UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 11: More Control Flow

Taylor Johnson

Announcements and Outline

- Quiz 3 on Blackboard (due 11:59PM Friday 9/26)
  - Review binary arithmetic and Boolean operations
- Homework 3 due today
  - Finish reading chapter 2 (ARM version on Blackboard site)
- Homework 4 assigned today, due 10/7
- Midterm 10/9
  - Chapter 1, 2 (ARM), Appendices A1-A8, Appendices B1-B2 (ARM)

- Review: Control Flow, Stack, Recursion
  - Basic function calls
  - Intro to Recursion and the Stack
- More control flow
- Macros, pseudoinstructions, assembler directives

Review: The Stack

- Last-in, first-out (LIFO) data structure
  - Last data put in comes out first
  - Common analogy: like a quarter / coin holder in your car, the last coin put in comes out first
- Stack pointer (SP) register: points to current address of stack (i.e., the last thing in)
  - **YOU** must initialize it! Typically use address 0x100000
  - `mov sp, #0x100000`
- Stack instructions
  - `PUSH {r0}` means:
    - `SUB sp, sp, #4`
    - `STR r0, [sp]`
  - `POP {r0}` means:
    - `LDR r0, [sp]`
    - `ADD sp, sp, #4`
  - Can use lists of registers, e.g., `PUSH {r0,r1}` is:

```
SUB sp, sp, #8
STR r0, [sp]
STR r1, [sp,#4]
```

Review: Summary of Caller and Callee Steps

- Caller steps:
  - Step 1: Put arguments in the registers r0, r1, r2, r3.
  - Step 2: Branch to the function, using the bl instruction.
  - Step 3: After the function has returned, recover the return value (if any), and use it.
- Callee (called function) steps:
  - Step 1 (preamble): Allocate memory on the stack, and save register rl, and other registers that the function modifies, to the stack.
  - Step 2: Do the main body of the function.
  - Step 3 (wrap-up):
    - Store the return value (if any) on r0, second return value (if any) on r1.
    - Restore, from the stack, the original values of all registers that the function modified, as well as the value of register lr.
    - Deallocate memory on the stack (increment sp).
    - Branch to the return address using instruction bx.

Review: Basic Function Call Example

```
int ex(int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

```
r0 = g, r1 = h, r2 = I, r3 = j, r4 = f
```

Review: Basic Function Call Example Assembly

```
ex:                  ; label for function name
SUB sp, sp, #12  ; adjust stack to make room for 3 items
STR r6, [sp,#8]  ; save register r6 for use afterwards
STR r5, [sp,#4]  ; save register r5 for use afterwards
STR r4, [sp,#0]  ; save register r4 for use afterwards

ADD r5,r0,r1     ; register r5 contains g + h
ADD r6,r2,r3     ; register r6 contains i + j
SUB r4,r5,r6     ; f gets r5 – r6, ie: (g + h) – (i + j)
MOV r0,r4        ; returns f (r0 = r4)

