ARQCP Course

Arquitetura de Computadores
Licenciatura em Engenharia Informática

2023/24

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

Data Transfer

Addressing Memory

- `offset(base,index,scale)` where `base` and `index` are registers, `scale` is a constant 1,2,4, or 8, and `offset` is a constant or symbolic label.
  - The effective address corresponding to this specification is `(offset + R[base] + R[index] * scale)`.
  - Any of the various fields may be omitted if not wanted; in effect, the omitted field contributes 0 to the effective address (except that scale defaults to 1).

Syntax	**Meaning**
0x104	Address 0x104 (no $)
(`%rax`)	What's in `%rax`
4(`%rax`)	What's in `%rax`, plus 4
(`%rax`, `%rdx`)	Sum of what's in `%rax`and `%rdx`
4(`%rax`, `%rdx`)	Sum of values in `%rax`and `%rdx`, plus 4
(, `%rcx`, 4)	What's in `%rcx`, times 4 (multiplier can be 1, 2, 4, 8)
(`%rax`, `%rcx`, 2)	What's in `%rax`, plus 2 times what's in `%rcx`
8(`%rax`, `%rcx`, 2)	What's in `%rax`, plus 2 times what's in `%rcx`, plus 8

Moving Data (I)

- Uasge: `mov[b|w|l|q] S, D`
  - Actually, `mov` instruction copies data
  - For most cases, the `mov` instructions will only update the specific register bytes or memory locations indicated by the destination operand
    - The only exception is that when `movl` has a register as the destination, it will also set the high-order 4 bytes of the register to 0
    - This exception arises from the convention, adopted in x86-64, that any instruction that generates a 32-bit value for a register also sets the high-order portion of the register to 0

```
                # %rax = 0x0011223344556677
movb $-1,%al    # %rax = 0x00112233445566FF
movw $-1,%ax    # %rax = 0x001122334455FFFF
movl $-1,%eax   # %rax = 0x00000000FFFFFFFF
movq $-1,%rax   # %rax = 0xFFFFFFFFFFFFFFFF
```

Moving Data (II)

- The regular `movq` instruction can **only have immediate source operands that can be represented as 32-bit two's-complement numbers**
  - This value is then sign extended to produce the 64-bit value for the destination
- The `movabsq` move absolute quad word) instruction c**an have an arbitrary 64-bit immediate value as its source operand** and can only have a register as a destination
  - Usage: `movabsq Imm, Reg`

```
movq $0x44556677,%rbx              # %rbx = 0x0000000044556677
movabsq $0x0011223344556677,%rax   # %rax = 0x0011223344556677
```

`movq` operand combinations

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

- Cannot do **memory-memory transfer with a single instruction**.

Function arguments

- The **first six arguments of a function** are stored as follows:

Operand size (bits)	Arguments					
	1st	**2nd**	**3rd**	**4th**	**5th**	**6th**
64	`%rdi`	`%rsi`	`%rdx`	`%rcx`	`%r8`	`%r9`
32	`%edi`	`%esi`	`%edx`	`%ecx`	`%r8d`	`%r9d`
16	`%di`	`%si`	`%dx`	`%cx`	`%r8w`	`%r9w`
8	`%dil`	`%sil`	`%dl`	`%cl`	`%r8b`	`%r9b`

- If there are **more than six** they are stored on the **stack**.

void swap(long* px, long *py) (I)

```
void swap(long* px, long* py)
{
  long t0 = *px;
  long t1 = *py;
  *px = t1;
  *py = t0;
}
```

```
#include <stdio.h>

int main(void) {
  long x = 20;
  long y = 50;
  printf("before swap: x= %ld: y= %ld\n", x,y);
  swap(&x, &y);
  printf("after swap: x= %ld: y= %ld\n", x,y);
  return 0;
}
```

`void swap(long* px, long *py)` (II)

