

*It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years.*

*John Von Neumann*



universidade de aveiro  
teoria poesisis praxis



2

# ARQUITETURA DE COMPUTADORES I

## **Atenção!**

---

Todo o conteúdo deste documento pode conter alguns erros de sintaxe, científicos, entre outros... **Não estudes apenas a partir desta fonte.** Este documento apenas serve de apoio à leitura de outros livros, tendo nele contido todo o programa da disciplina de Arquitetura de Computadores I, tal como foi lecionada, no ano letivo de 2014/2015, na Universidade de Aveiro. Este documento foi realizado por Rui Lopes.

---

mais informações em [ruieduardofalopes.wix.com/apontamentos](http://ruieduardofalopes.wix.com/apontamentos)

Através dos conhecimentos obtidos em a1s1 - Introdução aos Sistemas Digitais e a1s2 - Arquitetura e Sistemas Operativos, esta disciplina é um aglomerado de conhecimentos acerca da área de arquitetura de computadores. Pretende-se assim, com o seu estudo, que se comprehenda a organização dos computadores digitais, que se adquira familiaridade com a arquitetura de microprocessadores através da programação em Assembly, que se comprehenda a estrutura interna dos processadores e que se conheçam as formas de representação da informação nos computadores digitais, com relevo para a representação da informação numérica (inteiros e vírgula flutuante) e as operações aritméticas básicas.

## 1. Introdução à Arquitetura de Computadores

Todos nós, hoje em dia, temos acesso constante a vários tipos e formas de computadores, quer seja no trabalho, em casa ou até mesmo na rua. Usando tanto computadores de secretaria como telemóveis, ambos conseguem realizar os serviços que temos necessidade de cumprir no dado instante de tempo. Mas já devemos saber que os telemóveis não partilham da mesma arquitetura que os computadores - enquanto que uns podem ser ARM, os outros podem ser Intel x86. Mas o que é que isso interfere com a garantia e funcionamento da mesma aplicação tanto no dispositivo móvel como no computador? Acontece que pelos dispositivos terem arquiteturas diferentes, estes também não deverão suportar todos os programas, mas antes programas que tenham sido compilados para uma linguagem máquina que o nosso dispositivo seja capaz de interpretar. Os processadores é que sustêm essas vertentes tecnológicas e são eles quem acabam por interpretar dados e convertê-los em informação, daí que seja importante estudar a arquitetura do computador, percebendo inicialmente os detalhes de um componente de processamento de dados, como o CPU, e só depois perceber toda a envolvente periférica a este.

### Modelo de von Neumann

Até ao ano de 1945, a conceito de processamento e de arquitetura baseava-se muito num **modelo de computação** muito primitivo. Com os poucos programas que eram construídos até à data, a programação era sempre realizada no momento e através de cabos, pelo que a informação que saía processada das aplicações que eram realizadas eram todas salvaguardadas em memórias.

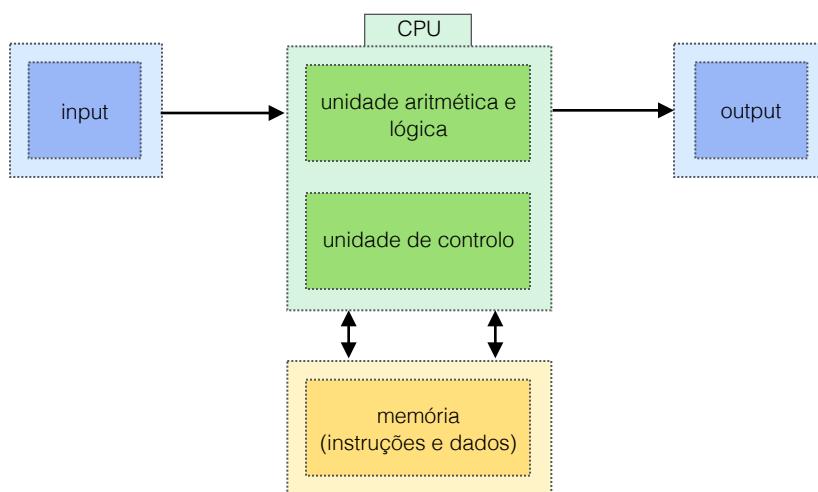
Em 1945, um matemático húngaro, John von Neumann (1903-1957), em *First Draft of a Report on the EDVAC*, escreveu uma nova teoria sobre um novo modelo computacional, inovador, que poderia preservar a informação acerca das instruções executadas pelos programas que eram realizados. Assim, ele criou o conceito de um computador digital eletrónico cujos componentes consistiam numa unidade de processamento que contém uma unidade aritmética e lógica e registos, uma unidade de controlo contendo um registo de instruções (em inglês *instruction register*) e um contador de programa (*program counter*), uma memória onde se guardasse tanto informações/dados, como instruções de execução, uma unidade de armazenamento em massa e periféricos de entrada/saída. Isto automaticamente criou um novo conceito de máquina, denominado de **stored-program computer**, no qual uma execução de uma instrução (*instruction fetch*) e uma operação de dados não podem ocorrer ao mesmo tempo, dado que ambas partilham o mesmo *bus*. Na Figura 1 temos uma representação esquemática do funcionamento de uma máquina baseada no modelo de von Neumann, isto é, baseada na lógica de stored-program computer.

**modelo de computação**

© John von Neumann

**stored-program computer**

# 3 ARQUITETURA DE COMPUTADORES I



Então, como já estudámos atrás, o processador tem duas componentes internas específicas - a unidade aritmética e lógica, também denominada de ALU (acrônimo de *Arithmetic and Logical Unit*), e a unidade de controlo. Mas como é que o processador funciona? Os processadores contam com um **relógio** a uma determinada frequência, que controla ciclos de execução. Esses ciclos de execução têm, em suma, dois grandes passos: primeiro, o processador lê da memória o código da instrução seguinte a executar e segundo, o processador executa a operação especificada no código da instrução a que se chama de **fetch** (executar ciclo - em inglês *execute cycle*).

**figura 1**  
modelo de computação de  
von Neumann

**relógio (clock)**

**fetch**

**ISA**

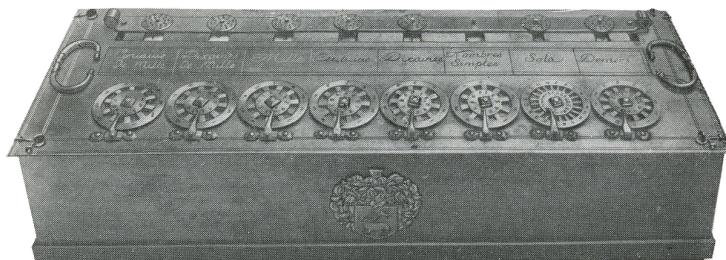
## Conjunto de instruções de uma arquitetura

Já sabemos, embora por alto, como é que um processador funciona, mas não sabemos que tipo de informação é que ele consome. Por outras palavras, ainda não sabemos responder à questão: qual o conjunto de instruções que o processador executa? Para responder a essa questão temos ter o conhecimento de uma tabela de instruções possíveis e realizáveis num processador com uma determinada arquitetura - **ISA** (acrônimo de *Instruction Set Architecture*). O ISA contém os atributos de um sistema de computação tal como visto por um programador, sendo definida uma estrutura conceptual e um comportamento funcional. No ISA estão definidos parâmetros que permitem uma organização da memória, que compreendam tipos e estruturas de dados - codificação e representação -, formatos de instrução, códigos de operação, modos de endereçamento e de acesso a dados e instruções e condições de exceção.

Até ao momento já somos capazes de concluir algo mais acerca da arquitetura de computadores, respondendo à nova questão: que fatores influenciam a arquitetura? Para responder, basta recuar no nosso estudo e relembrar que é a arquitetura interna de um processador aquela que executa o conteúdo de um programa (as suas instruções), ou seja, dois dos fatores serão a aplicação da arquitetura e as linguagens de programação. Os sistemas de operação fornecem uma organização mais estruturada para a criação de novas e gestão de correntes aplicações, sendo um outro fator. A tecnologia também é um fator que é decisivo para a arquitetura de computadores, esta, que tem evoluído com um último fator que é a história. A história do computador e de todos os seus componentes é bastante importante para a percepção do estado evolutivo da tecnologia atual e da provável tecnologia futura.

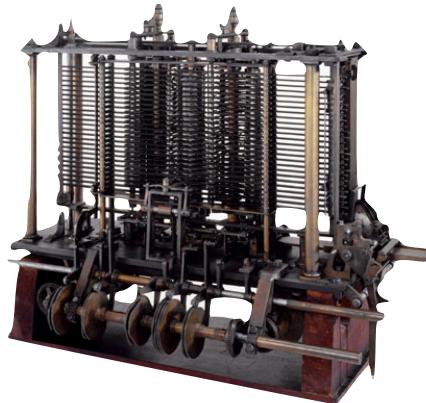
## Evolução dos computadores

A história dos computadores é algo que perdura desde o século XVII, embora a primeira máquina realmente considerada como computador tenha sido inventada no século XVIII. Mas com Blaise Pascal (1623-1662) surge uma máquina mecânica calculadora, a **Pascaline**, representada na Figura 2.



A Pascaline somava e subtraía com transportes (*carries*), tudo, num procedimento puramente mecânico.

Mais tarde, Charles Babbage (1791-1871) desenhou uma máquina, a **máquina analítica** (em inglês *analytical engine*), que nunca chegou a ser construída enquanto fora vivo, dadas muitas impossibilidades relacionadas com a tecnologia da época, que não o permitira materializar grande parte das suas peças. Muito mais tarde, através dos planos de Babbage, o sonho do fabrico da máquina foi superado, e foi criado um exemplar à escala do planeamento, representado na Figura 3.



A máquina analítica também chegou a ter programas escritos para serem executados nela, por Ada Lovelace (1815-1862), pessoa a quem foi criada a linguagem Ada por homenagem.

Já no século XX (1944), através da tecnologia eletromecânica, introduzindo o uso de **relés**, criou-se um computador que era capaz de fazer 3 adições por segundo e fazer multiplicações em 6 segundos. Conceptualizado originalmente por Howard Aiken (1900-1973), o **Harvard Mark I** não possuía a lógica do modelo de von Neumann. Os relés que ele continha foram mais tarde substituídos por transístores. A Figura 4 mostra uma imagem deste computador.

Segundo, temos o **ENIAC** (acrônimo de *Electronic Numerical Integrator And Computer*), criado pela Universidade da Pensylvania, que usava **válvulas eletrônicas**, ao invés dos transístores. Esta máquina foi desenvolvida durante a Segunda Guerra

© Blaise Pascal  
**Pascaline**

**figura 2**  
Pascaline: a calculadora de Pascal

© Charles Babbage  
**máquina analítica**

**figura 3**  
máquina analítica de Babbage

© Ada Lovelace

**relés**  
© Howard Aiken  
**Harvard Mark I**

**ENIAC**  
válvulas eletrônicas

## 5 ARQUITETURA DE COMPUTADORES I

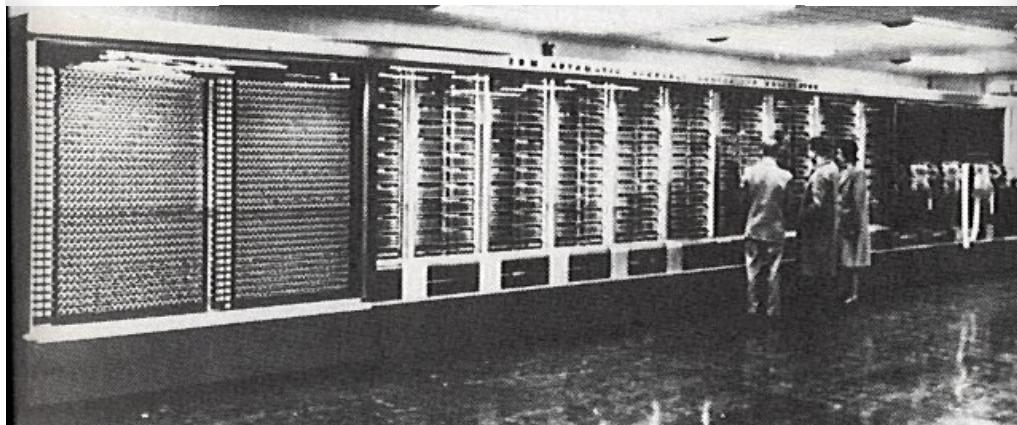


figura 4  
Harvard Mark I

Mundial e tinha como principal objetivo calcular tabelas balísticas. Constituída por válvulas eletrónicas, estas têm um funcionamento semelhante ao tubo de raios catódicos, que podemos encontrar nas nossas televisões antigas e ecrãs CRT (também chamados de cinescópios). Esta máquina tinha um custo aproximado de \$500.000. Com programação por cablagem, von Neumann analisa esta máquina, o que o faz avançar com uma nova ideia de modelo computacional. O ENIAC está representado na Figura 5.

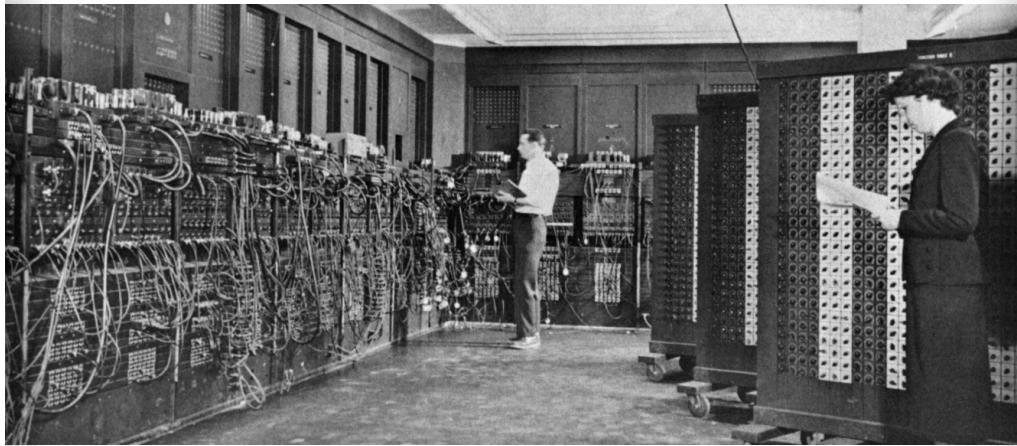


figura 5  
ENIAC

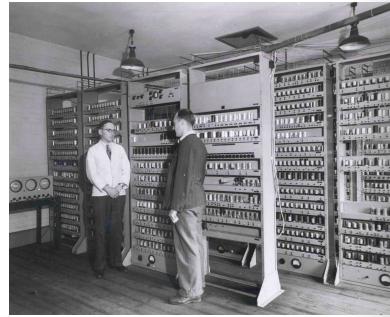
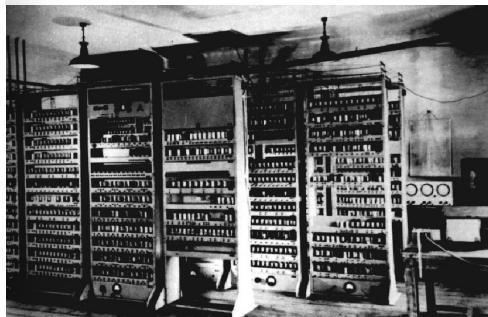
Em 1951 o **EDVAC** (acrônimo de *Electronic Discrete Variable Automatic Computer*), pouco tempo depois da criação do ENIAC, inventado por John Mauchly (1907-1980) e John Presper Eckert (1919-1995), surgiu com os mesmos fins que o ENIAC. Este foi o primeiro computador a surgir com a lógica de von Neumann de *stored-program computer*. Na Figura 6 está uma imagem do EDVAC (à esquerda) e do **EDSAC** (acrônimo de *Electronic Delay Storage Automatic Calculator*), criado por Maurice Wilkes (1913-2010), o segundo computador que usou o modelo de von Neumann (à direita).

Já nos anos '60 a empresa IBM (sigla de *International Business Machines*) lançou o **System/360**, um computador que já continha transistores, estes, criados em 1949. Não possuindo uma tecnologia SSI (sigla de *Small Scale Integration*), ou seja, não possuindo circuitos que incorporem portas lógicas, tinham SLI (sigla de *Solid Logic Technology*), um método da IBM para empacotar os circuitos eletrônicos, uma tecnologia híbrida.

• John Mauchly  
• John Prosper Eckert

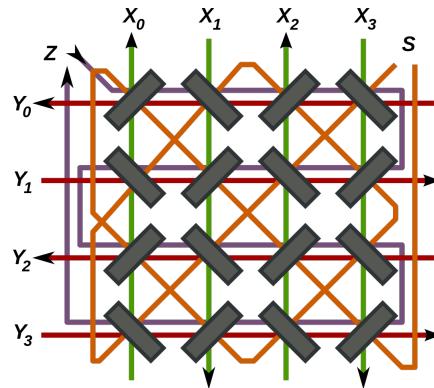
**EDSAC**  
• Maurice Wilkes

**System/360**



**figura 6**  
EDVAC (à esquerda) e  
EDSAC (à direita)

Do final da década de '50 até inícios de '70, uma nova tecnologia de memórias foi introduzida, com a lógica **RAM** (acrônimo de *Random Access Memory* - memória de acesso aleatório) que, com toros de ferrite, gravavam estados através de mudanças de polarização.



Nos toros de ferrite, se a polarização fosse positiva, então o toro preservava a informação 1, mas se a polarização fosse negativa, então o toro preservava a informação 0. A esta tecnologia dá-se o nome de **Magnetic-core Memory** (em português, memória de núcleo magnético).

Prosseguindo os anos '70 chegaram os **minicomputadores**, isto é, computadores com propósito mais pessoal e que usam circuitos integrados, quer de baixa, como de média escala de integração. Estes computadores continham memórias de semi-condutores, não competitivos com toros de ferrite.

Em 1971 nasce o primeiro **microprocessador**, pela Intel - o **Intel 4004**. Nesta mesma época nasce uma memória, com tecnologia NMOS (sigla de *N-type Metal-Oxide-Semiconductor*), que tem a capacidade de 1024 Bytes (1Kb), também desenvolvida pela Intel - a **Intel 1103**, usada na máquina HP 9800 series (calculadora com impressora).

## Evolução do software

Tal como os computadores evoluíram ao longo do tempo, os softwares tiveram que se adaptar, tanto às necessidades do utilizador, como às capacidades da máquina onde são executados. A evolução do software é passível de ser dividido em duas partes ao que chamamos de gerações.

Da primeira geração faz parte a **linguagem máquina**, também chamada de *bytecode*, que é o código binário que a máquina comprehende, e a **linguagem assembly**, uma linguagem simbólica que permite tornar o *bytecode* legível aos olhos humanos.

**figura 7**  
Magnetic-core Memory

**Magnetic-core Memory**

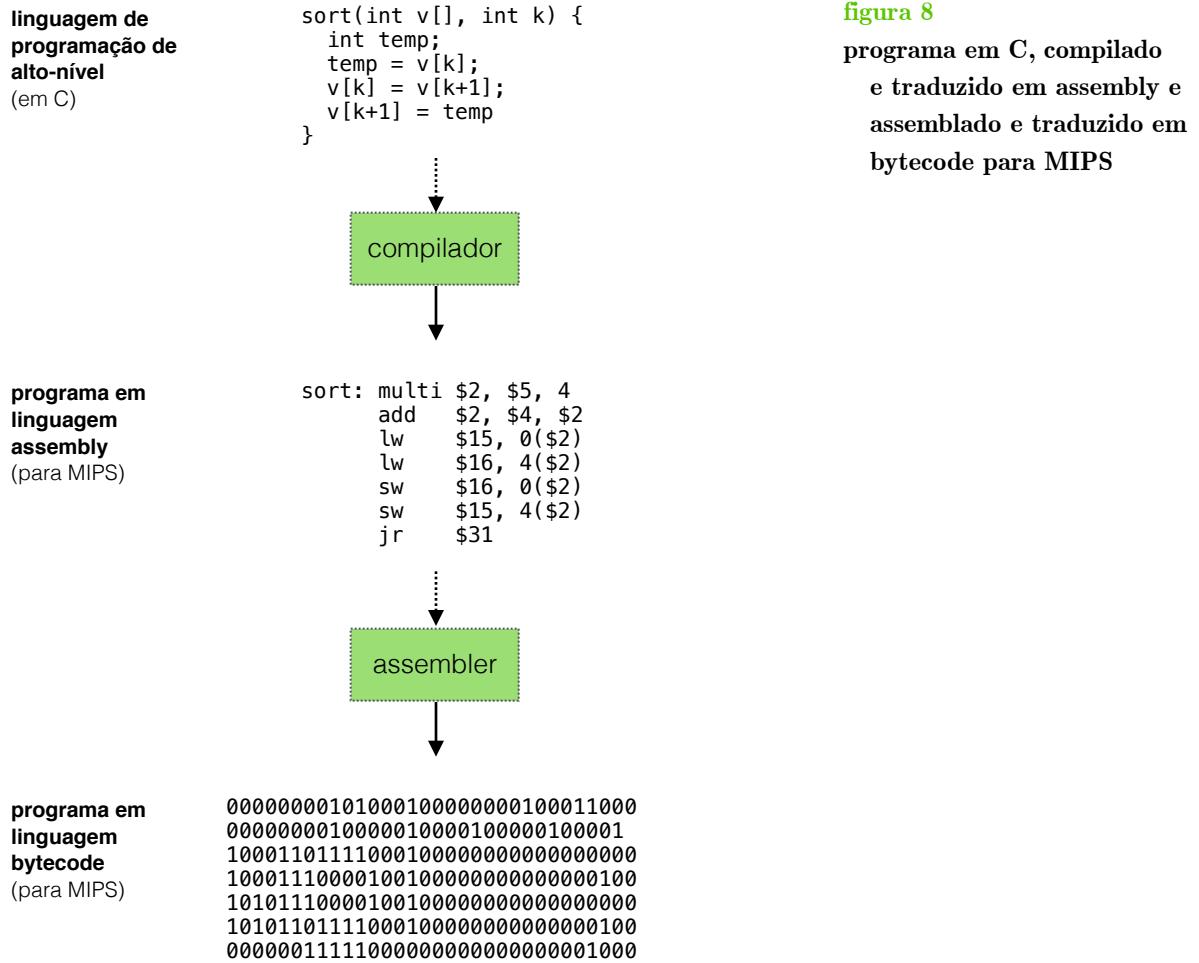
**minicomputadores**

**microprocessadores, Intel 4004**

**Intel 1103**

**linguagem máquina**  
**linguagem assembly**

Já da segunda geração fazem parte linguagens de alto-nível, como o FORTRAN (acrônimo de *Formula Translator*) para cálculo científico e COBOL (acrônimo de *Common Business-Oriented Language*) para processamento de dados administrativo e comerciais. Esta geração usava uma lógica, como a representada na Figura 8.



As **linguagens de alto-nível** têm várias vantagens como a maior produtividade, portabilidade e a conservação do investimento feito no desenvolvimento dos programas.

**linguagens de alto-nível**

## Tecnologia integrada

A partir dos anos '70 os computadores, na altura minicomputadores, já usavam a **tecnologia integrada**. Esta tecnologia baseia-se principalmente no conceito de processamento e integração de outros componentes, como a memória, a uma escala mesmo muito precisa e pequena. Mas como é que estes componentes são feitos? Tudo começa com um lingote de silício (Si) que passa por uma serra muito fina que o corta em várias fatias, a que chamamos de **bolachas**. Estas bolachas, ainda em branco, através de 20 a 40 passos intermédios de processamento, são padronizadas, ficando com um recorte em xadrez, sendo que cada quadrado é preenchido com um chip. De seguida os chips são testados e são eliminados os que apresentam danos. Depois do primeiro passo de teste, os chips são todos separados através de uma segunda secção

**tecnologia integrada**

**bolachas**

de corte. De novo, os chips passam por uma unidade de teste e são eliminados aqueles que não garantem os requisitos. Os que passaram neste último teste são empacotados, sendo encaixados e prontos para utilização. Todo este processo encontra-se descrito na Figura 9.

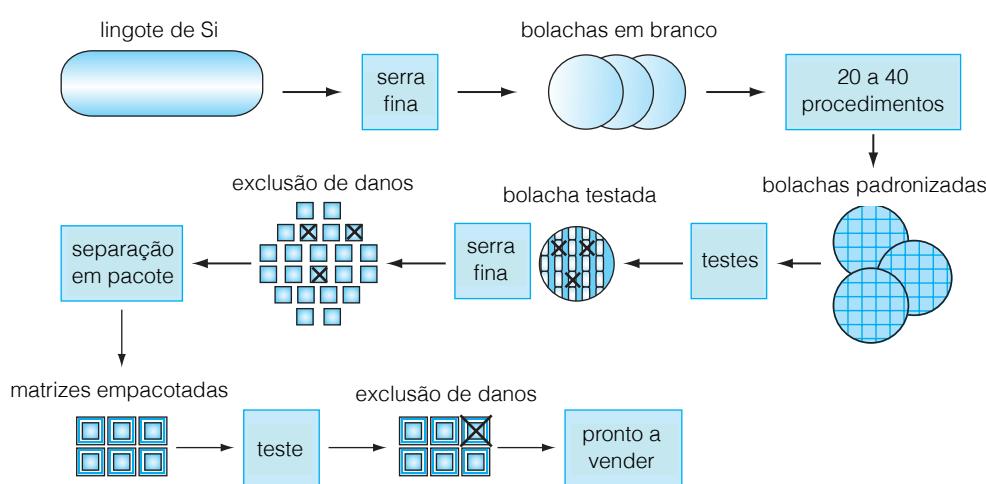


figura 9

criação de processadores  
através de um lingote de  
silício (Si)

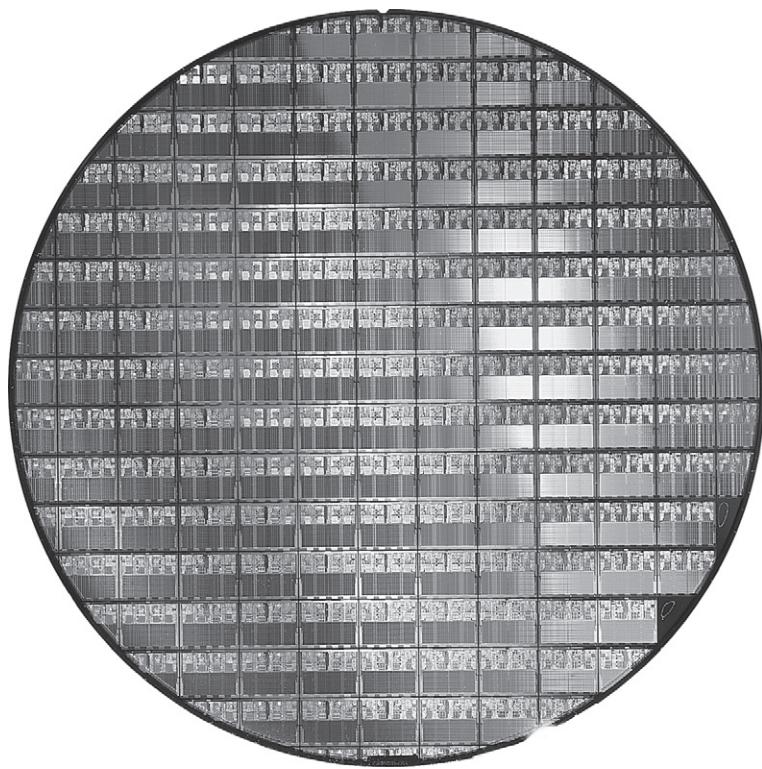


figura 10

bolacha padronizada

## 2. Introdução à Arquitetura MIPS

Nesta disciplina de Arquitetura de Computadores I (a2s1), de forma a estudar o nível de linguagem baixo no qual os processadores trabalham, precisamos de escolher uma arquitetura que nos seja capaz de mostrar grande parte das capacidades e da importância de parâmetros para a criação de linguagens de baixo nível - escolhemos assim a **arquitetura MIPS**. MIPS é um acrónimo em inglês de *Microprocessor without Interlocked Pipeline Stages* e provém de uma família de processadores do tipo RISC (acrónimo de *Reduced Instruction Set Computer*). Desenvolvido pela MIPS Technology, este processador é baseado numa arquitetura de 32 bits, embora também já exista a tecnologia de 64 bits. Nós vamos estudar a arquitetura de 32 bits.

Os processadores MIPS são relevantes a nível de estudo por duas grandes razões: é regular e é rápido. A **regularidade** do processador MIPS provém do facto de todas as instruções do ISA terem o mesmo número de bits e pelas operações aritméticas e lógicas se realizarem sempre sobre registos do processador, ao contrário de muitas outras arquiteturas que usam a memória nestes processos. A **rapidez**, por sua vez, está diretamente relacionada com este fator, dado que evitando a utilização direta da memória, os programas correm com muito mais rapidez pelo uso de registos. Mais do que esta razão, quando um operando é uma constante - muitas vezes chamado de imediato -, este deve fazer parte da instrução, dado que mais de 50 por cento das operações da ALU envolvem constantes.

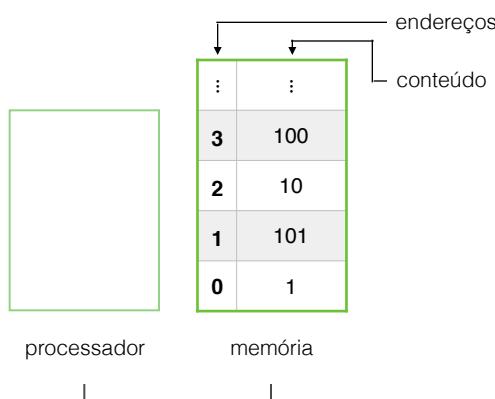
**arquitetura MIPS**

**regularidade**

**rapidez**

### Relação entre endereço e conteúdo da memória

Quando falamos em memória temos de criar uma analogia algo como um armário com várias gavetas. Cada gaveta, para ser acedida, necessita de uma legenda, de forma a que nunca nos enganemos a usá-la. Sabendo qual é a gaveta pretendida, nós podemos verificar o seu conteúdo, usando-o da forma que pretendemos. Vejamos a Figura 11.



**figura 11**  
relação entre memória e  
processador

Na Figura 11 podemos verificar que o conceito de memória é algo semelhante ao conceito de *array*, dado que são células onde podem ser armazenadas informações, entre as quais, instruções e operandos. Pelo número de células de memória endereçáveis temos o nome de **espaço de endereçamento**. À menor célula de memória dá-se o nome de *byte*, daí o nome também, a este tipo de memória, de **byte-addressable memory**, característica da maior parte das arquiteturas.

**espaço de endereçamento**

**byte-addressable memory**

O espaço de endereçamento do MIPS é de  $2^{32}$  bits, o que são cerca de 4 Gb. A arquitetura MIPS permite duas grandes operações de interação com a memória: leitura

e escrita de dados. A escrita armazena os dados na célula de memória que está indicada por um dado endereço ( $Mem[address] = data$ ). Por outro lado, a leitura obtém a informação armazenada na célula de memória cujo endereço é fornecido ( $data = Mem[address]$ ). Os dados ( $data$ ) podem ser do tipo **byte** (8 bits), **half-word** (16 bits) ou **word** (32 bits).

