Common programming constructs

Systems Programming

Michael Sonntag
Institute of Networks and Security

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

Decisions: IF constructs

■ Basic model

   ■ Calculate expression

   ■ Convert result into one/several flags

   ■ Jump based on flag to ELSE part

   ■ Perform IF part

   ■ Jump over ELSE part

   ■ Perform ELSE part

■ Example: IF(RAX!=0) {A;} ELSE {B;}

```
        movq $???,%rax    # Calculate expression: Here fixed value
        cmpq $0,%rax      # Compare to 0; set flags (CF, OF, SF, ZF, AF, PF)
        je else_part      # Add current index to result
        ...               # Perform "A" part
        jmp continue      # At the end of the loop?
else_part:                # Part executed if condition is false
        ...               # Perform "B" part
continue:                 # Continue with program
```

IF equals/not equals

- Comparison: CMP instruction

  - Internally "Operand2 – Operand1; Set Flags; Ignore result;"

- cmp %rax,%rbx

  - Temp=RBX-RAX

  - Flags are set like in the SUB instruction

  SUB/CMP set flags "twice": For parameters interpreted as both signed and unsigned integers!

- The flags set and their meanings:

  - CF = Carry flag: Overflow for unsigned integers (1 = Carry)

  - SF = Sign flag: Sign of result for signed integers (1 = Negative)

  - ZF = Zero flag: Result is zero (RAX==RBX) (1 = Zero)

  - OF = Overflow flag: Overflow for signed integers (1 = Overflow)

  - PF = Parity flag: Number of 1 in least significant byte (1 = Even)

  - AF = Auxiliary carry flag: Carry for bit 3; used in BCD arithmetics

Jump instructions

■ There exist LOTS of jump instructions! The most important are:

■ JE/JZ: Jump equal/zero → ZF=1

■ JNE/JNZ: Jump not equal/zero → ZF=0

■ JB: Jump below → CF=1

■ JA: Jump above → CF=0 && ZF=0

Unsigned

■ JL: Jump less → SF=1

■ JG: Jump greater → SF=0 && ZF=0

Signed

■ JA/JB/JL/JG+"E": Same as above, but jump also if equal

■ Example: JLE → Jump if less or equal

■ They exist in a "negated" version too: "j**n**"…

■ Some are technically identical (=same opcode)

■ je = jz; ja = jnbe

JꓝU | INSTITUTE OF NETWORKS AND SECURITY

Jump examples (1): 0 <-> 0 ?

```
movb $0,%al        # Set first value
movb $1,%bl        # Set second value
movb $255,%cl      # Set third value
movb $55,%dl       # Set fourth value

cmpb %al,%dl       # 0==0? --> 0-0 --> ZF=1, CF=0, SF=0 (PF=1, OF=0, AF=0)
je true            # Will jump, as ZF == 1
jne false          # Will NOT jump, as ZF != 0
jb false           # Will NOT jump, as CF != 0
ja false           # Will NOT jump, as ZF != 0
jl false           # Will NOT jump, as SF != 0
jg false           # Will NOT jump, as ZF != 0
```

Jump examples (2): 0 <-> 1 ; 1 <-> 0

```
cmpb %al,%bl        # 0==1? --> 1-0 --> ZF=0, CF=0, SF=0 (PF=0, OF=0, AF=0)
je false            # Will NOT jump, as ZF != 1
jne true            # Will jump, as ZF == 0
jb false            # Will NOT jump, as CF != 1
ja true             # Will jump, as CF == 0 && ZF == 0
jl false            # Will NOT jump, as SF != 1
jg true             # Will jump, as SF = 0 && ZF == 0

cmpb %bl,%al        # 1==0? --> 0-1 --> ZF=0, CF=1, SF=1 (PF=1, OF=0, AF=1)
je false            # Will NOT jump, as ZF != 1
jne true            # Will jump, as ZF == 0
jb true             # Will jump, as CF == 1
ja false            # Will NOT jump, as CF != 0
jl true             # Will jump, as SF == 1
jg false            # Will NOT jump, as SF != 0
```

