UNIVERSITY OF TEXAS ARLINGTON

# Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 9: Control Flow and Procedure Introduction

Taylor Johnson

# Announcements and Outline

- Quiz 3 on Blackboard site soon (due 11:59PM next Wednesday 9/24)
  - Review binary arithmetic and Boolean operations
- Homework 3 assigned
  - Finish reading chapter 2 (ARM version on Blackboard site)

- Review
  - Addressing modes
- Control flow
  - Branches
  - Conditional execution
  - Basic function calls
  - Intro to Recursion and the Stack

# Review: Addressing Modes

- Many instructions have operands (inputs), so where do they come from?
  - Addressing mode specifies this
- Addressing modes
  - Immediate addressing
  - Direct addressing
  - Register addressing
  - Register indirect addressing
  - Indexed addressing
  - Based-index addressing
  - Stack addressing
  - …
- Only some of these are available in typical modern ISAs
  - ARM has many addressing modes
  - Don't have to use them all, but good to be aware of them

# Review: Immediate/Literal Addressing

1. Immediate: ADD r2, r0, #5

| cond | f | opcode | rn | rd | Immediate |
|---|---|---|---|---|---|

- Operand comes from the instruction
- Example: 32-bit instruction to move 4 into R1
  - Result is R1 := 4

MOV R1 #4

*destination register field* (R1)