**byte, half-word  
word**

## Execução de instruções

Para a execução de uma instrução, há que obedecer a duas grandes fases decisivas do processador MIPS: *fetch* e *execute*. Na fase **fetch** o processador endereça a memória para ir buscar o código da instrução seguintes executar, usando o registo especial **program counter (PC)** - registo onde o CPU guarda o endereço da instrução seguinte a ser executada - e o **instruction register (IR)** - registo onde o CPU guarda o código da instrução a executar, sendo que  $IR = Mem[PC]$ . A fase **execute** é quando o processador executa a operação indicada no código de instrução.

**fetch**

No cálculo de instruções com operações aritméticas e lógicas, os operandos podem estar em dois locais: no processador ou na memória. Se estiverem na memória, é necessário ir primeiro buscar o operando e copiá-lo para um registo (que é também o caso da arquitetura Intel x86 e outras CISC, que se baseiam num processo de **von Neumann bottleneck**). No caso dos registos do processador - o acesso é mais rápido - o cálculo é direto e guardado num outro registo (isto é possível tanto no MIPS como em outras arquiteturas RISC). O MIPS tem 32 registos de 32 bits, assinalados de \$0 a \$31.

**program counter (PC)  
instruction register (IR)  
execute**

**von Neumann bottleneck**

## Conjunto de instruções do MIPS

Como já foi referido antes, o MIPS tem um conjunto de instruções com um determinado formato e codificação. Sendo assim, as instruções terão todas um tamanho fixo ou um tamanho variável? Fixo. O tipo e dimensão de dados está definido para que os inteiros sejam apresentados em 32 bits, os caracteres em byte e os reais em vírgula flutuante - aspeto que vamos estudar mais adiante -, quer de 32 ou 64 bits.

Existem várias instruções para o MIPS. Um grupo de instruções são as instruções aritméticas e lógicas (*add, sub, or, nor, ...*) que operam sobre registos. Um segundo grupo do ISA do MIPS é o das instruções de transferência de dados entre CPU e memória (*lw, lb, sw, sb, ...*) que operam sobre a memória. Por fim temos as instruções de controlo da execução de outras operações, permitindo a criação de ciclos e iterações (*bne, beq, blt, j, jal, jr, ...*).

Em Assembly, quer seja do MIPS, como de outro processador qualquer, as operações tomam um formato ligeiramente diferente, dado que esta linguagem, por exemplo, impossibilita o cálculo de uma operação com mais de dois operandos, isto porque só aceita um par de operandos de cada vez. Mais ainda, a apresentação, por exemplo, da atribuição da soma  $z = a + b$  é feita pela ordem de apresentação das variáveis, sendo que a sua representação em Assembly é a representada no Código 1.

```
add $z, $a, $b    # soma do conteúdo do registo a e b a guardar em z
```

**código 1**

**soma em Assembly MIPS**

No que respeita a este tipo de operações, diretamente presentes na ALU, temos que a sua representação é sempre da forma *operação \$registro\_de\_destino, \$registro\_operando\_1, \$registro\_operando\_2*.

```
opp $dst, $op1, $op2    # operação aritmética e lógica generalizada
```

**código 2**

**operação aritmética  
generalizada**

Mas existem exceções à regra e como já foi referido atrás, mais de metade das operações de uma ALU são constituídas por dois operandos, sendo um deles uma constante (imediato). Por essa razão, a linguagem Assembly tem uma construção de instruções que permitem o envolvimento de constantes. Como se tratam de imediatos, as operações que permitem a utilização direta de constantes terminam sempre com um *i*, como a operação *addi* (adição com imediato), *subi* (subtração com imediato) ou até mesmo *andi* (e lógico *bitwise* com imediato). Nestas operações, na sua representação, a constante é apresentada sempre no fim da operação, como no Código 3.

```
addi $z, $a, 5 # $z = $a + 5
```

A nível de operações lógicas, já várias vezes elas foram identificadas como sendo operações *bitwise*, isto é, o cálculo lógico é efetuado bit a bit. Por exemplo, as operações *and*, *or*, *xor*, *nor* são quatro possibilidades de execução em Assembly. Vejamos o Código 4 - se o registo *\$a* tiver o valor  $01010011_2$  e o registo *\$b* tiver o valor  $10011101_2$ , o registo *\$z* ficará com os seguintes valores para cada uma das operações indicadas.

```
and $z, $a, $b # $z = 000100012
or $z, $a, $b # $z = 110111112
xor $z, $a, $b # $z = 110011102
nor $z, $a, $b # $z = 001000002
```

Do mesmo modo que as operações aritméticas podem ter a intervenção de constantes imediatas, as operações lógicas também pode ter. Para tal, basta anexar um *i* ao final de cada uma das operações acima descritas e obterá o mesmo resultado (com os mesmos valores dos operandos).

```
andi $z, $a, 157 # $z = 000100012
ori $z, $a, 157 # $z = 110111112
xori $z, $a, 157 # $z = 110011102
nori $z, $a, 157 # $z = 001000002
```

O segundo grupo de instruções que indicámos atrás foi um que permite transferir dados dos registos para a memória e vice-versa. Para tal, existem as instruções *load* e *store*. Estas têm uma codificação diferente das instruções aritméticas e lógicas, tal como a sua forma.

Para poder trabalhar com a memória precisamos de saber indicar o endereço de memória pretendido para a transferência. Sendo assim, o MIPS integra uma instrução cujo endereço de memória é igual à soma do registo-base com o deslocamento. O **registro-base** é o registo pelo qual pretendemos iniciar uma leitura ou uma escrita na memória - é análogo ao ponto de origem num sistema de coordenadas cartesiano -, dado que contém o endereço da memória a ler/escrever. O **deslocamento** (em inglês **offset**) é a distância, sempre múltipla de 4, da origem, que é o registo-base, até ao endereço pretendido. Dado isto, eis a operação *load word*, no Código 6.

```
lw $t0, offset($s3) # ($t0) = Mem[($s3)+8]
```

Da mesma forma que podemos ler informação da memória, também a podemos escrever na memória. Para tal usamos a operação *store word*, como no Código 7.

```
sw $t0, offset($s3) # Mem[($s3)+8] = ($t0)
```

### código 3

**operação aritmética  
com imediato**

### código 4

**operações lógicas**

### código 5

**operações lógicas com  
imediato**

**registro-base**

**deslocamento  
offset**

### código 6

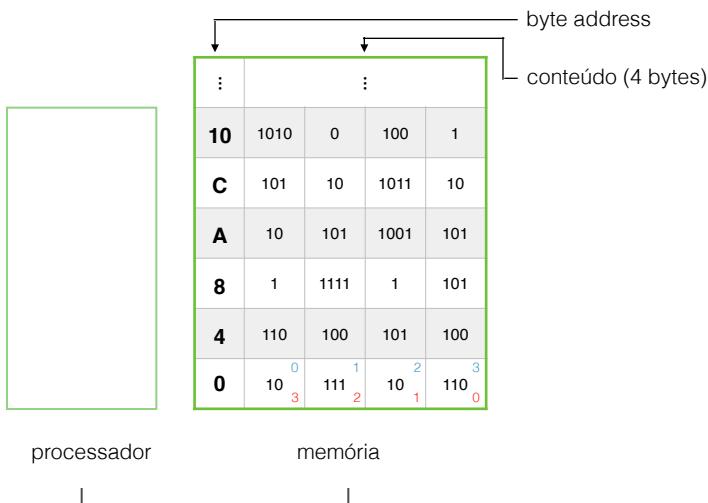
**load word genérico**

### código 7

**store word genérico**

## Organização da memória

Como já referimos anteriormente, o MIPS tem uma memória endereçável byte a byte - a bytes sucessivos correspondem endereços sucessivos (lógica byte-addressable memory). Vimos também que enquanto a 8 bits chamamos de byte, a 32 bits damos o nome de *word*, composição de 4 bytes.



**figura 12**  
relação entre memória  
4 bytes e processador

Na Figura 12 podemos ver um exemplo da relação entre a memória e o processador. O **datapath** que os liga, é que é usado no momento da execução de uma instrução do tipo de transferência de dados. Já a forma como os dados são preservados na memória depende da lógica a que a memória se rege - ou é **big endian**, representado pela cor azul na Figura 12, ou é **little endian**, representado pela cor vermelha. Vejamos um exemplo de um excerto em Assembly do MIPS para um acesso a um *array* *A*, de forma a fazer com que o índice 12 de *A* seja preenchido com um valor de  $h + A[8]$  (Código 8).

```
lw    $t0, 32($s3)  # ($t0) = Mem[$s3]+32] porque 32 = 4*8
add $t0, $s2, $t0  # $t0 = $t0 + h (sendo que h está em $s2)
sw    $t0, 48($s3)  # Mem[$s3]+48] = ($t0) porque 48 = 4*12
```

## Representação de sistemas numéricicos com e sem sinal

Da disciplina de Introdução aos Sistemas Digitais (a1s1) aprendemos que um número inteiro binário com  $n$  bits, isto é, do bit 0 ao bit  $n - 1$ , tem uma **gama de representação** de 0 a  $2^n - 1$ .

A representação de números negativos em binário pode ser feita de dois grandes modos: em sinal e módulo ou em complemento. No caso do **sinal e modo**, em Introdução aos Sistemas Digitais remos que o bit mais significativo não tem qualquer tipo de peso e serve de indicador para o sinal da representação, isto é, 0 se for positivo e 1 se for negativo. Mas este método tem muitos inconvenientes: para além de ter o caso especial do bit-sinal, este método exige domadores e subtraíres com lógicas distintas e tem, mais-que-tudo, das representações para a quantidade 0, quer 00000000 como 10000000, respetivamente 0 e -0.

Dadas as limitações do sinal e módulo, criaram-se os **complementos**. Estes já não carecem de domadores e subtraíres com lógicas distintas e todos os bits são tratados de igual modo. Foram criados dois tipos de complemento. Primeiramente, o

**datapath**

**big endian**

**little endian**

**código 8**

**excerto de código para  
aceder a array**

**gama de representação**

**sinal e modo**

**complemento**

# 13 ARQUITETURA DE COMPUTADORES I

tipo de complemento para um parecia resolver os problemas, mas rapidamente se chegou à conclusão do contrário, porque, embora se baseando no processo de inverter os bits individualmente, continuava-se a ter duas representações para o 0, quer 00000000, quer 11111111, respetivamente 0 e -0. O **complemento para 1** (também chamado de complemento falso) obtém-se então aplicando uma lógica *bitwise not*, pelo que a representação de 15 em complemento para 1 será 00001111<sub>2</sub> e a representação de -15 em complemento para 1 será 11110000<sub>2</sub>. Neste método o bit mais significativo tem o peso de -(2<sup>n</sup> - 1).

Mais do que a desvantagem da dupla representação do 0 temos também o **end-around carry**, que acontece quando ambos os números que pretendemos operar sobre são negativos e os queremos somar.

Para resolver todos os problemas dos métodos anteriores criou-se o **complemento para dois**. A gama de representação para  $n$  bits é de  $-2^{n-1}$  até  $2^{n-1} - 1$ . Nesta representação, por ser linear, não existe a limitação da dupla representação do zero, dada a métrica da gama de representação. Em complemento para dois o número mais positivo de todos é o 01111...11<sub>2</sub> e o mais negativo de todos há de ser o 100...00<sub>2</sub>.

Para calcular um número em complemento para dois basta fazer o complemento para 1 de um número e somar 1. De forma mais direta, a representação em complemento para 2 é igual, da direita para a esquerda, até ao primeiro 1, seguida da inversão de todos os números à esquerda.

Uma outra vantagem do complemento para dois é que caso precisemos de representar um dado número em mais bits, basta replicar o valor do bit de sinal para a esquerda, o que não altera o sinal o valor representado. Por exemplo, no ISA do MIPS o *addi* é extraído a 32 bits, o *lb* e o *lw* extendem a 32 bits o byte e o half-word transferido da memória e o *bne* e *beq* extendem o valor do imediato que indica o deslocamento. De 8 bits para 16 bits, o número 2<sub>10</sub>, em complemento para 2 pode ser extendido de 0000 0010<sub>2</sub> para 0000 0000 0000 0010<sub>2</sub> e o número -2<sub>10</sub> pode ser extendido de 1111 1110<sub>2</sub> para 1111 1111 1111 1110<sub>2</sub>.

## Codificação das instruções em MIPS

Como é que são representadas as instruções do repertório do MIPS? Vamos até ao processo de conversão do Assembly para *bytecode*, pelo **Assembler**.

As instruções no MIPS estão codificadas em binário (código máquina). Estas estão codificadas em 32 bits e têm sempre um comprimento fixo com um número reduzido de formatos de instrução. Elas codificam um **código de operação** (denominado de *opcode*), número dos registos, ... com uma regularidade própria.

Existem vários formatos de instrução. O tipo principal, o **tipo R**, é utilizado aquando de operações aritméticas e lógicas e tem o seguinte formato representado na Figura 13.



Na Figura 13 está então representado um tipo de instrução do ISA do MIPS - o tipo R. Este tipo caracteriza-se por se usar, como já foi referido, em instruções do tipo aritméticas e lógicas. Na sua codificação, os bits estão agrupados por 6 grupos com funções muito específicas: os primeiros 6 bits são reservados para o **opcode**, que se traduz pelo código de operação, o qual, juntamente com os últimos 6 bits (**funct** - function code), que extende o opcode, preparam uma dada operação aritmética ou

**complemento para 1**

**end-around carry**

**complemento para dois**

**Assembler**

**código de operação**

**tipo R**

**figura 13**  
**tipo de instrução R**

**opcode**  
**funct**

lógica. Continuando, os primeiros 5 bits reservados, com a sigla *rs*, traduzem o número do primeiro registo usado como operando, sendo que os 5 bits seguintes, *rt*, traduzem o número do registo do segundo operando e os seguintes, *rd*, o número do registo de destino da operação. Dado que não estamos a tratar de operações de *shift*, os últimos 5 bits, com o nome de **shamt**, manter-se-ão a 00000, pelo que a sua designação provém de **shift amount**.

Quando fazemos operações com imediatos, mesmo que aritméticas ou lógicas, o armazenamento da constante na instrução codificada não pode ser integrada na ordem de um registo com apenas 5 bits, dado que uma constante não se restringe a apenas 5 bits. Sendo que existe esta limitação, as instruções aritméticas e lógicas com imediatos, necessitam de um outro tipo de instrução, denominada de **tipo I**. Este tipo de instrução também é usado no caso das operações de transferência de dados para a/da memória. A Figura 14 tem a representação do tipo I de instrução.

<b>op</b>	<b>rs</b>	<b>rt</b>	<b>constant or address</b>	<b>tipo I</b>
6 bits	5 bits	5 bits	16 bits	

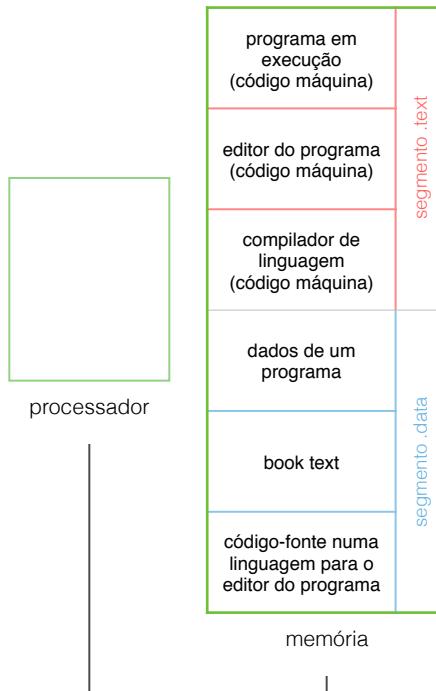
**figura 14**  
**tipo de instrução I**

Mais uma vez, e tal como na instrução do tipo R, na Figura 14, o indicado como *op* significa opcode e traduz o código de operação, embora, neste caso, sem a complementação de código funct, dado que ele não existe no tipo I. O *rs* e *rt* são dois registos: o *rs* serve de registo-base nas operações de *load/store* e de operando nas operações com imediato; o *rt* nas operações de *load/store* serve de registo para/de onde a transferência de dados é realizada, pelo que nas operações aritméticas e lógicas com imediato serve de registo de destino do resultado de um cálculo pretendido por uma determinada instrução. Até ao momento já reunimos 16 bits da instrução do tipo I - os 16 bits restantes, para os totais 32 bits, estão reservados para a constante (no caso das operações aritméticas e lógicas com imediato, sendo que este é entre  $-2^{15}$  e  $2^{15}-1$ ) ou para o endereço de memória (no caso das operações de transferência de dados para a memória, enquanto deslocamento - offset - adicionado ao registo-base em *rs*).

Até agora várias vezes classificámos o MIPS como um processador com bastantes regularidades de protocolo. Um princípio importante a nível do *digital design* é que “*good design demands good compromises*” (em português, “um bom desenho pede bons compromissos”). A arquitetura MIPS segue esse princípio, pelo que diferentes formatos complicam a descodificação, e esta arquitetura colmata isso, simplificando o número de formatos disponíveis, permitindo que todas as instruções sejam codificadas em apenas 32 bits. Mais ainda, a procura por formatos que sejam o mais semelhantes possível entre si tornam a descodificação o mais simples possível, tal como a sua implementação no processador - o que fornece rapidez ao sistema.

Em suma, cumulativamente ao que estudámos anteriormente acerca da organização da memória, temos então que a memória se encontra segmentada da forma visível na Figura 15, na qual as instruções são apresentadas em binário, tal como os dados, sendo estes armazenados na memória; os programas podem operar sobre programas (compiladores, *linkers*, assembler, ...); e a compatibilidade binária permite aos programas compilados serem executados em diferentes computadores, dado que o ISA é estandardizado.

Na Figura 15, então temos a interação entre o processador e a memória, sendo que nesta última residem os segmentos *data* e *text*, nos quais se armazenam os dados e as instruções do programa e se executam os programas, estes, preservados em código máquina para processamento direto, como os compiladores, *linkers*, ...



**figura 15**  
organização da memória  
revisitada

## Operadores lógicos

As operações lógicas são muito úteis para extrair e inserir grupos de bits numa *word* de 32 bits. Noutras disciplinas muito provavelmente já deves ter usado este tipo de operações para fazer seleções de dados, filtrar receitas, ... Em Assembly também o podes fazer, e com quase toda a certeza, de um modo ainda mais aproximado do real, que em linguagens de alto nível.

Nas linguagens de C e de Java podes encontrar os operadores “<<” e “>>”, que significam *shift left logical* (deslocamento lógico para a esquerda) e *shift right logic* (deslocamento lógico para a direita), respetivamente. Em Assembly também o podemos fazer, sendo que para tal basta usar a operação *sll*, para o *shift left logical*, e o *srl* para o *shift right logical*. Vejamos o seguinte Código 9:

```
sll $t0, $s3, 4 # $t0 = $s3 << 4
srl $t0, $t0, 4 # $t0 = $t0 >> 4
```

A operação de **shift left logical** faz com que os bits de uma dada sequência sejam movidos *n* bits à esquerda. Por exemplo, se em *\$s3*, do Código 9, estivesse o número, em binário, 1001001<sub>2</sub>, após a primeira operação, *\$t0* teria o valor 0010000<sub>2</sub>. Por outro lado, a operação de **shift right logical** faz com que os bits de uma dada sequência sejam movidos *n* bits à direita. A partir de onde ficámos no Código 9, na sua análise, se *\$t0* agora tem o valor 0010000<sub>2</sub>, com a operação *srl* temos 0000001<sub>2</sub>, em *\$t0*.

Agora já podemos responder à questão de para que serve o valor do *shamt* no tipo R de instruções do MIPS. O *shamt* indica quantas posições de deslocamento é que vão ser realizadas. Sendo assim, para uma instrução de deslocamento, o campo *rs* é sempre igual a 0. Para o *sll* há um movimento de *n* casas para a esquerda, criando *n* zeros à direita - a instrução *sll* multiplica por  $2^n$ . Para o *srl* há um movimento de *n* bits à direita, criando *n* bits 0 à esquerda - *srl n* bits divide (operandos sem sinal - *unsigned*) por  $2^n$ .

**código 9**  
instruções sll e srl

**shift left logical**

**shift right logical**

Uma outra operação lógica é o *sra*, **shift right arithmetic**. Esta operação é semelhante à operação de *srl* com a ligeira diferença que divide operandos com sinal, isto é, operandos *signed*. Este tipo de operandos, após uma operação *sra*, permanece sempre com o mesmo sinal, através de uma extensão de sinal. Vejamos o Código 10, onde  $\$s3$  tem o valor 1001001<sub>2</sub>.

```
srl $t0, $s3, 4 # $t0 = 0000100
sra $t1, $s3, 4 # $t1 = 1111100
```

**shift right arithmetic****código 10****comparação entre sra e srl**

Dado o Código 10, podemos dizer que para o *sra* há um movimento de  $n$  bits à direita, preenchendo à esquerda  $n$  bits iguais ao bit de sinal, pelo que *sra* de  $n$  bits permite a divisão de operandos *signed* por  $2^n$ .

A operação “e” lógico, também designada por AND, é uma instrução útil para mascarar, isto é, selecionar, bits numa *word*. Ela seleciona alguns bits, deixando os outros a 0, sendo que a conjugação de um 0 dá sempre 0 e a conjugação de um valor  $x$  com 1, depende do valor de  $x$ , sendo que se este for 1, o resultado é 1. Em Assembly podemos usar a operação *and* ou a *andi*, para operar sobre um imediato, sendo que este é primeiro (e automaticamente) convertido para binário. Sendo assim, vejamos o Código 11.

```
and $t0, $s3, $s2 # $t0 = $s3 & $s2
andi $t1, $s3, 4 # $t1 = $s3 & 4 = 1100 se $s3 = 1101
```

**“e” lógico****código 11****instruções and e andi**

A instrução “ou” lógico, também designada por OR, é uma instrução útil para incluir bits numa *word*. Esta operação coloca alguns bits a 1, enquanto que os restantes não se alteram, sendo que a disjunção de um elemento de qualquer estado com 1 é sempre 1, e a disjunção de um elemento  $x$ , com 0, é sempre dependente do valor de  $x$ . Em Assembly podemos usar a operação *or* ou *ori*, para operar sobre um imediato, sendo que este é primeiro (e automaticamente) convertido para binário.

A instrução “não” lógico, também designada por NOT é uma operação que permite a inversão de todos os bits numa *word*. Acontece que esta operação não é uma operação nativa do MIPS, embora o MIPS tenha uma que lhe corresponde em curto-circuito de entradas - o NOR. Na máquina virtual existe a operação NOT, mas é conveniente saber que esta se converte numa NOR. Vejamos o Código 12.

```
not $t0, $s3 # $t0 = ($s3)'
nor $t0, $s3, $s3 # $t0 = ($s3 | $s3)' = ($s3)' & ($s3)' = ($s3)'
```

**“ou” lógico****“não” lógico****código 12****instruções not e nor**

## Instruções de escolha e operações condicionais

As operações condicionais em Assembly são operações com um nome especial - chamam-se de **branches**. Os branches são saltos que apenas ocorrem quando a condição que estes avaliam é de facto verificável e verdadeira, isto é, dado que o assembler lerá as instruções uma a uma, por ordem de escrita, quando ele atingir um branch ele terá de saber quando é que salta, sem que tenha que voltar para trás e saltar de novo no código. Vejamos uma aplicação desta explicação: num dado programa temos uma condição *if...else* na qual o *if* é lido só caso uma variável *a* seja igual a 0. O programa em Assembly terá que ter primeiro a instrução correspondente à verificação da condição e só depois o corpo. Mas o corpo do *if* tem de ser imediatamente a seguir à verificação, de modo a que caso a verificação resulte num argumento falso, o programa possa saltar as linhas de código correspondentes ao *if* e passar assim ao *else*. Mas aqui há um problema: como já dissemos, os branches só saltam quando a condição é de

**branches**

facto verificável e verdadeira. Neste caso temos uma condição que resulta num argumento falso, pelo que não podemos saltar sem lei. Como é que podemos resolver esta situação? Primeiro há que ver que tipos de instrução branch é que temos e depois é que podemos resolver o nosso grande problema.

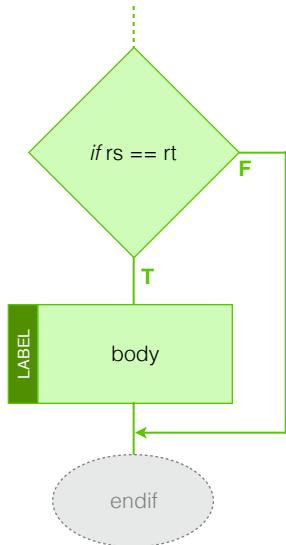
As instruções branch são várias, mas nativamente, na arquitetura do MIPS, só existem duas, que são as *beq* (**branch on equal**) e a *bne* (acrónimo inglês de **branch on not equal**). O *beq* funciona da seguinte forma, interpretada através de um fluxograma na Figura 16 e do Código 13.

```
beq    rs, rt, LABEL  # if rs == rt jumps to LABEL
```

**branch on equal**  
**branch on not equal**

**código 13**  
**branch on equal**

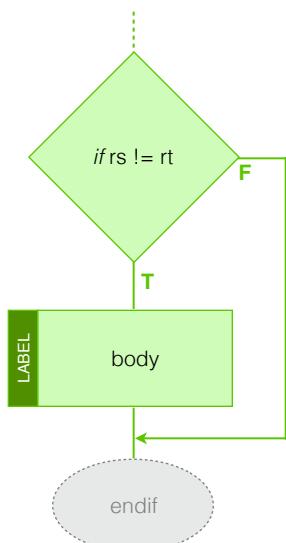
**figura 16**  
**branch on equal**



A operação *bne*, por outro lado, sendo semelhante, em termos de fluxograma, à operação anterior, tem a forma representada pela Figura 17 e pelo Código 14.

```
bne    rs, rt, LABEL  # if rs != rt jumps to LABEL
```

**código 14**  
**branch on not equal**  
**figura 17**  
**branch on not equal**



Como iremos ver daqui em diante, os fluxogramas serão uma grande ajuda em termos de compreensão e de adequação da escolha de branches à resolução de vários problemas. Um deles, logo de imediato, são os saltos. Não esquecendo o nosso grande problema de ainda há bocado (saltos em verdadeiro apenas), vejamos o que é ao certo, um salto.

Um salto pode ser dado de duas grandes formas distintas - condicionalmente, através de uma instrução branch (efetua a avaliação de uma condição) ou incondicionalmente, através de uma instrução **jump** (que não efetua qualquer avaliação de condições). Numa analogia vejamos um jogo, como o jogo da glória, no qual existem casas onde o jogador é obrigado efetuar um salto com a sua peça em  $x$  outras casas (operação jump) ou onde o jogador só efetua um salto se cumprir os requisitos suficientes para o fazer (instrução branch, com avaliação de uma condição). A instrução jump tem a sua forma representada no Código 15.

**j      LABEL      # jump to label**

**jump**

**código 15**  
**instrução jump**

Resolvamos então o problema que colocámos mais atrás. Dizia então que se o salto de um branch só pode ser dado em verdadeiro, como é que podemos lidar com situações em que o salto, por um pseudocódigo, tem de ser dado em falso? Para tal analisemos caso-a-caso, isto é, sendo cada caso, cada tipo de operação de controlo da operação de um programa, entre os quais o *if...else*, o *for*, o *while* e o *do...while*.

Comecemos pelo *if...else*. Vejamos o seguinte código em C, representado no Código 16.

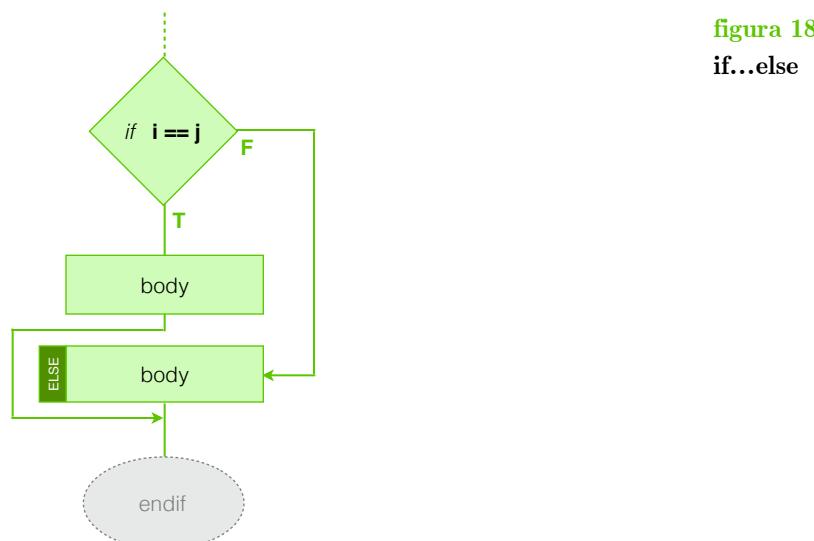
```
if (i == j) {
    f = g + h;
}
else {
    f = g + h;
}
```

**código 16**  
**if...else**

**nota!!** consideremos que as variáveis  $f$ ,  $g$ ,  $h$ ,  $i$  e  $j$  estão guardadas em  $\$s0, \dots, \$s4$ .

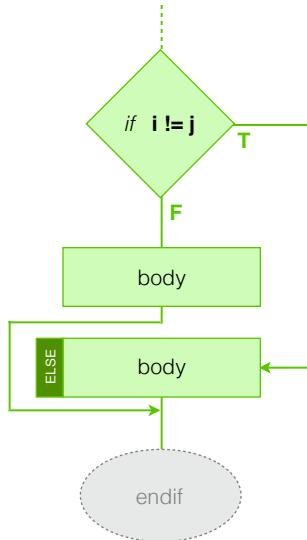
**nota**

Para escrever isto em Assembly façamos primeiro um fluxograma que represente o funcionamento da engrenagem-base deste programa (Figura 18).



**figura 18**  
**if...else**

Se olharmos bem para o fluxograma podemos reparar que realmente o salto está a ser dado quando a condição avaliada é falsa. Para reparar isto, apenas temos que negar a condição, sendo que quando esta é falsa, o assembler segue a leitura do código pelo tronco principal, como representado na Figura 19.



**figura 19**  
correção de if...else

Através do fluxograma, agora, basta interpretar cada uma das instruções e escrevê-las em código Assembly, como no Código 17.

<pre> bne      \$s3, \$s4, ELSE add      \$\$0, \$\$1, \$\$2 j       END ELSE: sub      \$\$0, \$\$1, \$\$2 ENDIF: # . . </pre>	<pre> # if (i == j) { #   f = g + h # } # else { f = g - h } </pre>
---	---

**código 17**  
código assembly

Os **ciclos while** funcionam executando uma verificação do tipo condicional antes da execução da primeira leitura do corpo ciclo. Vejamos assim o seguinte código genérico em C.

```

while (i < 20) {
    body;
    i++;
}

```

**ciclos while**

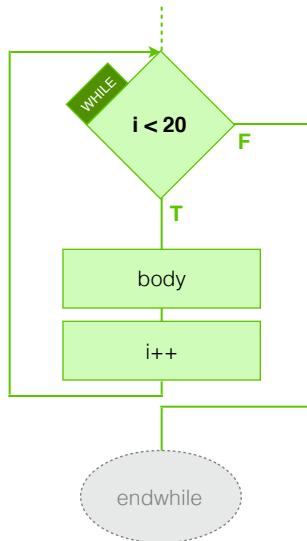
**código 18**  
while

Fazendo o fluxograma para averiguar o funcionamento deste pequeno programa temos a Figura 20.