Jump examples (3): 255 <-> 55

```
cmpb %cl,%dl        # 255/-1==55? Signed (55 - -1)= 56, Unsigned = "-200"
                    # ZF=0, CF=1, SF=0 (PF=0, OF=0, AF=1)
je false            # Will NOT jump, as ZF != 1
jne true            # Will jump, as ZF == 0
jb true             # Will jump, as CF == 1 (55<255)
ja false            # Will NOT jump, as CF != 0
jl false            # Will NOT jump, as SF != 1
jg true             # Will jump, as SF == 0 && ZF == 0 (55>-1)
```

■ Note: 255 is just a binary number (11111111b)

   ■ We can interpret it as an unsigned integer: 255

      ■ jb, ja: Unsigned → Compare 255 and 55

   ■ We can interpret it as a signed integer: -1

      ■ jl, jg: Signed → Compare -1 and 55

■ The CMP instruction does not care; it handles both cases simul-
taneously, as they use different flags (signed → SF, unsigned → CF)

JⴼU INSTITUTE OF NETWORKS AND SECURITY

Different kinds of simple loops

■ As seen before:
   □ Check condition
   □ Exit if no longer matching
   □ Do something
   □ Jump to begin of loop

■ Alternative: Check loop condition at the end
   □ "Repeat" vs. "While" loop

■ For loops:
   □ Use a separate counter as the basis for the condition

■ The "**loop**" instruction…
   □ is followed by a label
   □ is based on the register ECX
   □ will decrement ECX, compare with 0, if not zero jump to label
   □ does not "count-up" or "end with 1/3/…"
   □ is slower than comparison + jump (and so "deprecated" on 64 Bit)

Loop examples: For 2...7

```
.equ      END, 7

          movq $0,%rax     # End result - initialize with 0
          # FOR-LOOP
          movq $2,%rcx     # Initialize loop counter
for_start:
          cmpq $END,%rcx   # At the end of the loop?
          jg for_end       # If not, go to end
          addq %rcx,%rax   # Add current index to result
          incq %rcx        # Increment loop counter
          jmp for_start    # Go to loop begin
for_end:
```

■ How often is this loop going to be evaluated?
    ☐ With the following indices: 2, 3, 4, 5, 6, 7
    ☐ Result: 27
☐ Value of RCX afterwards: 8

Loop examples: Repeat

```
.equ       END, 7

           movq $0,%rax       # End result - initialize with 0
           # REPEAT-LOOP
           movq $2,%rcx       # Initialize loop counter
repeat_start:
           addq %rcx,%rax     # Add current index to result
           incq %rcx          # Increment loop counter
           cmpq $END,%rcx     # At the end of the loop?
           jle repeat_start   # If not, go to top
repeat_end:
```

■ How often is this loop going to be evaluated?
  □ With the following indices: 2, 3, 4, 5, 6, 7
  □ Result: 27
□ Value of RCX afterwards: 8

Loop examples: While

```
.equ      END, 7

        movq $0,%rax      # End result - initialize with 0
        # WHILE-LOOP
        movq $2,%rcx      # Initialize loop counter
while_start:
        cmpq $END,%rcx    # At the end of the loop?
        jge while_end     # If not, go to end
        addq %rcx,%rax    # Add current index to result
        incq %rcx         # Increment loop counter
        jmp while_start   # Go to loop begin
while_end:
```

■ How often is this loop going to be evaluated?
  □ With the following indices: 2, 3, 4, 5, 6
  □ Result: 20
□ Value of RCX afterwards: 7

Loop examples: Loop

```
.equ      END, 7

          movq $0,%rax       # End result - initialize with 0
          # LOOP
          movq $END,%rcx     # Initialize loop counter
loop_start:
          addq %rcx,%rax     # Add current index to result
          loop loop_start    # Decrement RCX and jump to start until it
                             # reaches zero
loop_end:
```

■ How often is this loop going to be evaluated?
    □ With the following indices: 7, 6, 5, 4, 3, 2, 1
    □ Result: 28
  □ Value of RCX afterwards: 0

Strings

■ Strings in Assembler are really C strings
  □ Because we run our programs in Linux, which is written in C/C++
    ● As almost all operating systems!
  □ This has no technical reason, it's just a "convention" as every operating system function accepting a string expects a "C" string!

