Campus de Gualtar
4710-057 Braga

UNIVERSIDADE DO MINHO
ESCOLA DE ENGENHARIA

Departamento de
Informática

Introdução aos Sistemas de Computação

Notas de estudo

Alterações à 1ª versão

Alberto José Proença

2002/03

Nota introdutória

Este documento é um texto de apoio ao funcionamento de disciplinas na área da Arquitectura de Computadores, complementando apenas a bibliografia básica recomendada. Não pretendendo substitui-la, adapta e resume alguns aspectos considerados essenciais durante a leccionação da matéria; baseia-se em documentos de anos anteriores e integra e complementa excertos de documentos de livros recomendados:

- "*Computer Organization and Architecture - Designing for Performance*", 6th Ed., W. Stallings, Prentice Hall, 2002 (mais informação em http://williamstallings.com/COA6e.html); adiante referido por COA
- "*Computer Systems: A Programmer's Perspective*", Randal Bryant and David O'Hallaron, Prentice Hall, 2003 (mais informação em http://csapp.cs.cmu.edu/); adiante referido por CSAPP
- *Structured Computer Organization*, 4th Ed., Andrew S. Tanenbaum, Prentice Hall, 1999; adiante referido por SCO

Índice

1. Representação da informação num computador **2**

1.1 *Information is Bits in Context* 5

2. Estrutura interna dum computador **7**

2.1 A origem do computador "moderno" 10

2.2 A hierarquia duma arquitectura de barramentos 11

2.3 *Hardware Organization of a System* 12

3. Níveis de abstracção num computador e mecanismos de conversão **14**

3.1 *Programs are Translated by Other Programs into Different Forms* 15

3.2 *Processors Read and Interpret Instructions Stored in Memory* 17

3.3 *Running the* `hello` *Program* 18

4. Execução de instruções num computador **20**

4.1 Acessos à memória na execução de instruções 21

4.2 *Accessing Main Memory* 22

4.3 *Instruction-Level Parallelism* 24

4.4 *Caches Matter* 26

4.5 *Storage Devices Form a Hierarchy* 27

5. Análise do nível ISA (*Instruction Set Architecture*) 28

5.1 Operações num processador 28

5.2 Formato de instruções em linguagem máquina 29

5.3 Tipos de instruções presentes num processador 30

5.4 Registos visíveis ao programador 30

5.5 Modos de acesso aos operandos 32

5.6 Instruções de *input/output* 33

5.7 Caracterização das arquitecturas RISC (resumo) 33

Anexo A: Representação de inteiros **A1**

Anexo B: Representação de reais em vírgula flutuante **B1**

Anexo C: Arquitectura e conjunto de instruções do IA32 **C1**

4.2 Acessos à memória na execução de instruções

Durante a execução de cada instrução, o CPU necessita de aceder à memória para ir buscar a instrução (*Fetch*), e o(s) operando(s) sempre que estes estejam em memória. O processo de aceder à memória para ir buscar uma instrução é idêntico ao de buscar um operando para efectuar uma operação (apresentado com detalhe na próxima sub-secção); i.e., o CPU coloca o conteúdo do IP/PC no barramento de endereços, activa o sinal de leitura à memória no barramento de controlo, e o conteúdo da(s) célula(s) de memória indicada(s) no barramento de endereços é colocado no barramento de dados, de modo que o CPU o possa ler e colocar no registo de instrução (IR).

A sub-secção seguinte apresenta com mais detalhe os acessos à memória nas operações de leitura e escrita de dados na memória.

4.3 Accessing Main Memory (*retirado de CSAPP*)

Data flows back and forth between the processor and the DRAM main memory over shared electrical conduits called *buses*. Each transfer of data between the CPU and memory is accomplished with a series of steps called a *bus transaction*. A *read transaction* transfers data from the main memory to the CPU. A *write transaction* transfers data from the CPU to the main memory.

A *bus* is a collection of parallel wires that carry address, data, and control signals. Depending on the particular bus design, data and address signals can share the same set of wires, or they can use different sets. Also, more than two devices can share the same bus. The control wires carry signals that synchronize the transaction and identify what kind of transaction is currently being performed. For example, is this transaction of interest to the main memory, or to some other I/O device such as a disk controller? Is the transaction a read or a write? Is the information on the bus an address or a data item?

