UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 10: Stack and Recursion

Taylor Johnson

Announcements and Outline

- Quiz 3 on Blackboard (due 11:59PM Friday 9/26)
  - Review binary arithmetic and Boolean operations
- Homework 3 assigned and due Thursday
  - Finish reading chapter 2 (ARM version on Blackboard site)

- Review: Control flow
  - Branches
  - Conditional execution
- More control flow
  - Basic function calls
  - Intro to Recursion and the Stack

Review: Branch and Branch with Link

- Branch Conditional and Unconditional

```
b{cond} label
beq label      @ conditional
b label        @ unconditional
```

  - Updates:
    - PC = address of label
- Branch with Link (function/procedure calls)
  - Link register: keeps return address (for procedure calls)

```
bl{cond} label
```

  - Updates:
    - LR = PC (before call, so we can return)
    - PC = address of label

Review: Conditional Execution

- Current Program Status Register (CPSR)
  - Keeps track of arithmetic / logic (ALU) status
    - Example: last result was negative, zero, positive, had a carry, etc.
    - N (negative) / Z (zero) / C (carry) / V (overflow) bits
- Allows us to make conditional branches, etc. for conditional control flow changes (ifs, finite loops, etc.)
- Examples:

```
cmp r0, #0    @ compare r0 and #0
beq label     @ branch if r0 == 0
adds r0, r0, #1
bgt label     @ branch if r0 positive
```

Review: ALU Status Bits

- N (negative) bit
  - Set to 1 when the result of the operation is negative, cleared to 0 otherwise
- Z (zero) bit
  - Set to 1 when the result of the operation is zero, cleared to 0 otherwise
- C (carry) bit
  - if the result of an addition is greater than or equal to 2^{32}
  - if the result of a subtraction is positive or zero
  - as the result of an inline barrel shifter operation in a move or logical instruction
- V (overflow) bit
  - Overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{31}, or less than -2^{31}

Review: Condition Code Suffixes

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.

Review: Control Flow Translation Example

- Suppose our ISA does not have a multiply instruction
- How can we perform multiplication?
  - Create equivalent sequence of computations yielding the same result
  - Use addition, branch, comparisons, etc.
  - $A * B = \sum_{i=1}^{B} A = \underbrace{A + A + \cdots + A}_{B\ times}$
  - Example: $5 * 9 = \sum_{i=1}^{9} 5 = \underbrace{5 + 5 + \cdots + 5}_{9\ times} = 45$
- Generalizing: this is the basis of all our modern computations
- CPU does not have “visit website, buy shoes“ instruction

Review: Procedures: Iterative Multiply

- How do we write a loop?
- How does flow of control change with function (procedure) calls?

```
int multiply(int A, int B) {
    int product = 0;
    while (B > 0) {
        product += A;
        B--;
    }
    return product;
}
```

Review: ARM Assembly for Iterative Multiply

```
.globl _start
_start:     mov   r0, #5       @ A = 5
            mov   r1, #3       @ B = 3
            bl    imul         @ call iterative multiply procedure

iloop:      b     iloop        @ infinite loop (for “termination”)

imul:       mov   r2,#0        @ initialize result to 0
imul_loop:  cmp   r0,#0        @ r0 == 0?
            beq   imul_done    @ if r0 == 0, set PC = imul_done
            add   r2,r2,r1     @ r2 += r1
            sub   r0,r0,#1     @ r0 -= 1
            b     imul_loop    @ branch to imul_loop
imul_done:  mov   r0,r2        @ r0 = r2
            bx    lr           @ set PC = LR
```

Procedures

- How does a function get its input arguments and return its result?
- What if one function calls another?
- How can we accomplish this allowing for recursion (functions calling themselves)?
  - The stack

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]
```

Caller Steps

- **Step 1: Put arguments in the right place.**
- Specific machines use specific conventions.
  - "R0-R3 hold parameters to the procedure being called".
- So:
  - Argument 1 (if any) goes to r0.
  - Argument 2 (if any) goes to r1.
  - Argument 3 (if any) goes to r2.
  - Argument 4 (if any) goes to r3.
- If there are more arguments, they have to be placed in memory. We will worry about this case only if we encounter it.

Caller Steps

- **Step 2: branch to the first instruction of the function.**
  - Here, we typically use the bl instruction, not the b instruction.
  - The bl instruction, before branching, saves to register lr (the link register, aka r14) the return address.
  - The **return address** is the address of the instruction that should be executed when the function is done.
- **Step 3: after the function has returned, recover the return value, and use it.**
  - We will follow the convention that the return value goes to r0.
  - If there is a second return value, it goes to r1.

Called (callee) Function Steps

- Step 1: Do the preamble:
  - Allocate memory on the stack (more details in a bit).
  - Save to memory the return address. Why?
  - Save to memory all registers (except possibly for r0) that the function modifies. Why?
- Step 2: Do the main body of the function.
  - Assume arguments are in r0, r1, r2, r3.
  - This is where the actual work is done.
- Step 3: Do the wrap-up:
  - Store the return value (if any) on r0, and second return value (if any) on r1.
  - Retrieve from memory the return address. Why?
  - Retrieve from memory, and restore to registers, the original values of all registers that the function modified (except possibly for r0). Why?
  - Deallocate memory on the stack.
  - Branch to the return address.

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.

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
```