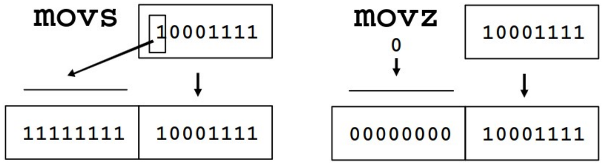
```
.section .text
.global swap
swap:
 # prologue

 movq (%rdi), %rax
 movq (%rsi), %rdx
 movq %rdx, (%rdi)
 movq %rax, (%rsi)

 # epilogue
 ret
```

`movs` and `movz`

- The `movs` class fills out the remaining bytes of the **destination with bit sign**
  - Usage: `movs[bw|bl|bq|wl|wq|lq] S,D`
- The `movz` class fills out the remaining bytes of the **destination with zeros**
  - Usage: `movz[bw|bl|bq|wl|wq] S,D`
    - Note the absence of an explicit instruction to zero-extend a 4-byte source value to an 8-byte destination `[lq]`
    - This type of data movement can be implemented using a `movl` instruction having a register as the destination



Arithmetic Operations

void sumstore(long x, long y, long *dest) (I)

```c
void sumstore(long x, long y, long *dest)
{
 long t = x + y;
 *dest = t;
}
```

```c
#include <stdio.h>

int main(void) {
 long x = 0;
 sumstore(20, 30, &x);
 printf("x= %ld\n", x);
 return 0;
}
```

`void sumstore(long x, long y, long *dest)`(II)

```
.section .text
.global sumstore
sumstore:
 # prologue

 movq %rdi, %rbx
 addq %rsi, %rbx
 movq %rbx, (%rdx)


 # epilogue
 ret
```

Arithmetic operand sizes

- Most of the operations are given as instruction classes, as they can have different variants with different operand sizes, specified by suffixes (b, w, l, and q)
- However, **in an operation the operands must be of the same size**, according **to the instruction suffix**.
  - If not, the **type conversion has higher precedence than arithmetic operations**
- Moving **from a smaller to a larger** data size can involve either **sign extension** (for signed values) or **zero extension** (for unsigned values)
- There are two `mov` instructions that can be used to copy a smaller source to a larger destination:
  - `movz` fills the remaining bytes with zeros
  - `movs` fills the remaining bytes by sign-extending the most significant bit in the source.
  - The source could be from memory or a register, and the destination is a register.

void sumstore2(char x, short y, long *dest) (I)

```c
void sumstore2(char x, short y, long *dest)
{
 long t = x + y;
 *dest = t;
}
```

```c
#include <stdio.h>

int main(void) {
 long x = 0;
 sumstore2(20, 30, &x);
 printf("x= %ld\n", x);
 return 0;
}
```

void sumstore2(char x, short y, long *dest)(II)

```
.section .text
.global sumstore2
sumstore2:
  # prologue

  movsbq %dil, %rbx
  movswq %si, %rcx
  addq %rbx, %rcx
  movq %rcx, (%rdx)

  # epilogue
  ret
```

Large Multiplication

- Multiplying 64-bit numbers can produce a 128-bit result.
- How does x86-64 support this with only 64-bit registers?
  - **Join two 64-bit registers to hold a 128-bit result**.

Instruction	Effect	Description
imulq S	R[%rdx]:R[%rax] ← S * R[%rax]	Signed full multiply
mulq S	R[%rdx]:R[%rax] ← S * R[%rax]	Unsigned full multiply

Division and Remainder

- The high-order 64 bits of the **dividend are in `%rdx`, and the low-order 64 bits are in `%rax`**.
- The **divisor is the operand to the instruction**.
- The **quotient is stored in `%rax`**, and the **remainder in `%rdx`**.