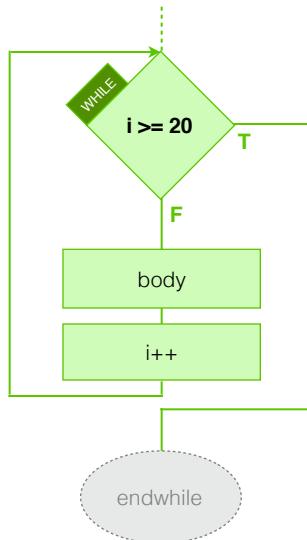
Analisando a Figura 20 temos que o salto, mais uma vez, está a ser dado cada vez que a condição  $i < 20$  é avaliada como falsa. Para tal, a solução é negar a condição, de forma a que o salto apenas seja realizado em verdadeiro e nunca em falso. A solução, então, é visível no fluxograma da Figura 21.

Os **ciclos for** são um redesenho dos ciclos que estudámos anteriormente, sendo que sistematizam o funcionamento de um ciclo while num intervalo fechado, mais visível e concreto em termos de compilação, tal como mais simples. O que acontece num ciclo for é que há uma inicialização de uma variável, uma avaliação de uma condição e uma pós incrementação da variável inicial, concretizada no final da execução do corpo do ciclo. No Código 19 podemos ver uma implementação genérica de um ciclo for, em C.

**ciclos for**



**figura 20**  
**while**



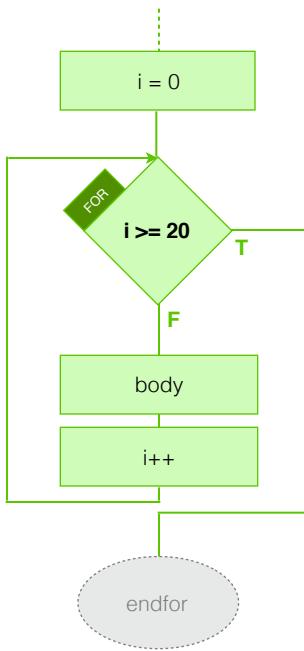
**figura 21**  
**correção de while**

```
for (int i = 0; i < 20; i++) {
    body;
}
```

**código 19**  
**ciclo for**

Num fluxograma, o princípio de resolução de um ciclo for é algo semelhante ao dos ciclos while. Quando representamos um ciclo while num fluxograma temos de nos preocupar com o mapeamento das etiquetas para a condição e para o ponto de conclusão do ciclo, e com o incremento da variável de controlo. Da mesma forma, num ciclo for, há que inicializar a variável de controlo - por exemplo,  $i$  -, verificar a condição, tendo em conta que o salto deve ser dado em verdadeiro (pelo que numa instrução do tipo for existe sempre uma negação da condição), execução do corpo do ciclo e incrementação da variável de controlo. É importante especificar etiquetas para parametrizar os saltos (jumps e branches) que o programa tem de fazer.

Na Figura 22 temos o fluxograma respetivo à instrução for, seguida da execução do conteúdo do seu corpo, até à conclusão no passo etiquetado de *endfor*.



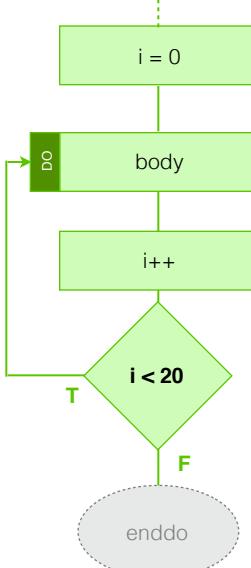
**figura 22**  
**ciclo for**

O ciclo que nos falta analisar é o **do...while**. A diferença entre este ciclo e todos os outros ciclos já estudados é que este, em particular, avalia a condição depois de uma execução completa do ciclo - existe **pós-validação**. O que podemos verificar é que dada esta pós-validação da condição de execução, esta, não tem de se negar para que o programa salte quando a condição é verdadeira, porque, por raiz, nós pretendemos, num ciclo do...while, que um determinado ciclo se efetue (ou seja, que salte) cada vez que a condição que a delimita se encontre no estado verdadeiro. Sendo assim, no Código 20 podemos encontrar uma implementação em C, genérica, e na Figura 23, o fluxograma do ciclo respetivo.

**do...while**  
**pós-validação**

```
do {
    body
} while (i < 20)
```

**código 20**  
**do...while**



**figura 23**  
**do...while**

Tendo os fluxogramas dos ciclos feitos, agora é-nos possível traduzir os códigos em C para Assembly do MIPS. Assim, no Código 21 temos o código do ciclo while, no Código 22 o código do ciclo for, e no Código 23 o código do ciclo do...while.

```
WHILE:      bge    $s3, 20, ENDWHILE      # while (i < 20) {  
. . .  
addi   $s3, $s3, 1  
j      WHILE  
ENDWHILE:      # body;  
# i++;  
# }  
# . . .
```

---

código 21  
ciclo while

```
FOR:        li     $s3, 0  
bge   $s3, 20, ENDFOR      # i = 0  
. . .  
addi  $s3, $s3, 1  
j     FOR  
ENDFOR:      # body;  
# i++;  
# }  
# . . .
```

---

código 22  
ciclo for

```
DO:         li     $s3, 0  
. . .  
addi  $s3, $s3, 1  
blt   $s3, 20, FOR      # i = 0  
# do { body  
# i++;  
# } while (i < 20)  
# . . .
```

código 23  
ciclo do...while

Concluindo, podemos reparar que sempre que existe uma pré-validação da condição, esta tem de ser negada, pelo que no caso do ciclo do...while, no qual não existe pré-validação, mas antes pós-validação, não há negação da condição.

## Basic Blocks

Nas construções condicionadas e de controle de ciclos há um conceito muito importante e aplicado em Assembly - o de **basic block**. Os compiladores fazem as suas operações de otimização de código sob basic blocks, maioritamente. Os basic blocks são sequências de instruções em que não há saltos (exceto se estivermos a referir o fim de uma sequência) e em que os alvos de instrução de salto só podem existir no início. Esta criação e organização do próprio código em Assembly permite que o CPU acelere a execução, por saber da existência de basic blocks. Mais à frente, quando abordaremos uma temática de *pipelines*, iremos perceber com mais detalhe o propósito desta rapidez de leitura de dados.

basic block

## Construção de instruções case/switch

Muitas vezes, a programar, sentimos a necessidade de criar um método de implementação de **case/switch**, de forma a que este permita a escolha de uma entre várias alternativas de execução. Mas em Assembly, como é que podemos codificar este tipo de operação?

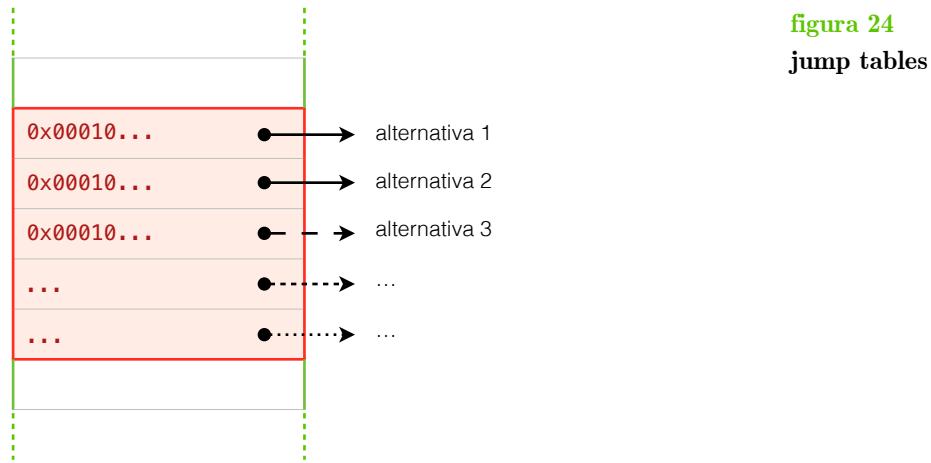
case/switch

A raiz de um sistema de case/switch provém, na sua forma mais básica, de uma sequência de parâmetros de controlo *if-then-else*. Sendo assim, em Assembly é-nos permitido criar uma sequência de branches de forma a que estes possam criar o nosso sistema de case/switch.

Uma outra forma de gerar uma instrução de case/switch é criando um novo conceito de organização de dados, através de uma tabela de endereços de registos. O

# 23 ARQUITETURA DE COMPUTADORES I

que acontece é que criando uma tabela deste tipo, denominadas de **jump tables**, criamos um array de palavras (words - 32 bits) contendo os endereços que correspondem às respetivas etiquetas do programa (labels). A Figura 24 pretende ilustrar este princípio.



A escolha dos casos da forma representada na Figura 24 é garantida pela instrução jr (jump register).

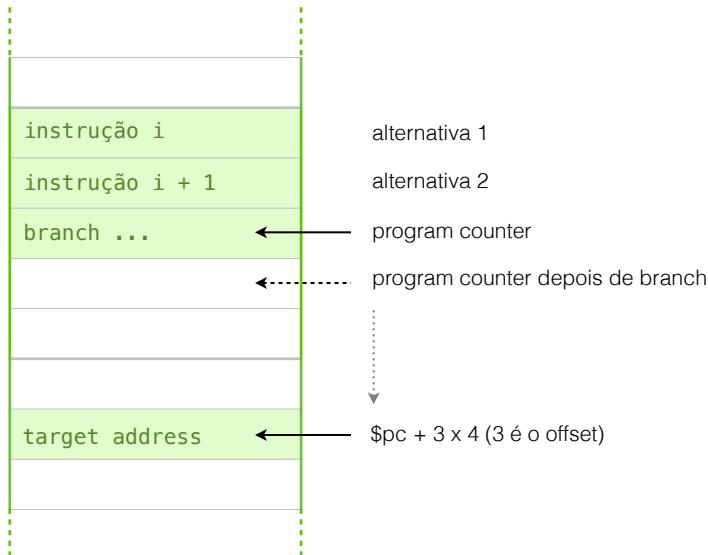
## Modos de endereçamento das instruções branch e jump

Tanto no caso das instruções de operações aritméticas e lógicas, com ou sem imediato, como no caso das instruções de transferência de dados entre memória e processador nós já devemos saber como é que se processam as codificações de instruções e como é que elas são endereçadas. Entretanto abordámos novas instruções, as quais ainda não foram incluídas no esquema de nenhum tipo de instrução: refiro-me às instruções de branch e jump.

Antes de avançar mais é bom saber qual é, de facto, a diferença entre a instrução branch (básica, representada por um *b*) e a instrução jump (representada por um *j*). A única diferença que separa estas duas operações relaciona-se com o grau de abrangência de cada uma, sendo que a instrução jump permite que sejam feitos saltos com maior espelho que as instruções do tipo branch, que esperam sempre que o salto seja realizado para as proximidades da instrução. Por definição, uma instrução do tipo branch permite que seja feito um salto entre  $-2^{15}$  (para trás) e  $2^{15}$  endereços (para a frente), pelo que é visível uma simetria, de acordo com a notação de complemento para dois, aplicada ao espelho do salto branch, sendo que tem base no índice -1 (para trás).

As instruções de branch especificam então um determinado *opcode*, acompanhado de dois registos e de um **target address**. O target address é um valor estipulado para o endereço de leitura após salto branch. Dado que este valor não é aleatório e depende da própria definição da instrução branch, tal como do estado atual da máquina que processa este código, para calcular o target address basta adicionar ao *program counter* (registo \$pc) a quantidade total de deslocamento que multiplica por 4, dado o alinhamento por omissão de 32 bits (4 bytes). Após a introdução de uma instrução branch o *program counter* já deve estar a apontar para a instrução seguinte (previamente incrementado de 4).

Na Figura 25 podemos verificar, graficamente, uma representação do movimento do *program counter* num programa que integre um branch.



O endereçamento de uma instrução branch pertence ao tipo I, mas as instruções jump não. No Código 24 temos uma instrução jump genérica - analisemo-la.

**j      LABEL**      # jump to label LABEL

**código 24**  
sumário de ciclos

Como podemos ver no Código 24, a instrução jump apenas requer a etiqueta, a qual representa o endereço para onde pretendemos concretizar o salto. Os endereços no MIPS têm 32 bits. Faria algum sentido ter um tipo novo de instrução, na qual apenas se colocasse o endereço de salto pretendido. Mas se assim fosse teríamos um sério problema, entre muitos outros: a operação jump não é exclusiva - também temos a jump and link (representada por *jal*). Para tal precisamos também de um *opcode*, sendo assim - um código que traduza a operação que pretendo efetuar, conseguindo distinguir entre jump e jump and link. Dado isto, cria-se um novo tipo de instrução - o **tipo J**. Este tipo caracteriza-se por ter reservados 6 bits para o *opcode*, mais 28 bits para o endereço, sendo que corresponde a 30 bits com extensão de sinal - de 28 corresponde a 30 porque os endereços estão todos alinhados por 32 bits e todos os números binários com dois zeros nos bits menos significativos são múltiplos de 4.

op	address	tipo J
6 bits	28 bits	

**figura 26**  
instrução do tipo J

## Arrays e Ponteiros

Muitas vezes é importante, antes de fazermos operações com operandos fixos ou imediatos, fazê-las com endereços, de forma a ganharmos mais precisão e generalidade no que toca ao desenvolvimento de um programa. Para tal, algumas linguagens de alto-nível, como a linguagem C, usa um conceito de desenvolvimento de **ponteiros**. Um ponteiro (*pointer*, em inglês) é uma variável que preserva o endereço de outra variável. Num conceito mais generalizado, trata-se de uma versão em linguagem de alto-nível, do endereço de código máquina.

**ponteiros**

# 25 ARQUITETURA DE COMPUTADORES I

A utilidade dos ponteiros traduz-se na necessidade de, por vezes, só haver esse método para criar um método de computação. Por outro lado, e como já foi referido, os ponteiros aumentam o rigor de um programa, sendo que o torna muito mais simples, compacto e eficiente.

Em C, os ponteiros designam-se por caracteres especiais - denotam-se o “\*” e o “&”. Estes caracteres são sempre adjacentes às variáveis pretendidas. Consideremos, por exemplo, uma variável *c* que tem o valor 100 e que se encontra localizada na posição, isto é, endereço 0x10000000. O operador “&” dá-nos o endereço, como se pode ver no Código 25.

```
p = &c;           // atribui a p o endereço de c, i.e. p = 0x10000000
```

**código 25**  
endereço de variável

Por outro lado, o operador “\*” dá o valor para o qual o ponteiro aponta, como podemos verificar no Código 26.

```
if (p == &c)      *p = c           // desreferenciação de p
```

**código 26**  
ponteiros

Através de ponteiros, podemos assim dizer que uma operação simples de atribuição, da forma *y* = *x*, é equivalente a copiar o endereço de *x* para uma variável externa, como *px* (através de *px* = *&x*), seguido da salvaguarda do conteúdo de *px* para referenciar *y* (através de *y* = *\*px*).

Em C, na declaração de variáveis, declaram-se os ponteiros com um determinado tipo de dados. Se declararmos, por exemplo, *int \*p1*, o que se está a fazer ao certo é assumir a criação de um ponteiro para variáveis do tipo inteiro (*int*). No Código 27 temos o caso de um ponteiro para carateres.

```
char *p;           // p é ponteiro para carateres
```

**código 27**  
ponteiro para carateres

Nesta linguagem de alto-nível os argumentos são passados por valor, isto é, a função não tem acesso à variável, mas sim apenas ao seu valor. De forma a ser possível a função ter acesso à variável, para alterar o seu valor, tem que ser passado como argumento um ponteiro para a variável pretendida, isto é, o seu endereço, e não o seu valor - **passagem por referência**. Vejamos o exemplo do Código 28.

```
int a[50];        // array de 50 inteiros
int *p;           // ponteiro para inteiro
int x;
p = &a[0];         // p fica com endereço de a[0]
x = p;            // atribui-se a x o endereço de a[0] em p1
```

**passagem por referência**  
**código 28**  
exemplo de aplicação

Em Assembly do MIPS os ponteiros simplificam o cálculo da leitura e escrita de dados na memória, sendo que assumir a variável com o operador “\*” antes e depois de uma atribuição corresponde às duas operações de transferência de dados: store e load, respetivamente, como podemos verificar no Código 29.

```
*p = . . .;       // (store) armazena o valor na posição apontada por p
. . . = *p;       // (load) obtém o valor armazenado na pos. apontada por p
```

**código 29**  
load e store

Vejamos então uma comparação entre o código escrito com base em ponteiros e com base em arrays. Pretende-se assim traduzir em linguagem Assembly do MIPS o

<sup>1</sup> Se *p* aponta para um dado elemento do array, então *p*++ apontará para o seguinte. No caso dos arrays, a variável que designa o array indica o respetivo endereço-base, isto é: *p* = *a* ⇔ *p* = *&a[0]*;

código em C que está presente no Código 30. Primeiro iremos dispor e estudar o código com base em arrays e só depois em ponteiros.

```
void main(void) {
    static char str[20];
    int num, i;
    read_str(str, 20);
    num = 0;
    i = 0;
    while (str[i] != '\0') {
        if ((str[i] >= '0') && (str[i] <= '9'))
            num++;
        i++;
    }
    print_int10(num);
}
```

código 30

código de estudo em C

Como dito acima uma solução deste problema através de indexação por arrays é a seguinte (Código 31):

```
.data
str:    .space 20
        .globl main
.text
main:   la $t2, str          # load str's address to $t2
        move $a0, $t2          # copy $t2 value to $a0
        li $a1, 20             # $a1 = 20
        li $v0, 8              # prepare for read_str($a0, $a1)
        syscall               # print_str($a0, $a1)
        li $t0, 0              # num = 0
        li $t1, 0              # i = 0
while:   addu $t3, $t1, $t2  # $t3 -> &str(i)
        lb $t4, 0($t3)         # load byte (char) from $t3
        beqz $t4, endwhile    # if ($t4 == 0) goto endwhile
        blt $t4, 0x30, endif   # if ($t4 < 0x30) goto endif
        bgt $t4, 0x39, endif   # if ($t4 > 0x39) goto endif
        addi $t0, $t0, 1        # num++
        addi $t1, $t1, 1        # i++
endif:   j while             # jump to while
endwhile: move $a0, $t0        # copy $t0 value to $a0
        li $v0, 36              # prepare for print_int10(num)
        syscall               # print_int10(num)
        jr $ra                 # end of program
```

código 31

exemplo de aplicação

Através da utilização de ponteiros, o Código 31 reescreve-se da seguinte forma:

```
.data
str:    .space 20
        .globl main
.text
main:   la $t1, str          # load str's address to $t1
        move $a0, $t1          # copy $t1 value to $a0
        li $a1, 20             # $a1 = 20
        li $v0, 8              # prepare for read_str($a0, $a1)
        syscall               # print_str($a0, $a1)
        li $t0, 0              # num = 0
        lb $t2, 0($t1)         # p = &str(0)
        beqz $t2, endwhile    # if ($t2 == 0) goto endwhile
        blt $t2, 0x30, endif   # if ($t2 < 0x30) goto endif
        bgt $t2, 0x39, endif   # if ($t2 > 0x39) goto endif
        addi $t0, $t0, 1        # num++
        addi $t1, $t1, 1        # p++
endif:   j while             # jump to while
endwhile: move $a0, $t0        # copy $t0 value to $a0
        li $v0, 36              # prepare for print_int10(num)
        syscall               # print_int10(num)
        jr $ra                 # end of program
```

código 32

exemplo de aplicação

A simplificação que é feita em ponteiros, em relação à indexação por arrays é feita através da incrementação e leitura da memória. O que acontece é que enquanto que em arrays nós temos que carregar o endereço da memória do array para um registo e depois ir incrementando o conteúdo do registo por quatro e voltar a ler, assim consecutivamente, através de ponteiros basta criar o ponteiro, isto é, carregar o endereço do array e incrementá-lo para leitura direta. O processo é mais direto, como se pode ver no Código 32. É necessário apenas ter atenção ao incremento de um ponteiro  $p$ , sendo que no caso de inteiros é  $p = p + 4$  e no caso de caracteres é  $p = p + 1$ .

## Utilização dos registos

Os registos do MIPS têm normas próprias para a sua utilização, sendo que os seus nomes não foram criados aleatoriamente. Os registos \$a0 até \$a3 (numerados de registos de 4 a 7) servem para alojar os variados argumentos de uma rotina. Os registos de \$v0 e \$v1 (numerados de registos 2 e 3) servem para registar os valores dos resultados de rotinas. Um registo que temos usado constantemente para terminar os nossos programas é o \$ra, sigla de *return address*. Este registo está numerado de registo 31 e guarda o endereço da instrução executada a seguir ao fim de execução de uma rotina, sendo que isto é feito automaticamente após uma instrução *jal* (acrónimo de *jump and link*), para além de que a última instrução deve ser *jr \$ra*.

Um registo também deveras importante é o \$sp, sigla de **stack pointer**, numerado de registo 29. Ele é usado para alocar e libertar memória de um procedimento. Por conseguinte, os registos de \$t0 a \$t9 são temporários e o seu conteúdo pode ser destruído pelo procedimento invocado (*callee*), ao contrário dos registos de \$s0 a \$s7, que têm de ser preservados pelo *callee*. Esta divisão de salvaguarda existe para simplificar o trabalho do processador a nível de dados inúteis para rotinas que se encontram a ser executadas.

O registo \$gp serve para alojar o **global pointer** (numerado por registo 28) e o \$fp serve para alojar o **frame pointer** (numerado por registo 30). O \$fp é um ponteiro para o **stack**. É usado apenas por alguns compiladores, porque a sua principal função é dar uma referência fixa para o *stack*, que também pode ser dado pelo \$sp. O problema é que o *stack pointer* aponta para o topo do *stack*. Por outro lado, o \$gp aponta sempre para uma zona de memória que contém variáveis globais (designadas pelas diretivas *.globl* e *.eqv*).

## Invocação de procedimentos/funções

Quando escrevemos um programa numa linguagem de alto nível devemos ter sempre um certo cuidado em termos de modularidade do código. A modularidade tem vários aspetos importantes, que têm as suas consequências refletidas no código traduzido em Assembly. Estudando apenas a modularidade ao nível da criação de funções, as quais chamar-lhe-emos de **procedimentos**, vejamos como é que elas são geradas e ligadas em Assembly.

Para executar um procedimento há que o invocar, e, já dentro dele, também temos de retornar à posição seguinte de onde invocámos. Para tal existem duas instruções relevantes do Assembly do MIPS - a **jump and link** (*jal*) e a **jump register** (*jr*). A instrução *jump and link* tem como síntese a *label* do início do procedimento, enquanto que a instrução *jump register* usa um registo, sendo o seu conteúdo, a próxima instrução a ser executada.

**stack pointer**

**global pointer**

**frame pointer**

**stack**

**procedimentos**

**jump and link**

**jump register**

Vejamos o seguinte exemplo genérico (Código 33):

```
main:    . . .
          jal proc1
. . .
proc1:   . . .
          jr $ra
```

código 33  
procedimentos

Mas o que é que acontece no Código 33? No corpo do *main* é executada uma instrução *jump and link* para a etiqueta que representa o procedimento *proc1*. Esta tarefa faz com que o *program counter* considere como conteúdo o endereço da primeira instrução da rotina *proc1*. Em simultâneo, no registo designado de *return address* (\$ra) fica o endereço da instrução seguinte à invocação que fora realizada. Assim, quando o procedimento termina, o Assembler saberá que deve voltar para o endereço guardado em \$ra.

## Introdução à Stack

A execução de rotinas em Assembly segue um formato de pilha (em inglês, **stack**). Quando executamos um procedimento, temos de guardar espaços para as variáveis que nela iremos utilizar. Para tal, preservam-se espaços na stack, iniciada pelo endereço do registo *stack pointer* (\$sp). A representação de uma stack segue os mesmos parâmetros que antes usámos para descrever a memória, só que de uma forma inversa, isto é, o endereço com maior valor encontra-se no topo da stack (Figura 27).

stack



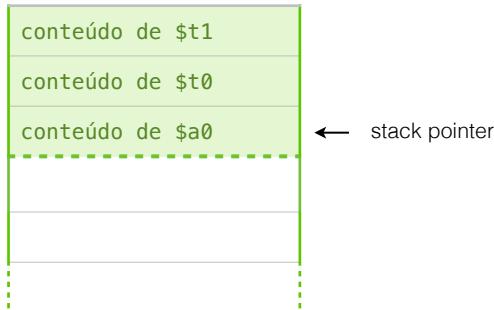
figura 27  
stack limpo

Para reservarmos espaço para, por exemplo, três variáveis na stack temos de subtrair 3 espaços inteiros ao endereço atual do \$sp, isto é,  $3 \times 4 = 12$  espaços, e neles guardar as devidas variáveis, como fazemos no Código 34.

```
subi $sp, $sp, 12
sw   $t1, 8($sp)
sw   $t0, 4($sp)
sw   $s0, 0($sp)
```

código 34  
reservar espaço na stack

Com esta salvaguarda de dados o nosso stack mudou de conteúdo, sendo que lhe adicionámos três novas entradas. Ao adicionarmos estas entradas podemos representar a nossa pilha como em Figura 28, não esquecendo que a lógica da estrutura da nossa pilha tem como endereços mais altos os presentes no topo e os mais baixos no fundo.



**figura 28**  
stack com conteúdo

Depois do procedimento terminar é importante repor ou restaurar as variáveis locais do procedimento que executou o último. Para tal temos de carregar o conteúdo das células gravadas para os registos de origem, da forma representada no Código 35.

```
lw    $t1, 8($sp)
lw    $t0, 4($sp)
lw    $s0, 0($sp)
addi $sp, $sp, 12
```

**código 35**  
libertar espaço na stack

Dado que carregámos todos os conteúdos anteriormente salvaguardados e depois adicionámos ao valor de \$sp 12, fazemos com que o nosso stack volte para o estado em que estava em Figura 27 (limparamos a pilha).

## Tipos de procedimento e processo de invocação

Um compilador, quando lê o código de um programa e o pretende compilar deve ter a capacidade de, ao ser apresentado um procedimento, ser capaz de o distinguir de outros e de o avaliar, classificando-o entre um de dois tipos. Sendo assim, os procedimentos podem ser de dois tipos: **non-leaf routines**, se são funções que se invocam a si próprias ou outras funções; **leaf routines**, se não invocarem outras funções, requerendo, ou não, espaço no stack para variáveis locais.

**non-leaf routines**  
**leaf routines**

Então mas, em suma, como é que podemos apresentar uma invocação de procedimentos? Primeiro temos de colocar os parâmetros em registos, guardando no stack todos os registos temporários em utilização que são essenciais; em segundo temos de transferir a execução do programa para a execução do procedimento, através da instrução do MIPS *jal*; em terceiro devemos adquirir algum espaço de memória no stack para a execução do procedimento (variáveis locais) - devem guardar-se todos os registos \$s que o procedimento use, no stack; em quarto executamos o procedimento; em quinto devemos colocar o resultado num registo para passar ao invocador e restaurar, copiando do stack, o conteúdo dos registos do tipo \$s, e libertar o espaço de memória do procedimento, isto é, restaurar o valor de \$sp; por fim devemos regressar ao ponto de partida, isto é, ao ponto do programa onde foi invocado o procedimento, através da instrução *jr*. Ainda mais em suma, podemos descrever os dois primeiros passos como intervenções do **caller** (em português “invocador”) e os outros restantes passos como intervenções do **callee** (em português “invocado”).

**caller**  
**callee**

A nossa memória fica com uma forma, em comparação com outros estados de representação apresentados anteriormente, muito mais detalhado, sendo que agora podemos incluir o **text segment** - código do programa -, **static data** - variáveis globais -, **dynamic data** - designada por **heap** (corresponde a *malloc* em linguagem C e *new* em Java) - e **stack** - armazenamento automático de variáveis locais. (Figura 29)

**text segment, static data**  
**dynamic data, heap**  
**stack**

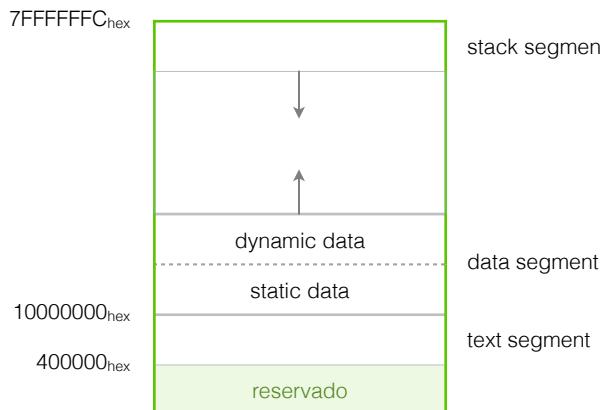


figura 29  
memória

## Preservação de valores sob invocações de procedimentos

Existem responsabilidades a cumprir aquando da invocação de procedimentos pelo MIPS. Da parte do invocador, isto é, do *caller*, temos que os registos \$t0 a \$t9, denominados de registos temporários não são preservados, tal como os registos de argumentos \$a0 a \$a3 ou de retorno de funções \$v0 e \$v1. Do mesmo modo o conteúdo abaixo de \$sp é sempre perdido, sob a responsabilidade do *caller*.

Por outro lado, sob a responsabilidade do *callee*, isto é, do invocado, temos que os registos de \$s0 a \$s7 são preservados, tal como o \$sp (*stack pointer*) e o registo \$ra (*return address*). Do mesmo modo, o conteúdo acima do valor em \$sp é preservado.

## Aplicação de procedimentos

Vejamos um exemplo de procedimento que não invoca outro, no Código 36.

```
int leaf_example (int g, h, i, j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

código 36  
código de estudo em C

Como é que traduzimos o Código 35 para código Assembly do MIPS? Vejamos o Código 37.

```
addi $sp, $sp, -4
sw $s0, 0($sp)
add $t0, $a0, $a1
add $t1, $a2, $a3
sub $s0, $t0, $t1
add $v0, $s0, $zero
lw $s0, 0($sp)
addi $sp, $sp, 4
jr $ra
```

código 37  
exemplo de aplicação

Quando analisamos o Código 35 que aspectos deverão ser analisados de modo a que consigamos completar o Código 36? Primeiro devemos classificar se as funções são terminais ou não, isto é, se apenas chama uma função ou não. Neste caso estamos a estudar uma função que é terminal, pelo que dentro da função não há novas invocações. Depois devemos enquadrar as variáveis em registos (aqui será nos registos \$a0 a \$a3 e f em \$s0, o que é necessário preservar na stack) e o resultado será em \$v0.

No caso de procedimentos que invocam outros procedimentos, isto é, para invocações em cadeia, o *caller* precisa de guardar na stack o seu endereço de retorno - dado que o endereço em \$ra será sempre substituído por cada instrução *jal* - e os valores de argumentos e variáveis temporárias de que necessite depois da invocação. No final da execução há que restaurar o stack quando o procedimento invocado retorna.

Vejamos o caso seguinte, onde se representa o algoritmo de ordenação **bubble sort** em C (Código 38).

