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

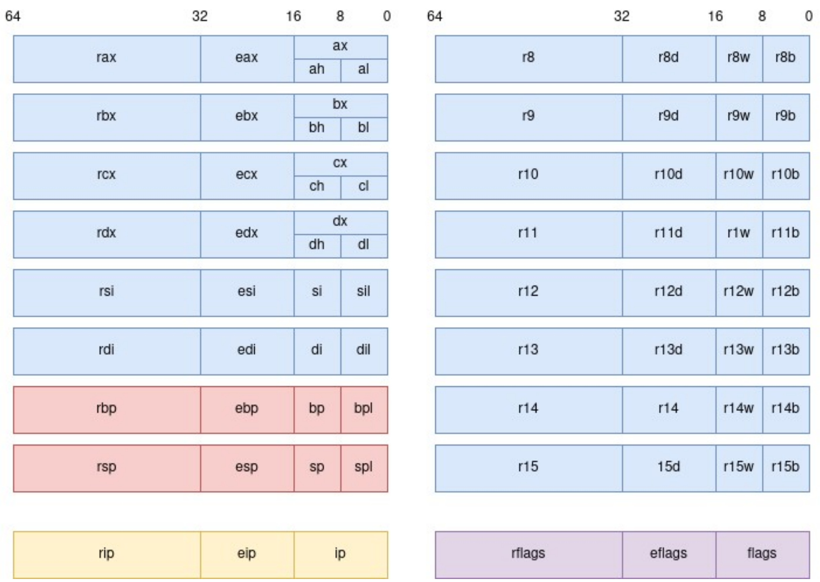
Some of the material/slides are adapted from various:

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

Outline

x86-64

Registers (I)


64 32 16 8 0
rax eax ax ah al
rbx ebx bx bh bl
rcx ecx cx ch cl
rdx edx dx dh dl
rsi esi si sil
rdi edi di dil
rbp ebp bp bpl
rsp esp sp spl
rip eip ip
64 32 16 8 0
r8 r8d r8w r8b
r9 r9d r9w r9b
r10 r10d r10w r10b
r11 r11d r1w r11b
r12 r12d r12w r12b
r13 r13d r13w r13b
r14 r14 r14w r14b
r15 15d r15w r15b
rflags eflags flags

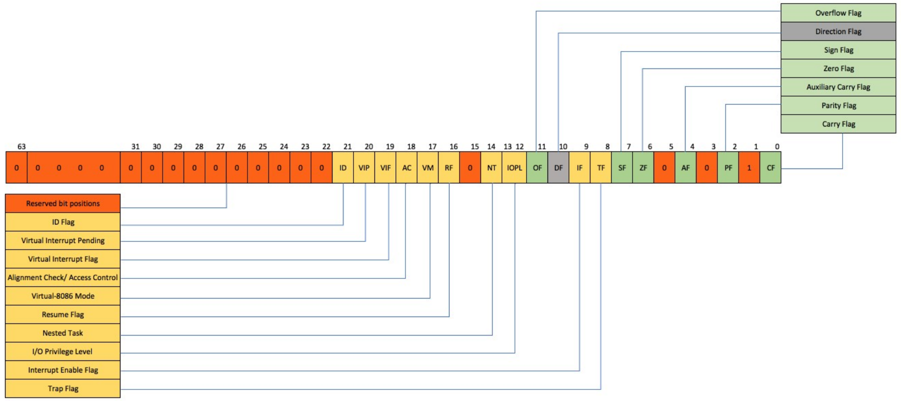
Registers (II)

- There are **16** 64-bit (`%rax` ... `%rsp`, `%r8` ... `%r15`) general-purpose registers;
  - The **low-order 32, 16, and 8 bits** of each register **can be accessed** independently under other names.
- Almost any register can be used **to hold operands** for almost any logical and arithmetic operation, but **some have special or restricted uses**.
  - Register `%rsp` is reserved as the **stack pointer**.
  - Register `%rbp` is used as a frame pointer, i.e., **the base of the current stack frame**.
  - A few other instructions make implicit use of certain registers;
    - For example, the integer multiply and divide instructions require the `%rax` and `%rdx`.
  - `%rax` register is used for specifying the **syscall number** and it is also used **for storing the return value of functions**.
  - Calling functions: The **first six arguments of a function** are stored as follows: `%rdi`(arg0), `%rsi`(arg1), `%rdx`(arg2), `%rcx` (arg3), `%r8` (arg4), and `%r9` (arg5).
    - If there are more arguments than six they are stored on the stack.