Instruction	**Effect**	**Description**
idivq S	R[%rax] ← R[%rdx]:R[%rax] / S R[%rdx] ← R[%rdx]:R[%rax] mod S	Signed divide
divq S	R[%rax] ← R[%rdx]:R[%rax] / S R[%rdx] ← R[%rdx]:R[%rax] mod S	Unsigned divide
cqto	R[%rdx]:R[%rax] ← SignExtend(R[%rax])	Convert to oct word

- Most **division uses only 64 bit dividends**.
- The **`cqto` instruction sign-extends the 64-bit value in `%rax` into `%rdx` to fill both registers with the dividend**, as the division instruction expects.

void divide(long dividend, long divisor, long* quotient, long* remainder) (I)

```c
void divide(long dividend, long divisor, long* quotient, long* remainder)
{
 *quotient = dividend / divisor;
 *remainder = dividend % divisor;
}
```

```c
#include <stdio.h>

int main(void) {
 long q = 0, r = 0;
 divide(100, 30, &q, &r);
 printf("q= %ld, r= %ld\n", q,r);
 return 0;
}
```

`void divide(long dividend, long divisor, long* quotient, long* remainder)` (II)

```
.section .text
.global divide
divide:
# prologue

 movq %rdi, %rax
 movq %rdx, %rdi
 cqto
 idivq %rsi
 movq %rax, (%rdi)
 movq %rdx, (%rcx)

 # epilogue
 ret
```

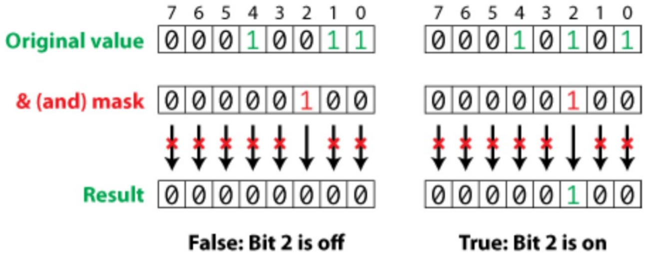
Logic Operations

Bit masking (I)

- Logical instructions **play a pivotal role in manipulating data at the bit level**.
- These instructions **allow for precise control over individual bits**, making them essential for various tasks.
- One of the most common uses of logical operations is **bit masking**.
- Bit masking **is a technique in programming used to test or modify the states of the bits** of a given data.

Bit masking (II)

- The main usage of bitwise logical instructions is: **to set**, **to clear**, **to invert**, **to isolate** some selected bits in the Destination operand.
- To do this, a Source bit pattern known as **a mask** is constructed.
- The mask bits **are chosen so that the selected bits are modified in the desired manner when an instruction is executed**.



Bit masking (II)

- The Mask bits **must be constructed based on the properties of `and`, `or`, and `xor` operations**.
  - The `and` instruction:
    - Can be used to clear specific Destination bits while preserving the others.
    - A zero mask bit clears the corresponding Destination bit;
    - A one mask bit preserves the corresponding Destination bit.
  - The `or` instruction:
    - Can be used to set specific Destination bits while preserving the others.
    - A one mask bit sets the corresponding Destination bit;
    - A zero mask bit preserves the corresponding Destination bit.
  - The `xor` instruction:
    - Can be used to invert specific Destination bits while preserving the others.
    - A one mask bit inverts the corresponding Destination bit;
    - A zero mask bit preserves the corresponding Destination bit.

