

Computer Architecture (Practical Class)

Assembly: Controlling Execution Flow

Luís Nogueira

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

lmn@isep.ipp.pt

2023/2024

- So far, we have only considered the behavior of straight-line code, where instructions follow one another in sequence
- Some constructs in C, such as conditionals, loops, and switches, require conditional execution
 - Where the sequence of operations that get performed depends on the outcomes of tests applied to the data
- The CPU has a *FLAGS* register, where a set of single-bit condition codes describe the attributes of the most recent arithmetic or logical operation
- The execution order of a set of instructions can be altered with a *jump* instruction, indicating that control should pass to some other part of the program, possibly contingent on the result of some test on condition codes

The RFLAGS Register

- 64-bit register used as a collection of bits representing Boolean values to store the results of operations and the state of the processor
- Each bit is a Boolean flag (1 - active/true, 0 - inactive/false)
- As instructions execute, they may change some of these flags

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15
%rip	Instruction pointer
%rflags	Status, control and system flags

The RFLAGS Register - Important flags for control flow

- CF - carry flag
 - Set on most significant bit carry or borrow; cleared otherwise
- ZF - zero flag
 - Set if result is zero; cleared otherwise
- SF - sign flag
 - Set equal to the most significant bit of result (0 if positive, 1 if negative)
- OF - overflow flag
 - Set if result is too large (a positive number) or too small (a negative number) to fit in destination operand; cleared otherwise

The RIP Register

- The program counter (called *%rip* in x86-64) indicates the address in memory of the next instruction to be executed
- After the instruction's execution, *%rip* is automatically increased to the address of the next instruction
- A jump instruction can cause the execution to switch to a completely different position in the program
 - Unconditionally - the instruction pointer is set to a new value
 - Conditionally - the instruction pointer is set to a new value if a condition is true

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15
%rip	Instruction pointer
%rflags	Status, control and system flags

`jmp address`

- The `jmp` instruction changes the RIP register to address, a location within the program to jump to, usually denoted by a label

Unconditional Jump Example

```
.global jmptest
jmptest:
    ...
    movq %rax, %rcx
    addq %rdx, %rcx
    jmp end

    # this line is never executed!
    movq $1, %rax

end:
    movq $10, %rax

    ...
    ret
```

- Conditional jumps are taken or not depending on the state of the RFLAGS register at the time the branch is executed
- Each conditional jump instruction examines specific flag bits to determine whether the condition is proper for the jump to occur. Some examples:
 - JE – Jump if equal (ZF=1)
 - JL – Jump if less (SF<>OF)
 - JG – Jump if greater (ZF=0 e SF=OF)
- Similarly to the `jmp` instruction, they only take one argument indicating the address within the program to jump to

Important note

- Before a conditional jump, **condition codes in RFLAGS must be set appropriately by some operation...**

`cmp operand1, operand2`

- The compare instruction is the most common way to evaluate two values for a conditional jump
- Compares the second operand with the first operand by executing a subtraction ($\text{operand2} - \text{operand1}$)
- Does not change the operands, but changes the condition codes in the RFLAGS register
- Examples:
 - if $\text{operand2} == \text{operand1}$ then ZF (zero flag) = 1
 - if $\text{operand2} > \text{operand1}$ then SF (sign flag) = 0
 - if $\text{operand2} < \text{operand1}$ then SF (sign flag) = 1
- The CMP instruction can be applied to 8 (b), 16 (w), 32 (l), or 64(q) bits

jX	Condition	Description
jmp	1	Unconditional
j_e	ZF	Equal / Zero
j_{ne}	~ZF	Not Equal / Not Zero
j_s	SF	Negative
j_{ns}	~SF	Nonnegative
j_g	~(SF^OF) & ~ZF	Greater (signed)
j_{ge}	~(SF^OF)	Greater or Equal (signed)
j_l	(SF^OF)	Less (signed)
j_{le}	(SF^OF) ZF	Less or Equal (signed)
j_a	~CF & ~ZF	Above (unsigned)
j_b	CF	Below (unsigned)

Controlling Execution Flow

```
...  
# compares %rcx with %rsi through %rsi - %rcx  
cmpq %rcx, %rsi  
jg jmp_rsi_is_greater  
je jmp_rsi_is_equal  
jl jmp_rsi_is_less  
jmp_rsi_is_greater:  
    movq $1, %rax  
    jmp end  
jmp_rsi_is_equal:  
    movq $0, %rax  
    jmp end  
jmp_rsi_is_less:  
    movq $-1, %rax  
end:  
    ret
```

Exercise

Are you able to reduce the number of jumps without changing the program behaviour?