LDR r4, [sp,#0] ; restore register r4 for caller
LDR r5, [sp,#4] ; restore register r5 for caller
LDR r6, [sp,#8] ; restore register r6 for caller
ADD sp,sp,#12    ; adjust stack to delete 3 items
MOV pc, lr       ; jump back to calling routine
```

Review: Basic Function Call Example Stack

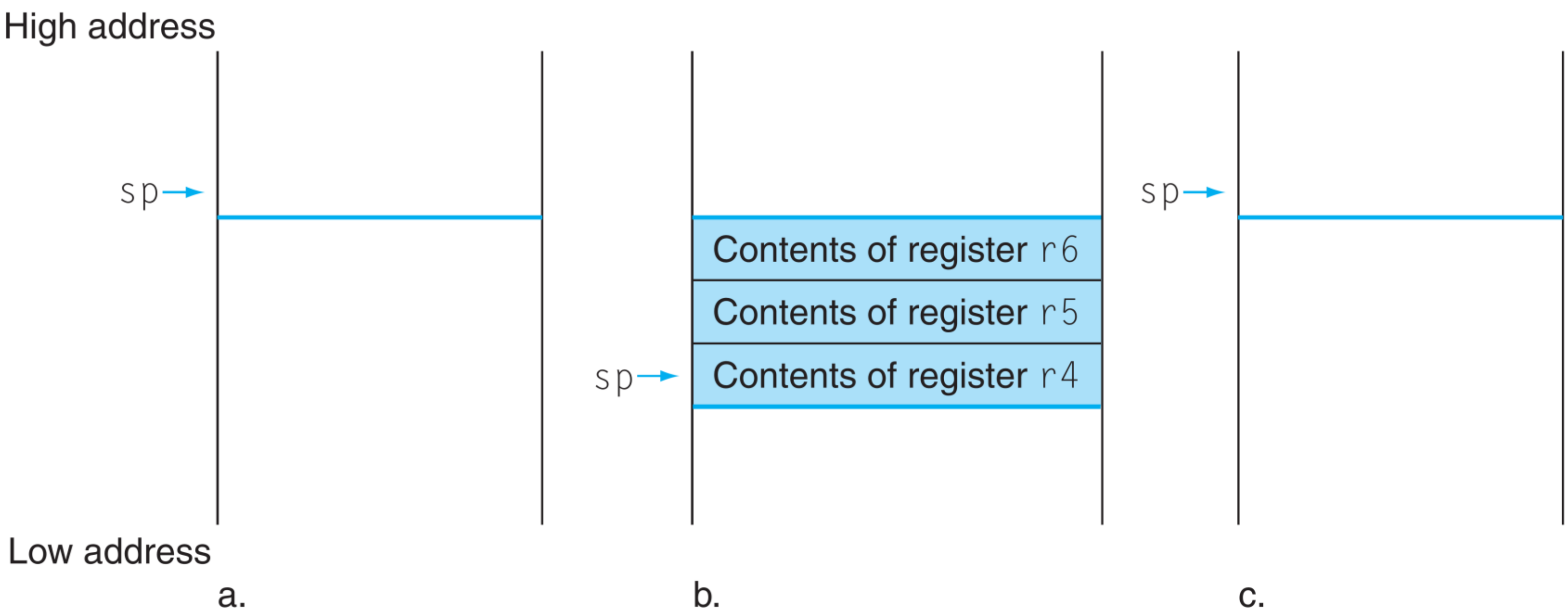

FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Review: Basic Function Call Example Assembly (Push/Pop)

```
ex:                 ; label for function name
PUSH {r4,r5,r6}  ; save r4, r5, r6, decrement sp by 12

ADD r5,r0,r1    ; register r5 contains g + h
ADD r6,r2,r3    ; register r6 contains i + j
SUB r4,r5,r6    ; f gets r5 – r6, ie: (g + h) – (i + j)
MOV r0,r4       ; returns f (r0 = r4)

POP {r4,r5,r6} ; restore r4, r5, r6, increment sp by 12
MOV pc, lr      ; jump back to calling routine
```

Review: State Preservation Across Procedure Calls

Preserved	Not preserved
Variable registers: `r4-r11`	Argument registers: `r0-r3`
Stack pointer register: `sp`	Intra-procedure-call scatch register: `r12`
Link register: `lr`	Stack below the stack pointer
Stack above the stack pointer	

Review: ARM Assembly for Recursive Multiply

```
.globl _start
_start:       mov    sp, #0x11000   @ set up stack
              mov    r0, #5         @ A = 5
              mov    r1, #3         @ B = 3
              mov    r7, #0         @ set up result before call
              bl     rmul           @ first recursive call
              mov    r0,r7          @ put result in r0

iloop: b      iloop                 @ infinite loop ("termination")

rmul:         push   {lr}           @ save link register on stack
              add    r7,r7,r0       @ r7 += r0
              sub    r1, r1, #1     @ r1 -= 1
              cmp    r1, #0         @ r1 == 0?
              beq    rmul_exit      @ if r1 == 0, quit
              bl     rmul           @ else, recursive call
rmul_exit:    pop    {lr}           @ restore link register
              b      lr             @ branch to calling location
```

Recursive Factorial

```
int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * factorial(n - 1);
    }
}
```

Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?

- How do we write function factorial in assembly?

Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?

```
int fact(int n)
{
  if (n== 0) return 1;
  return n * fact(n - 1);
}
```

- How do we write function factorial in assembly?

Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?

