

# Lecture 8: Control flow

David Hovemeyer

September 18, 2026

601.229 Computer Systems Fundamentals



# Control flow!

- ▶ Control flow:
  - ▶ Decisions (if/then, switch)
  - ▶ Loops (for, while)
- ▶ Today's example programs are linked as `control.zip` on the course website

# Decisions

# Unconditional jump

- ▶ Sometimes we want to jump *unconditionally*
  - ▶ Continue a loop
  - ▶ Complete a decision construct
- ▶ This is the `jmp` instruction
- ▶ Because unconditional, not directly useful for implementing decisions and loops
  - ▶ But, definitely useful and necessary

# Condition codes

- ▶ *Condition codes* are status bits updated by most ALU instructions to indicate the outcome of the instruction
- ▶ Most important condition code bits:
  - ▶ CF: carry flag (unsigned operation overflowed)
  - ▶ ZF: zero flag (result was 0)
  - ▶ SF: sign flag (result was negative)
  - ▶ OF: overflow flag (signed operation overflowed)
- ▶ Condition code bits can be used to make decisions
  - ▶ If/else logic, loops

# Comparing values

- ▶ `cmp` instruction: essentially the same as `sub`, except that it doesn't modify the "result" operand
  - ▶ Useful for comparing integer values
- ▶ Annoying quirk: AT&T syntax puts the operands in the opposite of the order you might expect
  - ▶ E.g., `cmpl %eax, %ebx` computes  $\%ebx - \%eax$  and sets condition codes appropriately

# Testing bits

- ▶ test instruction: essentially the same as and, but doesn't modify the "result" operand
- ▶ Example:

```
testl $0x80, %eax
```

Sets ZF (zero flag) IFF bit 7 of %eax is 0

# set instructions

- ▶ The `setX` instructions set a single byte to 0 or 1 depending on whether a condition code bit is set
  - ▶ Useful to get the result of a comparison as a data value
- ▶ Example:

```
setz %al
```

Set `%al` (low byte of `%rax`) to 1 IFF ZF (zero flag) is set

# Conditional jump

Most often, we want to use the result of a comparison in order to influence a *conditional jump* instruction (used for implementing if/else logic and eventually-terminating loops)

Examples ( $\wedge$  means XOR,  $\sim$  means NOT,  $\&$  means AND,  $|$  means OR):

| Instruction | Condition for jump                     | Meaning                           |
|-------------|----------------------------------------|-----------------------------------|
| je, jz      | ZF                                     | jump if equal                     |
| j1          | SF $\wedge$ OF                         | jump if less                      |
| jle         | (SF $\wedge$ OF) $ $ ZF                | jump if less than or equal        |
| jg          | $\sim$ (SF $\wedge$ OF) $\&$ $\sim$ ZF | jump if greater                   |
| jge         | $\sim$ (SF $\wedge$ OF)                | jump if greater than or equal     |
| ja          | $\sim$ CF $\&$ $\sim$ ZF               | jump if above (unsigned)          |
| jae         | $\sim$ CF                              | jump if above or equal (unsigned) |
| jb          | CF                                     | jump if below (unsigned)          |
| jbe         | CF $ $ ZF                              | jump if below or equal (unsigned) |

# Implementing decisions (if, if/else)

Basic approach for implementing an if statement (C and assembly):

```
/* C code */  
if (compare op1 and op2) {  
    conditionally-executed code  
}  
rest of code...
```

```
/* assembly code */  
    cmp op2, op1  
    jX .Lout  
    conditionally-executed code  
  
.Lout:  
    rest of code...
```

Idea is that *jX* jumps to *.Lout* if the condition evaluates as *false*

# Implementing decisions (if, if/else)

Basic approach for implementing an if/else statement (C and assembly):

```
/* C code */  
if (compare op1 and op2) {  
    code if true  
} else {  
    code if false  
}  
rest of code...
```

```
/* assembly code */  
    cmp op2, op1  
    jX .LelsePart  
    code if true  
    jmp .Lout  
.LelsePart:  
    code if false  
.Lout:  
    rest of code...
```