Registers (III)

- The instruction pointer register (`%rip`) **points to the next instruction to be executed**;
  - **It cannot be directly accessed by the programmer**, but is heavily used as the base for **position-independent code addressing**.
    - It is security feature.
    - It specifically supports **position-independent executables** (PIE), which are programs that work independently of where their code is loaded into memory
    - In a PIE, the operating system loads the program at varying locations: every time it runs, the program's functions and global variables have different addresses. This makes the program harder to attack (though not impossible)
    - Therefore, global variables are referenced relatively to the current value of the program counter (the `%rip` register in x86-64)

- When the ALU performs some operations, **it flags the results of these operations** in a special 64-bits register called it `%rflags`.



`%rflags` (II)

- **Carry Flag(CF)**:
  - Set if the **arithmetic operation generates a carry or a borrow out** of the most significant bit of the result;
    - Cleared otherwise.
  - The flag indicates **an overflow condition for unsigned-integer arithmetic**.
- **Zero Flag (ZF)**:
  - Set if the **result is zero**;
    - Cleared otherwise.
- **Sign Flag (SF)**:
  - Set **equal to the most significant bit of the result**, the sign bit for a signed integer.
    - 0 indicates a positive value;
    - 1 a negative value.
- **Overflow Flag (OF)**:
  - Set if the integer **result is too large** excluding the sign-bit **to fit in the destination operand**.
    - Cleared otherwise.

Carry Flag

- The rules for turning on the carry flag in binary/integer math are two:
  1. The carry flag is set if the addition of two numbers causes a carry out of the most significant (leftmost) bits added.
     ```
     1111 + 0001 = 0000 (carry flag is turned on)
     ```
  2. The carry (borrow) flag is also set if the subtraction of two numbers requires a borrow into the most significant (leftmost) bits subtracted.
     ```
     0000 - 0001 = 1111 (carry flag is turned on)
     ```
- Otherwise, the carry flag is turned off (zero).
  ```
  0111 + 0001 = 1000 (carry flag is turned off [zero])
  1000 - 0001 = 0111 (carry flag is turned off [zero])
  ```
- In unsigned arithmetic, watch the carry flag to detect errors.
- In signed arithmetic, the carry flag tells you nothing interesting.

Overflow Flag

- The rules for turning on the overflow flag in binary/integer math are two:
  1. If the sum of two numbers with the sign bits off yields a result number with the sign bit on, the "overflow" flag is turned on.

     ```
     0100 + 0100 = 1000 (overflow flag is turned on)
     ```

  2. If the sum of two numbers with the sign bits on yields a result number with the sign bit off, the "overflow" flag is turned on.

     ```
     1000 + 1000 = 0000 (overflow flag is turned on)
     ```

- Otherwise, the overflow flag is turned off (zero).

  ```
  0100 + 0001 = 0101 (overflow flag is turned off)
  0110 + 1001 = 1111 (overflow flag is turned off)
  1000 + 0001 = 1001 (overflow flag is turned off)
  1100 + 1100 = 1000 (overflow flag is turned off)
  ```

- In signed arithmetic, watch the overflow flag to detect errors.
- In unsigned arithmetic, the overflow flag tells you nothing interesting.
  - Mixed-sign addition never turns on the overflow flag.

Instruction Size Specifiers

- The x86-64 registers, memory and operations use the following data types (among others):

Data type	Suffix	Size (nr bytes)
byte	b	1
word	w	2
double (or long) word	l	4
quad word	q	8

- The **suffix** is used by the GNU assembler (gas) to specify appropriately-sized variants of instructions.

Data Types

- Assembly language defines intrinsic data types, each of which describes a set of values that can be assigned to variables and expressions of the given type.
- The essential characteristic of each type is its **size in bits**.
  - `.octa` – 128 bits (16 bytes) integer
  - `.quad` – 64 bits (8 bytes) integer
  - `.long` – the same as `.int`
  - `.int` – 32 bits (4 bytes) integer
  - `.short` – 16 bits (2 bytes) integer
  - `.byte` – 8 bits (1 byte) integer
  - `.ascii` – string (with no automatic trailing zero byte)
  - `.asciz` – string automatically terminated by zero (The “z” stands for “zero”)
  - `.float` – floating point number (4 bytes)
  - `.double` – floating point number with double precision (8 bytes)
- The `.lcomm` and `.comm` directives allocate storage in the `.bss` section.