■ Properties of a C string:
  □ It has a starting address: the address of its first character
  □ It does **not** have an explicit length **anywhere**
  □ It ends with a "zero"-byte: 0x00, '\0', 0 (=all are the same)

■ Security issues:
  □ You can easily change a string, just overwrite its bytes
  □ Take care to not extend it, typic. there is no free space at the end
  □ Do not overwrite the terminating zero, or your string will "extend" for any string functions to the end of the memory (or usually: some zero encountered before, resp. a memory access violation)

Defining "text" in assembler

■ **`.string:`**   Define a String, will be zero terminated automatically

■ **`.ascii:`**   Define an ASCII string – use it directly and unchanged
  □ No zero byte added at end (if needed → Put it into string!)

■ **`.asciz:`**   Define an ASCII string and add a zero byte at the end
  □ Do not manually add the zero byte or you get two!
    ● Just a waste of space, but no danger
  □ Attention: "ascii" but "asciz" (only one "i"!)

■ **`.byte:`**   Manually define a string as individual bytes
  □ Not recommended; use for binary data!

■ **`.lcomm:`**   Reserve space for a "string", but do not initialize it
  □ This is not really a string, but some generic area of memory!
  □ Typically used for buffers (file I/O)

String example

■ Print a fixed string and the first parameter on the console

■ How do we print? Write to file descriptor 1 (STDOUT)
    ☐ Note: This "file" is already open!

■ Where do we get the first parameter?
    ☐ See the stack!
    ☐ Remember: the first "parameter" is the argument count, the second the name of the program itself
    ☐ What is really on the stack?
        ● Not the parameter, but the address in memory where it is stored
            ○ We do not care where specifically this is (=logical address)!
    ☐ We can let the assembler count the characters in a static string, but for the program parameter we must do this manually…

Printing a string (1)

```
.section .data

prefix: .ascii "Hello \0" # Must be terminated manually
.set prefix_len,.-prefix  # Calculate length as current address
                          # minus start address of string

postfix: .asciz ", how are you?\n" # Automatically terminated
.set postfix_len,.-postfix

error: .string "Program parameter(s) incorrect\n"   # Auto-terminated
.set error_len,.-error

.section .text
.globl _start

_start:
        movq %rsp,%rbp
        # Check parameter count
        cmpq $2,0(%rbp)
        jne print_error
```

Printing a string (2)

```asm
        # Print prefix
        movq $1,%rdi            # File descriptor of STDOUT
        movq $prefix,%rsi       # Print prefix
        movq $prefix_len,%rdx   # Length of string
        movq $1,%rax            # Write to stream
        syscall

        # Print parameter 1
        movq $0,%rdx            # Set count to 0
        movq 16(%rbp),%rsi      # Retrieve start address
len_loop:
        cmpb $0,(%rsi,%rdx,1)   # Retrieve first byte of string
        je end_len              # If zero -> End of string
        incq %rdx               # One more character found
        jmp len_loop            # Continue loop
end_len:
        movq $1,%rdi            # File descriptor of STDOUT
        movq 16(%ebp),%rsi      # Print parameter 1
        # EDX (=Length of string) has been calculated above!
        mov1 $1,%rax            # Write to stream
        syscall
```

Printing a string (3)

```
        # Print postfix
        movq $1,%rdi                # File descriptor of STDOUT
        movq $postfix,%rsi          # print postfix
        movq $postfix_len,%rdx      # Length of string
        movq $1,%rax                # Write to stream
        syscall
        jmp program_end

print_error:
        movq $1,%rdi                # File descriptor of STDOUT
        movq $error,%rsi            # print postfix
        movq $error_len,%rdx        # Length of string
        movq $1,%rax                # Write to stream
        syscall

program_end:                        # Terminate program
        movq $0,%rdi
        movq $60,%rax
        syscall
```
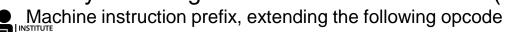
Special string instructions

■ For handling strings, special commands exist

■ Note: While these are intended for strings, they really operate on bytes/words/… and can therefore be used for any other data as well!