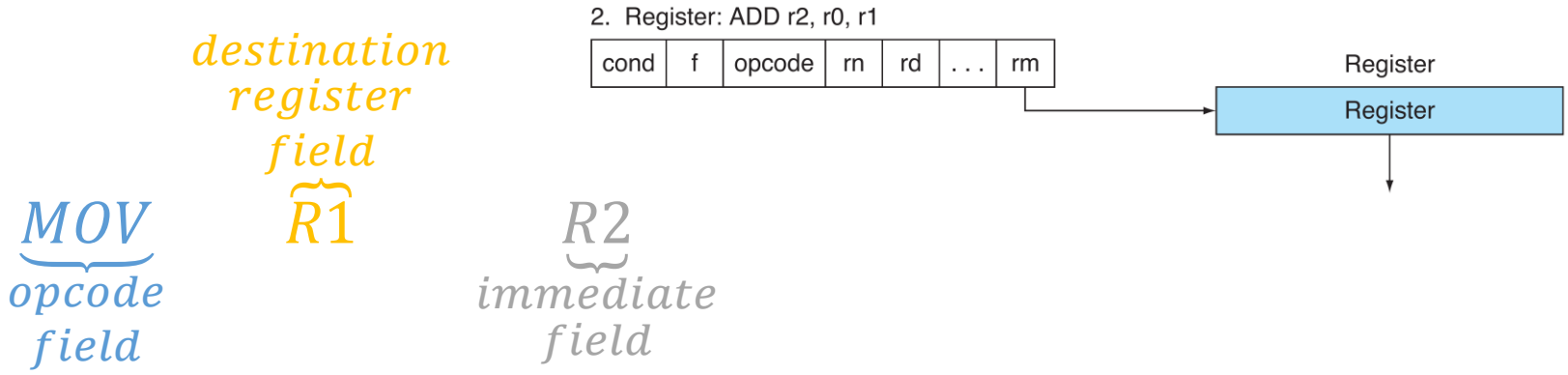
*MOV* — *opcode field*

#4 — *immediate field*

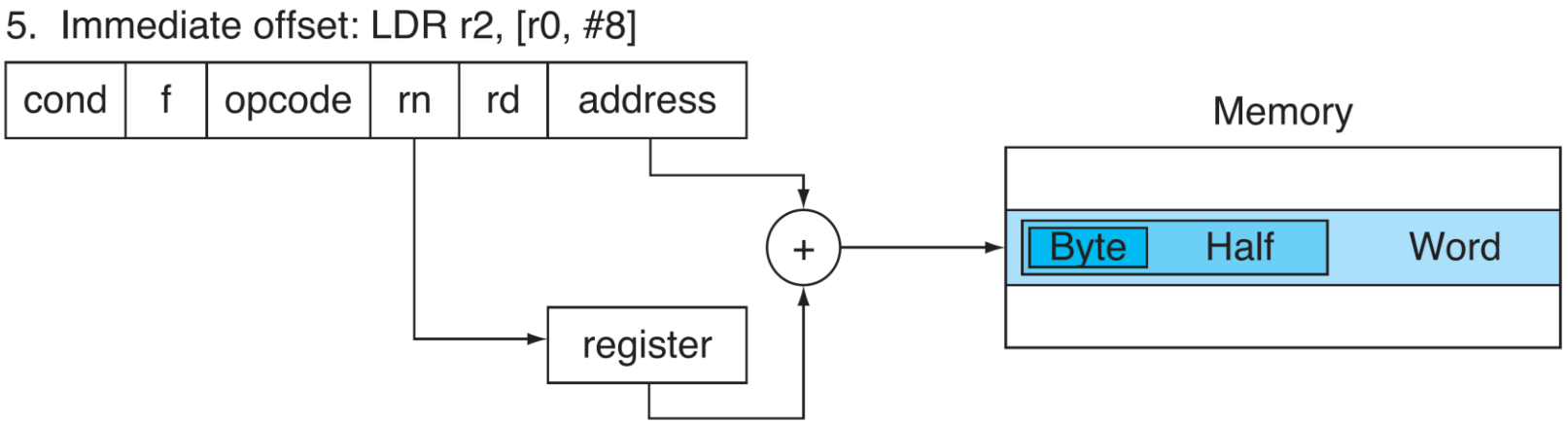
- Useful for specifying small integer constants (avoids extra memory access)
- Can only specify small constants (limited by size of immediate field)
  - ARM: typically 8-12-bits

# Review: Register/Register-Direct Addressing

- Operand(s) come(s) from register(s)
  - Seen this many times already: ADD R0 R1 R2 does R0 := R1 + R2
  - Also: MOV R1 R2: the destination operand is specified by its register address (Result is R1 := R2)



# Review: Indirect with Immediate Offset




- Uses a register value and an immediate offset
- Example: LDR r2, [r0, #8]
  - Updates r2 = MEM[r0 + #8]

# Review: Addressing Modes

- Specify where to get operands (inputs) for instructions
- Why have all these different modes?
- What are some tradeoffs?
- What could machine language instruction formats look like for each of the addressing modes?

# Review: Assembly Language

- Assembly: source language is symbolic representation for numerical machine language
  - Assembler (not compiler) used in this case to perform translation
- Pure assembly: one-to-one correspondence between assembly language and machine language
- Practical assembly: extra commands that the assembler takes care of
  - Examples: computing addresses from labels, filling consecutive addreses with zeros, placing strings in memory, etc.

# Review: Assembly Language Format

| Label | Opcode | Operands | Comments |
|---|---|---|---|
| iloop: | add | r1,r1,#1 | @ r1 := r1 + 1 |
| | b | iloop | @ pc := iloop |
| val: | .byte | 0x9F | @ put 0x96 at address val |
| s: | .asciz | "hello!" | @ put "hello!" at sequential addresses starting at address s |

# Review: Assembly Instructions vs. Directives

- Instructions
  - Machine language equivalents
- Directives / Pseudoinstructions
  - Special commands given to the assembler that are converted to equivalent machine language during assembly process
  - Used for placing data in memory, etc.