Statements

- Input to the assembler is a **text file**, with extension ".s", consisting of a **sequence of statements**.
- A statement consists of **tokens** separated by **whitespace** and terminated by either a newline character or a semicolon (`;`).
- **Whitespace** consists of spaces and tabs that are not contained in a string or comment.
- A statement can consist of a **comment**, started by `#`.
  - The comment is terminated by the newline that terminates the statement.
- An empty statement is one that contains nothing other than spaces, tabs, newlines or other similar characters.
  - Empty statements have no meaning to the assembler.

Labels

- A label **can be placed at the beginning of a statement**.
- During assembly, the label **is assigned the current value of the active location address** and serves as **an instruction operand**.
- A symbolic label consists of an identifier (or symbol) followed by a colon (`:`).
  - Symbolic labels must be defined only once.

Tokens (I)

- Identifiers (symbols)
  - An identifier is an arbitrarily-long **sequence of letters and digits**.
  - The first character must be a letter; the underscore (_) and the period (`.`) are considered to be letters.
  - Case is significant: uppercase and lowercase letters are different.
- Keywords
  - Keywords such as x86-64 instruction mnemonics (“opcodes”) and assembler directives are **reserved for the assembler and should not be used as identifiers**.
- Numerical Constants
  - Numbers can be integers or floating point [1].
  - Integers can be signed or unsigned, with signed integers represented in two’s complement representation.

[1] Out of scope

Tokens (II)

- Numerical Constants (cont.)
  - Can be expressed
    - Decimal integers begin with a non-zero digit followed by zero or more decimal digits (0–9).
    - Binary integers begin with “0b” or “0B” followed by zero or more binary digits (0, 1).
    - Octal integers begin with zero (0) followed by zero or more octal digits (0–7).
    - Hexadecimal integers begin with “0x” or “0X” followed by one or more hexadecimal digits (0–9, A–F). Hexadecimal digits can be either uppercase or lowercase.
  - String Constants
    - A string constant consists of a sequence of characters enclosed in double quotes (`"`).

Directives

- Directives are commands that are part of the assembler syntax but are not related to the x64 processor instruction set.
- All assembler directives begin with a period (.).

Instructions

- The general form of an assembly instruction is: **mnemonic operands**
  - Each mnemonic **opcode** represents a CPU instruction.
  - Instructions **can have a variable number of operands** and when more that one are separated by commas ",".
    - For instructions with two operands, the first (lefthand) operand is the source operand, and the second (righthand) operand is the destination operand.
    - An operand specifies data being operated on or manipulated.
    - An operand has a type that can either be a register, a memory location, an immediate value or an address.
- The term addressing modes refers to the way in which the operand of an instruction is specified.

Addressing modes

- **Immediate mode**: the operand is specified by a `$` followed by an numerical constant value.
- **Register mode**: the operand is the name of a register.

Addressing modes (II)

- **Memory mode**: the operand is the value stored in a memory location, which is specified by an offset from the value in a register.
  - `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 `(base + index × scale + offset)`
    - 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).
  - **RIP-relative addressing**: this is new for x64 and allows accessing data tables and such in the code relative to the current instruction pointer, making position independent code easier to implement.
    - For example, we would write the address of a global value stored at location labeled `a` as `a(%rip)`, meaning that the assembler and linker should cooperate to compute the offset of a from the ultimate location of the current instruction.

Data transfer instructions

- The instructions in the `mov` class **copy their source (S) values to their destinations (D)**.
  - The source operand designates a value that is immediate (I), stored in a register (R), or stored in memory (M).
  - The destination operand designates a location that is either a register or a memory address.
  - A move instruction cannot have both operands refer to memory locations.

Instruction	**Description**
mov *S*, *D*	Move source to destination
movs *S*, *D*	Move byte to word (sign extended)
movz *S*	Move byte to word (zero extended)

Arithmetic and Logical Operations (I)

- Unary Operations

Instruction	Description
inc *D*	Increment by 1
dec *D*	Decrement by 1
neg *D*	Arithmetic negation
not *D*	Bitwise complement

- Binary Operations

Instruction	Description
add *S*, *D*	Add source to destination
sub *S*, *D*	Subtract source from destination
imul *S*, *D*	Multiply destination by source
xor *S*, *D*	Bitwise XOR destination by source
or *S*, *D*	Bitwise OR destination by source
and *S*, *D*	Bitwise AND destination by source