Controlling execution flow - Another Example

Consider the following C code

```
long x;  
long y;  
  
long test_xy(){  
    if(x > y)  
        return 1;  
    else  
        return 0;  
}
```

Can be written in Assembly as

```
test_xy:  
    movq x(%rip), %rdi  
    cmpq y(%rip), %rdi  
    jle false  
    movq $1, %rax  
    jmp end  
false:  
    movq $0, %rax  
end:  
    ret
```

Practice problem

Consider the following Assembly code

```
test_xyz:
    movq x(%rip), %rdi
    movq y(%rip), %rsi
    movq z(%rip), %rdx
    movq %rdi, %rax
    addq %rsi, %rax
    subq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    jmp .L4
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    jmp .L4
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    ret
```

Fill in the missing expressions in C code

```
long x;
long y;
long z;

long test_xyz()
{
    long val = _____;
    if (_____) {
        if (_____)
            val = _____;
        else
            val = _____;
    } else if (_____)
        val = _____;

    return val;
}
```

Practice problem

Consider the following Assembly code

```
test_xyz:
    movq x(%rip), %rdi
    movq y(%rip), %rsi
    movq z(%rip), %rdx
    movq %rdi, %rax
    addq %rsi, %rax
    subq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmpq %rdx, %rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    jmp .L4
.L3:
    movq %rsi, %rax
    imulq %rdx, %rax
    jmp .L4
.L2:
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4:
    ret
```

Fill in the missing expressions in C code

```
long x;
long y;
long z;

long test_xyz()
{
    long val = x + y - z;
    if (x < -3){
        if(y < z)
            val = x * y;
        else
            val = y * z;
    }else if(x > 2)
        val = x * z;

    return val;
}
```

JC – Jump if carry (CF=1)

- The *carry* flag is used in **unsigned** integer arithmetic when it generates a carry or borrow for the most significant bit
- The jump is taken if the *carry* flag is active (1)

Test Carry Example

```
.global addtest_carry

addtest_carry:
    ...
    addq %rax, %rcx

    # jump if carry
    jc carry_detected
    movq $0, %rax
    jmp end

carry_detected:
    movq $1, %rax

end:
    ret
```

JO – Jump if overflow (OF=1)

- The *overflow* flag is used in **signed** integer arithmetic when a positive value is too large, or a negative value is too small, to be properly represented in the register
- The jump is taken if the *overflow* flag is active (1)

Test Overflow Example

```
.global addtest_overflow

addtest_overflow:
    ...
    movb $-127, %cl
    addb $-10, %cl

    # jump if overflow
    jo overflow_detected
    movq $0, %rax
    jmp end

overflow_detected:
    movq $1, %rax

end:
    ret
```

- In C, we do not have access to these flags and overflows are not signaled as errors
- We can check if an overflow has occurred on $x + y$ by seeing, if and only if, $sum < x$ (or equivalently, $sum < y$) for unsigned addition
- For signed addition, the computation of sum has had positive overflow if and only if $x \geq 0$ and $y \geq 0$ but $sum < 0$. The computation has had negative underflow if and only if $x < 0$ and $y < 0$ but $sum \geq 0$