*jX* jumps to *.LelsePart* if the condition evaluates as *false*

# Example: can you vote?

```
/* vote.S */
.section .rodata
sAgePrompt: .string "What is your age? "
sInputFmt: .string "%d"
sCanVoteMsg: .string "You can vote, yay!\n"
sCannotVoteMsg:
.string "You're not old enough to vote yet\n"

.section .bss
age: .space 4

.section .text
.globl main
main:
    subq $8, %rsp
```

```
    movl $0, %eax
    movq $sAgePrompt, %rdi
    call printf
```

```
    movl $0, %eax
    movq $sInputFmt, %rdi
    movq $age, %rsi
    call scanf
```

```
    cmpl $18, age
    jl .LtooYoungToVote
    movq $sCanVoteMsg, %rdi
    jmp .LprintMsg
```

```
.LtooYoungToVote:
    movq $sCannotVoteMsg, %rdi
```

```
.LprintMsg:
    movl $0, %eax
    call printf
```

```
    addq $8, %rsp
    ret
```

# Running the program

```
$ gcc -c -no-pie -o vote.o vote.S
$ gcc -no-pie -o vote vote.o
$ ./vote
What is your age? 17
You're not old enough to vote yet
$ ./vote
What is your age? 18
You can vote, yay!
```

# Clicker quiz!

Clicker quiz omitted from public slides

# Implementing decisions (switch)

switch statement: multiway branch based on an integer value

Example:

```
int month;
scanf("%d", &month);
switch (month) {
case 1: case 3: case 5: case 7:
case 8: case 10: case 12:
    printf("31 days\n"); break;
case 4: case 6: case 9: case 11:
    printf("30 days\n"); break;
case 2:
    printf("28 or 29 days\n"); break;
default:
    printf("not a valid month\n");
}
```

# Switch implementation

One approach: translate into equivalent of `if/else if/...`

This might be the best approach if the range of tested integers is not dense

If the range of tested integers is dense, can use a *jump table*

- ▶ Jump table = array of code addresses
- ▶ Look up entry, jump to that location
- ▶  $O(1)$  time!
- ▶ Full demo program `months.S` in `control.zip`

# Jump tables

Assume that `%esi` contains an integer value input by the user

```
    cmpl $1, %esi
    jl .LDefaultCase
    cmpl $12, %esi
    jg .LDefaultCase
    dec %esi
    jmp *.LJumpTable(,%esi,8)
.L31DaysCase:
    code to handle months 1, 3, 5, etc.
    jmp .LSwitchDone
.L30DaysCase:
    code to handle months 4, 6, 9, etc.
    jmp .LSwitchDone
.LFebCase:
    code to handle month 2
    jmp .LSwitchDone
.LDefaultCase:
    code to handle invalid month values
.LSwitchDone:
```

# Jump tables

Assume that `%esi` contains an integer value input by the user

```
    cmpl $1, %esi
    jl .LDefaultCase
    cmpl $12, %esi
    jg .LDefaultCase
    dec %esi
    jmp *.LJumpTable(,%esi,8)    <-- jump table lookup
.L31DaysCase:
    code to handle months 1, 3, 5, etc.
    jmp .LSwitchDone
.L30DaysCase:
    code to handle months 4, 6, 9, etc.
    jmp .LSwitchDone
.LFebCase:
    code to handle month 2
    jmp .LSwitchDone
.LDefaultCase:
    code to handle invalid month values
.LSwitchDone:
```

# Jump tables

The actual jump table is simply an array of pointers, where the element values are code addresses specified using labels

```
.LJumpTable:  
    .quad .L31DaysCase  
    .quad .LFebCase  
    .quad .L31DaysCase  
    .quad .L30DaysCase  
    .quad .L31DaysCase  
    .quad .L30DaysCase  
    .quad .L31DaysCase  
    .quad .L31DaysCase  
    .quad .L30DaysCase  
    .quad .L31DaysCase  
    .quad .L30DaysCase  
    .quad .L31DaysCase
```

# Loops

# Implementing loops

One way to implement a loop (essentially a `while`):

```
.Ltop:
    cmp value, reg
    jX .Ldone

    loop body
    jmp .Ltop

.Ldone:

    code following loop...
```

Assumes that:

- ▶ *reg* is a loop counter
- ▶ *jX* is a conditional jump which, when taken, terminates loop