Arithmetic and Logical Operations (II)

- Shift Operations
  - *k* is a numeric literal

Instruction	Description
sal/shl *k*, *D*	Left shift destination by *k* bits
sar *k*, *D*	Arithmetic right shift destination by *k* bits
shr *k*, *D*	Logical right shift destination by *k* bits

- Special Arithmetic Operations

Instruction	**Description**
imulq *S*	Signed full multiply of `%rax` by *S* Result stored in `%rdx`:`%rax`
mulq *S*	Unsigned full multiply of `%rax` by *S* Result stored in `%rdx`:`%rax`
idivq *S*	Signed divide `%rdx`:`%rax` by *S* Quotient stored in `%rax` Remainder stored in `%rdx`
divq *S*	Unsigned divide `%rdx`:`%rax` by *S* Quotient stored in `%rax` Remainder stored in `%rdx`

Comparison and Test Instruction

Instruction	**Description**
cmp *S2*, *S1*	Set condition codes according to *S1* - *S2*
test *S2*, *S1*	Set condition codes according to *S1* & *S2*

Accessing Condition Codes (I)

- Conditional Set Instructions

Instruction	Description	Condition
sete / setz *D*	Set if equal/zero	ZF
setne / setnz *D*	Set if not equal/nonzero	~ZF
sets *D*	Set if negative	SF
setns *D*	Set if nonnegative	~SF
setg / setnle *D*	Set if greater (signed)	~(SF^OF)&~ZF
setge / setnl *D*	Set if greater or equal (signed)	~(SF^OF)
setl / setnge *D*	Set if less (signed)	SF^OF
setle / setng *D*	Set if less or equal	(SF^OF)|ZF
seta / setnbe *D*	Set if above (unsigned)	~CF&~ZF
setae / setnb *D*	Set if above or equal (unsigned)	~CF
setb / setnae *D*	Set if below (unsigned)	CF
setbe / setna *D*	Set if below or equal (unsigned)	CF|ZF

Accessing Condition Codes (II)

- Jump Instructions

Instruction	**Description**	**Condition**
jmp *Label*	Jump to label	
je / jz *Label*	Jump if equal/zero	ZF
jne / jnz *Label*	Jump if not equal/nonzero	~ZF
js *Label*	Jump if negative	SF
jns *Label*	Jump if nonnegative	~SF
jg / jnle *Label*	Jump if greater (signed)	~(SF^0F)&~ZF
jge / jnl *Label*	Jump if greater or equal (signed)	~(SF^0F)
jl / jnge *Label*	Jump if less (signed)	SF^0F
jle / jng *Label*	Jump if less or equal	(SF^OF)|ZF
ja / jnbe *Label*	Jump if above (unsigned)	~CF&~ZF
jae / jnb *Label*	Jump if above or equal (unsigned)	~CF
jb / jnae *Label*	Jump if below (unsigned)	CF
jbe / jna *Label*	Jump if below or equal (unsigned)	CF|ZF

Accessing Condition Codes (III)

- Conditional Move Instructions

Instruction	Description	Condition
cmove / cmovz *S*, *D*	Move if equal/zero	ZF
cmovne / cmovnz *S*, *D*	Move if not equal/nonzero	∼ZF
cmovs *S*, *D*	Move if negative	SF
cmovns *S*, *D*	Move if nonnegative	∼SF
cmovg / cmovnle *S*, *D*	Move if greater (signed)	∼(SFˆ0F)&∼ZF
cmovge / cmovnl *S*, *D*	Move if greater or equal (signed)	∼(SFˆ0F)
cmovl / cmovnge *S*, *D*	Move if less (signed)	SFˆ0F
cmovle / cmovng *S*, *D*	Move if less or equal	(SFˆOF)|ZF
cmova / cmovnbe *S*, *D*	Move if above (unsigned)	∼CF&∼ZF
cmovae / cmovnb *S*, *D*	Move if above or equal (unsigned)	∼CF
cmovb / cmovnae *S*, *D*	Move if below (unsigned)	CF
cmovbe / cmovna *S*, *D*	Move if below or equal (unsigned)	CF|ZF

Procedure Call Instruction

Instruction	Description
call *Label*	Push return address and jump to label
ret	Pop return address from stack and jump there

Other Instructions

- Stack

Instruction	Description
push *S*	Push source onto stack
pop *D*	Pop top of stack into destination