```
int fact(int n)
{
  if (n== 0) return 1;
  return n * fact(n - 1);
}
```

- How do we write function factorial in assembly?

```
@ factorial main body
mov r4, r0
cmp r4, #0
moveq r0, #1
beq fact_exit

sub r0, r4, #1
bl fact
mov r5, r0
mul r0, r5, r4
```

Recursive Function Example: Factorial


```
    @ factorial preamble
fact:     ???

    @ factorial body
    mov r4, r0
    cmp r4, #0
    moveq r0, #1
    beq fact_exit

    sub r0, r4, #1
    bl fact
    mov r5, r0
    mul r0, r5, r4
```

```
 @ factorial wrap-up
fact_exit:          ???
```

Recursive Function Example: Factorial


```
    @ factorial preamble
fact: sub sp, sp, #12
    str lr, [sp, #0]
    str r4, [sp, #4]
    str r5, [sp, #8]

    @ factorial body
    mov r4, r0
    cmp r4, #0
    moveq r0, #1
    beq fact_exit

    sub r0, r4, #1
    bl fact
    mov r5, r0
    mul r0, r5, r4
```

```
@ factorial wrap-up
fact_exit:
    ldr lr, [sp, #0]
    ldr r4, [sp, #4]
    ldr r5, [sp, #8]
    add sp, sp, #12
    bx lr
```

Recursive Function Example: Factorial


```
    @ factorial preamble
fact: push {r4,r5,lr}

    @ factorial body
    mov r4, r0
    cmp r4, #0
    moveq r0, #1
    beq fact_exit

    sub r0, r4, #1
    bl fact
    mov r5, r0
    mul r0, r5, r4
```

```
@ factorial wrap-up
fact_exit:
    pop {r4,r5,lr}
    bx lr
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r

r0             0x5        5
r1             0x183      387
r2             0x100      256
r3             0x0        0
r4             0x0        0
r5             0x0        0
r6             0x0        0
r7             0x0        0
r8             0x0        0
r9             0x0        0
r10            0x0        0
r11            0x0        0
r12            0x0        0
sp             0xfff4     0xfff4
lr             0x1000c    65548
pc             0x10014    0x10014 <fact+4>
cpsr           0x600001d3      1610613203
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r

r0             0x4       4
r1             0x183     387
r2             0x100     256
r3             0x0       0
r4             0x5       5
r5             0x0       0
r6             0x0       0
r7             0x0       0
r8             0x0       0
r9             0x0       0
r10            0x0       0
r11            0x0       0
r12            0x0       0
sp             0xffe8    0xffe8
lr             0x1002c   65580
pc             0x10014   0x10014 <fact+4>
cpsr           0x200001d3      536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x3        3
r1             0x183      387
r2             0x100      256
r3             0x0        0
r4             0x4        4
r5             0x0        0
r6             0x0        0
r7             0x0        0
r8             0x0  0
r9             0x0  0
r10            0x0  0
r11            0x0  0
r12            0x0  0
sp             0xffdc      0xffdc
lr             0x1002c     65580
pc             0x10014     0x10014 <fact+4>
cpsr           0x200001d3 536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r
r0            0x2        2
r1            0x183      387
r2            0x100      256
r3            0x0        0
r4            0x3        3
r5            0x0        0
r6            0x0        0
r7            0x0        0
r8            0x0  0
r9            0x0  0
r10           0x0  0
r11           0x0  0
r12           0x0  0
sp            0xffd0     0xffd0
lr            0x1002c    65580
pc            0x10014    0x10014 <fact+4>
cpsr          0x200001d3 536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0
(gdb) i r
r0             0x1       1
r1             0x183     387
r2             0x100     256
r3             0x0       0
r4             0x2       2
r5             0x0       0
r6             0x0       0
r7             0x0       0
r8             0x0  0
r9             0x0  0
r10            0x0  0
r11            0x0  0
r12            0x0  0
sp             0xffc4      0xffc4
lr             0x1002c     65580
pc             0x10014     0x10014 <fact+4>
cpsr           0x200001d3 536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r

r0             0x0       0
r1             0x183     387
r2             0x100     256
r3             0x0       0
r4             0x1       1
r5             0x0       0
r6             0x0       0
r7             0x0       0
r8             0x0  0
r9             0x0  0
r10            0x0  0
r11            0x0  0
r12            0x0  0
sp             0xffb8       0xffb8
lr             0x1002c      65580
pc             0x10014      0x10014 <fact+4>
cpsr           0x200001d3 536871379
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Breakpoint 2, fact () at
example2.s:12, mov r4, r0