```
void swap(int a[], int k) {
    int temp;
    temp = a[k];
    a[k] = a[k+1];
    a[k+1] = temp;
}
void sort(int v[], int n) {
    int i, j;
    for (i = 0; i < n; i+=1) {
        for (j = i - 1; j >= 0 && v[j] > v[j+1]; j-=1) {
            swap(v,j);
        }
    }
}
```

Em Assembly do MIPS, então, faz-se a seguinte tradução (Código 39):

```
sort:      addi $sp, $sp, -20          # 5
           sw   $ra, 16($sp)          # save $ra
           sw   $s3, 12($sp)          # save $s3
           sw   $s2, 8($sp)           # save $s2
           sw   $s1, 4($sp)           # save $s1
           sw   $s0, 0($sp)           # save $s0
           move $s2, $a0
           move $s3, $a1
           move $s0, $zero
           for1tst:    slt  $t0, $s0, $s3
                         beq $t0, $zero, exit1
                         addi $s1, $s0, -1
           for2tst:    slti $t0, $s1, 0
                         bne $t0, $zero, exit2
                         sll  $t1, $s1, 2
                         add  $t2, $s2, $t1
                         lw   $t3, 0($t2)
                         lw   $t4, 4($t2)
                         slt  $t0, $t4, $t3
                         beq $t0, $zero, exit2
                         move $a0, $s2
                         move $a1, $s1
                         jal  swap
                         addi $s1, $s1, -1
           exit2:     j    for2tst
           exit1:     addi $s0, $s0, 1
                         j    for1tst
           swap:      lw   $s0, 0($sp)          # restore $s0
                         lw   $s1, 4($sp)          # restore $s1
                         lw   $s2, 8($sp)          # restore $s2
                         lw   $s3, 12($sp)         # restore $s3
                         lw   $ra, 16($sp)         # restore $ra
                         addi $sp, $sp, 20          # restore $sp
                         jr   $ra                  # return to callee
                         sll  $t1, $a1, 2
                         add  $t1, $a0, $t1          # $t1 = endereço de v[k]
                         lw   $t0, 0($t1)          # $t0 (temp) = v[k]
                         lw   $t2, 4($t1)           # $t2 = v[k+1]
                         sw   $t2, 0($t1)
                         sw   $t0, 4($t1)
                         jr   $ra
```

### bubble sort

#### código 38

#### bubble sort em C

#### código 39

#### bubble sort em assembly do MIPS

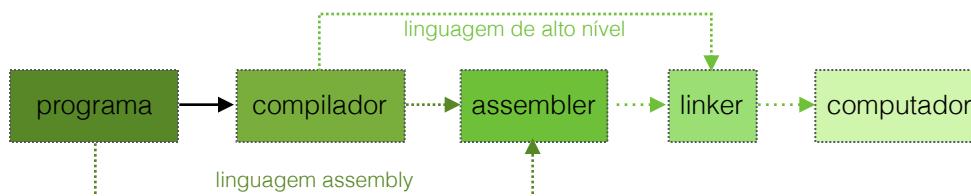
Um outro exemplo, possível de ser feito em Assembly, é de **recursividade**. Vejamos o caso clássico do procedimento de factorial (Código 40):

```
int fact(int n) {
    if (n < 1) return (1);
    else return n * fact(n - 1);
}
```

Em Assembly do MIPS, a função em Código 39 escreve-se como em Código 41.

```
fact:   addi $sp, $sp, -8      # ajusta o stack para 2 items
        sw  $ra, 4($sp)       # save return address
        sw  $a0, 0($sp)       # save argument
        slti $t0, $a0, 1       # verify if n < 1
        beq $t0, $zero, L1     #
        addi $v0, $zero, 1       # if n<1, resultado=1
        addi $sp, $sp, 8       # clean stack
        jr   $ra                # return
L1:    addi $a0, $a0, -1       # else n-=1
        jal  fact              # recursive call
        lw   $a0, 0($sp)       # restore n original value
        lw   $ra, 4($sp)       # restore return address
        addi $sp, $sp, 8       # pop 2 items from stack
        mul  $v0, $a0, $v0       # multiply to obtain result
        jr   $ra                # return
```

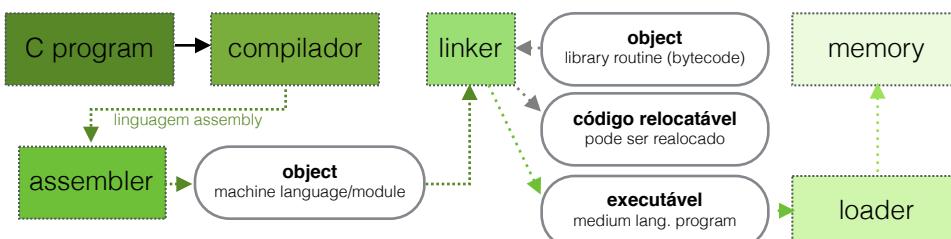
## O processo de assemblagem



Um programador tem duas alternativas para criar um programa: pode criá-lo numa linguagem de alto-nível, código que é depois traduzido num **compilador** para Assembly da máquina em questão, sendo este novamente traduzido para uma linguagem máquina, por um **assembler**, denominada de **bytecode**; pode criá-lo em Assembly, diretamente, sendo diretamente criada em baixo-nível.

O Assembly permite ao utilizador recorrer a funções da biblioteca, em código máquina, já executável, sendo que para as utilizar tem que fazer a ligação do código do nosso programa com o código das funções que ele utiliza, o que é garantido pelo **linker** - editor de ligações -, o que é possível verificar na Figura 30.

Vejamos o seguinte exemplo representativo do processamento de um programa em linguagem C (Figura 31).



**código 40**  
código de estudo em C

**código 41**  
exemplo de aplicação

**figura 30**  
processo de assemblagem

**compilador**

**assembler, bytecode**

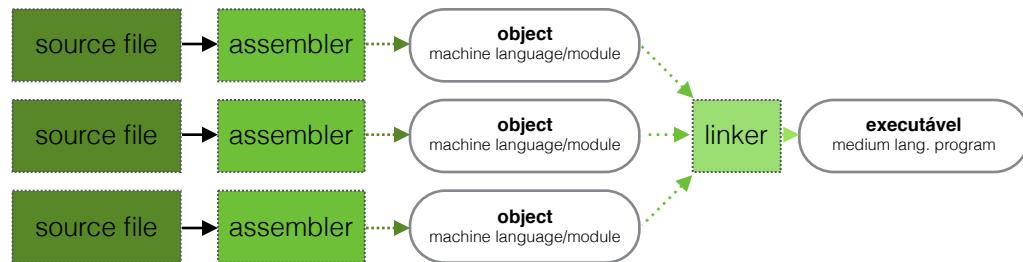
**linker**

**figura 31**  
exemplo de assemblagem

# 33 ARQUITETURA DE COMPUTADORES I

Na Figura 31 há um passo de processamento que contém um **loader**. Um loader é um programa que carrega em memória o ficheiro executável, o qual deve poder ser executado em qualquer posição de memória. O loader pode substituir os endereços que forem carregados anteriormente, sendo que o código relocatável torna-se código máquina que se encontra em uso.

De uma forma muito mais generalizada, temos que o processo de assemblagem se pode representar da seguinte forma, como representada pela Figura 32.



**nota!** denota-se a capacidade de modularidade que é concretizável no processo de assemblagem.

Mas quais são as tarefas do assembler? A que é que ele se deve reger? Primeiro ele deve seguir as diretivas expressas em código (`.data`, `.text`, `.eqv`, ...) que indicam diferentes percursos de dados e detalhes acerca de alocações de memória. Segundo há que dar relevo a possíveis pseudoinstruções, também denominadas de **instruções virtuais**, e traduzi-las para **instruções nativas**. Terceiro, há que construir uma tabela, com efeito meramente interno, onde se preservam a lista de *labels* e endereços, esta denominada de **symbol table**. A symbol table conterá símbolos que são usados na assemblagem, internos, e símbolos que são levados para o linker, externos. Quarto, há que traduzir para linguagens máquina e só depois, (quinto e último passo) é que se cria um *object file*.

## Diretivas do MIPS

Como já pudemos reparar, o Assembly do MIPS permite-nos caracterizar determinadas zonas e dados dos nossos códigos. Para tal usam-se as **diretivas**, isto é, palavras reservadas que se destacam por se iniciarem com um carácter “.”. Estas diretivas, como também já foi referido atrás, permitem ao compilador saber como traduzir o programa, sendo que estas não executam instruções máquina. Entre muitas outras, as diretivas que mais usamos são as seguintes:

- **.text** (addr) - indica que o que se segue é para ser colocado no segmento de texto, a partir de “addr”;
- **.align** n - alinha o dado que se segue com  $2^n$  byte boundary (por exemplo, se escrevermos “align 2” estamos a referir um *word boundary* seguinte);
- **.globl** sym - declara a *label* “sym” como global. Assim, “sym” fica declarado na symbol table acessível para o linker. Isto faz com que “sym” possa ser referenciado noutras ficheiros no mesmo diretório, o que permite modularidade;
- **data** (addr) - os itens subsequentes são colocados no segmento de dados, a partir de “addr”;
- **asciiz** str - permite armazenar na memória uma string incluindo o carácter NUL ('\0'), no final da string (terminador).

**loader**

**figura 32**  
**exemplo de assemblagem**

**nota**

**instruções virtuais,**  
**instruções nativas**  
**symbol table**

**diretivas**

## Pseudoinstruções

Algumas das instruções podem ser escritas de forma pseudoinstrutiva, sendo que o assembler deve ser capaz de saber traduzir para uma ou mais instruções nativas, como se pode ver a seguir (Código 42).

```
# a pseudoinstrução
move $t0, $t1
# pode ser convertida nas seguintes instruções nativas
add $t0, $zero, $t1

# a pseudoinstrução
blt $t0, $t1, L1
# pode ser convertida nas seguintes instruções nativas
slt $at, $t0, $t1
bne $at, $zero, L1

# a pseudoinstrução
li $t0, 0x12345
# pode ser convertida nas seguintes instruções nativas
lui $t0, 0x1
ori $t0, $t0, 0x2345
add $t1, $t0, $at

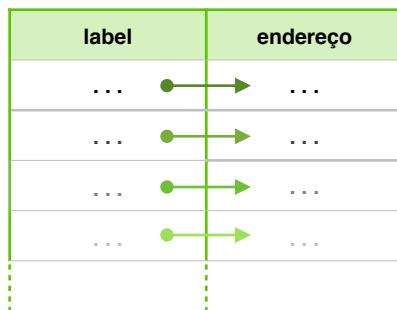
# a pseudoinstrução
add $t1, 0x12345
# pode ser convertida nas seguintes instruções nativas
lui $at, 0x1
ori $at, $at, 0x2345
add $t1, $t1, $at
```

**código 42**  
pseudoinstruções

## Assembler de duas passagens

Como é que o assembler faz uma tradução? O assembler percorre duas vezes o texto todo do programa, isto é, faz duas passagens: na primeira passagem constroi uma symbol table, como já foi referido atrás, e como é visível na Figura 33 - nesta figura é visível que para cada símbolo que o assembler encontra, definidos pelo utilizador (como *labels*, variáveis, constantes, ...) ele regista um **endereço relativo**; na segunda passagem o assembler gera o então chamado código máquina.

endereço relativo



**figura 33**  
symbol table

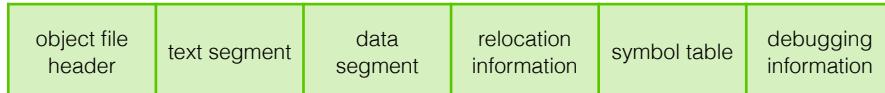
As *labels* (em português “etiquetas”) podem ser de dois tipos:

- podem designar algo local, isto é, o objeto pode ser referenciado no ficheiro;
- podem ser globais, ou seja, podem ser referenciados fora do ficheiro.

Dado que as *labels* podem designar objetos contidos dentro de ficheiros, só nos falta saber como caraterizar um ficheiro e estudar como os seus parâmetros são importantes para a arquitetura.

## Formato de um ficheiro

Consideremos o seguinte ficheiro em UNIX, caracterizado pelos parâmetros nele contidos, representado na Figura 34.



Saber caracterizar um ficheiro em UNIX é saber defini-lo consoante os parâmetros explícitos na Figura 34. Para tal, vejamos o que significa cada um desses aspectos: o **object file header** serve para descrever o tamanho e a posição de outros campos do ficheiro do tipo *object*; o **text segment** contém o código máquina das rotinas no ficheiro-fonte, o que pode não ser executável devido a possíveis referências externas; o **data segment** contém a representação em binário dos dados no ficheiro-fonte, sendo que podem estar incompletos devido a referências externas; a **relocation information** identifica as instruções e dados que dependem dos endereços absolutos quando o programa é carregado em memória, o que é necessário dado que o assembler não sabe que posições de memória é que o código e os dados vão ocupar quando forem ligados ao resto do programa; a **symbol table** que contém as *labels* que ainda não estão definidos (referências externas, ...) e as variáveis que são globais (referenciados em outros objetos); e a **debugging information** que se traduz pela informação que permite associar instruções máquina com as instruções no ficheiro-fonte.

**figura 34**  
ficheiro em UNIX

**object file header**  
**text segment**  
  
**data segment**  
**relocation information**  
  
**symbol table**  
  
**debugging information**

## Linker

O que é que faz um **linker**? O linker, ou link editor, pega em diferentes programas do tipo objeto e liga-os entre si. Ele tem três passos de processamento: primeiro ele procura nas bibliotecas as rotinas usadas pelo programa; segundo, determina as posições de memória que o código de cada módulo ocupará e realoca as respetivas instruções ajustando as referências absolutas; terceiro, e por fim, resolve as referências entre ficheiros (o ficheiro executável terá todo o código a ser executado no programa, sendo que já se sabe a que correspondem todas as referências externas).

**linker**

## Loader

Os ficheiros executáveis têm de ser submetidos ao **loader**. Este processo lê o header do ficheiro executável para determinar o tamanho dos segmentos de texto e dados. Para tal, cria um espaço de endereçamento para o programa (texto, dado e stack). Copiando instruções e dados para o espaço de endereçamento atribuído, também copia os argumentos passados ao programa para o stack. No fim, inicializa os registos \$sp (topo do stack) e salta para a rotina de inicialização que copia os argumentos do stack para os registos.

**loader**

## 3. Aritmética Binária

Da disciplina de Introdução aos Sistemas Digitais (a1s1) nós já devemos saber efetuar cálculos em base binária. De qualquer das formas, iremos rever os conceitos base, pois a partir deste ponto, a sua inteira interpretação e compreensão são assunto mais-que-importante ao longo de todo o curso.

## Soma e subtração de quantidades binárias

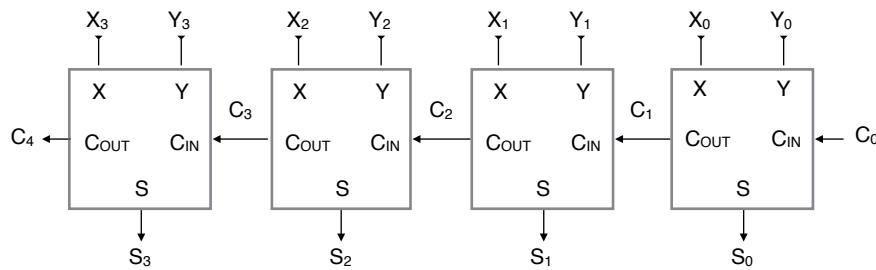
Já sabemos que em MIPS a representação de inteiros é feita em 32 bits, sen que a representação de números negativos é feita em complemento para dois. Sendo assim, para valores sem sinal (*unsigned*), se o valor da soma for maior que  $2^n - 1$  e o *carry-out* da posição mais significativa for 1 ( $c_{out} = 1$ ), então o valor da soma excede a gama de representação, isto é, ocorre **overflow** - o valor não é representado em  $n$  bits. Em complemento para dois, se o *carry-out* do bit mais significativo for diferente do bit anterior, então ocorre overflow.

Em MIPS, quando ocorre overflow em operações como *add*, *addi*, *sub*, ... é sempre gerada uma **exceção**.

No caso da subtração, em complemento para dois, o MIPS tem como algoritmo, ela ser equivalente à **soma simétrica**, sendo que é somado ao subtraendo, o complemento para dois do subtrator. Dado isto, temos que é realizada a seguinte operação:  $A - B = A + (2^n - B)$ . Lembremo-nos que para obter o complemento para dois basta negar todos os bits (complemento para um) e somar 1.

Em suma, os resultados excedem a gama de representação se forem quantidades sem sinal e o *carry-out* do bit mais significativo for 1 ou se, em complemento para dois, o *carry-out* do bit mais significativo for igual ao do bit anterior.

Para implementar estas operações em hardware criam-se sistemas digitais: os **somadores** (neste caso, somadores completos). Um somador de 1 bit, completo - denominado em inglês de *full-adder* (FA) -, tem três entradas ( $A$ ,  $B$  e  $C_{in}$ ) e duas saídas ( $S$  e  $C_{out}$ ).



Na Figura 35 podemos ver uma aplicação de quatro somadores completos, formando um **ripple adder**. Os ripple adder são somadores completos ligados em cascata, só que têm um inconveniente - são componentes muito lentos, pois o seu atraso é calculado através do número de somadores a multiplicar pelo tempo de atraso de cada um -  $t_{ripple} = n t_{FA}$ .

## Unidade aritmética e lógica (ALU)

Um processador como o MIPS não pode servir-se apenas de somadores completos, dado que precisa de ser capaz de efetuar outras operações. Por esta razão terá de ser criada uma unidade que seja capaz de atribuir a designação de operação a ser executada e executá-la ativando segmentos (exertos) de um determinado circuito. Cria-se então a **unidade aritmética e lógica** (em inglês *arithmetic and logic unit*, daí o acrônimo **ALU**). Na Figura 36 encontra-se representada uma ALU de tratamento de 1 bit de dados, sendo que a preto se designam os sinais de dados (decorrentes do datapath) e a verde se designam os sinais de controlo, provenientes de uma unidade de

**overflow**

**exceção**

**soma simétrica**

**somadores**

**figura 35**  
**ripple adder**

**ripple adder**

**unidade aritmética e lógica**  
**ALU**

controlo que opera sobre todo um circuito do processador, como já referimos anteriormente.

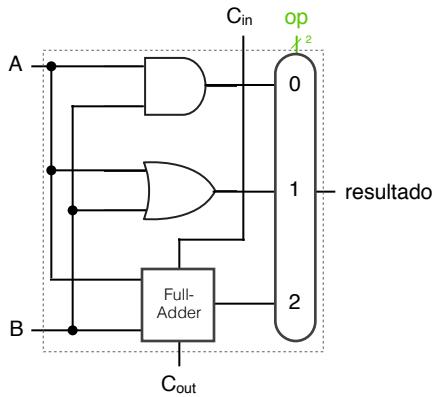


figura 36

ALU

A unidade aritmética e lógica da Figura 36 tem como códigos de operação 00 para a instrução AND, 01 para OR e 10 para ADD.

Graficamente, de forma abreviada, as unidades aritméticas e lógicas representam-se como na Figura 37.

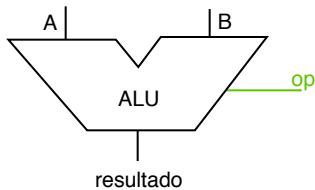


figura 37

ALU (forma abreviada)

## Multiplicação binária

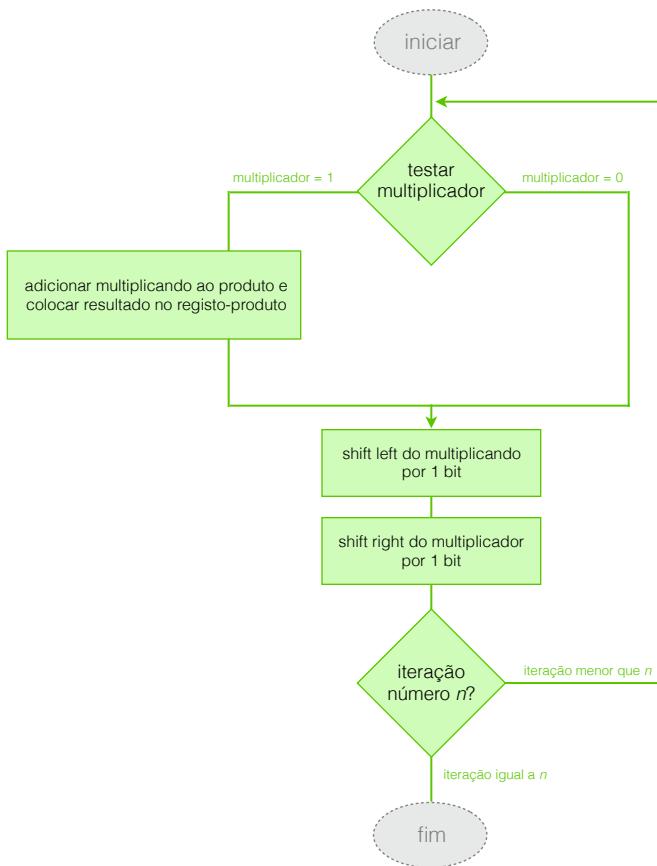
A **multiplicação** binária não tem um algoritmo muito fácil de compreensão. Este algoritmo é diferente entre quantidades com sinal e sem sinal. Comecemos por analisar o caso de quantidades não negativas, isto é, sem sinal (*unsigned*). Como podemos multiplicar então as quantidades  $1000_2$  e  $1001_2$ ? Em decimal seria  $8_{10} \times 9_{10} = 72_{10}$ . Em binário o algoritmo para quantidades sem sinal é igual ao algoritmo da multiplicação em decimal, sendo que fazemos multiplicações bit-a-bit do multiplicador ( $1001$ ), seguido de somas sucessivas, até obter o resultado final com o dobro dos bits iniciais (de multiplicador e multiplicando com 4 bits, temos um produto de 8 bits), sendo que o nosso produto final é  $01001000_2$  que é, de facto, igual a  $72_{10}$ .

No caso de quantidades com sinal a abordagem é totalmente diferente. Primeiro temos de avaliar as quantidades que temos em complemento para dois, dado que deixámos de ter  $1000_2$  como  $8_{10}$  e  $1001_2$  como  $9_{10}$ , para termos  $1000_2$  como  $-8_{10}$  e  $1001_2$  como  $-7_{10}$ . Neste caso, ao estudar as multiplicações sucessivas com o primeiro bit menos significativo do multiplicador, no final faz-se uma extensão do sinal, passando a seguir para o bit seguinte. A grande diferença dos algoritmos é que neste, quando se chega ao bit mais significativo do multiplicador, para a soma sucessiva coloca-se o complemento para dois do multiplicando, neste caso  $01000_2$ . Fazendo as contas, dá o resultado de  $00111000_2$  o que é igual a  $56_{10}$ .

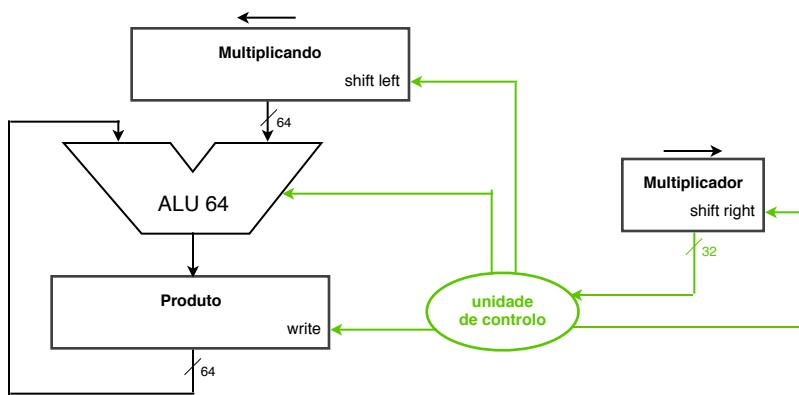
Para implementar isto num circuito primeiro tentemos sintetizar passo-a-passos, a instrumentação para o cálculo de multiplicações na base binária, com e sem sinal. Na Figura 38 podemos ver então um fluxograma que pretende sumariar todos os passos do algoritmo para uma multiplicação de  $n$  bits.

multiplicação

**figura 38**  
algoritmo de multiplicação



Tendo agora o fluxograma feito é-nos mais fácil implementar num circuito todo este algoritmo denominado de **add-shift**. Sendo assim, com recurso a ALU e a unidade de controlo, temos a Figura 39.

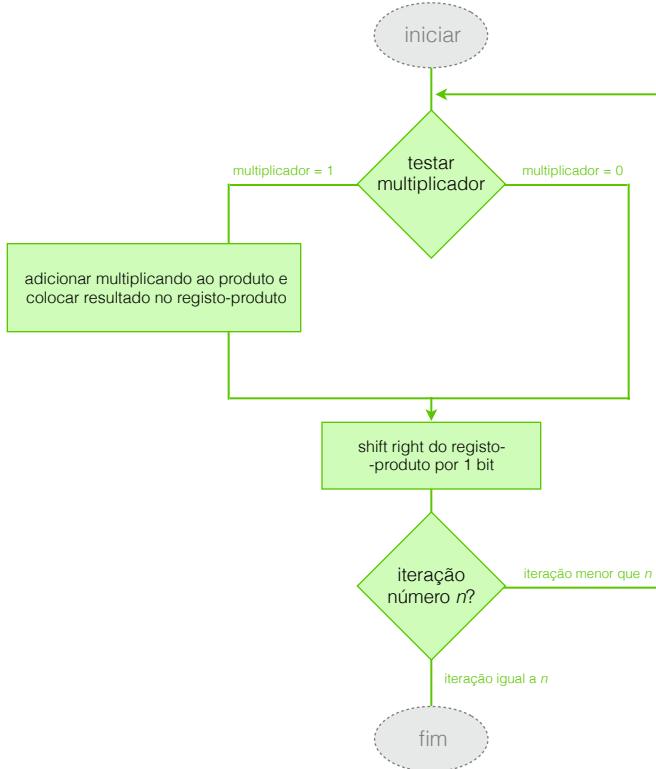
**add-shift**

**figura 39**  
círcuito multiplicador

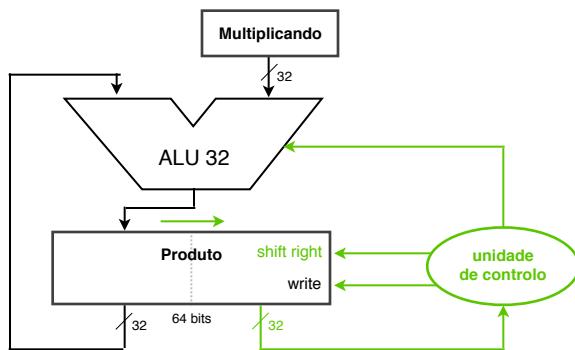
Se agora analisarmos bem o circuito da Figura 39 podemos reparar que este tem um custo muito grande, pelo que temos um multiplicando de 32 bits e um registo de 64 bits para o armazenar. Se num exemplo mais básico usámos  $n$  bits, porque é que aqui estamos a usar o dobro dos bits ( $2n$  bits)? Isto só pode significar que podemos fazer melhor. Para tal, vamos usar o registo-produto para armazenar, na parte direita, o multiplicador (sendo que a parte esquerda é inicializada a zeros, a qual ficará com os

# 39 ARQUITETURA DE COMPUTADORES I

produtos parciais). Com esta melhoria podemos reduzir a complexidade do circuito para metade. Vejamos então os efeitos da melhoria no nosso algoritmo, começando por analisar o fluxograma da Figura 40.



Dadas estas alterações, agora estamos em condições plenas de implementar o circuito (Figura 41).



Os multiplicadores também são passíveis de ser ligados em cascata, tornando-se em **multiplicadores em paralelo**, em inglês *array multiplier* (isto, referindo quantidades *unsigned*).

## Algoritmo de Booth

Para a multiplicação com quantidades com sinal existem outros algoritmos que podem ser aplicados, entre os quais, o **algoritmo de Booth**. Através de um recodificação de um multiplicador, as suas entradas podem ficar mapeadas de um

**figura 40**  
algoritmo de multiplicação  
melhorado

**figura 41**  
círculo multiplicador  
melhorado

**multiplicadores em paralelo**

**algoritmo de Booth**

forma como a seguinte, na qual o bit  $y_i$  e o bit anterior  $y_{i-1}$  correspondem a diferentes operações, consoante a sua combinação de entradas. Tomando a ordem dada de bits em consideração, temos então que se tivermos 00 a operação a realizar é de shift right, 01 a operação será adição seguida de shift, 10 a operação a realizar é de subtração seguida de shift (ou adição de complemento para dois seguido de shift) e 11 é shift. Vejamos o seguinte exemplo: pretende-se calcular  $2_{10} \times -4_{10}$ .

- **passo 1** - preencher o registo-produto com zeros na primeira metade e com o multiplicador na segunda metade ( $-4_{10}$ );
- **passo 2** - preencher o multiplicando ( $2_{10}$ );
- **passo 3** - preencher o espaço “bit anterior” com valor 0;
- **passo 4** - preencher o espaço “bit atual” e executar operação em questão;
- **passo 5** - executar shift right no registo-produto (para manter o sinal correto);
- **passo 6** - atualizar espaço “bit anterior” e repetir passos desde *passo 4*, até que o número de ciclos seja maior ou igual que o número de bits.

Eis uma tabela já preenchida para o caso em estudo (Figura 42).

registo de produto	multiplicando	bit atual (c)	bit anterior (p)	operação	ciclo
00000 11100	00010	0	0	shift right arithmetic	ciclo 0
00000 01110	00010	0	0	shift right arithmetic	ciclo 1
00000 00111	00010	1	0	subtração + sra	ciclo 2
11110 00111	00010			shift right arithmetic	
11111 00011	00010	1	1	shift right arithmetic	ciclo 3
11111 10001	00010	1	1	shift right arithmetic	ciclo 4
11111 11000	00010	0	1	soma + sra	ciclo 5
00001 11000	00010				

A resposta são, neste caso, os cinco bits menos significativos, sendo que tal se confirma: em decimal  $2 \times -4 = -8$ , pelo que o resultado sendo  $11000_2$  em complemento para dois dá  $-8_{10}$ .

## Multiplicação no MIPS

Existem dois registos no processador MIPS que se definem para a execução de um produto (multiplicação). Quando multiplicamos dois números nós podemos facilmente exceder a nossa gama de representação de 32 bits. Porque é preferível ter um registo de 64 bits, mas tal não é possível no MIPS, temos dois registos que se comportam quase como um só: são eles os **HI** e o **LO**. Estes registos foram há pouco representados na Figura 41, sendo que na representação do registo-produto que se encontra dividida por uma linha a traço interrompido, a primeira metade é o registo HI e a segunda metade corresponde ao registo LO.

As **instruções** de multiplicação no MIPS são as seguintes (Código 43):

```
mult    $rs, $rt      # HI_L0 = rs * rt
multu   $rs, $rt      # HI_L0 = rs * rt (unsigned)
```

**figura 42**  
algoritmo de Booth  
(exemplo)

**HI, LO**

**instruções**

**código 43**  
**instruções mult e multu**

Se fizermos a seguinte operação, apenas serão copiados os 32 bits menos significativos para o registo simbolicamente designado por *rd* (esta operação também se designa de multiplicação) - Código 44.