- Convert

Instruction	Description
cwtl	Convert word in `%ax` to doubleword in `%eax` (sign-extended)
cltq	Convert doubleword in `%eax` to quadword in `%rax` (sign-extended)
cqto	Convert quadword in `%rax` to octoword in `%rdx`:`%rax` (sign-extended)

The 64 bit x86 C Calling Convention

Calling Convention

- Whenever you have assembler coded function invoked from C code **some issues arise**:
  - How are parameters passed to a function?
  - Can functions overwrite the values in a register, or does the caller expect the register contents to be preserved?
  - Where should local variables in a function be stored?
  - How should results be returned from functions?
- The C calling convention is based heavily on the use of the hardware-supported **stack**.
  - To understand the C calling convention, you should first make sure that you fully understand the `push`, `pop`, `call`, and `ret` instructions
- A **caller** is a function that calls another function;
- A **callee** is a function that was called.

63	31		15 8	7 0	
%rax	%eax	%ax	%ah	%al	Return value
%rbx	%ebx	%bx	%bh	%bl	Callee saved
%rcx	%ecx	%cx	%ch	%cl	4th argument
%rdx	%edx	%dx	%dh	%dl	3rd argument
%rsi	%esi	%si		%sil	2nd argument
%rdi	%edi	%di		%dil	1st argument
%rbp	%ebp	%bp		%bpl	Callee saved
%rsp	%esp	%sp		%spl	Stack pointer
%r8	%r8d	%r8w		%r8b	5th argument
%r9	%r9d	%r9w		%r9b	6th argument
%r10	%r10d	%r10w		%r10b	Caller saved
%r11	%r11d	%r11w		%r11b	Caller saved
%r12	%r12d	%r12w		%r12b	Callee saved
%r13	%r13d	%r13w		%r13b	Callee saved
%r14	%r14d	%r14w		%r14b	Callee saved
%r15	%r15d	%r15w		%r15b	Callee saved

The Caller's Rules (I)

1. Before calling a function, the caller should save the contents of certain registers that are designated **caller-saved**.
   - The caller-saved registers are `%r10`, `%r11`, and any registers that parameters are put into.
     - If you want the contents of these registers **to be preserved across the function call, push them onto the stack**.
2. To pass parameters to the function, we put up to six of them into registers (in order: `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`).
   - If there are **more than six parameters** to the function, then **push the rest onto the stack in reverse order** (i.e. last parameter first)
     - Since the stack grows down, the first of the extra parameters (really the seventh parameter) parameter will be stored at the lowest address (this inversion of parameters was historically used to allow functions to be passed a variable number of parameters).

The Caller's Rules (II)

3. To call the function, use the `call` instruction.
   - This instruction places the **return address on top of the parameters on the stack**, and **branches to the subroutine code**.
4. After the function returns, (i.e. immediately following the call instruction) the caller must remove any additional parameters (beyond the six stored in registers) from stack.
   - This restores the stack to its state before the call was performed.
5. The caller can expect to find the **return value of the function** in the register `%rax`.
6. The caller restores the contents of caller-saved registers (`%r10`, `%r11`, and any in the parameter passing registers) by **popping them off of the stack**.
   - The caller can assume that no other registers were modified by the function.

The Callee's Rules (I)

1. Allocate **local variables by using registers or making space on the stack**.
   - Recall, the stack grows down, so to make space on the top of the stack, the stack pointer (`%rsp`) should be decremented.
     - The amount by which the stack pointer is decremented depends on the number of local variables needed
2. The values of any registers that are designated **callee-saved that will be used by the function must be saved**.
   - To save registers, **push them onto the stack**.
     - The callee-saved registers are `%rbx`, `%rbp`, and `%r12` through `%r15` (`%rsp` will also be preserved by the call convention, but need not be pushed on the stack during this step).
3. When the function is done, the **return value for the function** should be placed in `%rax` if it is not already there.

The Callee's Rules (II)

4. The function must **restore the old values of any callee-saved registers** (`%rbx`, `%rbp`, and `%r12` through `%r15`) that were modified.
   - The register contents are restored **by popping them from the stack**.
     - Note, the registers should be popped in the inverse order that they were pushed.
5. Next, we deallocate local variables.
   - The easiest way to do this is to add to `%rsp` the same amount that was subtracted from it in step 1.
6. Finally, we return to the caller by executing a `ret` instruction.
   - This instruction will find and remove the appropriate **return address from the stack**, pushed by `call` instruction