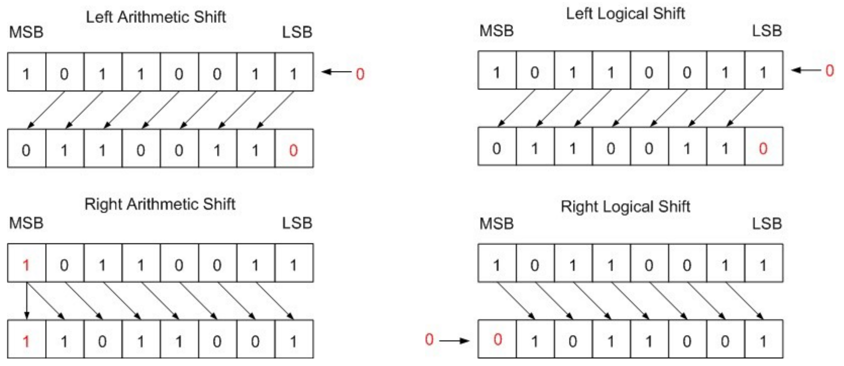
Bit masking: Example

- Swap values

```
xorq %rdi, %rsi
xorq %rsi, %rdi
xorq %rdi, %rsi
```

- Clear a register

```
andq $0, %rax
```



- Arithmetic Shift operations **can be used for dividing or multiplying** an integer variable **by powers of 2**.
  - **Multiplication by left shift**: the result of a Left Shift operation is a multiplication by 2^n , where n is the number of shifted bit positions.
  - **Division by right shift**: The result of a Right Shift operation is a division by 2^n , where n is the number of shifted bit positions.

Control

`%rflags`

- The CPU contains a 64-bit register, `%rflags`, **where individual bits (flags) are set or cleared as a consequence of arithmetic and other operations**.
- `%rflags` register holds information **about the result of the most recent instructions that has affected flags**.
- These can be **updated and read to influence what to do next**.
- Most common flags, **condition codes (cc)**, are:
  - **Carry flag (CF)**.
    - The most recent operation generated a carry out of the most significant bit.
    - Used to detect overflow for unsigned operations.
  - **Zero flag (ZF)**.
    - The most recent operation yielded zero.
  - **Sign flag (SF)**.
    - The most recent operation yielded a negative value.
  - **Overflow flag (OF)**.
    - The most recent operation caused a two's-complement overflow-either negative or positive.
- **They can be set implicitly or explicitly**

Condition codes: Implicit setting

- Implicitly set (**think of it as side effect**) by arithmetic operations
- `add a, b`
  - `t = a + b` setting destination to `t`
  - Sets condition codes based on value of `a + b`
    - CF set if carry out from most significant bit (unsigned overflow)
    - ZF set if `t == 0`
    - SF set if `t < 0` (as signed)
    - OF set if two's-complement (signed) overflow, whenever
      `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

Condition codes: Explicit setting

- `cmp a, b`
  - The `cmp` instruction is like the subtraction instruction, but it does not store the result anywhere.
  - Sets condition codes based on value of `b - a`
    - CF set if carry out from most significant bit (unsigned overflow)
    - ZF set if `a == b`
    - SF set if `b - a < 0` (as signed)
    - OF set if two's-complement (signed) overflow, whenever `(a<0 && b>0 && (b-a)<0) || (a>0 && b<0 && (b-a)>0)`
- `test a, b`
  - The `test` instruction is like the `and` instruction, but it does not store the result anywhere.
  - Sets condition codes based on value of `a & b`
    - ZF set if `a & b == 0`
    - SF set if `a & b < 0`
    - OF and CF are always set to 0
  - Typically, the same operand is repeated (e.g., `testl %eax,%eax` to see whether `%eax` is negative, zero, or positive), or one of the operands is a mask indicating which bits should be tested.

Condition codes: `cmp`

- `cmp a, b`
  - Sets condition codes based on value of `b - a`

Suffix *cc*	Condition tested	Meaning after `cmp`
e	ZF	equal to zero
ne	~ZF	not equal to zero
s	SF	negative
ns	~SF	non-negative
g	~(SF^OF) & ~ZF	greater (signed >)
ge	~(SF^OF)	greater or equal (signed >=)
l	SF^OF	less (signed <)
le	(SF^OF) | ZF	less or equal (signed <=)
a	~CF & ~ZF	above (unsigned >)
ae	~CF	above or equal (unsigned >=)
b	CF	below (unsigned <)
be	CF | ZF	below or equal (unsigned <=)

Condition codes: `cmp` and `test`

Suffix *cc*	Meaning	`cmp a,b`	`test a,b`
e	Equal	b == a	b&a == 0
ne	Not equal	b != a	b&a != 0
s	Sign (negative)	b-a < 0	b&a < 0
ns	(non-negative)	b-a >=0	b&a >= 0
g	Greater	b > a	b&a > 0
ge	Greater or equal	b >= a	b&a >= 0
l	Less	b < a	b&a < 0
le	Less or equal	b <= a	b&a <= 0
a	Above (unsigned >)	b > a	b&a > 0U
b	Below (unsigned <)	b < a	b&a < 0U