```
mul      $rd, $rs, $rt    # rd = rs * rt (LSB – 32 bits)
```

código 44

instrução mul

Para ter acesso aos registo HI e LO há que executar duas instruções especiais (Código 45).

```
mfhi    $rd          # copia de HI para rd  
mflo    $rd          # copia de LO para rd
```

código 45

instruções mfhi e mflo

## Divisão binária

A última operação passível de ser feita e que ainda não abordámos foi, de facto, a **divisão**. Em decimal, quando queremos dividir uma quantidade *x* por uma *y* comparamos o divisor com a parte mais significativa do dividendo, consecutivamente, pedindo o divisor mais próximo de *y*. Por outras palavras, se quisermos saber quanto é 16 a dividir por 4, perguntamos quantas vezes é que 4 cabe em 16. Então como é que podemos processar este algoritmo para o caso de quantidades binárias?

Com quantidades binárias o algoritmo é o seguinte:

- **passo 1** - verificar se divisor é 0:
  - se **não**:
    - divisor é menor que bits do dividendo?
    - se **sim**: novo bit do quociente é 1 e subtrair;
    - se **não**: novo bit do quociente é 0;
    - juntar ao resto parcial o bit seguinte do dividendo;
- **passo 2** - subtrair sempre o divisor:
  - se resto for menor que 0:
    - somar de novo o divisor;

Se estivermos a trabalhar com quantidades negativas devemos dividir usando os valores absolutos e só no fim ajustar o sinal do quociente e do resto.

Para implementar isto num circuito devemos experimentar criar um fluxograma para estudar o seu funcionamento (Figura 43).

Tendo o fluxograma feito podemos passar para a criação do circuito (Figura 44).

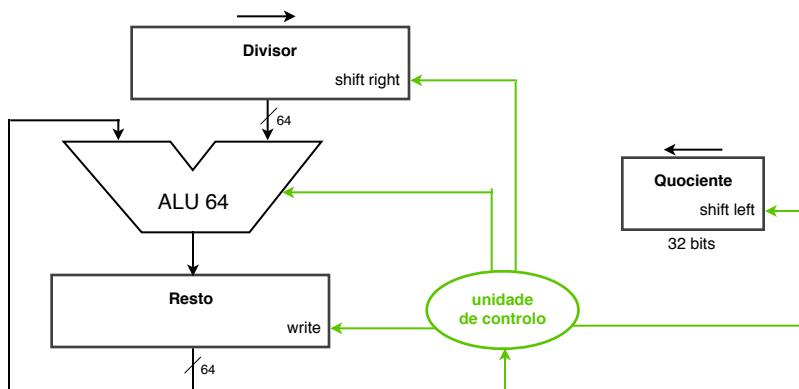
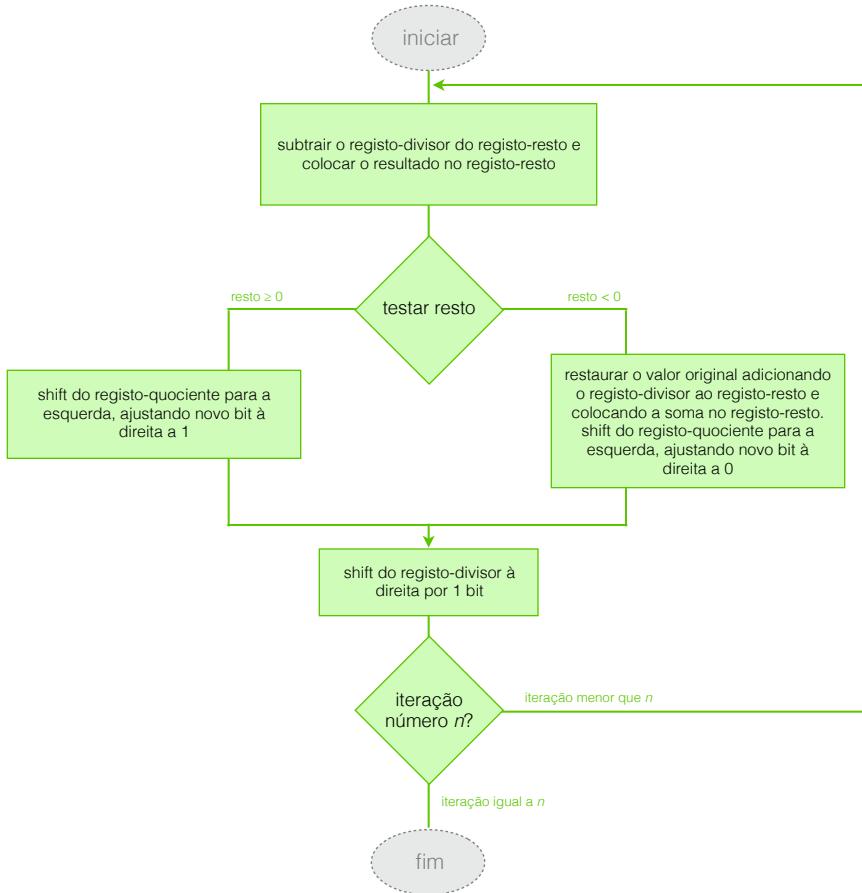
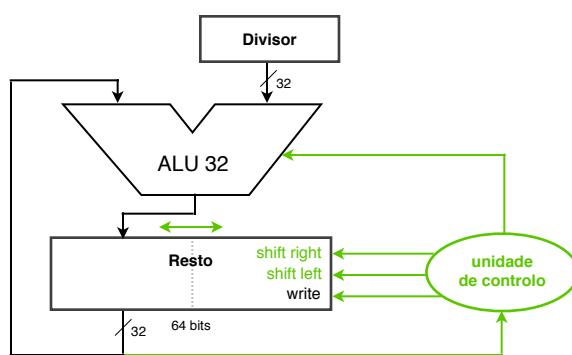


figura 44

círcuito divisor



**figura 43**  
algoritmo da divisão



**figura 45**  
circuito divisor melhorado

## Divisão no MIPS

No MIPS, mais uma vez, os registo HI e LO são usados para preservar os resultados obtidos através de uma operação de divisão. Sendo assim, e dado que o registo HI está representado como a primeira metade do registo-resto da Figura 45 e o registo LO como registo-quociente, o MIPS incorpora no seu ISA as seguintes instruções de divisão (Código 44).

```
div      $rs, $rt      # rs / rt -> L0 e HI
divu     $rs, $rt      # rs / rt -> L0 e HI (unsigned)
```

**nota!** o MIPS não faz verificação de exceções do tipo “divisão por zero” (*divide-by-zero*), sendo que cabe ao próprio software fazê-lo.

**código 46**

**instruções div e divu**

**nota**

## 4. Representação de Vírgula Flutuante

Muitas vezes temos necessidade de representar dados com uma precisão muito maior que a disponível através de quantidades inteiras. Para trabalhar com frações de quantidades os tipos inteiros não são propriamente úteis para o programador, dado que ele, com eles, terá sempre de fazer arredondamentos por excesso ou por defeito, no pior dos casos, com um erro de 0.5 (meio inteiro).

A matemática evoluiu desde que houve a necessidade de representar quantidades fracionárias, sendo que anos mais tarde, o grau de precisão que a comunidade científica, entre outras, necessitou para caracterizar certos passos dos seus trabalhos era de tal ordem grande que houve necessidade de desenvolver novos métodos para a caracterização de quantidades de forma a aumentar o grau de precisão - criou-se então a **notação científica**. Na notação científica representa-se sempre uma quantidade contável, com o acrescento da ordem de grandeza a que pertence. A quantidade contável, por norma, é uma quantidade normalizada, isto é, que segue uma convenção clara de número de algarismos significativos pré-ordem-de-grandeza.

Em computadores, a notação análoga à notação científica é a notação de **vírgula flutuante**. Assim, mantendo sempre uma base  $b$  (desnecessária de incluir na representação dado que é sempre a mesma), tem um **significando**  $m$  (por exemplo, 1.370, onde 370 é a **fração** ou **mantissa**) e um **expoente**  $exp$ . Sendo assim, a Equação 1 dá-nos uma representação de vírgula flutuante  $X$ .

$$X = m \times b^{exp}, \quad X = (m, exp)$$

**notação científica**

A **gama de representação** é determinada pelo número de bits do expoente. A **precisão**, por sua vez, é determinada pelo número de bits de significando.

**equação 1**

**vírgula flutuante**

Ao longo dos tempos foram existindo vários formatos de representação para vírgula flutuante, sendo que atualmente utiliza-se um standard (norma, em português) para a representação, denominado de **IEEE-754**. Esta norma especifica os seguintes tópicos:

**gama de representação**  
**precisão**

**IEEE-754**

- os formatos de vírgula flutuante de precisão simples e de precisão dupla normalizados;
- as exceções do formato são apenas seis, sendo elas o +0, -0, +∞, -∞, quantidades desnormalizadas e não-números (representados por NaN - *not-a-number*);
- as operações de adição, subtração, multiplicação, divisão, raiz quadrada, resto e comparação;
- conversões entre inteiros e vírgula flutuante;
- conversões entre formatos de vírgula flutuante;
- conversões entre vírgula flutuante e representação decimal;
- modos de arredondamento;
- exceções e seu tratamento.

Dado que esta norma esclarece globalmente o tópico da vírgula flutuante, estudemos ponto-por-ponto, sempre que possível, exemplificando um caso de aplicação.

## Formato de representação

Como o próprio standard já declara, existem dois graus de representação a nível de precisão: **single precision** e **double precision** (em português, precisão simples e precisão dupla, respectivamente). Estes dois formatos diferem no número de bits que dão suporte à representação.

Vejamos primeiro de uma forma global como é que se processa a representação de um número em vírgula flutuante, sem nos preocuparmos primeiro com o nível de precisão. Sendo assim, a Figura 46 pretende representar uma dada quantidade numérica.



**single precision, double precision**

Como podemos verificar pela Figura 46, nela não é representado qualquer parte inteira do significando, sendo que este tem sempre a forma  $1.(frac)$ , onde 1 é a quantidade inteira do significando - sempre 1, dado que estamos em formato binário e temos a normalização natural da representação - e  $(frac)$  é a fração ou mantissa, representada por  $m$  bits na Figura 46. Dado que a quantidade inteira do significado é sempre 1, a este bit damos o nome de **hidden bit**.

Vejamos então agora como é que enquadramos este raciocínio para o caso da precisão simples (Figura 47).



**figura 46**  
quantidade em vírgula flutuante

**hidden bit**

Para o caso da precisão simples, o sinal da quantidade representada tem espaço de 1 só bit (sendo que este pode tomar os valores convencionados anteriormente de 0 para positivo e 1 para negativo), seguido do expoente que tem espaço de 8 bits e da mantissa em 23 bits. Por partes, o expoente é uma representação efetuada por excesso<sup>2</sup>, sendo que o seu valor é calculado através da expressão em Equação 2.

$$\text{expoente} = \text{exp} + \text{bias} - \text{exp}(\text{unsigned}), \quad \text{bias} = 127$$

**figura 47**  
quantidade em precisão simples

**equação 2**  
**cálculo do expoente**

No caso da dupla precisão temos que a representação é feita de acordo com a Figura 48.



**figura 48**  
quantidade em precisão dupla

Neste caso o expoente é obtido através da expressão em Equação 2, com a particularidade do *bias* ser igual a 1023. Em suma, uma quantidade  $x$  em vírgula flutuante é obtida por:  $x = (-1)^{\text{sinal}} \times (1 + \text{frac}) \times 2^{(\text{exp}-\text{bias})}$ .

<sup>2</sup> a representação do expoente em excesso permite que seja aplicada a mesma lógica de comparação para inteiros e para vírgula flutuante.

## Representação de valores especiais

Dependendo do valor do expoente da representação, tal como do valor da própria mantissa, alguns **valores especiais** podem ser representados - valores esses já referidos anteriormente. Sendo assim, a tabela na Figura 49<sup>3</sup> contém todas as combinações necessárias para a representação de tais valores especiais.

**valores especiais**

expoente	fração	valor
-bias	zero	$\pm 0$
-bias	diferente de zero	$0.\text{frac} \times 2^{\exp\_min}$
entre $\exp\_min$ e $\exp\_max$	indiferente	$1.\text{frac} \times 2^{\exp}$
$\exp\_max + 1$	zero	$\pm\infty$
$\exp\_max + 1$	diferente de zero	NaN

**figura 49**  
**valores especiais**

## Gamas de representação por precisão

Então mas quais são as **gamas de representação** por precisão simples e por precisão dupla? Tendo sempre os expoentes 000...000 e 111...111 reservados, em ambas as precisões, em precisão simples, o menor valor representável é o que tem como expoente 00000001, isto é, com valor efetivo de  $1 - 127 = -126$  e com mantissa 0...000, sendo que o significando assim fica 1.0. Dado isto, visivelmente, a menor quantidade representável em vírgula flutuante de precisão simples é  $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$ . Já em relação ao maior valor representável, este é aquele que tem como expoente 11111110, tendo valor efetivo de 127 (expoente) e mantissa 1111...1111, sendo que o significando fica com o aspeto  $1.1111111 \approx 2.0$ . Sendo assim, visivelmente referimo-nos a  $\pm 2.0 \times 2^{127} \approx \pm 3.4 \times 10^{38}$ .

**gamas de representação**

No caso da precisão dupla apenas temos de aplicar o conhecimento usado para a precisão simples, mas extendendo para uma maior representação. Sendo assim, o menor valor representável em precisão dupla é  $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$ , e o maior valor representável em precisão dupla é  $\pm 2.0 \times 2^{1023} \approx \pm 1.8 \times 10^{308}$ .

**erro relativo**

Relativamente a **erro relativo** da representação, todos os bits do significando são significativos, sendo que em representação simples é aproximadamente  $2^{-23}$  o que é equivalente a  $23 \times \log_{10}2 \approx 23 \times 0.6 \approx 6$  dígitos decimais de precisão. Na representação dupla é aproximadamente  $2^{-52}$  o que é equivalente a  $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$  dígitos decimais de precisão.

## Conversão de quantidade decimal para vírgula flutuante

Como é que podemos converter uma quantidade decimal para a sua respetiva representação em vírgula flutuante? Consideremos como exemplo, então, a quantidade  $-0.75_{10}$ . Como é que representamos  $-0.75$  em vírgula flutuante? Primeiro convertemos para binário a quantidade  $0.75_{10}$ , através de multiplicações sucessivas por 2. Após este passo, neste caso, devemos ficar com  $0.11_2$ . Normalizando esta quantidade, temos que  $0.11_2 = 1.1_2 \times 2^{-1}$ . Só nos falta calcular o expoente - para tal apenas temos que ao expoente -1 somar o *bias* correspondente à precisão em questão (127 para simples e 1023 para dupla). A Figura 50 tem a representação final em precisão dupla e a Figura

<sup>3</sup> sabendo que para precisão simples,  $\exp\_max=127$  e  $\exp\_min=-126$ ; para precisão dupla,  $\exp\_max=1023$  e  $\exp\_min=-1022$ ; os desnormalizados têm  $\exp = -126$ , embora representado como -127.

51 a representação final em precisão simples. Note-se que a extensão da fração é preenchida por zeros, sendo que este valor não afeta o resultado real.

<b>1</b>	<b>0111 1110</b>	<b>1000 0000 0000 0000 0000 0000 0</b>	
neg	expoente	mantissa em 23 bits	simples
<b>1</b>	<b>0111 1111 110</b>	<b>1000 0000 0000 0000 0000 ... 0000</b>	
neg	expoente	mantissa em 52 bits	dupla

figura 50

resultado em precisão simples

figura 51

resultado em precisão dupla

Em suma, eis os passos para a conversão de decimal para vírgula flutuante:

- **passo 1** - representar a quantidade em causa em binário;
- **passo 2** - se a quantidade inicial for negativa, marcar bit sinal igual a 1, caso contrário, marcar bit sinal igual a 0;
- **passo 3** - normalizar a quantidade expressa em binário;
- **passo 4** - ao expoente da normalização, calcular expoente binário somando o *bias* de 127 ou 1023, consoante o tipo de precisão em questão.

## Conversão de quantidades em vírgula flutuante para decimal

Consideremos o número representado na Figura 52. Como é que podemos converter essa representação de vírgula flutuante em precisão simples, para decimal?

<b>0</b>	<b>0110 1000</b>	<b>1010 1010 1000 0110 1000 0100 0</b>	
pos	expoente	mantissa em 23 bits	simples

figura 52

quantidade em vírgula flutuante

Uma das expressões que referimos no nosso estudo anteriormente, para a representação de vírgula flutuante era que uma dada quantidade  $x$  era possível de ser calculada através de  $x = (-1)^{\text{sinal}} \times (1 + \text{frac}) \times 2^{(\text{exp}-\text{bias})}$ . Dado isto, primeiro vejamos o sinal: é positivo, pois o bit sinal é igual a 0. Segundo, o expoente é calculado através do ajuste do *bias*, sendo que sendo, em binário,  $01101000_2$ , em decimal faz-se representar pela quantidade  $104_{10}$ . Sendo o *bias* para precisão simples igual a 127, subtraindo a quantidade 127 ao 104 temos -13, o que representa  $2^{-13}$ . Por fim, em relação à fração (mantissa), dado que o expoente é diferente de zero, a fração é igual à representada na Figura 52. Convertendo o significando para decimal, temos que este é igual a  $1.0 + 0.66612$ , pelo que a quantidade final é  $1.66612_{10} \times 2^{-13} \approx 2.034 \times 10^{-4}$ .

Em suma, eis os passos para a conversão de vírgula flutuante para decimal:

- **passo 1** - converter expoente binário para decimal;
- **passo 2** - ajustar o *bias*, subtraindo 127 (1023, se for dupla precisão) à conversão obtida no passo anterior. O valor final é ordem de grandeza da expressão final;
- **passo 3** - converter significando para decimal;
- **passo 4** - calcular expressão final, juntando informação acerca do sinal.

## Adição em vírgula flutuante

Como é que podemos calcular somas diretamente em representação de vírgula, para que não corramos risco de perder precisão ao longo de todo o processo? Consideremos as quantidades  $9.999 \times 10^1$  e  $1.610 \times 10^{-1}$ . Como é que processa a soma destas duas quantidades? Primeiro temos de alinhar os operandos, sendo que o número que tiver menor expoente deve fazer shift à direita um número de bits suficientes até ter ambos os expoentes iguais. Neste caso teríamos  $9.999 \times 10^1$  e  $0.016 \times 10^1$ . Agora estamos em perfeitas condições para efetuar a soma:  $9.999 \times 10^1 + 0.016 \times 10^1 = = 10.015 \times 10^1$ . Como podemos reparar a quantidade final não se encontra normalizada, dado que excede a base numérica 10 - há que normalizar:  $1.0015 \times 10^2$ . Visto que não há underflow ou overflow, devemos arredondar e renormalizar, caso necessário:  $1.002 \times 10^2$ . Mas como é que se processa o **arredondamento**?

Os arredondamentos são processados da seguinte forma: consideremos a quantidade em soma  $4.584 \times 10^0 + 5.753 \times 10^3 = (5.753 + 0.004584) \times 10^3$  (em precisão total). Se supusermos que a adição é feita com quatro dígitos significativos, temos que  $(5.753 + 0.004) \times 10^3 = 5.757 \times 10^3$ . De outra forma, seria usando um **guard digit** - dígito que preserva mais precisão numérica:  $(5.753 + 0.0045) \times 10^3 = = 5.7575 \times 10^3 = 5.758 \times 10^3$ . O guard digit faz com que, caso o arredondamento providenciar um valor superior a 5, se some 1 ao último dígito fracionário, caso contrário, ignora-se. Por outro lado, caso o valor dê exatamente 5, há necessidade de existir mais um dígito, o **round digit**.

Mas como é que podemos aplicar este tipo de algoritmo em representações de vírgula flutuante?

Em vírgula flutuante temos que nos depender de vários seus constituintes, de forma a que possamos efetuar tal cálculo. Sendo assim, o primeiro passo para o nosso algoritmo de adição em vírgula flutuante é analisar o expoente das duas parcelas, calculando a sua diferença - considerando  $x^e$  e  $y^e$  como os dois expoentes, há que calcular  $y^e - x^e$ , sendo que  $y^e > x^e$ . O segundo passo é efetuar um deslocamento à direita do significando da parcela em  $x$ , de  $(y^e - x^e)$  posições, para obter  $x \cdot 2^{y^e - x^e}$ . Calculando de seguida o valor de  $x \cdot 2^{y^e - x^e} + y$ , sendo  $y$  o significando da segunda parcela, entramos num ciclo regido pelo seguinte pseudocódigo:

```

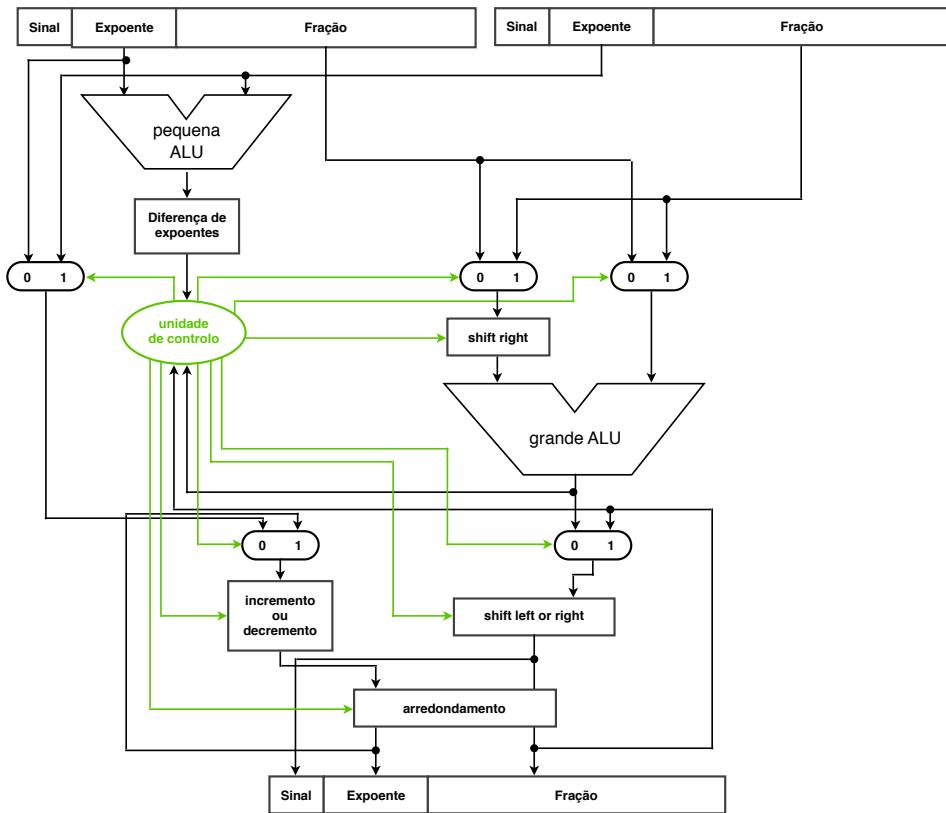
fazer:
  se ( $x \cdot 2^{y^e - x^e} + y == 0$ ):
    expoente_resultado = 0;
  caso contrário, se ( $x \cdot 2^{y^e - x^e} + y$  não normalizado):
    se ( $carry\_out == 1$ ):
      shift à direita significando_resultado
      incrementar expoente_resultado
    enquanto ( $MSB\_resultado \neq 1$ ):
      shift à esquerda significando_resultado
      decrementar expoente_resultado
    se ( $overflow$  ou  $underflow$ ): lançar exceção
    caso contrário: arredondar(resultado)
    enquanto ( $resultado$  não normalizado)
  
```

**arredondamento**

**guard digit**

**round digit**

Tendo o pseudocódigo realizado, agora estamos em plenas condições de passar para a fase de implementação do algoritmo em circuitos lógicos. Sendo assim, e usando alguns dos blocos já anteriormente usados, temos a Figura 53.



**figura 53**  
círcuito de adição em  
vírgula flutuante

Como podemos reparar na Figura 53, o somador com vírgula flutuante é um circuito muito mais complexo que um somador regular.

## Multiplicação em vírgula flutuante

Já que iniciámos o estudo de aritmética em vírgula flutuante também é importante falar na **multiplicação**. A multiplicação em *floating point* realiza-se segundo o seguinte pseudocódigo:

**multiplicação**

```

somar expoentes;
se (overflow ou underflow):
  gerar exceção;
multiplicar significandos;
fazer:
  se (overflow ou underflow):
    gerar exceção;
  arredondar significando para número de bits apropriado
  enquanto (resultado não normalizado)
    sinal = sinal_Y xor sinal_X
  
```

## Processamento de vírgula flutuante em MIPS

Num processador de arquitetura MIPS o processamento de quantidades numéricas com representação de vírgula flutuante é feita sobre um suporte diferente de outras quantidades. Para a vírgula flutuante há uma nova sub-unidade que coexiste

nos processadores, responsável pelo armazenamento deste tipo de quantidades - a essa sub-unidade damos o nome de **coprocessador**. Sendo assim, como coprocessador podemos dar o conceito de processador cuja função é complementar a função do processador central (CPU). Em MIPS há representação de dois coprocessadores, designados de C0 e C1, sendo o C0 o coprocessador do sistema (*system coprocessor*) e o C1 o coprocessador de vírgula flutuante. Originalmente este coprocessador existia numa placa-chip extra, com cerca de 75000 transístores. Hoje em dia ele já é integrado no mesmo chip que o próprio CPU, classificando-se de unidade aritmética de vírgula flutuante.

Para poder trabalhar com os valores pelo coprocessador de vírgula flutuante (C1) há que primeiro transferir os dados a usar do coprocessador de vírgula flutuante para registos inteiros, dado que o processador MIPS só cumpre tarefas com registos inteiros. Para tal existem as instruções *mfc1* e *mtc1*, representando “*move from coprocessor 1*” (mover de coprocessador 1) e “*move to coprocessor 1*” (mover para coprocessador 1), respetivamente (Código 47).

```
mfc1    $rX, $fX  # transfere do registo de vírgula flutuante para o inteiro
mtc1    $rX, $fX  # transfere do registo inteiro para o de vírgula flutuante
```

Apesar de transferirmos os dados de registos de vírgula flutuante para registos inteiros, isto não significa que as operações aritméticas efetuadas sobre estas quantidades sejam iguais, porque como já vimos antes, operações como a soma aritmética não são iguais entre diferentes representações. Sendo assim, o MIPS tem as seguintes instruções para vírgula flutuante (Código 48).

```
add.s   $f0, $f1, $f2      # soma vírgula flutuante em precisão simples
add.d   $f0, $f2, $f4      # soma vírgula flutuante em precisão dupla
sub.s   $f0, $f1, $f2      # subtração vírgula flutuante em precisão simples
sub.d   $f0, $f2, $f4      # subtração vírgula flutuante em precisão dupla
mul.s   $f0, $f1, $f2      # multiplicação vírgula flutuante em precisão simples
mul.d   $f0, $f2, $f4      # multiplicação vírgula flutuante em precisão dupla
div.s   $f0, $f1, $f2      # divisão vírgula flutuante em precisão simples
div.d   $f0, $f2, $f4      # divisão vírgula flutuante em precisão dupla
sqrt.s  $f0, $f1, $f2      # raiz quadrada vírgula flutuante em precisão simples
sqrt.d  $f0, $f2, $f4      # raiz quadrada vírgula flutuante em precisão dupla
abs.s   $f0, $f1           # valor absoluto vírgula flutuante em precisão simples
abs.d   $f0, $f2           # valor absoluto vírgula flutuante em precisão dupla
c.X.s   $f0, $f1           # comparação X vírgula flutuante em precisão simples
c.X.d   $f0, $f2           # comparação X vírgula flutuante em precisão dupla
bclt   $f0, $f2           # fazer salto se flag c.X.s ou c.X.d verdadeira
bclf   $f0, $f2           # fazer salto se flag c.X.s ou c.X.d falsa
```

Entre os outros tipos de instruções que existem para os registos inteiros há, quase sempre, uma correspondência para termos de vírgula flutuante. Sendo assim, no Código 49 representam-se as instruções de movimento de dados, e transferência de dados com a memória, tal como uma nova tarefa, designada de **conversão** entre precisão simples e precisão dupla.

```
mov.s   $f0, $f1           # cópia de dados vírgula flutuante em precisão simples
mov.d   $f0, $f2           # cópia de dados flutuante em precisão dupla
lwc1   $f1, 100($$2)     # $f1 = Memória[$$2 + 100]
swc1   $f1, 100($$2)     # Memória[$$2 + 100] = $f1
cvt.d.s $f1, $f2          # converter precisão simples em precisão dupla
cvt.s.d $f1, $f2          # converter precisão dupla em precisão simples
cvt.d.w $f1, $f2          # converter inteiro em precisão dupla
cvt.s.w $f1, $f2          # converter inteiro em precisão simples
cvt.w.d $f1, $f2          # converter precisão dupla em inteiro
cvt.w.s $f1, $f2          # converter precisão simples em inteiro
```

Para pormos agora os nossos conhecimentos em ação, tentemos traduzir para Assembly do MIPS o seguinte programa no Código 50, em C.

## coprocessador

### código 47 instruções mfc1 e mtc1

### código 48 vírgula flutuante em MIPS

## conversão

### código 49 vírgula flutuante em MIPS (conversões e movimentos)

```

void matrixMultiplication(double x[][], double y[][], double z[][]) {
    int i, j, k;
    for (i = 0; i != 32; i++) {
        for (j = 0; j != 32; j++) {
            for (k = 0; k != 32; k++) {
                x[i][j] = x[i][j] + y[i][k] * z[k][j];
            }
        }
    }
}

```

Neste exemplo, no Código 48 temos um pequeno programa que faz a operação  $X + Y * Z$  sobre matrizes, efetuando a multiplicação de matrizes - o algoritmo pode ser visto nos apontamentos de Álgebra Linear e Geometria Analítica (a1s1). Como já sabemos das rotinas, aqui as variáveis  $x$ ,  $y$  e  $z$  terão que ser armazenadas em registos \$a0, \$a1 e \$a2, dado que são argumentos da função. Já nos registos \$s0, \$s1 e \$s2 podemos guardar as variáveis  $i$ ,  $j$  e  $k$ . Por fim, de forma a termos sequências diretas de 64 bits em vírgula flutuante temos a pseudoinstrução *l.d* e *s.d*, que se traduz por séries de execuções de *lwc1* ou *swc1*. Sendo assim, primeiro há que inicializar as variáveis de controlo de ciclo (Código 51).