Test Overflow/Underflow in C

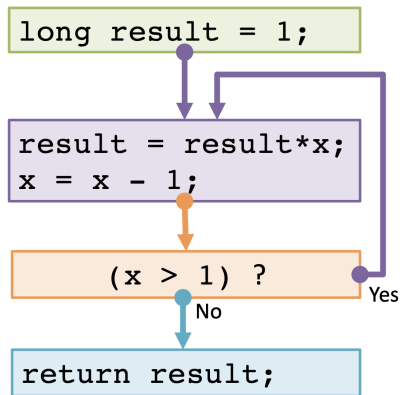
```
/* return 1 if arguments x and y can be added
   without causing overflow/underflow */
int add_ok(int x, int y) {
    int sum = x+y;
    int neg_over = x < 0 && y < 0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum < 0;
    return !neg_over && !pos_over;
}
```


- C provides several looping constructs - namely, *do-while*, *while*, and *for*
- No corresponding instructions exist in machine code
- Instead, combinations of conditional tests and jumps are used to implement the effect of loops
- Gcc and other compilers generate loop code based on several loop patterns
- We will study the translation of loops as a progression, starting with the *do-while* basic pattern

The *do-while* Loop

Consider the following C code

```
long x;  
long fact_do_while()  
{  
    long result = 1;  
  
    do{  
        result = result * x;  
        x = x - 1;  
    } while(x > 1);  
  
    return result;  
}
```



The *do-while* Loop

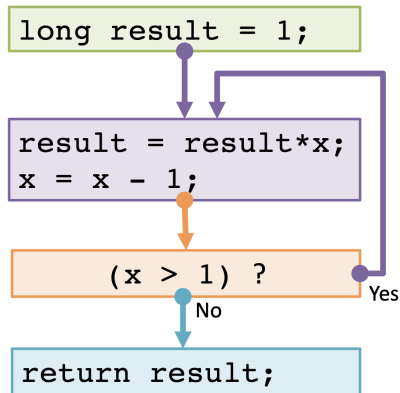
Corresponding Assembly code

```
fact_do_while:
    movq    x(%rip), %rdi
    movq    $1, %rax

my_loop:
    imulq   %rdi, %rax
    decq    %rdi

    cmpq    $1, %rdi
    jg      my_loop

    ret
```

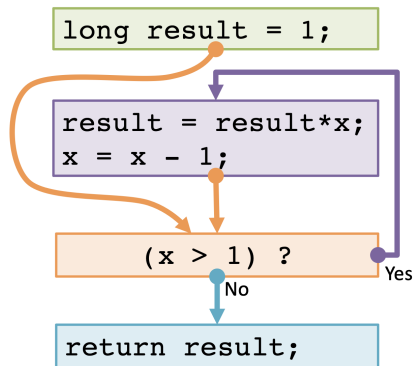


This pattern is frequently used by GCC in x86-64 code. Why?

The *while* Loop

Consider the following C code

```
long x;  
  
long fact_while()  
{  
    long result = 1;  
  
    while( x > 1 ){  
        result = result * x;  
        x = x - 1;  
    }  
  
    return result;  
}
```



It differs from *do-while* in that the test expression is evaluated and the loop is potentially terminated before the first execution of the body statement

The *while* Loop

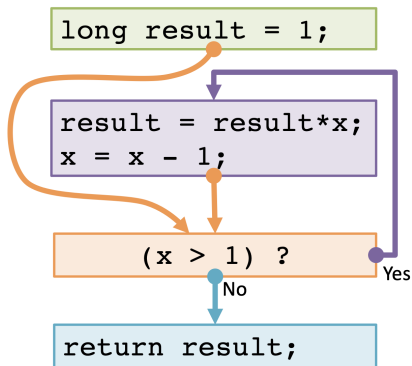
Corresponding Assembly code

```
fact_while:
    movq    x(%rip), %rdi
    movq    $1, %rax
    jmp     test_expression

my_loop:
    imulq   %rdi, %rax
    decq    %rdi

test_expression:
    cmpq    $1, %rdi
    jg      my_loop

ret
```



The unconditional jump before the loop causes the program to first perform the test before modifying the values of `x` or `result`

The *while* Loop

Consider the following C code

```
long x;  
  
long fact_while()  
{  
    long result = 1;  
    while( x > 1 ){  
        result = result * x;  
        x = x - 1;  
    }  
  
    return result;  
}
```