Figure 6.6 shows the configuration of a typical desktop system. The main components are the CPU chip, a chipset that we will call an *I/O bridge* (which includes the memory controller), and the DRAM memory modules that comprise main memory. These components are connected by a pair of buses: a *system bus* that connects the CPU to the I/O bridge, and a *memory bus* that connects the I/O bridge to the main memory.

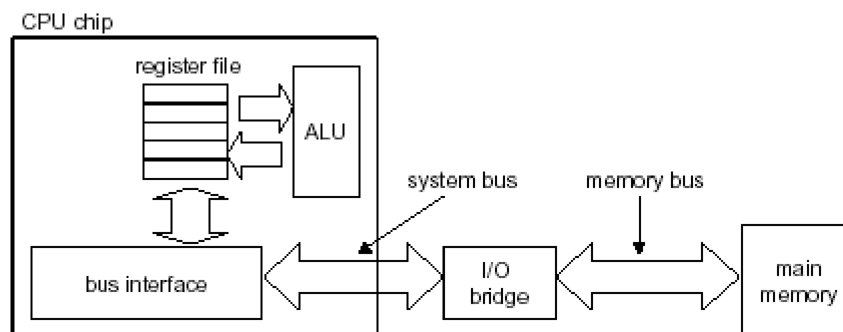


Figure 6.6: Typical bus structure that connects the CPU and main memory.

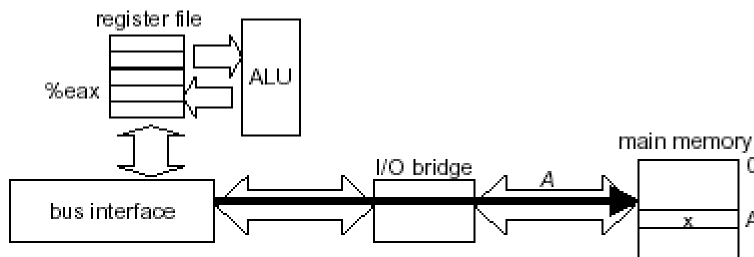
The I/O bridge translates the electrical signals of the system bus into the electrical signals of the memory bus. As we will see, the I/O bridge also connects the system bus and memory bus to an I/O bus that is shared by I/O devices such as disks and graphics cards. For now, though, we will focus on the memory bus.

Consider what happens when the CPU performs a load operation such as

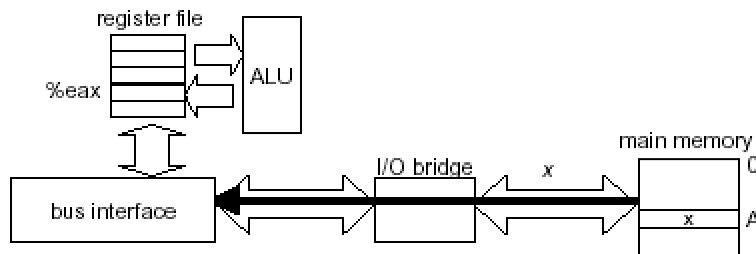
```
movl A,%eax
```

where the contents of address `A` are loaded into register `%eax`. Circuitry on the CPU chip called the *bus interface* initiates a read transaction on the bus.

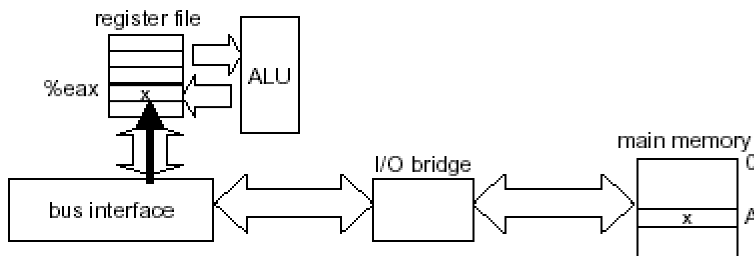
The read transaction consists of three steps. First, the CPU places the address `A` on the system bus[1]. The I/O bridge passes the signal along to the memory bus[2] (Figure 6.7(a)). Next, the main memory senses the address signal on the memory bus[3], reads the address from the memory bus, fetches the data word from the DRAM, and writes the data to the memory bus[4]. The I/O bridge translates the memory bus signal into a system bus signal, and passes it along to the system bus (Figure 6.7(b)). Finally, the CPU senses the data on the system bus, reads it from the bus, and copies it to register `%eax` (Figure 6.7(c)).