- `cmp a,b` is like `sub` (`b-a`), `test` is like `and` (`a & b`)
- Result is not stored anywhere, they only set condition code bits

Condition Code-Dependent Instructions

- There are **three common instruction types that use condition codes (cc)**:
  - `set`*cc* instructions **conditionally set a byte to 0 or 1**
  - `cmov`*cc* instructions **conditionally move data**
  - `j`*cc* instructions **conditionally jump to a different next instruction**

`set`*cc* instructions

- Each `set`*cc* instruction **has a designated destination**:
  - **Byte length register**
    - Do not alter remaining bytes in register
  - **Single-byte memory location**

```
int less(int a, int b){
 return ( a < b );
}
```

```
less:
  cmpl %esi, %edi
  setl %al
  movzbl %al, %eax
  ret
```

`cmov`*cc* instructions

- The `cmov`*cc* instructions **check the condition codes and perform or not a move operation**.
- If the **condition is not satisfied**, a **move is not performed** and **execution continues** with the instruction following the `cmov`*cc* instruction.

```c
int equals(int a, int b){
 int x = 0;
 if ( a == b ) {
  x = 1;
 }
 return x;
}
```

```
equals:
 movl $0, %eax
 cmpl %esi, %edi
 cmove $1, %eax
 ret
```

Unconditional `jmp`

- The `jmp` instruction **jumps to another instruction** in the assembly code (“Unconditional Jump”).
- `jmp Label` (Direct Jump)

```
  jmp end
  ...
end:...
```

- `jmp *Operand` (Indirect Jump)

```
  jmp *%rax  # jump to instruction at address in %rax
```

Conditional `jcc`

- The `jcc` instruction **jumps conditionally (if certain conditions are true) to another instruction** in the assembly code.
- The `test` and `cmp` instructions are combined with the conditional and unconditional jump instructions **to implement most relational and logical expressions and all control structures**.

`if`, `switch` & loops

- The general form of an `if...else` statement in C is:

```
if (test-expr)
  then-statement
else
  else-statement
```

  - where `test-expr` is **an integer expression that evaluates either to 0** (interpreted as meaning “false”) **or to a nonzero value** (interpreted as meaning “true”).
  - Only **one of the two branch statements** (`then-statement` or `else-statement`) **is executed**.
- `if...else` statement in assembly implementation typically adheres to:

```
  t = test-expr;
  if (!t)
    goto false;
  then-statement
  goto done;
false:
  else-statement
done:
```

if...else (II)

```c
int equals (int a, int b){
  int x;
  if ( a == b ) {
    x = 1;
  }else{
    x = 0;
  }
  return x;
}
```

```
equals:
  cmpl %esi, %edi
  jne false
  movl $1, %eax
  jmp done
false:
  movl $0, %eax
done:
  ret
```

```c
int equals(int a, int b){
  int x = 0;
  if ( a == b ) {
    x = 1;
  }
  return x;
}
```

```
equals:
    movl $0, %eax
    cmpl %esi, %edi
    jne false
    movl $1, %eax
false:
    ret
```

if...else (III)

```c
int check(int x, int y){
  if (x < 3 && x == y) {
    return 1;
  } else {
    return 2;
  }
}
```