```

mm:
    li      $t1, 32          # $t1 = 32
    li      $s0, 0             # controlo do primeiro ciclo
L1:   li      $s1, 0             # controlo e restauro do segundo ciclo
L2:   li      $s2, 0             # controlo e restauro do terceiro ciclo

```

O segundo passo a fazer é obter a posição de  $x[i][j]$ , saltando  $i$  linhas  $i * 32$  e somar  $j$  (Código 52).

```

sll    $t2, $s0, 5          # $t2 = i * 2^5
addu   $t2, $t2, $s0         # $t2 = i * 2^5 + j

```

Depois de calcular a posição de  $x[i][j]$  é importante obter o endereço e carregar  $x[i][j]$  com precisão dupla (dado que cada elemento tem 8 bytes) (Código 53).

```

sll    $t2, $t2, 3          # $t2 = (i * 2^5 + j) * 2^3
addu   $t2, $a0, $t2         # endereço de x[i][j]
l.d    $f4, 0($t2)          # $f4 = x[i][j]

```

Tratando do ciclo controlado por  $k$ , temos de carregar  $z[]$  em \$f16 e  $y[]$  em \$f18 (Código 54).

```

L3:   sll    $t0, $s2, 5          # $t0 = k * 2^5
      addu   $t0, $t0, $t4         # $t0 = k * 2^5 + j
      sll    $t0, $t0, 3          # $t0 = (k * 2^5 + j) * 2^3
      addu   $t0, $a0, $t0         # endereço de z[k][j]
      l.d    $f16, 0($t0)          # $f16 = z[k][j]
      sll    $t0, $s0, 5          # $t0 = i * 2^5
      addu   $t0, $t0, $t5         # $t0 = i * 2^5 + k
      sll    $t0, $t0, 3          # $t0 = (i * 2^5 + k) * 2^3
      addu   $t0, $a1, $t0         # endereço de y[i][k]
      l.d    $f18, 0($t0)          # $f16 = y[i][k]

```

Para efetuar agora a multiplicação temos o Código 55.

```

mul.d  $f16, $f18, $f16 # y[][] * z[][]
add.d  $f4, $f4, $f16    # x[][] + y*z

```

**código 50**

**código de estudo em C**

**código 51**

**exemplo de aplicação**

**código 52**

**exemplo de aplicação**

**código 53**

**exemplo de aplicação**

**código 54**

**exemplo de aplicação**

**código 55**

**exemplo de aplicação**

# 51 ARQUITETURA DE COMPUTADORES I

Tendo a multiplicação completada, temos de incrementar  $k$  e caso seja o fim do ciclo controlado por  $k$ , guardar o valor de  $x$  (Código 56).

```
addiu $s2, $s2, 1      # k++
bne $t5, $t1, L3       # se k != 32, então ir para L3
s.d $t4, 0($t2)        # x[i][j] = $f14
```

Incrementamos  $j$  e caso  $j < 32$  completamos o ciclo intermédio, tal como incrementamos  $i$  se  $i \neq 32$ , pois caso contrário devolve-se o valor final (Código 57).