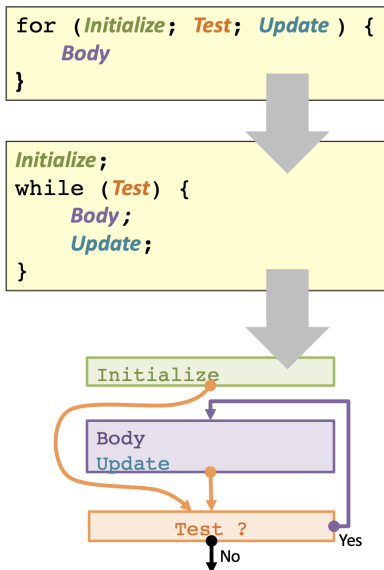
Another possible pattern in Assembly code

```
fact_while:  
    movq    x(%rip), %rdi  
    movq    $1, %rax  
  
my_loop:  
    cmpq    $1, %rdi  
    jle     end_my_loop  
  
    imulq   %rdi, %rax  
    decq    %rdi  
    jmp     my_loop  
  
end_my_loop:  
    ret
```

The loop itself has the same general structure as the previous pattern. One interesting feature, however, is that the loop test has been changed from $x > 1$ in the original C code to $x \leq 1$

The *for* Loop

- The C language standard defines a behavior of a *for* loop that can be easily translated to a *while* loop
- The code generated by gcc for a for loop then follows one of the two translation strategies for while loops that were discussed earlier



Consider the following C code

```
long x;

long fact_for()
{
    long result = 1;
    long i;

    for(i = 2; i <= x; i++) {
        result = result * i;
    }

    return result;
}
```

Can be translated to a while loop

```
long x;

long fact_for_while()
{
    long result = 1;
    long i;

    i = 2;
    while(i <= x){
        result = result * i;
        i++;
    }

    return result;
}
```


The *for* Loop

Consider the following C code

```
long x;  
  
long fact_for_while()  
{  
    long result = 1;  
    long i;  
  
    i = 2;  
    while(i <= x){  
        result = result * i;  
        i++;  
    }  
  
    return result;  
}
```

Corresponding Assembly code

```
fact_for_while:  
    movq    x(%rip), %rdi  
    movq    $1, %rax  
    movq    $2, %rdx  
  
my_loop:  
    cmpq    %rdi, %rdx  
    jg      end_my_loop  
  
    imulq   %rdx, %rax  
    incq    %rdx  
    jmp     my_loop  
  
end_my_loop:  
    ret
```

The loop, loope, loopz, loopne, and loopnz instructions

- LOOP instructions can be used in place of certain conditional jump instructions and give the programmer a simpler way of writing loop sequences
- They provide iteration control and combine loop index management with conditional branching
- LOOP is a single instruction that functions the same as a DECQ RCX instruction followed by a JNZ instruction

Important notes:

- The loop instructions test the flags, but do not change them
- The target label is encoded as a signed 8-bit offset. This means that only jumps offsets of -128 to +127 are allowed with these instructions

The `loop`, `loope`, `loopz`, `loopne`, and `loopnz` instructions

How to use:

- 1 Prior to enter the set of instructions to iterate, load the `%rcx` register with the number of required iterations
- 2 Then, use the `loop` instruction at the end of that set
- 3 The `loop` instruction automatically decrements `%rcx` by one and jumps to the label if `%rcx` is different from 0

Important notes:

- What will happen if the `%rcx` register is zero or less before the first call to any `loop` instruction?
- What will happen if the `%rcx` register is changed inside the loop by any other instruction or function call?

The loop, loope, loopz, loopne, and loopnz instructions

loop instruction example

```
...  
    movq $100, %rcx      # number of iterations  
my_loop:  
    ...                  # loop body  
    ...  
    loop my_loop  
    ...
```

- loop automatically decrements %rcx by one and jumps to the label if %rcx is different from 0
- loope/loopz: decrements %rcx by one and jumps to the label if %rcx is different from 0, *and the flag ZF is active*
- loopne/loopnz: decrements %rcx by one and jumps to the label if %rcx is different from 0, *and the flag ZF is **not** active*