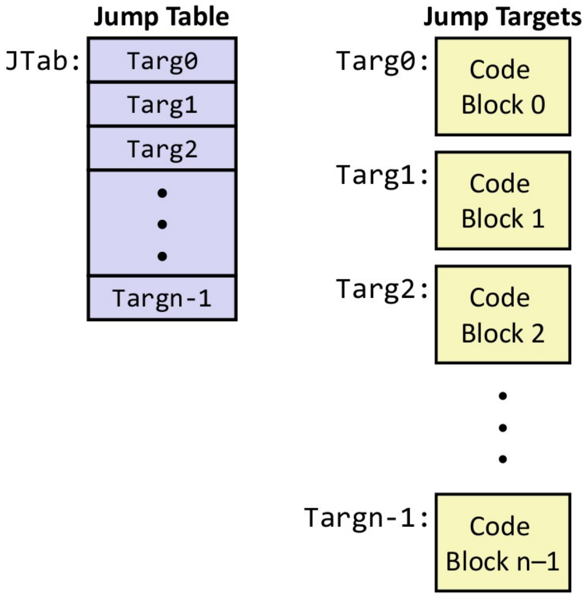
```
check:
  cmpl $3, %edi
  setl %dl
  cmpl %esi, %edi
  sete %al
  testb %al, %dl
  je false
  movl $1, %eax
  jmp done
false:
  movl $2, %eax
done:
  ret
```

`switch` (I)

- A `switch` statement provides a multi-way branching capability based on the value of an integer index.
- They are particularly useful when dealing with tests where there can be a large number of possible outcomes.
- An efficient implementation using a data structure called a **jump table**.
  - A jump table is an array where entry `i` is the address of a code segment implementing the action the program should take when the switch index equals `i`.
  - The code performs an array reference into the jump table using the switch index to determine the target for a jump instruction.
  - The advantage of using a jump table over a long sequence of if-else statements is that the time taken to perform the switch is independent of the number of switch cases.

Switch Form

```
switch (x) {
  case val_0:
     Block 0
  case val_1:
     Block 1
     • • •
  case val_n-1:
     Block n–1
}
```



Approximate Translation

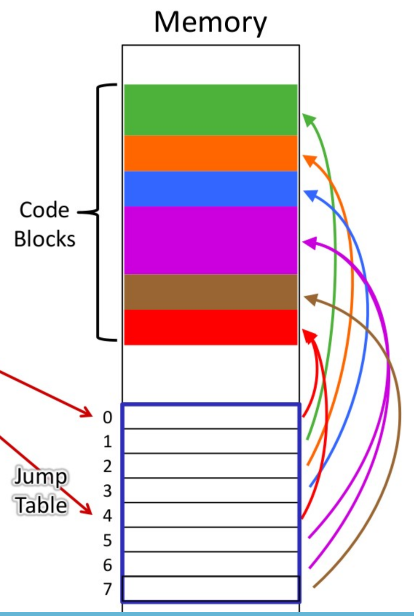
```
target = JTab[x];
goto target;
```

C code:

```
switch (x) {
  case 1: <code> break;
  case 2: <code>
  case 3: <code> break;
  case 5:
  case 6: <code> break;
  case 7: <code> break;
  default: <code>
}
```

Use the jump table when x ≤ 7:

```
if (x <= 7)
  target = JTab[x];
  goto target;
else
  goto default;
```



```c
int switcher(int a, int b, int c)
{
  int answer;
  switch(a) {
    case 5:
      c = b ^ 15;
      /* Fall through */
    case 0:
      answer = c + 112;
    break;
    case 2:
    case 7:
      answer = (c + b) << 2;
    break;
    case 4:
      answer = a;
    break;
    default:
      answer = b;
  }
  return answer;
}
```

```
.L7:
  .long .L3   #0
  .long .L2   #1
  .long .L4   #2
  .long .L2   #3
  .long .L5   #4
  .long .L6   #5
  .long .L2   #6
  .long .L4   #7
```

```
switcher:
  movl %edi, %eax
  cmpl $7, %eax
  ja .L2
  jmp *.L7(,%eax,4)
.L2:
  movl %esi, %eax
  jmp  .L8
.L5:
  movl $4, %eax
  jmp .L8
.L6:
  movl %esi, %eax
  xorl $15, %eax
  movl %eax, %edx
.L3:
  movl %edx, %eax
  addl $112, %eax
  jmp .L8
.L4:
  movl %edx, %eax
  addl %esi, %eax
  sall $2, %eax
.L8:
```

Loops

- C provides several looping constructs—namely, `do-while`, `while`, and `for`.
- Combinations of **conditional tests and jumps are used to implement the effect of loops**.
- Most compilers **generate loop code based on the `do-while` form of a loop**, even though this form is relatively uncommon in actual programs.
- Other loops are transformed into `do-while` form and then compiled into machine code.