■ Basic operations
- □ movs: Copy from source to destination
- □ lods: Load data from memory
  - ● Similar to "mov", but can be repeated (see below!)
- □ stos: Store data into memory
  - ● Similar to "mov", but can be repeated (see below!)
- □ cmps: Compare two data items in memory
  - ● This allows the use of two memory locations in one instruction!
- □ scas: Compare register (al, ax, eax, rax) with memory (="scan")
  - ● Find a specific value in memory
- □ rep: Repeat the following instruction until RCX is zero
  - ● Similar to the loop command
  - ● Works only for a single of the instructions above (e.g. "rep stos")!
    - ● Machine instruction prefix, extending the following opcode

REP variants

- The direction can be set with the flag "DF" (Direction Flag)
  - □ DF=0 ("cld" instruction) → Operation works left-to-right
    - ● Addresses are increased ("CLear Direction flag")
  - □ DF=1 ("std" instruction) → Operation works right-to-left
    - ● Addresses are decreased ("SeT Direction flag")
  - □ Note:
    - ● RCX is **always** decreased
    - ● This affects the addresses (RSI/RDI) only, not a comparison value!

- Variations:
  - □ REP: Repeat until RCX=0
  - □ REPE/REPZ: Repeat until RCX=0 or the zero flag is not set
    - ● "Repeat while equal/zero"; Checks byte/world/long/quad just handled
  - □ REPNE/REPNZ: Repeat until RCX=0 or the zero flag is set
    - ● "Repeat while not equal/zero"; checks data just handled

Which addresses are used?

- The addresses are **all** implicit. They do **not** appear in the instruction and therefore **cannot be changed at all**!
  - ☐ Make sure they contain the desired values before
  - ☐ If needed for other things, save the previous content on the stack and restore it afterwards

- Addresses used:
  - ☐ Source: DS:(E)SI / RSI
    - ● movs, cmps, lods
  - ☐ Destination: ES:(E)DI / RDI
    - ● movs, cmps, stos, scas
  - ☐ Content: RAX
    - ● scas, lods, stos
  - ☐ DS and ES are segment registers. These are not used for the segmentation memory model anymore, but for security reasons
    - ● For the string instructions in Linux IA-32 you can ignore them!
    - ● On IA-64 they are ignored and RSI/RDI alone are used

String instructions: Calc. string length

```
movq 16(%rbp),%rdi          # Store start address of string
movq $-1,%rcx               # Initialize with maximum possible length
cld                         # Set direction to for-/upward
movb $0,%al                 # Set the value to look for
repne scasb                 # Repeat while not equal a scan for %al
notq %rcx                   # Invert bits
decq %rcx                   # Decrement by one
movq %rcx,%rdx              # Store as length
```

■ How do we calculate the result?
  ☐ We start at -1, but rep uses unsigned values, so this is actually the maximum number (0xFF…FF)
  ☐ Even if the first byte is already the terminating-zero, RCX will still be decremented
  ☐ Result: Length = -1 (start) - RCX (end) - 1 (zero-byte)
  ☐ Or: Length = -RCX-2
  ☐ 2-complement: -RCX-2 = NEG(RCX)-2 = NOT(RCX)-1
    ● NEG(A) = NOT(A)+1

String instructions

```
cld                     # Set direction upward
movq $BUFFER,%rdi       # Set target address
movb $'!',%al           # The character to print
rep stosb               # Repeatedly store the byte in AL
movb $'\0',(%rdi)       # Add termination
```

■ Create as many exclamation marks in the buffer as RCX tells us:
  □ RCX must be set up before
  □ RDI is the destination, i.e. our buffer
    ● Which is hopefully large enough: RCX+1 !
    ● Note: For security we should check this somehow… (but can we?)
  □ AL is the value to store
    ● '$' is needed or the character would be interpreted as memory address!
  □ We add a termination here – might be necessary or not, depending on what happens next with the string we created

THANK YOU FOR YOUR ATTENTION!

JOHANNES KEPLER UNIVERSITÄT LINZ

INSTITUTE OF NETWORKS AND SECURITY

http**s**://www.ins.jku.at

Slides by: Michael Sonntag

michael.sonntag@ins.jku.at

+43 (732) 2468 - 4137

S3 235 (Science park 3, 2nd floor)