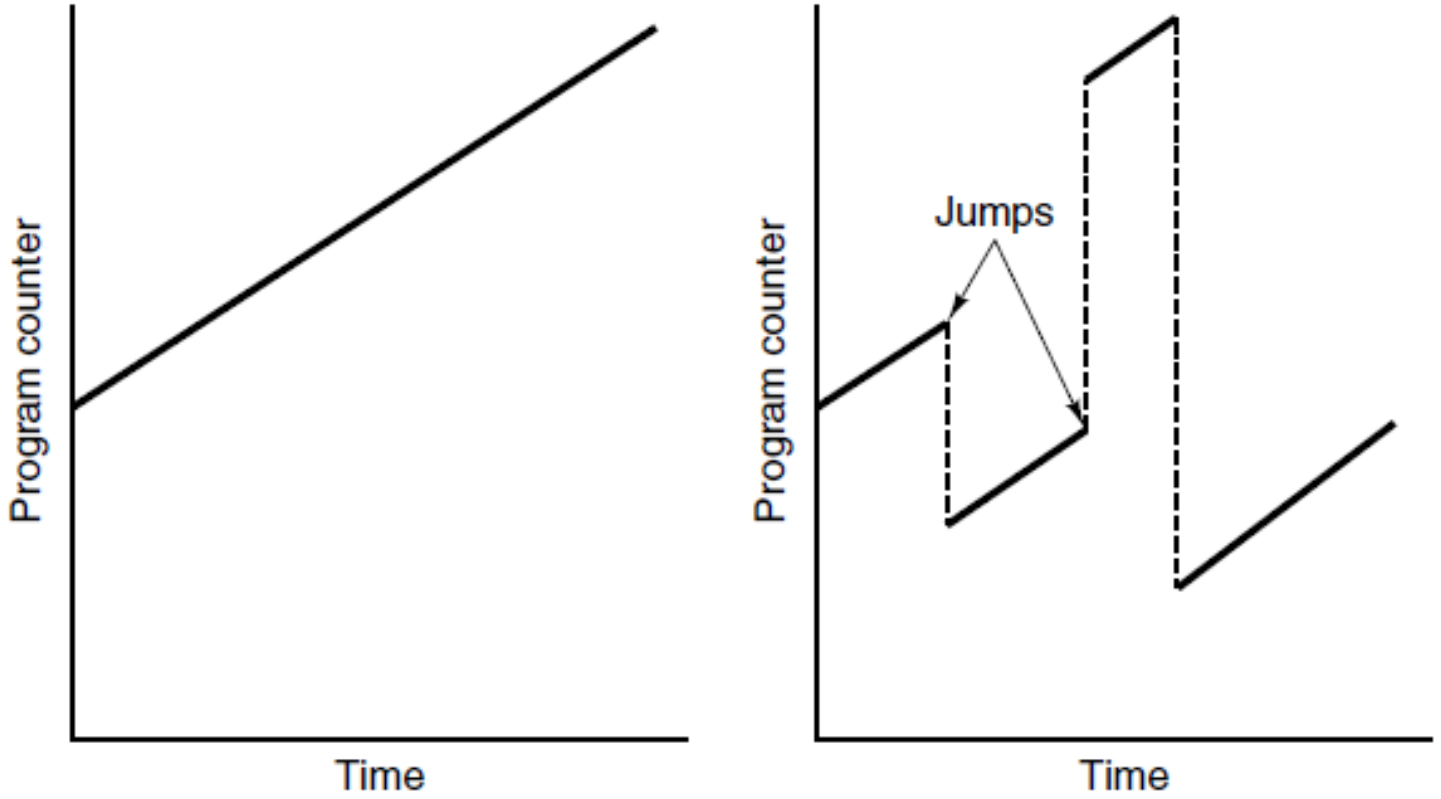
# Assembly Process

- Insuffiency of one pass
  - Suppose we have labels (symbols to addresses)
  - How do we calculate the addresses of labels later in the program?
  - Example:
    - `ADDR: 0x1000  init:b done`
    - `…  Other instructions and data`
    - `ADDR: 0x????  done:b init`

- Two-Pass Assemblers
  - First Pass: iterate over instructions, build a symbol table, opcode table, expand macros, etc.
  - Second Pass: iterate over instructions, printing equivalent machine language, plugging in values for labels using symbol table

# Flow of Control

- Left: no branches; Right: branches / jumps



# Loop Control with Branches / Gotos

Left: Test-at-the-end loop
Right: Test-at-the-beginning loop

```
        i = 1;
L1:     first-statement;
        .
        .
        .
        last-statement;
        i = i + 1;
        if (i < n) goto L1;
```

```
        i = 1;
L1:     if (i > n) goto L2;
        first-statement;
        .
        .
        .
        last-statement
        i = i + 1;
        goto L1;
L2:
```

# 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

# 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, #1
bgt label     @ branch if r0 positive
```

# 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 $-2^{31}$

# 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. |

# 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

# 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;
}
```