`do-while` (I)

- The general **form of a `do-while` statement** is as follows:

```
do
 body-statement
while (test-expr);
```

  - The effect of the loopis to repeatedly execute `body-statement`, evaluate `test-expr`, and continue the loop if the evaluation result is nonzero.
  - The `body-statement` is executed at least once.
- This general form can be translated into conditionals and goto statements as follows:

```
loop:
 body-statement
 t = test-expr;
 if (t)
  goto loop;
```

do-while (II)

```c
void do_while_loop(int n) {
  int i = 0;
  do{
    i++;
  } while (i < n);
}
```

```
do_while_loop:
  movl $0, %eax
loop:
  addl $1, %eax
  cmpl %edi, %eax
  jl loop
  ret
```

`while` (I)

- The general **form of a `while` statement** is as follows:

```
while (test-expr)
  body-statement
```

  - It differs from `do-while` in that `test-expr` is evaluated and the loop is potentially terminated before the first execution of `body-statement`.
  - There are a number of ways to translate a while loop into machine code.
  - One common approach is to transform the code into a `do-while` loop by using a conditional branch to skip the first execution of the body if needed:

```
  if (!test-expr)
    goto done;
  do
    body-statement
  while (test-expr);
done:
```

```
  t = test-expr;
  if (!t)
    goto done;
loop:
  body-statement
  t = test-expr;
  if (t)
    goto loop;
done:
```

```
  goto test;
loop:
  body-statement
test:
  t = test-expr;
  if (t)
    goto loop;
```

while (II)

```c
void while_loop(int n) {
  int i = 0;
  while (i < n) {
    i++;
  }
}
```

```
while_loop:
  movl $0, %eax
  jmp test
loop:
  addl $1, %eax
test:
  cmpl %edi, %eax
  jl loop
 ret
```

`for` (I)

- The general **form of a `for` statement** is as follows:

```
for (init-expr; test-expr; update-expr)
  body-statement
```

  - The C language standard states that the behavior of such a loop is identical to the following code, which uses a `while` loop:

```
  init-expr;
  while (test-expr) {
    body-statement
    update-expr;
  }
```

```
  init-expr;
  if (!test-expr)
    goto done;
  do {
    body-statement
    update-expr;
  } while (test-expr);
done:
```

```
  init-expr;
  goto test;
loop:
  body-statement
  update-expr;
test:
  t = test-expr;
  if (t)
    goto loop;
```

for (II)

```c
void for_loop(int n) {
  for (int i = 0; i < n; i++) {
    // body
  }
}
```

```
for_loop:
  movl $0, %eax
  jmp test
loop:
  addl $1, %eax
test:
  cmpl %edi, %eax
  jl loop
  ret
```

`loop` instruction

- Loop according to counter register (`%rcx`).

```
loop  label
```

- Where, `label` is the target label that identifies the target instruction as in the jump instructions.
- The `loop` instruction assumes that the **counter register contains the loop count**.
- When the **loop instruction is executed**, the **counter register is decremented and the control jumps to the target label**, until the **counter register value reaches zero**.

```
 movq $10, %rcx
iter:
 loop  iter
```

Conditional Loop Instructions

- `loope destination` and `loopz destination`
- Logic:
  - `%rcx` ← `%rcx` - 1
  - If `%rcx` > 0 and ZF=1, jump to destination
- `loopne destination` and `loopnz destination`
- Logic:
  - `%rcx` ← `%rcx` - 1
  - If `%rcx` > 0 and ZF=0, jump to destination