(a) CPU places address A on the memory bus.



(b) Main memory reads A from the bus, retrieves word x, and places it on the bus.



(c) CPU reads word x from the bus, and copies it into register `%eax`.

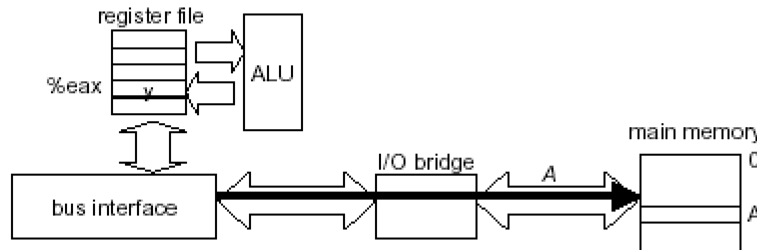
Figure 6.7: Memory read transaction for a load operation: `movl A,%eax`.

Conversely, when the CPU performs a store instruction such as

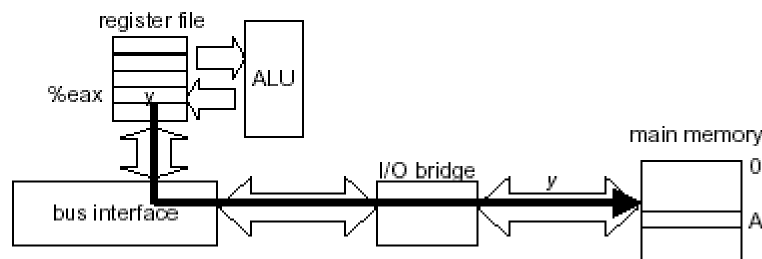
```
movl %eax,A
```

[1] Mais concretamente, no barramento de endereços, *Address Bus*
[2] Idem
[3] Em conjunto com o sinal de controlo, indicando que a operação é de leitura da memória
[4] Neste caso, no barramento de dados, *Data Bus*

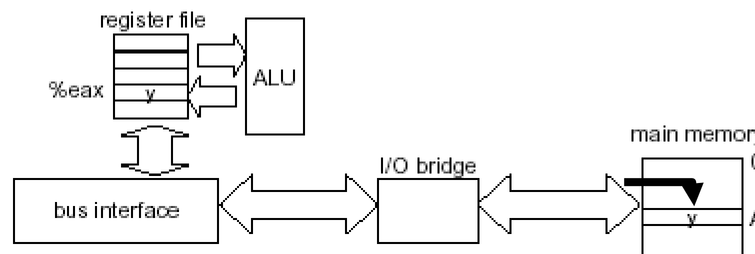
where the contents of register `%eax` are written to address `A`, the CPU initiates a write transaction. Again, there are three basic steps. First, the CPU places the address on the system bus. The memory reads the address from the memory bus and waits for the data to arrive (Figure 6.8(a)). Next, the CPU copies the data word in `%eax` to the system bus (Figure 6.8(b)). Finally, the main memory reads the data word from the memory bus and stores the bits in the DRAM (Figure 6.8(c)).



(a) CPU places address A on the memory bus. Main memory reads it and waits for the data word.



(b) CPU places data word y on the bus.



(c) Main memory reads data word y from the bus and stores it at address A.

Figure 6.8: Memory write transaction for a store operation: `movl %eax,A`.

4.4 Instruction-Level Parallelism (*retirado de SCO*)

Computer architects are constantly striving to improve performance of the machines they design. Making the chips run faster by increasing their clock speed is one way, but for every new design, there is a limit to what is possible by brute force at that moment in history. Consequently, most computer architects look to parallelism (doing two or more things at once) a way to get even more performance for a given clock speed.

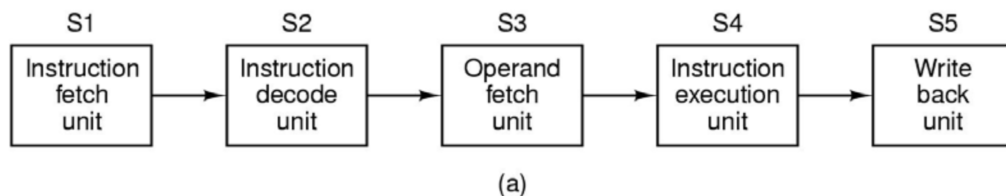
Parallelism comes in two general forms: instruction-level parallelism and processor level parallelism. In the former, parallelism is exploited within individual instructions to get more instructions/sec out of the machine. In the latter, multiple CPUs work together on the same problem. Each approach has its own merits. In this section we will look at instruction-level parallelism.

