 point to the return value and call RET to return to the calling subroutine.

```
STACK_BREAKDOWN
LDR R4, R5, -5 ;restore R4
LDR R3, R5, -4 ;restore R3
LDR R2, R5, -3 ;restore R2
LDR R1, R5, -2 ;restore R1
LDR R0, R5, -1 ;restore R0
ADD R6, R5, 0  ;bring R6 back down to R5
LDR R5, R6, 1  ;restore old frame pointer
LDR R7, R6, 2  ;restore return address
ADD R6, R6, 3  ;have R6 point to the return value
RET
```

In the following diagram, everything in red has been restored and the stack pointer (R6) points to the return value before calling RET.

TOP (LOWER MEMORY)			
	R4		
	R3		
	R2		
	R1		
	R0		
	local		
	old FP (R5)		
	RA (old R7)		
	RV	<- R6 when calling RET	
	1st param		
xF000			
BOTTOM (HIGHER MEMORY)			

6. For the recursive call in this problem, we first need to call fibonacci(n-1) and add this to fibonacci(n-2). To make a recursive call in assembly, decrement R6 to make room for another element on the stack and store the parameter to the next subroutine call there. Then call JSR fibonacci and the subroutine will have been called!

TOP (LOWER MEMORY)			
	param	<-R6 to call next sub	
	R4		
	R3		
	R2		
	R1		
	R0		
	local		
	old FP (R5)		
	RA (old R7)		
	RV		
	1st param		
xF000			
BOTTOM (HIGHER MEMORY)			

When the subroutine finishes, it will return with the stack pointer pointing to the return value of the callee subroutine. From there, we would need to do the logic to obtain the return value of the current subroutine and place that value in the return value slot of the stack (FP + 3), then increment R6 by 1 + number of parameters so that R6 points to the top of the saved registers before unconditionally branching to the stack breakdown.

Recursive Portion of our code:

```
ADD R6, R6, -1 ;R6 will now point to one above R4
ADD R1, R0, -1 ;getting n-1
STR R1, R6, 0  ;store (n-1) at first parameter
```

```
JSR fibonacci  ;fib(n-1)

LDR R1, R6, 0  ;when the subroutine ends, R6 will point to the return value
ADD R6, R6, 1  ;go back to the params
ADD R2, R0, -2 ;getting n-2
STR R2, R6, 0  ;store (n-2) at first parameter
JSR fibonacci  ;fib(n-2)

LDR R2, R6, 0  ;when the subroutine ends, R6 will point to the return value
ADD R6, R6, 2  ;we want r6 to point to the top of the register stack
ADD R3, R1, R2 ;have r3 hold the values of fib(n-1) + fib(n-2)
STR R3, R5, 3  ;store the return value!
BR STACK_BREAKDOWN
```

7. Below is the final code for the fibonacci problem:

```
.orig x3000

halt

fibonacci
ADD R6, R6, -4 ;place stack pointer to build up first portion of the stack (a)
STR R7, R6, 2  ;save old return address
STR R5, R6, 1  ;save old frame pointer
ADD R5, R6, 0  ;set current frame pointer
ADD R6, R6, -5 ;place stack pointer to save all of the registers on the stack (b)
STR R0, R6, 4  ;save R0
STR R1, R6, 3  ;save R1
STR R2, R6, 2  ;save R2
STR R3, R6, 1  ;save R3
STR R4, R6, 0  ;save R4

LDR R0, R5, 4  ;R0 <= n

ADD R2, R0, -1
BRnz BASE

ADD R6, R6, -1 ;R6 will now point to one above R4
ADD R1, R0, -1 ;getting n-1
STR R1, R6, 0  ;store (n-1) at first parameter
JSR fibonacci  ;fib(n-1)

LDR R1, R6, 0  ;when the subroutine ends, R6 will point to the return value
ADD R6, R6, 1  ;go back to the params
ADD R2, R0, -2 ;getting n-2
STR R2, R6, 0  ;store (n-2) at first parameter
JSR fibonacci  ;fib(n-2)

LDR R2, R6, 0  ;when the subroutine ends, R6 will point to the return value
ADD R6, R6, 2  ;we want r6 to point to the top of the register stack
ADD R3, R1, R2 ;have r3 hold the values of fib(n-1) + fib(n-2)
STR R3, R5, 3  ;store the return value!
BR STACK_BREAKDOWN
```

```
BASE
STR R0, R5, 3  ;place the return value in the return value place of the stack
BR STACK_BREAKDOWN

STACK_BREAKDOWN
LDR R4, R5, -5 ;restore R4
LDR R3, R5, -4 ;restore R3
LDR R2, R5, -3 ;restore R2
LDR R1, R5, -2 ;restore R1
LDR R0, R5, -1 ;restore R0
ADD R6, R5, 0  ;bring R6 back down to R5
LDR R5, R6, 1  ;restore old frame pointer
LDR R7, R6, 2  ;restore return address
ADD R6, R6, 3  ;have R6 point to the return value
RET

STACK .fill xF000
.end
```