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
```

Basic Function Call Example Stack

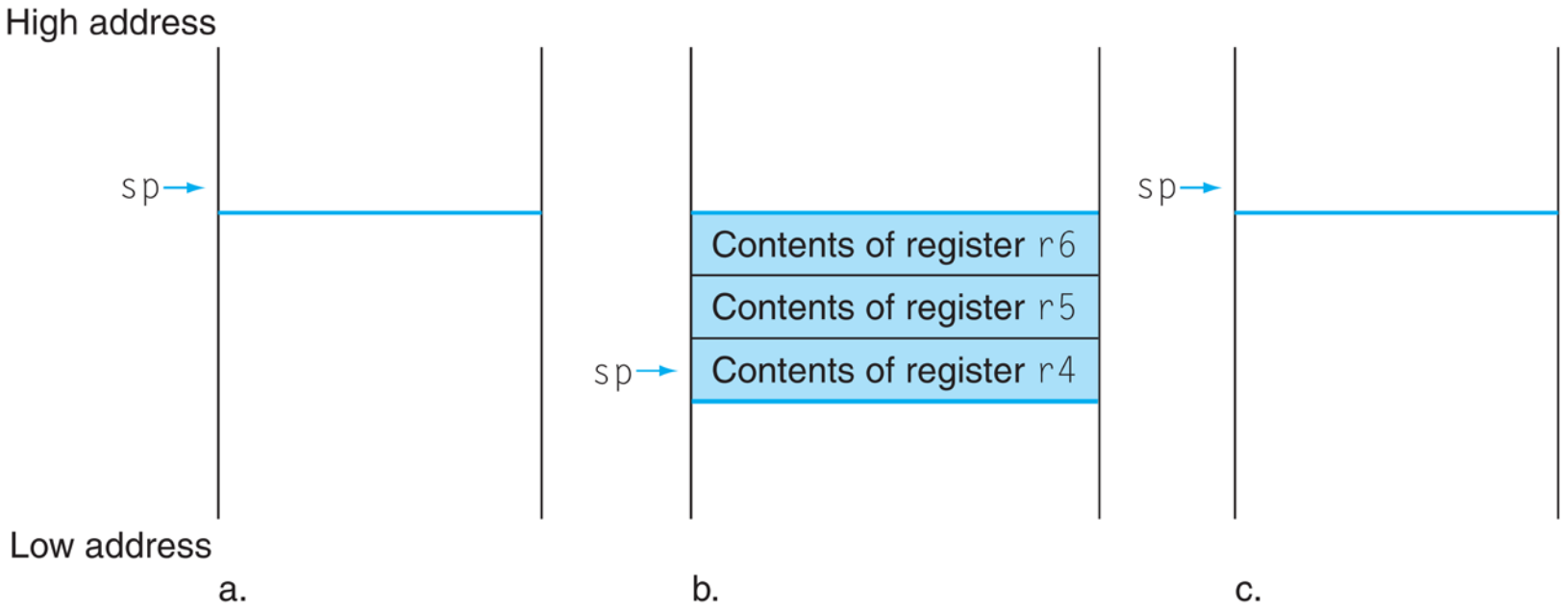


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

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
```

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	

Procedures: Recursive Multiply

- How does flow of control change with ***recursive*** procedure calls?

- Given what we know about the cycle loop (all computations use registers, etc.), how might this work?
  - Hardware support for function calls (and recursion)
  - Stack pointer (SP) register: pointer to stack in memory for passing function arguments and return addresses
  - Link register (LR): saves return address

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

UNIVERSITY OF TEXAS ARLINGTON

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

- How do we write function factorial in assembly?

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

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

Recursive Function Example: Factorial


```
@ factorial preamble
   ???

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

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

```
@ factorial wrap-up
???
```

Recursive Function Example: Factorial


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

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

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

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

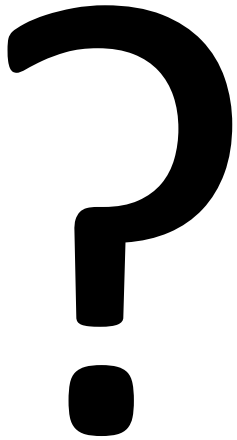
Exercise: Convert Iterative Factorial to Assembly

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

Summary

- More Control flow
  - The stack
  - Procedures
  - Basic recursion

Questions?



Macros

- Way to refer to commonly used or repeated code
- Similar to a procedure or function, but expanded at assembly time, not run time
- Macro call: use of macro as an opcode
- 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