Pipelining

It has been known for years that the actual fetching of instructions from memory is a major bottleneck in instruction execution speed. To alleviate this problem, computers going back at least as far as the IBM Stretch (1959) have had the ability to fetch instructions from memory in advance, so they would be there when they were needed. These instructions were stored in a set of registers called the **prefetch buffer**[5]. This way, when an instruction was needed, it could usually be taken from the prefetch buffer rather than waiting for a memory read to complete.

In effect, prefetching divides instruction execution into two parts: fetching and actual execution. The concept of a **pipeline** carries this strategy much further. Instead of dividing instruction execution into only two parts, it is often divided into many parts, each one handled by a dedicated piece of hardware, all of which can run in parallel.

Fig. 2-4(a) illustrates a pipeline with five units, also called stages. Stage 1 fetches the instruction from memory and places it in a buffer until it is needed. Stage 2 decodes the instruction, determining its type and what operands it needs. Stage 3 locates and fetches the operands, either from registers or from memory. Stage 4 actually does the work of carrying out the instruction, typically by running the operands through the data path of the CPU. Finally, stage 5 writes the result back to the proper register.



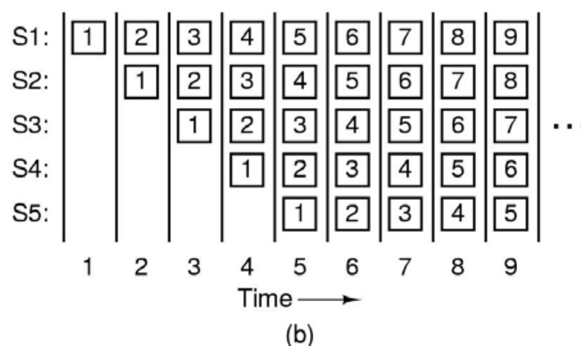


Figure 2-4. (a) A five stage pipeline. (b) The state of each stage as a function of time. Nine clock cycles are illustrated.

In Fig. 2.4(b) we show how the pipeline operates as a function of time. During clock cycle 1, stage S1 is working on instruction 1, fetching it from memory. During cycle 2, stage S2 decodes instruction 1, while stage S1 fetches instruction 2. During cycle 3, stage S3 fetches the operands for instruction 1, stage S2 decodes instruction, and stage S3 fetches the third instruction. (…) Finally, during cycle 5, S5 writes the result of instruction 1 back, while the other stages work on the following instructions.

Suppose that the cycle time of this machine is 2 nsec[6]. Then it takes 10 nsec for an instruction to progress all the way through the five-stage pipeline. At first glance, with an instruction taking 10 nsec, it might appear that the machine can run at 100 MIPS[7], but in fact it does much better than this. At every

---

[5] O CPU dos primeiros PC's, o Intel 8088, também tinha um *prefetch buffer* com 6 bytes.
[6] O que corresponde à utilização de um *clock* com uma frequência de 500MHz
[7] ***M**illions of **I**nstructions **P**er **S**econd*

clock cycle (2 nsec), one new instruction is completed, so the actual rate of processing is 500 MIPS, not 100 MIPS[8].

Superscalar Architectures

If one pipeline is good, then surely two pipelines are better. One possible design for a dual pipeline CPU, based on Figure 2-4, is shown in Fig. 2-5. Here a single instruction fetch unit fetches pairs of instructions together and puts each one into its own pipeline, complete with its own ALU for parallel operation. To be able to run in parallel, the two instructions must not conflict over resource usage (e.g., registers), and neither must depend on the result of the other. As with a single pipeline, either the compiler must guarantee this situation to hold (i.e., the hardware does not check ands gives incorrect results if the instructions are not compatible), or conflicts are detected and eliminates during execution using extra hardware.