(gdb) i r
r0             0x78      120
r1             0x183     387
r2             0x100     256
r3             0x0       0
r4             0x0       0
r5             0x0       0
r6             0x0       0
r7             0x0       0
r8             0x0  0
r9             0x0  0
r10            0x0  0
r11            0x0  0
r12            0x0  0
sp             0x10000     0x10000 <_start>
lr             0x1000c     65548
pc             0x1000c     0x1000c <iloop>
cpsr           0x600001d3 1610613203
```

Recursive Factorial Example for n = 5: Compute 5! Using fact(5)

```
Stack after final return:

0xff90:    0     0     0     0
0xffa0:    0     0     0     0
0xffb0:    0     0     1     0
0xffc0:    65580 2     0     65580
0xffd0:    3     0     65580 4
0xffe0:    0     65580 5     0
0xfff0:    65580 0     0     65548
0x10000
```

Exercise: Convert Iterative Factorial to Assembly

```
int factorial(int n) {
    int f = 1;
    while (n > 0) {
        f *= n;
        n--;
    }
    return f;
}
```

Array Example

```
.globl _start
_start: mov        r1,#0 @ r1 := 0
        ldr    r0,=arrayPtr        @ r0 := arrayPtr
        ldr    r3,=arrayEnd        @ r3 := arrayEnd
        ldrb   r4,[r3,#0]   @ r4 := MEM[R3 + 0]
loop:   ldrb   r2,[r0,#0]   @ r3 := MEM[r0]
        cmp    r2,r4        @ r0 == 0xFF ?
        beq    done         @ branch if done
        add    r1,r1,r2     @ r1 := r1 + r2
        add    r0,r0,#1     @ r0 := r0 + #1
        b      loop         @ pc = loop (address)
done:   strb   r1,[r2]             @ MEM[r2] := r1
iloop:         b      iloop        @ infinite loop
```

```
arrayPtr:
        .byte 2
        .byte 3
        .byte 5
        .byte 7
        .byte 11
        .byte 13
        .byte 17
        .byte 19
        .byte 23
        .byte 29
        .byte 31
        .byte 37
        .byte 41
        .byte 43
        .byte 47
arrayEnd:
        .byte 0xFF
```

Macros

- Another assembler directive
  - Like .byte, .word, .asciz, that we've seen a little of before
- Way to refer to commonly used or repeated code
- Similar to an assembly procedure or function, ***but expanded (evaluated) at assembly time***, not run time
- Similar to #define in C, which is replaced by compiler at compile time
- Macro call: use of macro as an instruction
- Macro expansion: replacement of macro body by the corresponding instructions

Macros vs. Procedures

Item	Macro call	Procedure call
When is the call made?	During assembly	During program execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	One

Macro Example

```
.globl _start
_start:   .macro addVals adA, adB
          ldrb     r2,[\adA]        @ r2 := MEM[adA]
          ldrb     r3,[\adB]        @ r3 := MEM[adB]
          sub      r5,r2,r3         @ r5 := r2 - r3 = A - B
          strb     r5,[\adA]        @ MEM[adA] = r5
          ldrb     r2,[\adA]        @ r2 := MEM[adA]
          add      \adA,\adA,#1     @ r0 := r0 + 1
          add      \adB,\adB,#1     @ r1 := r1 + 1
          .endm                     @ end macro definition
init:     ldr      r0,=A            @ r0 := A (address)
          ldr      r1,=B            @ r1 := B (address)
          ldr      r4,=A_end        @ r4 := A_end (address)
          addVals  r0,r1            @ call macro
done:     b        done             @ infinite loop
```

```
A:      .byte 9, 8, 7, 6
A_end:      .byte 0
B:      .byte 1, 1, 1, 1
B_end:      .byte 0
```

Summary

- More Control flow
  - Stack
  - Procedures
  - Basic recursion
- Macros

Questions?

?