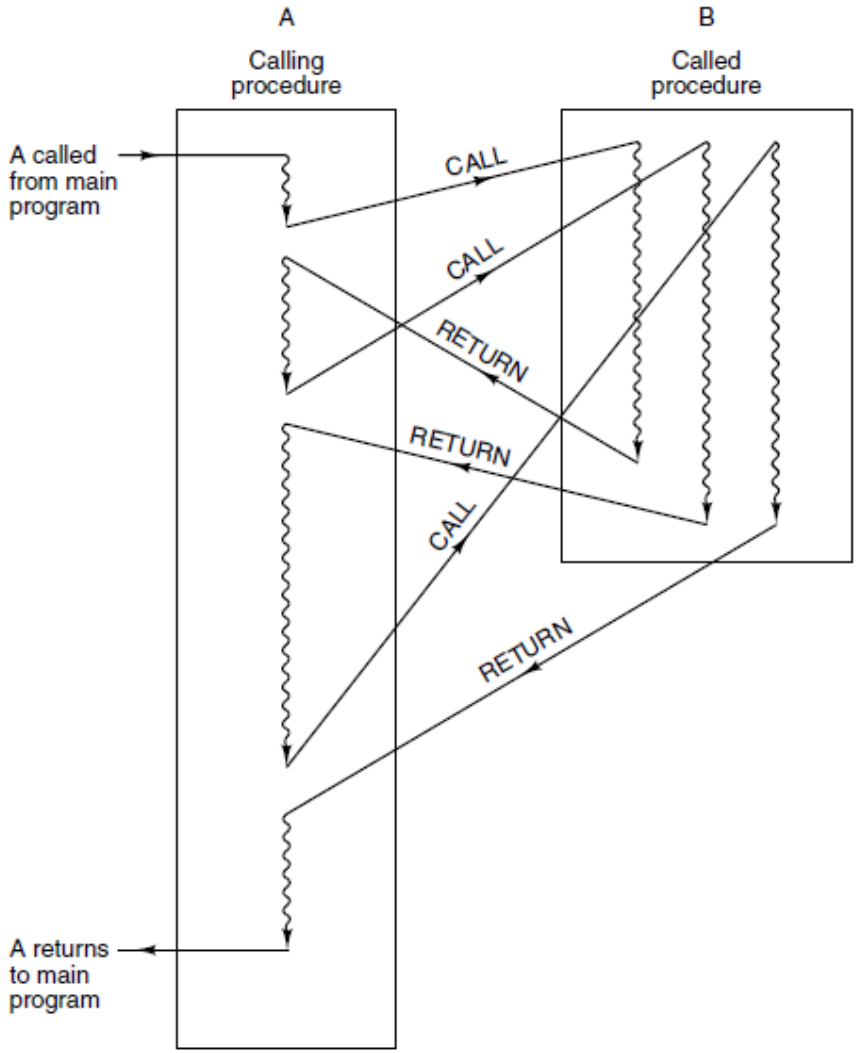
# ARM Assembly for Iterative Multiply

```
.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

# Control Flow with Procedure Calls



# Procedures

- ARM Convention
  - R0: stores the return value
  - R0-R4: may be modified by subroutine
- Callee-save: function A that calls function B must save registers to the stack
- Caller-save: called function must restore state to previous values before it was called

# Making a Function

- Why are functions useful in assembly?
- For the same reasons they are useful in any programming language:
  - Modularity, making code easy to design, write, read, debug
  - Reusability

# Making a Function

- Functions are easy to define and call in languages like C and Java
- In assembly, calling a function requires several steps
- This reflects that the CPU can do only a limited amount of work in a single step
- Note that, to correctly do a function call, both the caller (program/function making the function call) and the called function must do the right steps

# 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.

# 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

UNIVERSITY OF TEXAS ARLINGTON

- 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
```

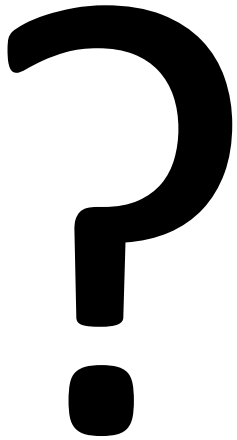
# Exercise: Convert Iterative Factorial to Assembly

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

# Summary

- Control flow
  - Branches
  - Conditional execution
  - Procedures
  - Basic recursion and the stack

# 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 |