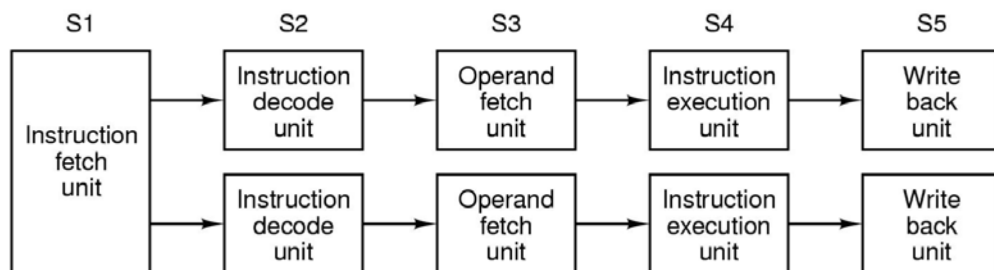


Figure 2-5. Dual five-stage pipelines with a common instruction fetch unit.

Although pipelines, single or double, are mostly used on RISC[9] machines (the 386 and its predecessors did not have any), starting with 486 Intel began introducing pipelines into its CPUs. The 486 had one pipeline and the Pentium had two five-stages pipelines roughly as Fig. 2-5, although the exact division of work between stages 2 and 3 (called decode-1 and decode-2) was slightly different than in our example. The main pipeline, called the **u pipeline**, could execute a arbitrary Pentium instruction. The second pipeline, called the **v pipeline**, could execute only simple integer instructions.

Complex rules determined whether a pair of instructions were compatible so they could be executed in parallel. If the instructions in a pair were not simple enough or incompatible, only the first one was executed (in the u pipeline). The second one was then held and paired with the instruction following it. Instructions were always executed in order.

---

[8]Nota: o aumento do desempenho muito raramente é proporcional ao nº de níveis de *pipeline*, pois nem sempre se consegue manter o *pipeline* cheio; por ex., quando uma instrução precisa do resultado da anterior, ou sempre que há instruções de salto, o *pipeline* poderá ter de ser empatado.

[9] *Reduced Instruction Set Computer*, conceito a ser detalhado adiante

Anexo C: Arquitectura e conjunto de instruções do IA32

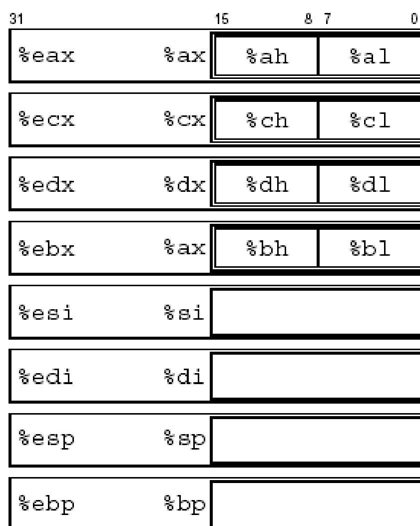


Ilustração 1- Conjunto de registos inteiros do IA32 (notação Linux)

Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$Imm\,(E_b)$	$M[Imm + R[E_b]]$	Base + displacement
Memory	(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$Imm\,(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm\,(, E_i, s)$	$M[Imm + R[E_i] \cdot s]$	Scaled Indexed
Memory	(E_b, E_i, s)	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm\,(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

Ilustração 2 - Modos de endereçamento IA32: imediato, registo e valores em memória. O factor de escala s pode tomar o valor 1, 2, 4 ou 8; *Imm* pode ser uma constante de 0, 8, 16 ou 32 *bits*. O modo de endereçamento em memória reduz-se à forma *Imm*(Eb, Ei, s), em que alguns dos campos podem não estar presentes.

Tabela 1- Códigos de condições (flags) . Descrevem atributos da última operação lógica ou aritmética realizada. Usadas para realizar saltos condicionais.

Símbolo	Nome	Descrição
CF	Carry Flag	A última operação gerou transporte.
ZF	Zero Flag	A última operação teve resultado zero
SF	Sign Flag	A última operação teve resultado negativo
OF	Overflow Flag	A última operação causou overflow em complemento para dois.