```
addiu $s1, $s1, 1      # j++
bne $s1, $t1, L2       # se j != 32, então ir para L2
addiu $s0, $s0, 1      # i++
bne $s0, $t1, L1       # se i != 32, então ir para L1
jr $ra
```

E assim se completa a nossa tradução do programa do Código 50.

## 5. Avaliação de Desempenho dos Sistemas de Computação

Existem diversos critérios a respeitar para se criar um repertório de instruções como é o ISA. Tais critérios de seleção têm de concordar com a simplicidade exigido pelo equipamento, conforme pedido pela execução de tarefas, à imagem de uma clareza de uma aplicação a problemas importantes, tendo um grande papel no grande interesse da resolução de problemas que é a velocidade de resolução.

Como já vimos anteriormente todas as instruções do MIPS têm um tamanho específico - 32 bits. Da mesma forma, já vimos que as operações aritméticas operam sobre registos e o resultado da operação também ele é preservado sob um registo. Estas duas evidências mostram e comprovam que a **regularidade** favorece a simplicidade. A simplicidade diz-nos que o mais comum também deve ser o mais rápido, pelo que, por exemplo, quando um determinado operando é uma constante, esta deve fazer parte da própria instrução, pelo que é habitual que cerca de mais de 50% das instruções que usam a ALU, envolvam constantes.

código 56

exemplo de aplicação

código 57

exemplo de aplicação

regularidade

tempo de execução

throughput

response time, elapsed time

desempenho do sistema

CPU time

### Medições de desempenho

Como num processador há sempre imensas variáveis que são passíveis de serem avaliadas, existem imensos métodos de avaliação de desempenho. Mas em geral, muitas delas podem ser englobadas em apenas dois grandes métodos: **tempo de execução** (tempo de execução dos programas passível de ser avaliado pelo próprio utilizador) e **throughput** (número de tarefas por dia, geridos por um *datacenter*). Nesta disciplina iremos focar-nos mais no método de tempo de execução.

O tempo de execução tem vários significados, pelo que pode ser determinado pelo intervalo de tempo limitado pelo início e pelo fim da execução de uma dada tarefa. A este significado damos o nome de **response time** ou **elapsed time** (termo inglês de tempo de resposta). Este tempo inclui a entrada e saída de dados, tal como o acesso ao disco, tempo despendido pelo sistema de operação, entre outros... Em suma, este primeiro significado resume o **desempenho do sistema** (em inglês *system performance*).

Por outro lado, o tempo de execução pode querer significar o tempo que o processador gasta a executar o programa do utilizador, na mesma incluindo o tempo de espera pelas entradas e saídas, entre outros..., ao que se dá o nome de **CPU time** (em português, tempo de CPU). O tempo de CPU resume-se à Equação 3.

$$\text{CPU}_{\text{time}} = \text{CPU}_{\text{time}}^{\text{user}} + \text{CPU}_{\text{time}}^{\text{system}}$$

**equação 3**  
**tempo de CPU**

**nota!!** na Equação 2 o *user CPU time* está relacionado com o desempenho do CPU em interação com o utilizador.

Vejamos mais em concreto o que é o tempo de CPU (Equação 4), sendo  $c$  o número de ciclos de relógio de CPU,  $t$  o tempo de cada ciclo (período) e  $f$  a frequência do relógio.

$$\text{CPU}_{\text{time}} = c \cdot t = \frac{c}{f}$$

**equação 4**  
**tempo de CPU**

Sendo que o número de ciclos de relógio é calculado multiplicando o número de instruções do programa ( $i$ ) pelo número médio de ciclos por instrução (também denominado de **CPI** - inglês para *clocks per instruction* -, o nosso tempo de CPU é determinado pela Equação 5.

$$\text{CPU}_{\text{time}} = \frac{i \cdot \text{CPI}}{f}$$

**equação 5**  
**tempo de CPU**

Avaliando a Equação 5, temos que para melhorar o desempenho é necessário diminuir o tempo de CPU, o que se obtém da diminuição do número de instruções ( $i$ ), pela redefinição do ISA, ou diminuindo o CPI ou aumentando a frequência do relógio ( $f$ ).

O CPI depende também do tipo de instrução em causa. Determinado pela Equação 6, este valor é sempre um valor médio do número de ciclos de relógio em que cada instrução é executada. Essa média é pesada pela frequência de execução de cada instrução, isto é, o CPI varia com os programas usados para o medir.

$$\text{CPI}_{\text{médio}} = \sum \text{instruções} (\text{CPI}_{\text{instrução}} \cdot f_{\text{instrução}})$$

**equação 6**  
**tempo médio de CPU**

Então mas que programas é que devemos escolher para medir o desempenho? Uma forma de resolver este problema é usando o **SPEC** (acrônimo de *Standard Performance Evaluation Corporation*). O SPEC é um conjunto de programas representativo do tipo de utilização dos computadores, que faz uma avaliação por utilização de operandos inteiros e outra por utilização de operandos em vírgula flutuante. Uma SPEC *program suite* tem os seguintes constituintes (Equação 7):

$$\text{SPEC} = \text{SPEC}_{\text{integer}} + \text{SPEC}_{\text{floating_point}}$$

**equação 7**  
**SPEC**

Uma outra forma de resolver o problema é usando um programa em que cada tipo de instrução aparecesse com a mesma frequência com que aparece no conjunto de programas representativo, o que são os casos **benchmark sintético**, como o *Dhrystone* (não contém operações com vírgula flutuante), *Whetstone* (contém operações com vírgula flutuante), ... Estas técnicas não se adequam aos novos compiladores, que detetam código redundante, eliminando-o, sendo este essencial para estes métodos, de forma a assimilar frequências de instruções. Em suma, este último método não dá resultados credíveis.

**benchmark sintético**

## Potência consumida

Um outro critério importante para a conceção e escolha de processadores (avaliação de desempenho) é a avaliação da **potência consumida**. Este, como seria de esperar, é um critério bastante importante para casos de computadores portáteis, dada a autonomia das baterias.

Todos nós devemos, por esta altura, saber que a capacidade que uma placa chip tem para a dissipação de calor é muito limitada. Por essa razão, e dependendo da frequência de relógio, foram criados processadores **multi-core**, pelo que este tipo de processadores, tal como já vimos antes na disciplina de Laboratórios de Informática (a1), permite a distribuição de tarefas e *threads* (assunto a ver mais à frente). Esta nova tecnologia permite igualar o desempenho em diferentes casos de frequência de relógio.

Para caracterizar a potência consumida também existem tabelas standard, da mesma corporação SPEC - as **SPECPower**.

**potência consumida**

**multi-core**

**SPECPower**

## Desempenho relativo

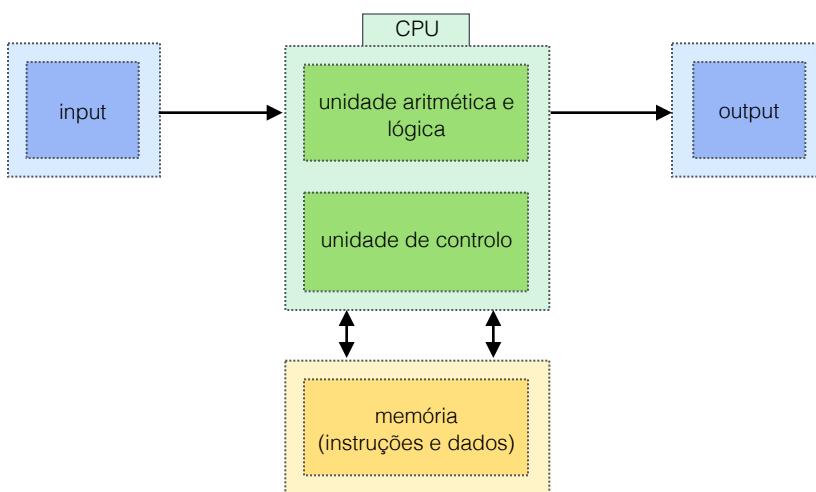
Uma forma de compararmos desempenhos de processadores é obtendo a razão dos seus desempenhos ou tempos de execução. A tal razão damos o nome de **desempenho relativo**. Sendo assim, o desempenho relativo pode ser obtido pela Equação 8.

$$\text{desempenho} = \frac{1}{t_{\text{execução}}} \therefore \frac{\text{desempenho}_A}{\text{desempenho}_B} = \frac{t_{\text{execução de B}}}{t_{\text{execução de A}}}$$

**equação 8**  
**desempenho relativo**

## 6. Implementação Single-cycle de um Processador

Como já estudámos no início da disciplina, um computador contém os seguintes cinco componentes, como é visível na Figura 54.



**figura 54**  
**computador em esquema**

A partir desta unidade, o nosso objeto de estudo será a estrutura interna do CPU, com o qual iremos tentar construir os componentes que interligam o seu interior, pelos registos e unidades de controlo.

Na unidade anterior tivemos a oportunidade de verificar que o desempenho de um processador é determinado pelo número de instruções, pelo CPI e pela frequência de relógio. Sendo assim, o desenho do CPU (datapath e unidade de controlo) terá de ser determinada pelo tempo de ciclo de relógio (período) e pelo CPI. Como é hábito, iremos começar pelo passo mais simples de implementação, acabando o programa numa estrutura muito mais complexa - começaremos então por tentar implementar um processador do tipo **single-cycle**. Um processador do tipo single-cycle significa que todas as instruções devem ser executadas num só ciclo de relógio, pelo que o CPI é 1. Isto não é propriamente bom, como teremos oportunidade de verificar mais à frente, dado que o período de cada ciclo é muito longo (frequência é muito baixa - relógio lento).

**single-cycle**

## Fases de construção de um processador

Para construirmos um processador teremos de seguir passos importantes para a obtenção de uma unidade coerente e coesa. Sendo assim, o primeiro a fazer será analisar o repertório de instruções (*instruction set*). É importante analisar isto primeiro, pois estes são os verdadeiros requisitos para todas as futuras ações - até propriamente para a funcionalidade do processador. O significado de cada instrução é dado pelas transferências entre registo, pelo que o datapath do processador terá de incluir hardware suficiente para os registo que o ISA suporta, criando oportunidade de haver transferências entre registo.

Tendo a análise feita, um segundo passo muito importante é o de selecionar todos os componentes para o datapath e definir uma metodologia para os impulsos de relógio. Isto permitirá-nos passar para o passo seguinte da construção do datapath, satisfazendo todas as especificações geradas nos passos 1 e 2.

Com o datapath construído, há que analisar a implementação de cada instrução, de forma a poder identificar os respetivos sinais de controlo que acionam as transferências entre registo, de forma a podemos executar o último passo, o de construção da unidade de controlo.

## Análise do repertório de instruções (ISA)

Como já devemos saber a unidade MIPS tem por si três tipos de instrução, visíveis na Figura 55.

<b>op</b>	<b>rs</b>	<b>rt</b>	<b>rd</b>	<b>shamt</b>	<b>funct</b>	<b>tipo R</b>			
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits				
<b>op</b>	<b>rs</b>	<b>rt</b>	<b>constant or address</b>			<b>tipo I</b>			
6 bits	5 bits	5 bits	16 bits						
<b>op</b>	<b>address</b>								
6 bits	28 bits								

**figura 55**  
tipos de instruções do MIPS

De forma a simplificarmos o nosso exercício, mas sem perder importância, vamos implementar apenas as instruções de adição sem sinal, subtração sem sinal, disjunção lógica com imediato, carregamento de palavra de 32 bits, salvaguarda de palavra de 32 bits e branch caso haja igualdade. As instruções são as visíveis no Código 58.

<code>addu</code>	<code>rd, rs, rt</code>
<code>subu</code>	<code>rd, rs, rt</code>
<code>ori</code>	<code>rt, rs, Imm16</code>
<code>lw</code>	<code>rt, rs, Imm16</code>
<code>sw</code>	<code>rt, rs, Imm16</code>
<code>beq</code>	<code>rt, rs, Imm16</code>

código 58  
instruções do ISA

Seguindo os passos explícitos em §§Fases de construção de um processador, temos agora que analisar quais são os requisitos exigidos pelo ISA estabelecido. Ora, para o nosso ISA de seis instruções precisamos de memória para as instruções e para os dados, tal como precisamos de registos (32 de 32 bits) com capacidade controlável de ler o conteúdo de *rs*, de *rt* e de escrever conteúdo em *rd* ou *rt* (conforme indicado no Código 56). Precisamos também de um *program counter*, de forma a podermos controlar o estado atual programável da máquina, tal como de um extensor de sinal (em inglês *sign extender*), um somador/subtrator para registos e imediatos com extensão de sinal e um outro somador para acrescentar o valor 4 ao *program counter*, para a sua atualização.

## Seleção dos componentes para o datapath

Selecionando então os componentes para o datapath e escolhendo a estratégia do relógio, nós já conhecemos a existência dos seguintes componentes combinatórios (Figura 56).

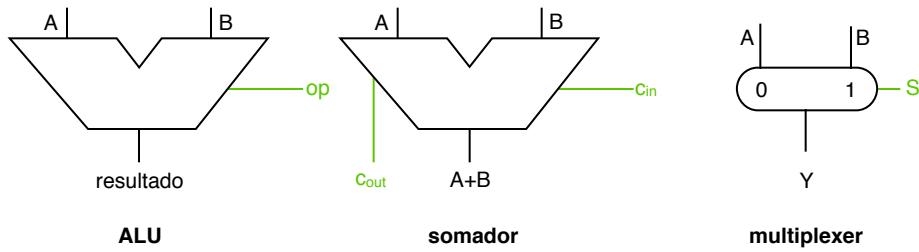


figura 56  
componentes combinatórios

Para além dos componentes combinatórios da Figura 56, também precisamos dos componentes sequenciais da Figura 57.

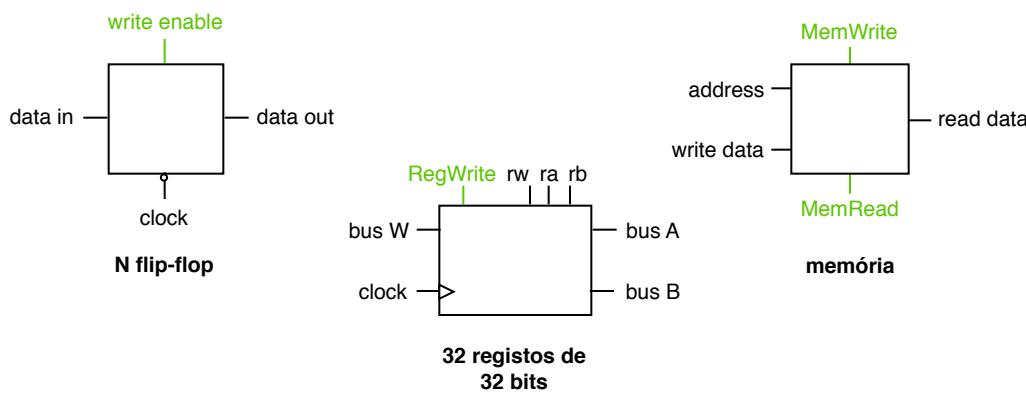


figura 57  
componentes sequenciais

Começando pelos 32 registos de 32 bits (Figura 57 (meio)), temos que cada um destes registos é constituído por componentes do tipo da Figura 57 (esquerda), sendo que cada um destes componentes denominados de *N* flip-flops, tem tecnologia

negative edge-triggered, sendo que tendo o sinal *write enable* a 0, o *data out* não muda, ao invés do que possa acontecer tendo o sinal *write enable* a 1, sendo que aí o *data out* assume o valor de *data in* na próxima frente descendente do relógio.

Focando a nossa atenção para a Figura 57 (meio), temos um bloco de 32 componentes sequenciais flip-flop, no qual a seleção de cada registo é feito através das entradas *ra*, *rb* e *rw* - *ra* escolhe o registo cujo conteúdo é colocado no *bus A*; *rb* escolher o registo cujo conteúdo é colocado no *bus B*; *rw* escolhe o registo onde é escrito o dado em *bus W*, quando *RegWrite* tem o valor 1.

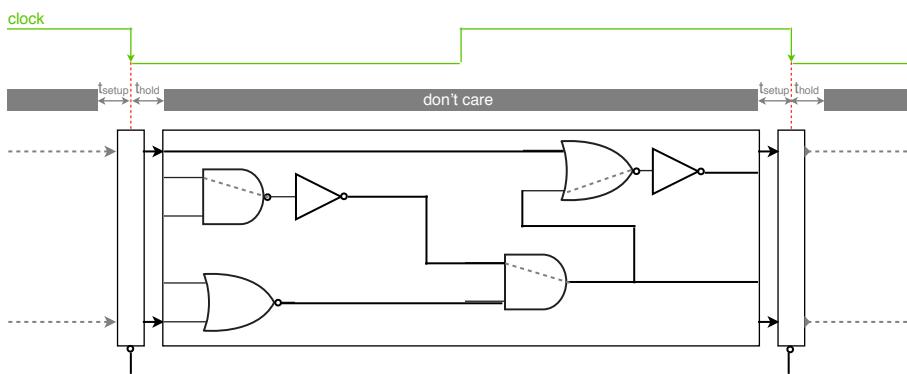
Já no caso da memória, na Figura 57 (direita) temos dois sinais de controlo, o *MemRead* e o *MemWrite*, sendo que se o primeiro sustiver o valor 1, será o valor *address* que escolherá a palavra a colocar em *read data*. Por outro lado, se o sinal *MemWrite* sustiver o valor 1, *address* escolherá a palavra de memória onde é escrito o dado presente em *write data*.

## Clocking

Falta-nos verificar a questão do relógio. Sabemos que todos os elementos de memória são sensíveis à mesma frente de relógio. Então será que podemos prever o tempo de ciclo de relógio (período)? Tendo os valores temporais dos componentes, é possível calcular o período (Equação 9), sendo *p* o tempo do caminho mais demorado e *k* o desvio do relógio (em inglês *clock skew*) e *s* o tempo de setup (Figura 58).

$$\begin{aligned} t_{\text{ciclo}} &= \text{CLK-to-Q} + p + s + k \\ \text{CLK-to-Q} + p + s + k &> t_{\text{hold}} \end{aligned}$$

**equação 9**  
tempo de ciclo



**figura 58**  
clocking

## Construção do datapath

Para conseguirmos definir os parâmetros das instruções a implementar, temos de compreender a lógica **RTL**. A lógica RTL (sigla de *Register Transfer Logic* - em português lógica de transferência de registos) permite-nos exprimir o sentido e significado das instruções. Sendo assim, é como efetuar um registo temporal de ações a cumprir desde o início de cada instrução, até à sua conclusão. Para tal, podemos criar uma tabela, como a Figura 59.

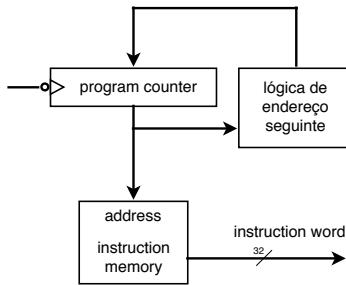
**RTL**

instrução	transferência de registos	instrução	transferência de registos
addu	R[rd] <- R[rs] + R[rt]; PC <- PC + 4	load	R[rd] <- Mem[ R[rs] + zero_ext(Imm16) ]; PC <- PC + 4
subu	R[rd] <- R[rs] - R[rt]; PC <- PC + 4	store	Mem[ R[rs] + sign_ext(Imm16) ] <- R[rt]; PC <- PC + 4
ori	R[rd] <- R[rs] + zero_ext(Imm16); PC <- PC + 4	beq	if (R[rs] == R[rt]): PC <- PC + sign_ext(Imm16)    00 else PC <- PC + 4

**figura 59**  
lógica RTL

No início como introdução ao processador MIPS vimos que ele cumpre dois passos de execução: o *instruction fetch* e o *instruction execution*. Recordando, no *instruction fetch* o *program counter* endereça a memória, lendo o código de instrução, e no *instruction execution*, a operação especificada no código de instrução é executada. Vejamos por partes, cada um destes processos.

Primeiro, como já revimos, a fase de *instruction fetch* é a fase na qual o valor do *program counter* endereça a memória e é atualizado, por um incremento de valor 4. Em termos de implementação por blocos, esta fase é aplicada na Figura 60.

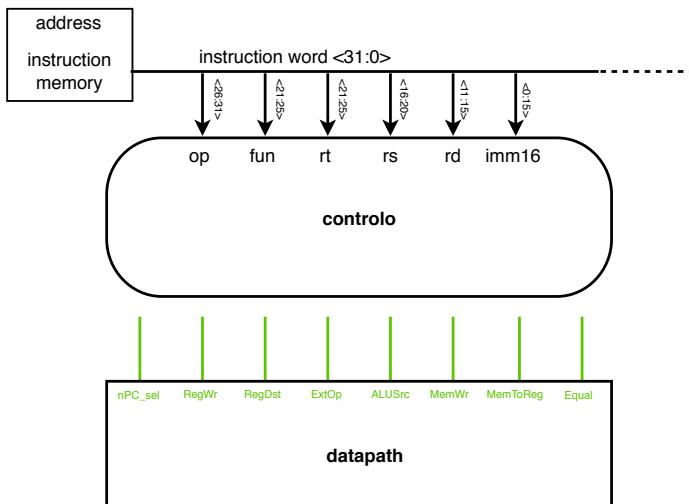


**figura 60**  
**instruction fetch**

Por outro lado, a fase de *instruction execution* tem uma implementação que depende do tipo de instrução em causa. De modo a tornar as coisas mais simples, abordemos a implementação desta fase em simultâneo com a identificação dos sinais de controlo.

### Identificação de sinais de controlo

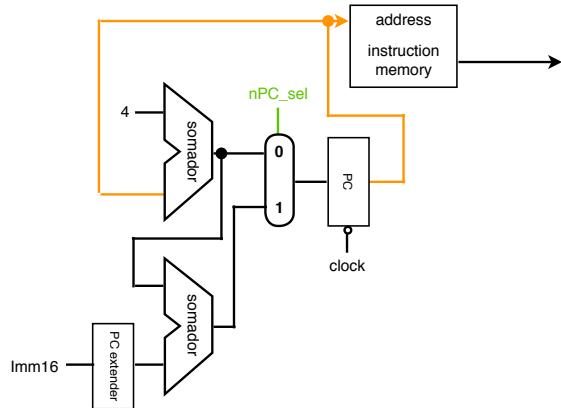
Aplicando os mesmo conhecimentos de RTL para a intuição de controlo relacionada com o funcionamento do nosso processador, começemos por saber a lista de sinais de controlo que precisamos designar ao longo de toda a nossa análise (Figura 61).



**figura 61**  
**sinais de controlo**

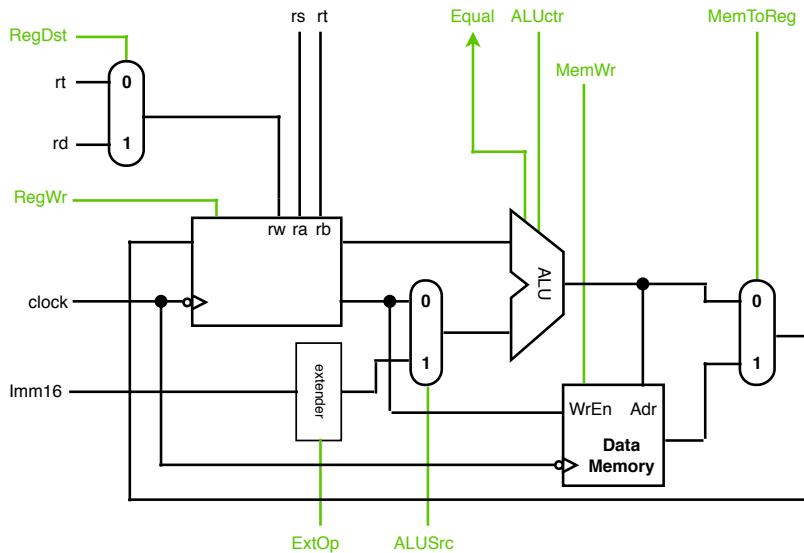
Passo-a-passo, iniciando pela instrução *addu*, como vimos por RTL, a instrução *addu rd, rs, rt* é executada em três passos - fetch da instrução da memória ( $Mem[\$pc]$ ), operação de adição ( $R[rd] \leftarrow R[rs] + R[rt]$ ) e cálculo do endereço da instrução seguinte ( $\$pc \leftarrow \$pc + 4$ ). A unidade responsável pela primeira fase de

execução (*instruction fetch*), no início da operação de *addu*, tem a seguinte forma (Figura 62).



**figura 62**  
fetch na operação addu

No *instruction fetch*, no início da instrução *addu*, consoante o valor do sinal de controlo *nPC\_sel*, o valor do registo \$pc é incrementado com '4' ou com '4' e *Imm16* (Figura 63).

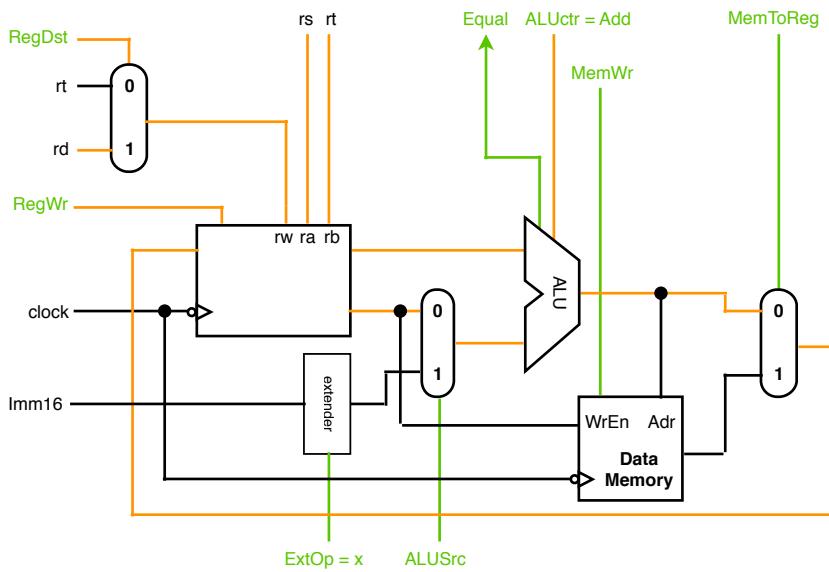


**figura 63**  
sinais de controlo (detalhe)

O significado dos sinais de controlo nesta fase é o seguinte:

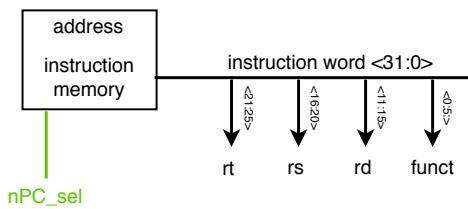
- *ExtOp* - se '0' não funciona, se '1' extende o sinal de *Imm16*;
- *ALUSrc* - se '0' passa valor de registo B, se '1' passa imediato;
- *ALUctr* - código para permitir operações soma, subtração ou soma lógica;
- *MemWr* - se '0' não permite escrita na memória, se '1' permite escrita;
- *MemToReg* - se '0' passa valor de ALU, se '1' passa valor atual da memória;
- *RegDst* - se '0' passa valor de *rt*, se '1' passa valor de *rd*;
- *RegWr* - se '0' não permite escrita em registo, se '1' permite escrita.

Então durante a instrução em estudo como é que esta fase se comporta em termos de implementação? A traço alaranjado encontram-se os caminhos ativos a '1' na Figura 64, consequência da instrução *addu* (incluindo sinais de controlo com cor).



**figura 64**  
instrução addu

Por complemento, a primeira fase de processamento deve encontrar-se da seguinte forma (Figura 65).

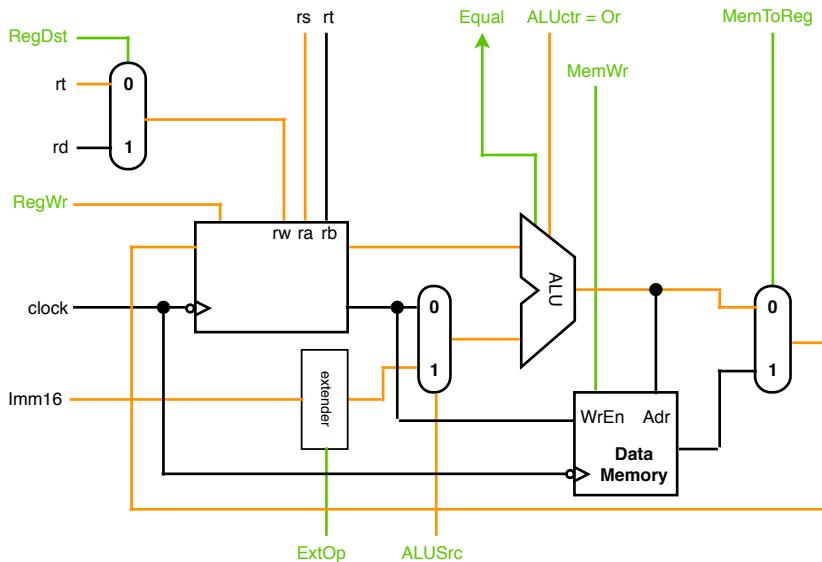


**figura 65**  
primeira fase

Se verificarmos bem, em todas as instruções, à exceção da branch, há um incremento de \$pc por '4' (Figura 61).

**nota!!** a operação de controlo para a instrução *subu* é semelhante à instrução analisada com a ligeira diferença de que, ao invés de ser injetado o valor de controlo respetivo à soma, para o ALU, injeta-se o valor respetivo à subtração.

Dado que já temos a instrução *addu* analisada, passemos para a operação de *ori*. Esta instrução é do tipo I, pelo que constitui uma interação importante com um valor constante - um imediato. Na Figura 66 temos o datapath durante o *ori*.

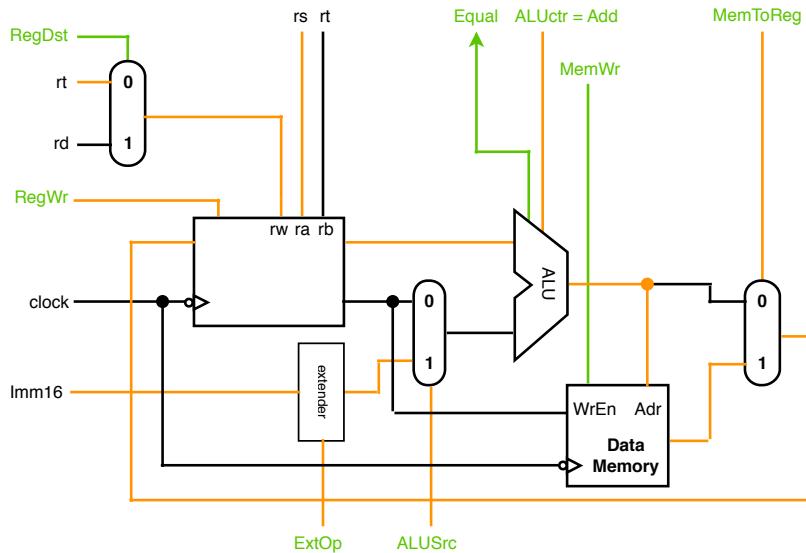


**figura 66**  
instrução ori

**nota**

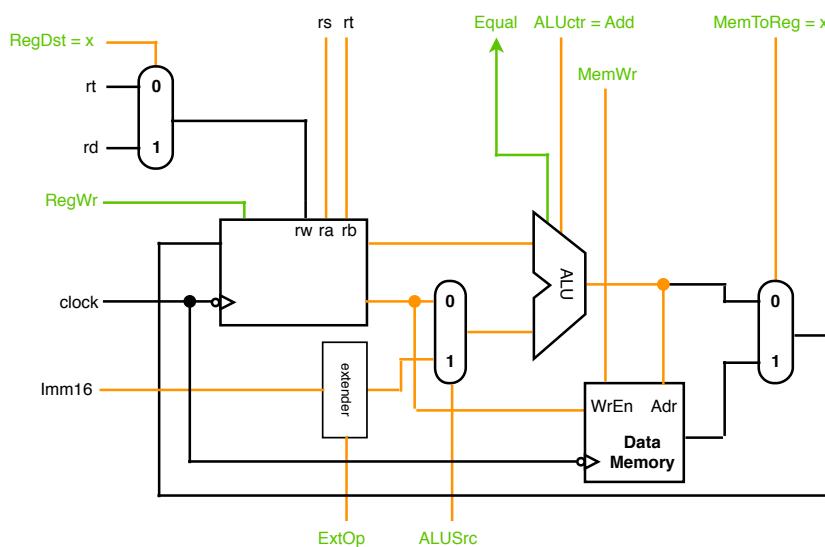
E durante um acesso à memória? A principal questão a fazer em relação a esta instrução é qual é o valor que o ALU terá de ser sujeito por controlo?

Ao fazer uma leitura da memória copiamos o conteúdo de uma ou mais células de memória para um determinado registo do processador. Dado isto, é claro que sinais de controlo como *MemToReg* tem de ter o valor de '1' e *MemWr* o valor de '0'. Ativando o controlo da ALU como 'Add', conseguimos adicionar o valor imediato ao do registo que passa pelo *bus A*, de forma a obter o endereço em questão. Sendo assim, na Figura 67 temos o estado do datapath para a instrução *lw*.



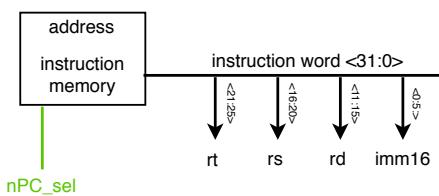
**figura 67**  
instrução *lw*

Analogamente ao caso da instrução *lw* (load word), a operação de salvaguarda de dados em memória, em palavras de 32 bits, denominada de *sw* (store word) também terá sinais de controlo fáceis de designar. Dado que estamos a estudar uma instrução que irá ter como resultado final uma escrita na memória, o sinal de controlo *MemWr* terá de ter o valor '1', tendo o valor *MemToReg* igual a '0' ou '1' (é indiferente - *don't care*). Note-se que o valor de controlo para a ALU terá de ser na mesma a operação 'Add', dado que pretendemos que se some o imediato de forma a obter o endereço final de memória. Na Figura 68 podemos ver o efeito de tal operação no nosso datapath.



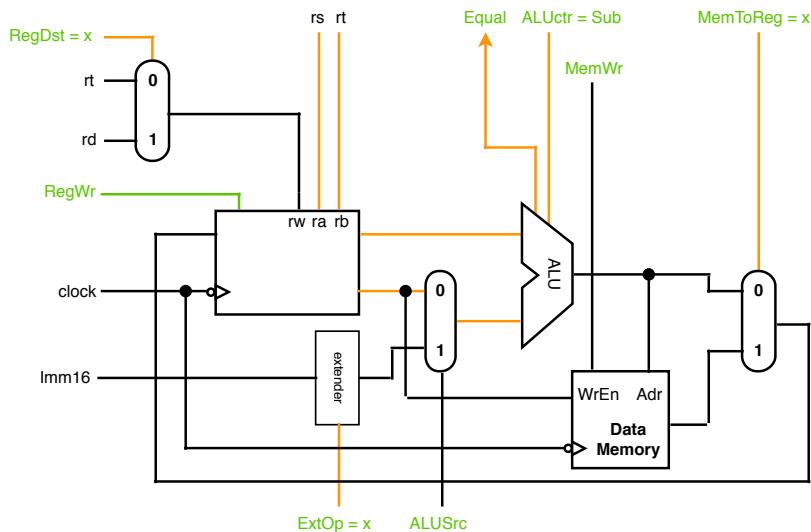
**figura 68**  
instrução *sw*

Como já averiguámos anteriormente, há semelhanças entre as instruções de *lw*, *sw* e *ori*, pois todas elas partilham o mesmo tipo de instrução - instrução do tipo I. Dado que isso acontece, para as três instruções, a fase de *instruction fetch* é igual para todas elas, podendo ser representada pela Figura 69.



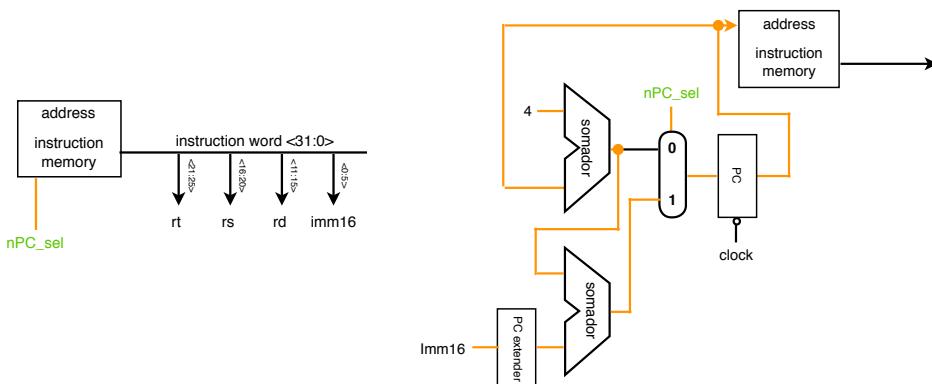
**figura 69**  
**instruction fetch**

A operação que nos falta analisar é a de *branch*, sendo ela diferente de todas as outras que já abordámos. Como já verificámos atrás através da lógica RTL, a instrução de branch ( neste caso *beq* - branch on equal) tem a seguinte manifestação de execução: if ( $R[rs] == R[rt]$ ):  $\$pc \leftarrow \$pc + ext(Imm16)$  || 00 else  $\$pc \leftarrow \$pc + 4$ . Dado isto, o estado da máquina aquando da execução desta instrução é representada pela Figura 69.



**figura 70**  
**instrução beq**

Pela fase de *instruction fetch*, a máquina deve fazer a atualização do valor do registo  $\$pc$ , somando, para além do valor '4', o valor do imediato, tendo no início da execução o aspeto em Figura 71 (esquerda) e o fim de execução em Figura 71 (direita).