# Implementing loops

Slightly more clever approach (also for implementing while):

```
    jmp .LcheckCond
```

```
.Ltop:
```

```
    loop body
```

```
.LcheckCond:
```

```
    cmp value, reg
```

```
    jX .Ltop
```

```
    code following loop...
```

Assumes that:

- ▶ *reg* is a loop counter
- ▶ jX is a conditional jump which, when taken, *continues* loop

This approach eliminates an unconditional jump from the loop body

# Loop example program

Compute  $fib(n)$  where:

$$fib(0) = 0$$

$$fib(1) = 1$$

$$\text{For } n > 1, fib(n) = fib(n - 2) + fib(n - 1)$$

# Loop example program

Note: this program will only work when  $N \geq 1$

```
/* fib.S */

#define N 9

.section .rodata
sResultMsg: .string "fib(%u) = %u\n"

.section .text
.globl main
main:
    subq $8, %rsp

    movl $1, %ecx    /* %ecx is the loop counter */
    movl $0, %r10d   /* %r10d stores fib(n-1) */
    movl $1, %r11d   /* %r11d stores fib(n) */

    jmp .LtestCond

.LoopTop:
    movl %r11d, %r9d
    addl %r10d, %r11d
    movl %r9d, %r10d
    inc %ecx

.LtestCond:
    cmpl $N, %ecx
    jl .LoopTop

    movl $0, %eax
    movq $sResultMsg, %rdi
    movl $N, %esi
    movl %r11d, %edx
    call printf

    addq $8, %rsp
    ret
```

# Loop example program

```
$ gcc -c -no-pie -o fib.o fib.S
$ gcc -no-pie -o fib fib.o
$ ./fib
fib(9) = 34
```

# Clicker quiz!

Clicker quiz omitted from public slides

# Practical assembly programming tips

# Know where to put things

- ▶ The `.section` directive specifies which “section” of the executable program assembled code or data will be placed in
- ▶ Put things in the right place!
- ▶ Code goes in `.text`
- ▶ Read-only data such as string constants go in `.rodata`
- ▶ Uninitialized (zero-filled) variables and buffers go in `.bss`
  - ▶ Use the `.space` directive to indicate how large these are
- ▶ Initialized (non-zero-filled) variables and buffers go in `.data`
  - ▶ There are various directives such as `.byte`, `.2byte`, `.4byte`, etc. to specify initialized data values

# Labels

- ▶ Labels are names representing addresses of code or data in memory
- ▶ For functions and global variables, use appropriate names
  - ▶ Functions and data exported to other modules must be marked with `.globl`
- ▶ For control-flow targets within a function, use *local labels*
  - ▶ These are labels which start with `.L` (dot, followed by upper case L)
  - ▶ The assembler will not add these to the module's symbol table
  - ▶ Using “normal” labels for control flow makes debugging difficult because `gdb` thinks they are functions!

# Using gdb

- ▶ You can debug assembly programs using gdb!
- ▶ “Debugging by adding print statements” is much less practical for assembly programs than programs in a high level language
  - ▶ Which isn't to say it's not possible or (occasionally) useful
- ▶ Being able to use gdb confidently will greatly enhance your ability to develop working assembly language programs

- ▶ Set breakpoints (`break main`, `break myProg.S:123`)
- ▶ `where`: see current call stack
- ▶ If you compiled your code with debugging symbols (i.e., using `-g` flag to `gcc`), `next` and `step` commands work as expected!
- ▶ If code is compiled without debug symbols, it's more difficult:
  - ▶ `disassemble` (or just `disas`): display assembly code of current function
  - ▶ `stepi`: step to next instruction
  - ▶ `nexti`: step to next instruction (stepping over `call` instructions)

## gdb tips (continued)

- ▶ Use \$ prefix to refer to registers (e.g., \$rax, \$edi, etc.)
- ▶ Use print and casts to C data types when inspecting data:
  - ▶ Print 64 bit value %rsp points to: `print *(unsigned long *)$rsp`
  - ▶ Print character string %rdi points to: `print (char *)$rdi`
  - ▶ Print fourth element of array of int elements that %r12 points to:  
`print ((int *)$r12)[3]`
  - ▶ Print contents of %rcx is hexadecimal: `print/x $rcx`