Tipo	Instrução	Efeito	Descrição
Transferência de Informação	mov? S, D	D←S	Move (? = b,w,l)
	movsbl S, D	D←SignExtend(S)	Move Sign-Extended Byte
	movzbl S, D	D←ZeroExtend(S)	Move Zero-Extended Byte
	pushl S	%esp ← %esp - 4; Mem[%esp] ← S	Push
	popl D	D←Mem[%esp]; %esp ←%esp+ 4	Pop
	leal S, D	D← &S	Load Effective Address
Operações Aritméticas e Lógicas	incl D	D← D +1	Increment
	decl D	D← D –1	Decrement
	negl D	D← -D	Negate
	notl D	D← ~D	Complement
	addl S, D	D← D + S	Add
	subl S, D	D← D – S	Subtract
	imull S, D	D← D * S	32 bit Multiply
	xorl S, D	D← D ^ S	Exclusive-Or
	orl S, D	D← D | S	Or
	andl S, D	D← D & S	And
	sall k, D	D← D << k	Left Shift
	shll k, D	D← D << k	Left Shift
	sarl k, D	D← D >> k	Arithmetic Right Shift
	shrl k, D	D← D >> k	Logical Right Shift
	imull S	%edx : %eax ← S × %eax	Signed 64 bit Multiply
	mull S	%edx : %eax ← S × %eax	Unsigned 64 bit Multiply
	cltd	%edx : %eax ← SignExtend(%eax)	Convert to Quad Word
	idivl S	%edx ← %edx : %eax mod S; %eax ← %edx:%eax ÷ S	Signed Divide
	divl S	%edx ← %edx : %eax mod S; %eax ← %edx : %eax ÷ S	Unsigned Divide
Teste	cmp? S2, S1	(CF, ZF, SF, OF) ← S1 – S2	Compare (? = b,w,l)
	test? S2, S1	(CF, ZF, SF, OF) ← S1 & S2	Test (? = b,w,l)
Instruções de set	sete R8	R8 ← ZF (Sinónimo: setz R8)	Equal/Zero
	setne R8	R8 ← ~ZF (Sinónimo: setnz R8)	Not Equal/Not Zero
	sets R8	R8 ← SF	Negative
	setns R8	R8 ← ~SF	Non Negative
	setg R8	R8 ← ~(SF^OF) & ~ZF (Sinónimo: setnle R8)	Greater (signed >)
	setge R8	R8 ← ~(SF^OF) (Sinónimo: setnl R8)	Greater or equal (signed >=)
	setl R8	R8 ← SF^OF (Sinónimo: setnge R8)	Less (signed <)
	setle R8	R8 ← (SF^OF) | ZF (Sinónimo: setng R8)	Less or equal (signed <=)
	seta R8	R8 ← ~CF & ~ZF (Sinónimo: setnbe R8)	Above (unsigned >)
	setae R8	R8 ← ~CF (Sinónimo: setnb R8)	Above or equal (unsigned >=)
	setb R8	R8 ← CF (Sinónimo: setnae R8)	Below (unsigned <)
	setbe R8	R8 ← CF & ~ZF (Sinónimo: setna R8)	Below or equal (unsigned <=)
Instruções de salto	jmp Label	%eip ← Label	Unconditional jump
	jmp *D	%eip ← *D	Indirect unconditional jump
	je Label	Jump if ZF (Sinónimo: jz)	Zero/Equal
	jne Label	Jump if ~ZF (Sinónimo: jnz)	Not Zero/Not Equal
	js Label	Jump if SF	Negative
	jns Label	Jump if ~SF	Not Negative
	jg Label	Jump if ~(SF ^OF) & ~ZF (Sinónimo: jnle)	Greater (signed >)
	jge Label	Jump if ~(SF^OF) (Sinónimo: jnl)	Greater or equal (signed >=)
	jl Label	Jump if SF^OF (Sinónimo: jnge)	Less (signed <)
	jle Label	Jump if (SF^OF) | ZF (Sinónimo: jng)	Less or equal (signed <=)
	ja Label	Jump if ~CF & ~ZF (Sinónimo: jnbe)	Above (unsigned >)
	jae Label	Jump if ~CF (Sinónimo: jnb)	Above or equal (unsigned >=)
	jb Label	Jump if CF (Sinónimo: jnae)	Below (unsigned <)
	jbe Label	Jump if CF & ~ZF (Sinónimo: jna)	Below or equal (unsigned <=)
Invocação de Procedimentos	call Label	pushl %eip; %eip= Label	Procedure call
	call *Op	pushl %eip; %eip= *Op	Procedure call
	ret	popl %eip	Procedure return
	leave	movl %ebp, %esp; pop %ebp	Prepare stack for return

D – destino [Reg | Mem] **S** – fonte [Imm | Reg | Mem] **R_8** – destino Reg 8 *bits*

D e **S** não podem ser ambos operandos em memória