**figura 71**  
**fetch da instrução beq**

## Realização da lógica de controlo

Comecemos, pela lógica RTL, por descodificar as ações a executar pelos sinais de controlo ao longo de uma operação da máquina:

- $nPC\_sel$  - se OP for igual a BEQ, então ‘Zero’, caso contrário ‘0’;
- $ALUSrc$  - se OP for igual ‘000000’ então retorna registo B, else ‘imediato’;
- $ALUctr$  - se OP for ‘000000’ dá ‘funct’, se for ORI dá ‘OR’, se for BEQ dá SUB, caso contrário ‘ADD’;
- $ExtOp$  - se OP for ‘ORI’ então ‘Zero’, caso contrário ‘Sign’;
- $MemWr$  - se OP for ‘Store’ então ‘1’, caso contrário ‘0’;
- $MemToReg$  - se OP for ‘Load’ então ‘1’, caso contrário ‘0’;
- $RegWr$  - se OP for ‘Store’ ou ‘BEQ’ então ‘0’, caso contrário ‘1’;
- $RegDst$  - se OP for ‘Load’ ou ‘ORI’ então ‘0’, caso contrário ‘1’.

Para implementarmos a lógica analisada por RTL temos que relembrar a Figura 60, na qual se pretende designar os sinais de controlo como interação entre o datapath e a unidade de controlo. Não estando designadas na figura, as ligações são orientadas de formas diferentes, sendo que no sentido da unidade de controlo para o datapath estão todas os sinais de controlo designados em RTL acima. Por outro lado, no sentido do datapath para a unidade de controlo encontra-se a ligação ‘Equal’, resultado de uma operação de branch, aqui designada por **condição**. De forma mais sintetizada, a Figura 72 fornece uma tabela na qual se fornecem os dados necessários para a execução dos sinais de controlo.

**condição**

func	10 0000	10 0010			don't care		
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	addu	subu	ori	lw	sw	beq	jump*
<b>RegDst</b>	1	1	0	0	X	X	X
<b>ALUSrc</b>	0	0	1	1	1	0	X
<b>MemToReg</b>	0	0	0	1	X	X	X
<b>RegWr</b>	1	1	1	1	0	0	0
<b>MemWr</b>	0	0	0	0	1	0	0
<b>nPC_sel</b>	0	0	0	0	0	1	0
<b>ExtOp</b>	X	X	0	1	1	X	X
<b>ALUctr&lt;3:0&gt;</b>	ADD	SUB	OR	ADD	ADD	SUB	-UNDEF-
<b>Jump*</b>	0	0	0	0	0	0	1

De forma a simplificar o funcionamento da interpretação tabular dos dados, podemos codificar os nossos parâmetros, dado que à exceção do sinal de controlo  $ALUctr$ , as operações de instrução *addu* e *subu*, têm os mesmos resultados dos sinais. Sendo assim, e dando significado à **descodificação local**, podemos simplificar a nossa tabela designando tais instruções por instruções do tipo R (Figura 73).

**descodificação local**

	tipo R	ori	lw	sw	beq	jump*
<b>RegDst</b>	1	0	0	X	X	X
<b>ALUSrc</b>	0	1	1	1	0	X
<b>MemToReg</b>	0	0	1	X	X	X
<b>RegWr</b>	1	1	1	0	0	0
<b>MemWr</b>	0	0	0	1	0	0
<b>nPC_sel</b>	0	0	0	0	1	0
<b>ExtOp</b>	X	0	1	1	X	X
<b>ALUctr&lt;3:0&gt;</b>	R-TYPE	OR	ADD	ADD	SUB	-UNDEF-
<b>Jump*</b>	0	0	0	0	0	1

**figura 73**

**tabela de sinais de controlo (simplificada)**

# 63 ARQUITETURA DE COMPUTADORES I

Até agora seguimos as operações da ALU de forma simbólica, isto é, designando cada instrução por um termo ('Add', 'Sub', 'R-Type', 'Or', ...). Por esta altura já devemos saber que tais designações não têm qualquer significado na máquina - precisamos de os **codificar**. Para codificar estes sinais vamos criar a variável  $ALUop$ , que se caracteriza por ter 3 bits para representar instruções do tipo R, e instruções 'OR', 'ADD' e 'SUB', com imediato (tipo I). Criamos então uma dada codificação de entradas com um dado código (Figura 74).

	tipo R	ori	lw	sw	beq	jump*
simbólico	R-TYPE	OR	ADD	ADD	SUB	-UNDEF-
ALUop <2:0>	100	010	000	000	001	-UNDEF-

**codificar**

**figura 74**

**codificação de entradas**

Nas instruções do tipo R, para além da definição e codificação do *opcode*, para designar uma instrução em particular usa-se o *funct* como código de especificidade. Sendo assim, definimos como convenção para este caso em estudo, a Figura 75 como codificação do *funct*.

	add	sub	and	or	set-on-less-than
funct <5:0>	10 0000	10 0100	10 0100	10 0101	10 1010
ALUctr	0010	0110	0000	0001	0111

**figura 75**

**codificação do funct**

Tendo o *opcode* e o *funct* de  $ALUop$  definidos, temos agora que definir as condições que  $ALUctr$  controla. Da mesma forma que definimos a estratégia de código para o *opcode*, agora falta definir uma estratégia para o sinal de controlo  $ALUctr$  (Figura 76).

	add	sub	and	or	set-on-less-than
ALUctr	0010	0110	0000	0001	0111
ALUop	000	010	000	001	011

**figura 76**

**estratégia de ALUctr**

Designados os termos em  $ALUctr$  e  $ALUop$ , consoante as suas variâncias de parâmetros, criemos agora uma tabela de verdade de  $ALUctr$  (Figura 77).

ALUop			func				operação da ALU	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	X	X	X	X	ADD	0	1	0
0	X	1	X	X	X	X	SUB	1	1	0
0	1	X	X	X	X	X	OR	0	0	1
1	X	X	0	0	0	0	ADD	0	1	0
1	X	X	0	0	1	0	SUB	1	1	0
1	X	X	0	1	0	0	AND	0	0	0
1	X	X	0	1	0	1	OR	0	0	1
1	X	X	1	0	1	0	SLT	1	1	1

**figura 77**

**tabela de verdade**

Sabendo que o sinal  $ALUctr$  é um sinal de controlo definido como sendo de 3 bits, temos que para cada um dos bits podemos estabelecer determinadas equações de saída, considerando-as como saídas de  $ALUop$ . Sendo assim, temos que o sinal de  $ALUctr$  pode ser dividido em três sinais diferentes: consideraremos assim  $ALUctr<2>$ ,  $ALUctr<1>$  e  $ALUctr<0>$ , sendo o número 2, 1 ou 0 como o bit respetivo ao código.

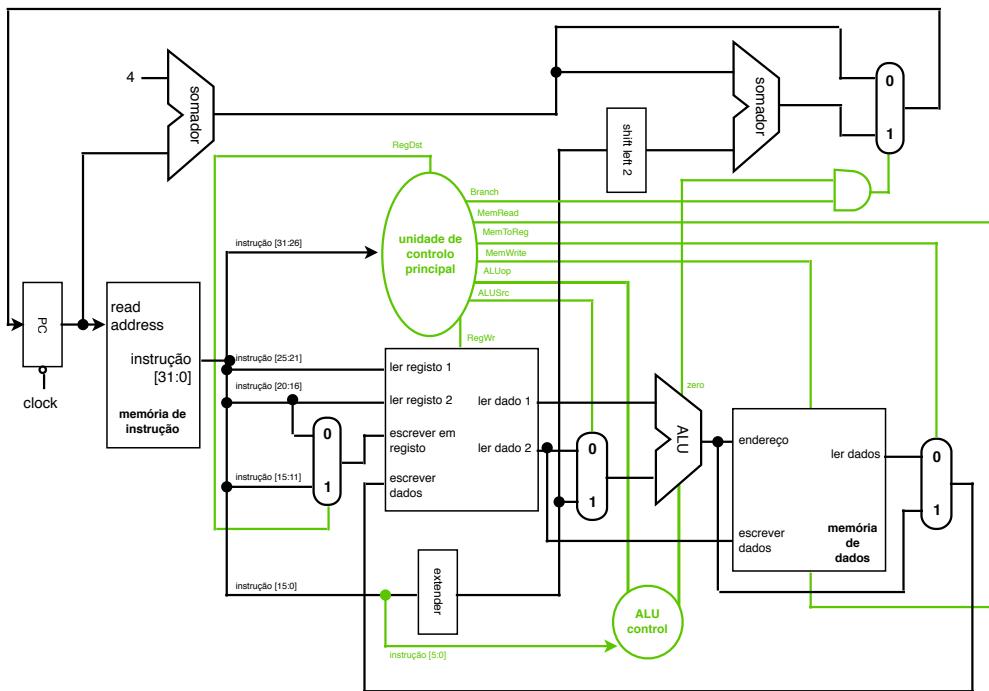
Começando com  $ALUctr<2>$ , temos que ele é correspondente à seguinte combinação:  $ALUctr<2> = !ALUop<2> \& ALUop<0> + ALUop<2> \& !func<2> \& func<1> \& !func<0>$ .  $ALUctr<2>$  não contém  $func<3>$  porque ele toma valores de '0' e '1', pelo que pode ser considerado como *don't care*.

Prosseguindo, com  $ALUctr<1>$  este tem como combinação geral a seguinte:  
 $ALUctr<1> = !ALUop<2> \& !ALUop<0> + ALUop<2> \& !func<2> \& !func<0>$ .  
Do mesmo modo,  $ALUctr<0> = !ALUop<2> \& ALUop<1> + ALUop<2> \& & !func<3> \& !func<2> \& !func<0> + ALUop<2> \& func<3> \& func<2> \& & func<1> \& !func<0>$ .

Tendo as expressões analisadas podemos criar um bloco de ALU de controlo local - denominamos-lhe de **ALU control**, o qual recebe como entradas os três bits de  $ALUop$  e os seis bits de  $funct$ . Este bloco está representado na Figura 78.

**ALU control****figura 78****bloco ALU control**

Agora já podemos concluir o nosso desenho projetado do processador single-cycle, dado que já possuímos todo o conhecimento necessário para compreender todo o seu esquema técnico (Figura 79).

**figura 79****unidade single-cycle**

## Caminho crítico

A entrada síncrona do relógio é um fator que interage apenas durante o processamento de execução da escrita em memória ou registo. Durante o período de leitura, a memória comporta-se como um bloco de lógica combinatória, pelo que as saídas apenas permanecem válidas depois de um dado tempo de acesso, caso o próprio endereço seja válido. A este tempo damos o nome de **tempo crítico** e o caminho que é realizado neste tempo **caminho crítico**. De forma a evitar este caminho crítico é possível fazer uma aproximação deste, através de um cálculo, este, expressido na Equação 10, onde  $q$  é o CLK-to-Q do registo \$pc,  $m$  é o tempo de acesso à memória de

**tempo crítico****caminho crítico**

instrução,  $r$  é o tempo de acesso aos registos,  $a$  é o tempo que uma ALU demora a fazer uma soma de 32 bits,  $d$  é o tempo de acesso à memória de dados,  $w$  é o tempo de setup para escrita em registo e  $k$  é o desvio do relógio (*clock skew*).

$$\text{path}_{\text{critical}} = q + m + r + a + d + w + k$$

**equação 10**  
caminho crítico

## Desvantagens do processador single-cycle

Como já tinhamos avançado ao início da unidade, uma grande desvantagem - a principal -, é que tudo é cumprido num só ciclo de relógio, pelo que para que tal possa acontecer, o ciclo de relógio tem de ser grande o suficiente longo para a execução de carregamento.

O longo período do ciclo de relógio, para além de ser, *per si*, enorme, é mais-que-necessário para algumas das instruções, pelo que se torna uma segunda grande desvantagem.

## 7. Implementação Multi-Cycle de um Processador

Relembrando o ciclo de instrução de um processador MIPS, sabemos que este inclui sempre duas fases: o *instruction fetch* e o *instruction execution*. Na unidade anterior vimos uma implementação simples de um datapath de um processador que tinha muitas desvantagens, sendo delas, a maior, o facto de todas as instruções durarem o período de um ciclo de execução. Conseguindo apontar tal desvantagem, podemos tentar quebrá-la, criando uma nova implementação que execute instruções em mais que um ciclo de relógio, dependendo do tipo de instrução. A esta implementação damos o nome de **multi-cycle** e será a que vamos abordar nesta unidade, analisando todos os seus parâmetros o mais rigorosamente possível.

Focando a nossa atenção na fase de *instruction execution*, sendo que é nesta precisa fase que a implementação anterior tinha a sua grande falha, podemos então dividir tarefas, criando novas sub-fases. Na execução deste processamento a primeira coisa a fazer é a descodificação da instrução, através de uma leitura dos registos. Esta será a nossa primeira sub-fase: **decode and read registers**. Tendo as instruções descodificadas o passo seguinte é o da execução. Nesta altura as instruções, já designadas por instruções do tipo I ou tipo R (ou tipo J) são passíveis de serem executadas por auxílio de operações do tipo R e tipo I imediato ou pelo cálculo do endereço de memória para instruções como *lw*, ou *sw*, ou ainda como cálculo das condições de branch. Esta é a nossa nova segunda sub-fase: **execute**. Como os cálculos já se encontram resolvidos agora há que preservar o resultado final. Sendo assim, é importante a escrita do resultado de uma operação do tipo R ou I imediato, ou ainda o acesso à memória, sub-fase à qual damos o nome de **memory access** (ou para o caso de escrita em registo **register write**). Por fim, há que escrever o conteúdo da posição de memória anteriormente calculada, sob efeito de instruções *load*. Esta situação é a nossa última sub-fase denominada de **write back**.

Estes são os procedimentos regulares de um processador do tipo multi-cycle, no qual cada sub-fase é executada em cada ciclo de relógio. Isto permite maximizar o desempenho em termos do single-cycle porque agora o tempo de cada ciclo é muito mais reduzido. Isto implica que os tempos de execução das operações em cada ciclo sejam todos iguais (pelo menos numa aproximação a esse ideal).

Em termos de componentes, para cada uma das fases designadas há funções a cumprir por determinados componentes. Na primeira fase de fetch precisamos de uma memória para a leitura de dados e de uma unidade ALU para a adição e atualização

**multi-cycle**

**decode and read registers**

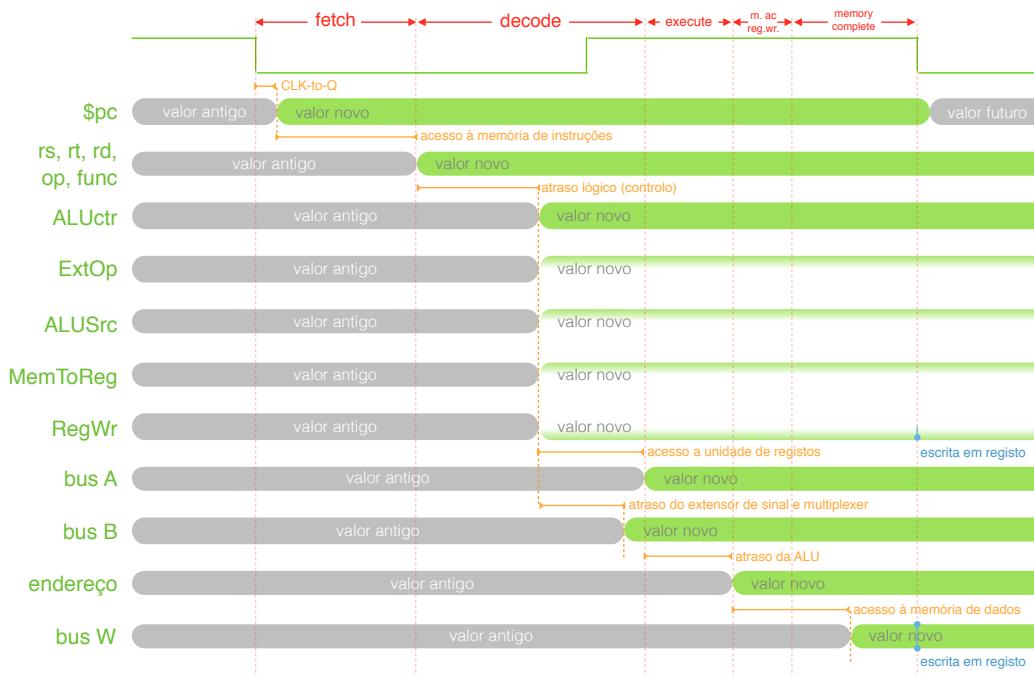
**execute**

**memory access**  
**register write**

**write back**

do registo \$pc, tal como na implementação anterior. De seguida, na fase de decode temos uma unidade de controlo responsável pela regularidade de todos os restantes procedimentos e de um datapath com ligação a uma unidade de registos para leitura. A fase de execução tem de ser claramente executada através de uma unidade aritmética e lógica (ALU) e a fase de memory access, também claramente, precisa de uma memória, tal como a fase de write back precisa de uma unidade de registos para escrita.

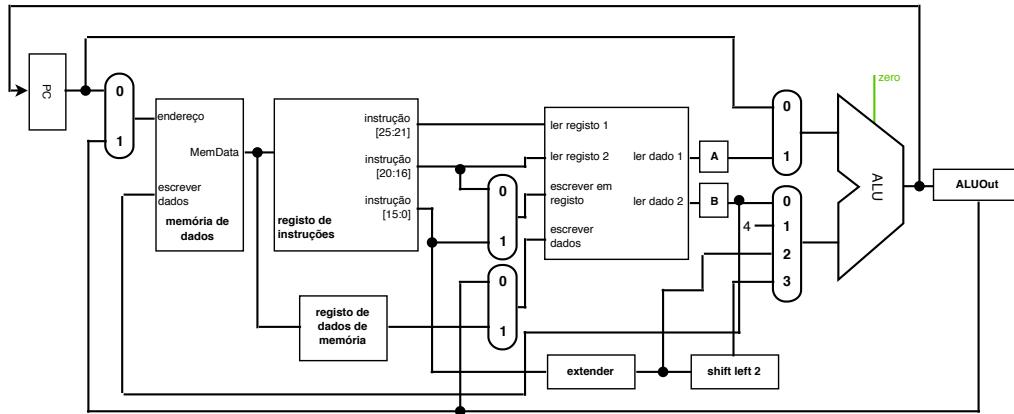
Dado estas configurações de implementação, o tempo de ciclo mímico, isto é, a frequência máxima de relógio, esclarece que o tempo de acesso à memória terá de ser aproximadamente igual ao tempo de acesso a registos, tal como aproximadamente igual ao tempo de operação da ALU, como já verificámos anteriormente. Na Figura 80 podemos ver um diagrama temporal, no qual verificamos os tempos algo semelhantes entre si.



**figura 80**  
diagrama temporal

Como podemos ver pela Figura 80 a implementação multi-cycle é muito mais vantajosa dado que permite que uma unidade funcional seja usada mais do que uma vez, desde que em ciclos de relógio diferentes, durante a execução da instrução. Do mesmo modo, também permite que diferentes instruções sejam executadas num número de ciclos de relógio diferente, não ficando assim o tempo de execução de todas as instruções dependente do tempo de execução da instrução mais lenta, que é o *lw*, dado o acesso à memória.

Em comparação direta com a implementação em single-cycle, a implementação multi-cycle tem uma memória única para instruções e dados. A unicidade da ALU substitui também o uso desmedido de uma ALU e mais dois somadores na *fetch single-cycle*. Outro pormenor é o de um ou mais registos serem colocados à saída de cada uma das principais unidades funcionais para que os valores possam ser usados no ciclo de relógio seguinte: na Figura 81 teremos oportunidade de verificar que o *instruction register* e o *data register* estão à saída da memória, os registos A e B estão à saída do banco de registos e o registo *ALUOut* está à saída da ALU. Também podemos verificar que o uso de multiplexeres em maior quantidade e maior dimensão também é diferença importante.



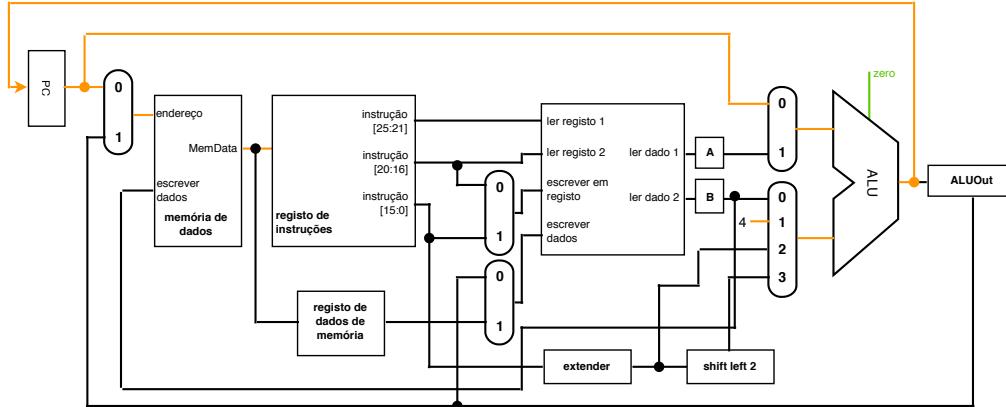
**figura 81**  
implementação multi-cycle

## Fases de Execução

De volta à lógica RTL designemos as fases de execução da máquina em estudo com implementação multi-cycle:

1. *instruction fetch* - leitura do código de instrução e atualização do conteúdo de \$pc:
  - ›  $IR \leftarrow Mem[\$pc];$
  - ›  $\$pc \leftarrow \$pc + 4.$
2. *instruction decode* - descodificação da instrução e leitura dos registros:
  - ›  $A \leftarrow Reg[IR[25:21]];$
  - ›  $B \leftarrow Reg[IR[20:16]];$
  - ›  $ALUOut \leftarrow \$pc + (sign-extend(IR[15:0]) \ll 2).$
3. *execute* - execução de operações:
  - › operação tipo R:
    - ›  $ALUOut \leftarrow A \text{ operação } B.$
  - › load ou store:
    - ›  $ALUOut \leftarrow A + (sign-extend(IR[15:0])).$
  - › branch on equal:
    - › if( $A == B$ ):  $\$pc \leftarrow ALUOut;$
  - › execução termina.
4. *memory access ou register write* - acesso à memória ou escrita de resultado em registro:
  - › operação tipo R:
    - ›  $Reg[IR[15:11]] \leftarrow ALUOut.$
    - › execução termina.
  - › load ou store:
    - ›  $Memory Data Register \leftarrow Mem[ALUOut];$  (load)
    - ›  $Mem[ALUOut] \leftarrow B.$  (store)
  - › store:
    - › execução termina.
5. *write back* - escrita do conteúdo da posição de memória calculada:
  - ›  $Reg[IR[20:16]] \leftarrow Memory Data Register.$

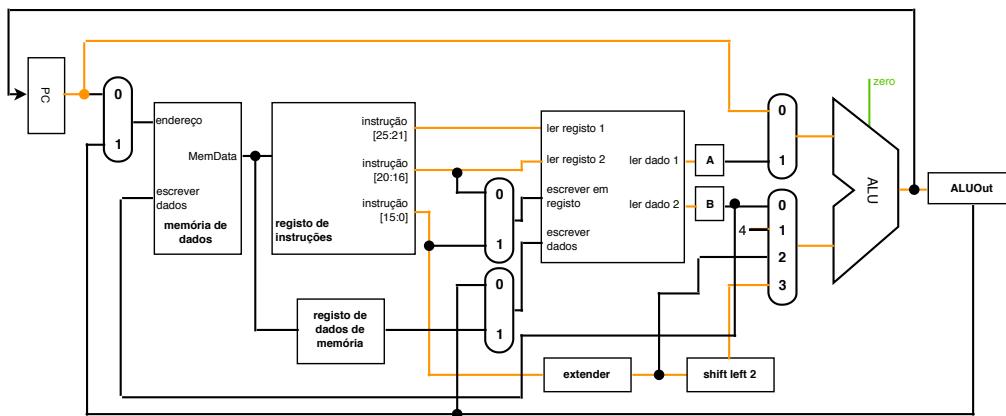
Por partes, vamos interpretando graficamente as consequências dos ciclos de execução. Sendo assim, começamos sempre pelo *instruction fetch*, o qual tem o resultado explícito em Figura 82.



**figura 82**  
instruction fetch

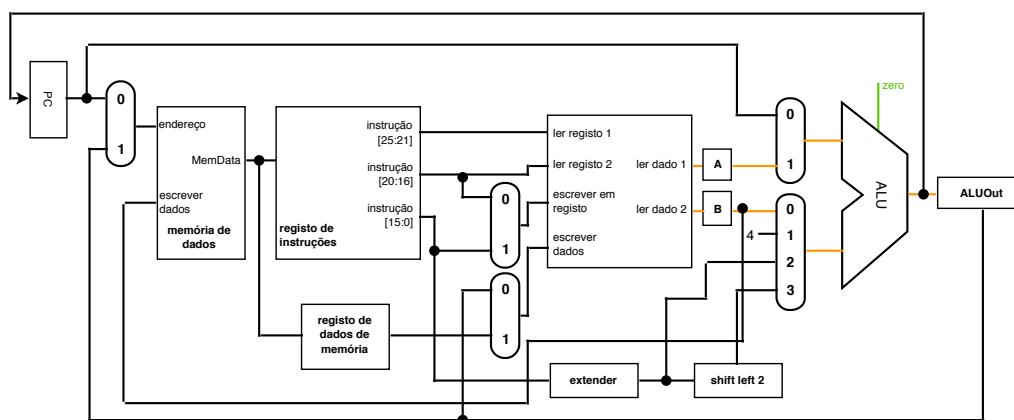
Como podemos interpretar na Figura 82, o *program counter* endereça a memória, sendo à frente, as entradas da ALU serem  $\$pc$  e ‘4’. Após a soma, o valor é retornado ao registo  $\$pc$ , atualizando-o. Ao fim do primeiro ciclo, IR é o código da instrução a executar e  $\$pc \leftarrow \$pc + 4$ .

Num segundo ciclo, pela fase de descodificação, leitura dos registos e cálculo do endereço-alvo, a nossa máquina apresenta o estado visível na Figura 83.



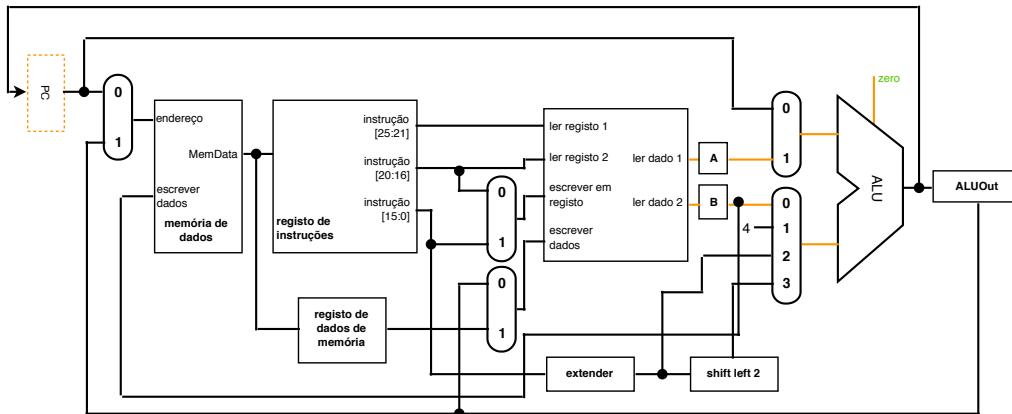
**figura 83**  
instruction decode

Vejamos agora dois casos possíveis, de resultados diferentes, para a fase *execute*, dado que para casos de branch on equal há resultado por *terminus* de funções. A Figura 84 resume o caso para a operação do tipo R.



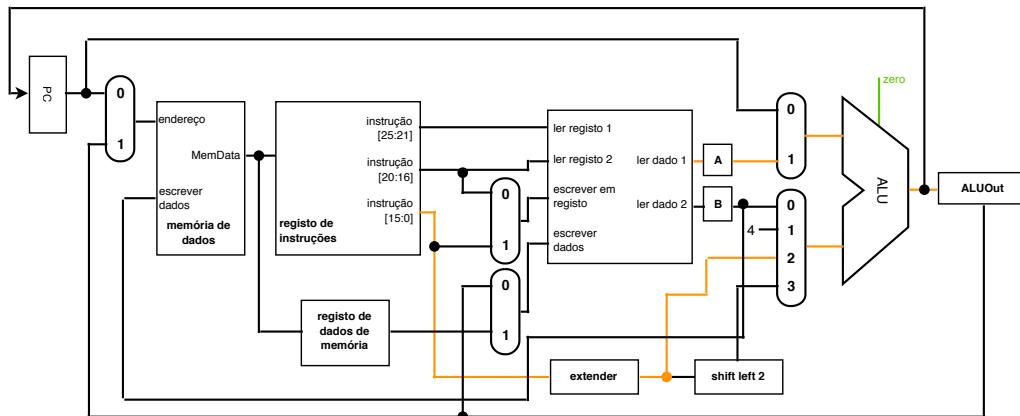
**figura 84**  
instrução tipo R

Agora, na Figura 85, para o caso do branch on equal, temos que o sinal de controlo *zero*, é ativada, sinalizando o registo \$pc e terminando execução.



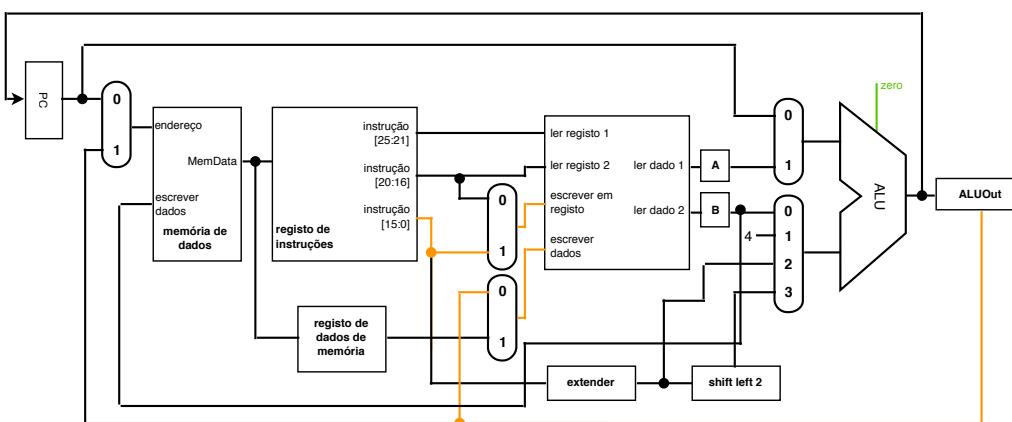
**figura 85**  
instrução branch

Um outro exemplo de operação em terceiro ciclo é o load ou store. A Figura 86 representa essa informação.



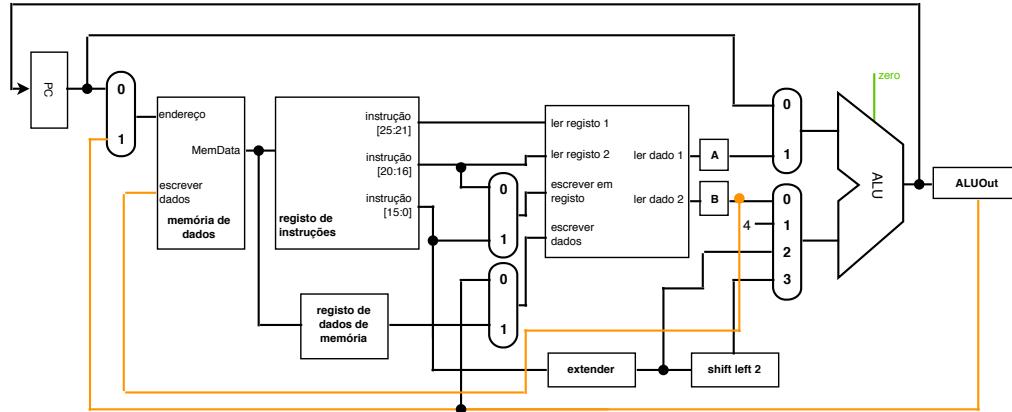
**figura 86**  
instrução load/store

Para o quarto ciclo temos o caso do fim da execução das operações do tipo R, com a escrita em registo, e o acesso à memória por parte das operações de load, store. A Figura 87 mostra o fim da execução das operações do tipo R.



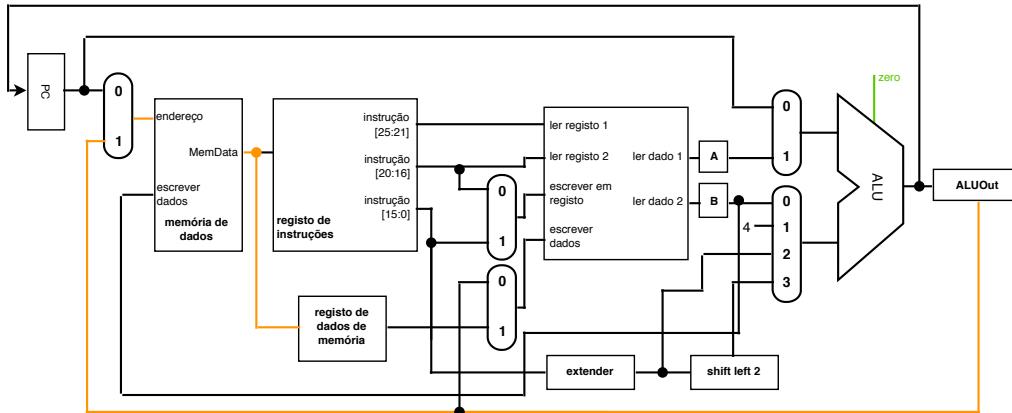
**figura 87**  
instrução tipo R

No quarto ciclo a segunda operação que pode terminar a execução é a operação de store. A Figura 88 mostra o fim da execução da operação store.



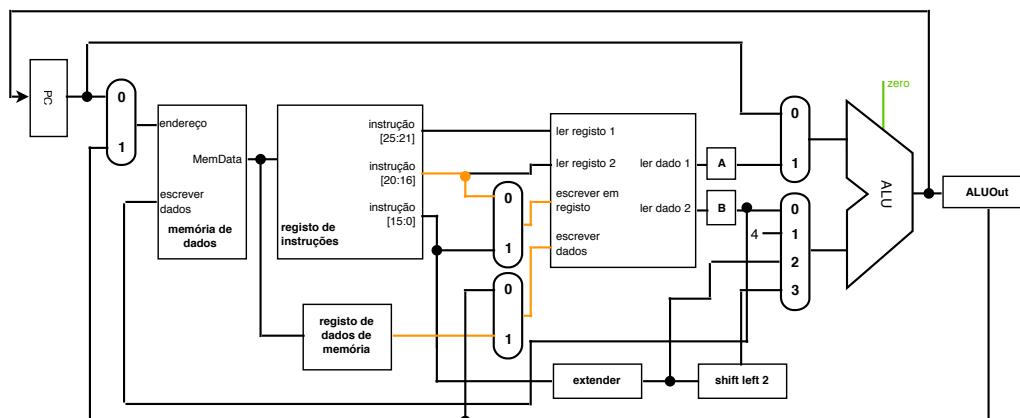
**figura 88**  
instrução store

Para terminar a nossa análise do ciclo de acesso à memória, temos o caso da operação de load, na qual não se termina execução. A Figura 89 pretende mostrar o significado da operação load na implementação multi-cycle.



**figura 89**  
instrução load

No último ciclo, a sub-fase de *write back*, a única operação cuja execução ainda é válida é a de load - daí ser a operação mais morosa. Sendo assim, a Figura 90 mostra o estado da máquina nesta operação em pleno quinto ciclo.



**figura 90**  
instrução load

## Identificação dos sinais e realização da lógica de controlo

De forma a podermos analisar a lógica de controlo da implementação em multi-cycle devemos antes ser capazes de identificar as necessidades de sinais de controlo. Na Figura 91 podemos encontrar o mesmo circuito com o acrescento dos sinais de controlo e dos seus nomes.

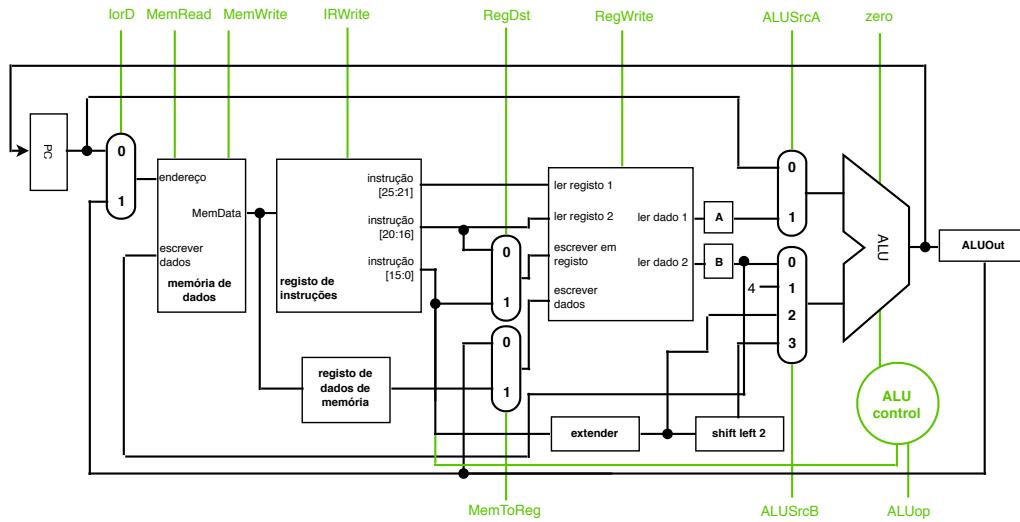


figura 91  
sinais de controlo

Ao contrário da lógica combinatória que fundamentava o exemplo da implementação single-cycle, a implementação multi-cycle é baseada em máquinas de estados (em inglês *finite state machines*). Sendo assim, um ciclo corresponde a tantos estados quantas as diferenças das operações a executar para cada categoria de instruções. Em cada estado a unidade de controlo gera os sinais que controlam as operações do datapath a executar nesse estado.

Como já pudemos designar muito atrás com a lógica RTL, aqui, com a lógica de uma máquina de estados finita, podemos definir um grafo no qual discriminamos o funcionamento da máquina (Figura 92).

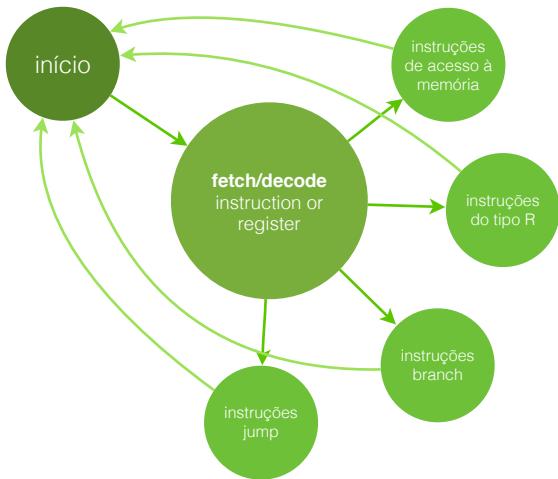
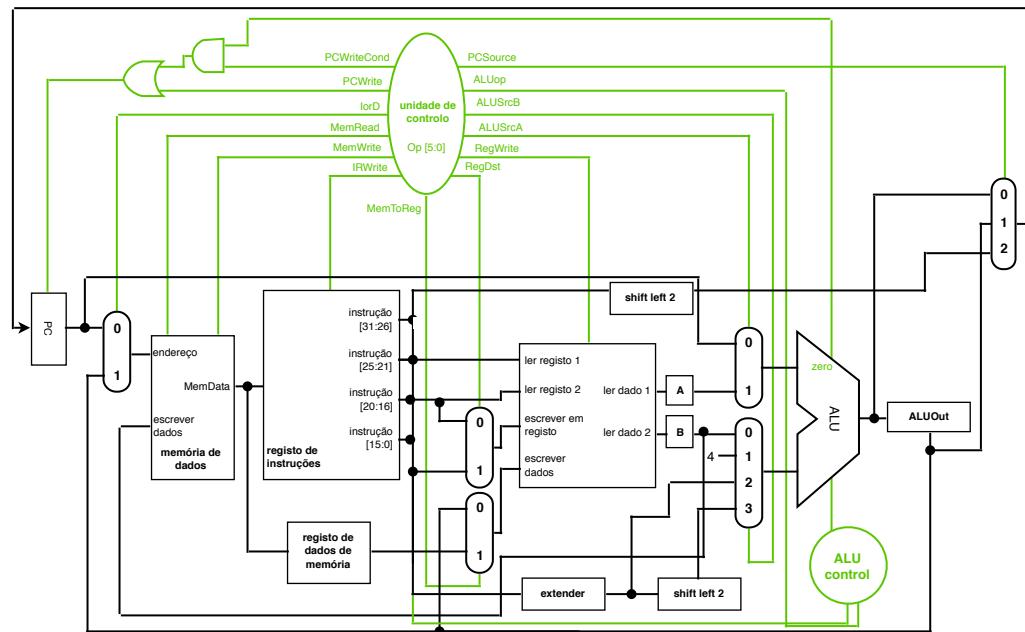


figura 92  
máquina de estados

Então será possível saber quantos estados tem a nossa máquina de estados? Sim, pois o *fetch* é um estado e o *decode* é também um estado, tal como as quatro possíveis tarefas de execução (por *execution*) têm a duração de um estado. Já a fase de

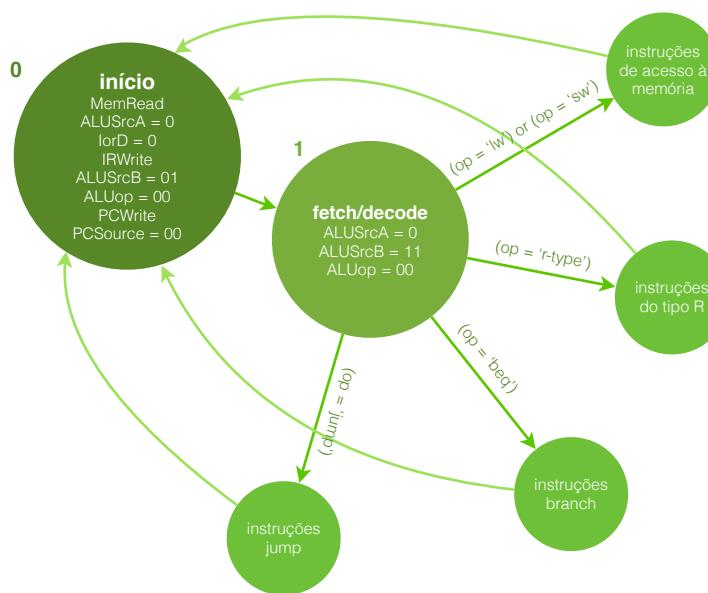
acesso à memória e escrita em registo dá um estado às instruções do tipo R e dois estados (load e store) para o acesso à memória. Por fim, a fase de *write back* tem a duração de apenas um estado. Somando tudo, podemos dizer que a nossa máquina de estados possui um total de 10 estados possíveis.

Revisitando a implementação da nossa máquina multi-cycle, incluindo a unidade de controlo, obtemos a Figura 93.



**figura 93**  
implementação multi-cycle

Tendo a implementação completa, analisemos agora as variações da máquina de estados para cada operação. Comecemos por revisitar a Figura 92 (Figura 94).



**figura 94**  
máquina de estados

Indo por partes, analisemos cada um dos quatro conjuntos de estados possíveis. Primeiro, para as instruções de acesso à memória, temos as Figura 95.

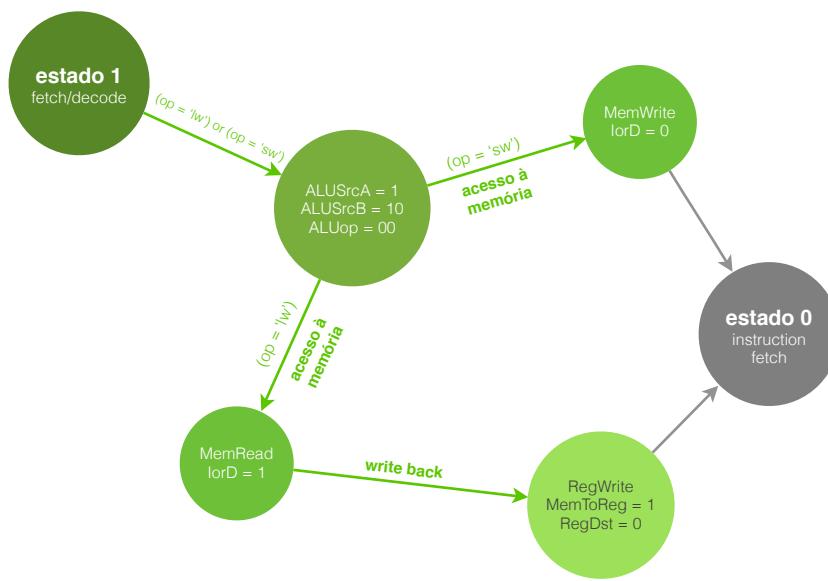


figura 95

máquina de estados (acesso à memória)

Continuando, agora para o conjunto de instruções do tipo R, temos a Figura 96.

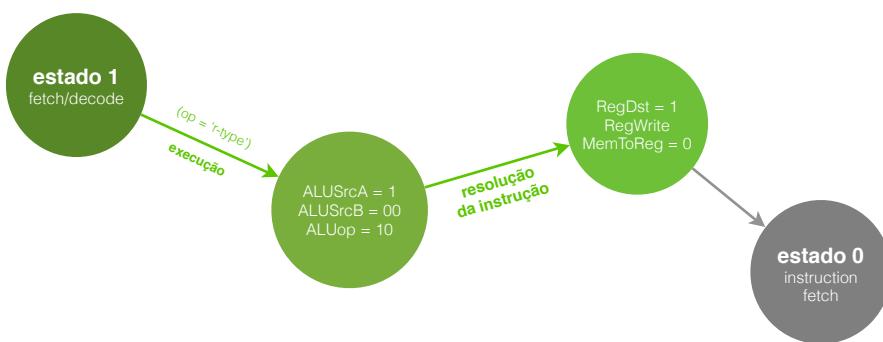


figura 96

máquina de estados (tipo R)

Em terceiro, para o conjunto de instruções branch temos a Figura 97.

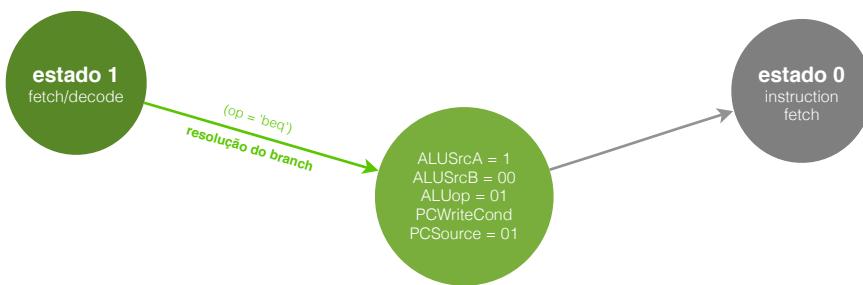


figura 97

máquina de estados (branch)

Finalmente, para as instruções de jump temos a Figura 98.

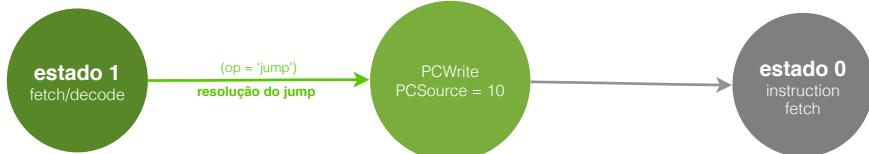


figura 98

máquina de estados (jump)

A máquina de estados completa pode ser vista na Figura 99.

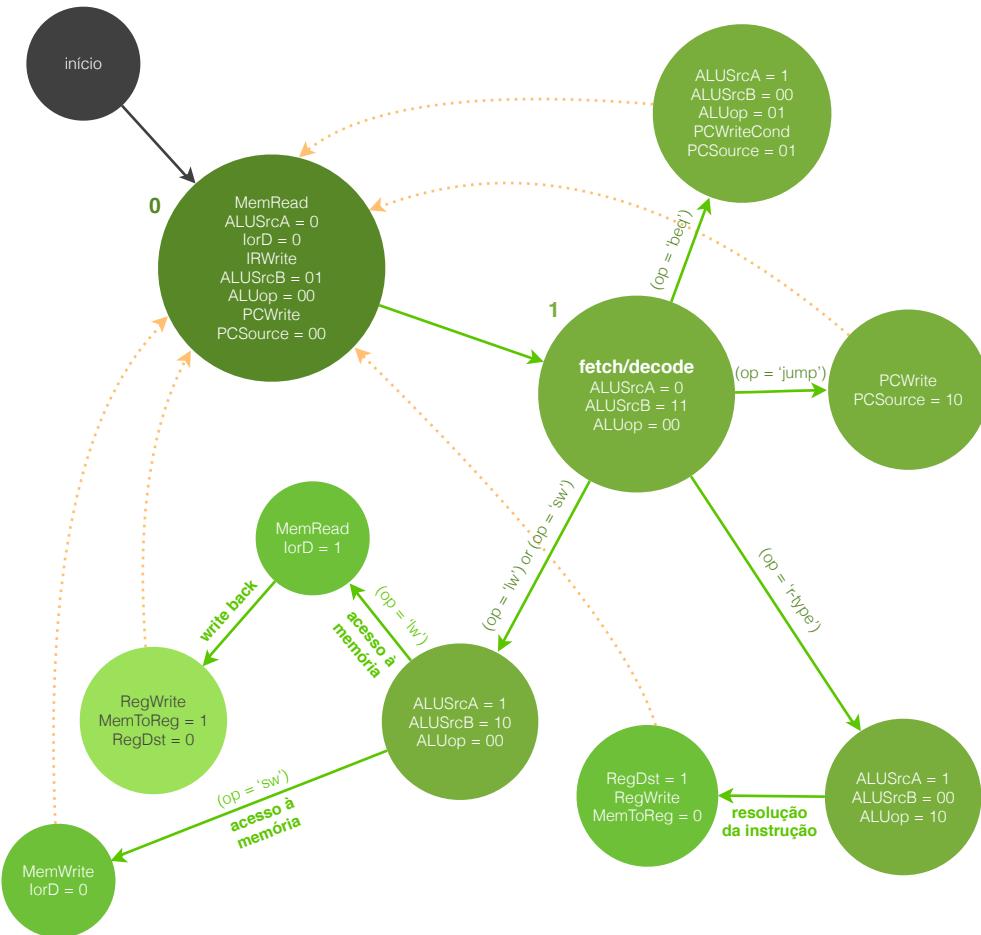


figura 99  
máquina de estados (final)

## 6. Implementação com Pipelining

No método estudado de implementação multi-cycle podemos reparar que há uma sincronia muito elementar em termos de temporização de ações. Pela Figura 79 podemos comprovar isso mesmo, sendo que a execução num todo é sempre dividida sob cinco ciclos de relógio. Desde a nossa primeira implementação em single-cycle também temos vindo a preocuparmo-nos com os recursos utilizados - para a atualização do valor do registo \$pc passámos a usar a ALU, ao invés de um somador dedicado para tal.

Introduzimos agora uma nova implementação - uma melhoria no processo de concretização de ações (execução). O que acontece é que podemos aplicar um método muito mais simples e rápido de execução. Numa analogia, pensemos no trabalho de uma pessoa que trabalhe numa fábrica de automóveis. Como é que uma fábrica monta os automóveis? Se pensarmos em termos de uma unidade single-cycle, se um automóvel tivesse seis passos para a sua montagem, então primeiro montava-se um carro e só depois é que se passava para o seguinte, cumprindo as seis tarefas, mas as próximas seis tarefas, e assim consecutivamente... Uma forma de simplificar este processo é, enquanto uma viatura está a ser montada no segundo passo, em simultâneo, no primeiro passo de execução já deve estar outra viatura preparada para a montagem. Cria-se o sentido de continuidade, mantendo-se o número de passos para a execução

# 75 ARQUITETURA DE COMPUTADORES I

total, mas gerando uma simplicidade protocolar muito maior. Aplicado nos sistemas digitais, a este tipo de raciocínio lógico damos o nome de **pipeline**, implementação a qual vamos a partir deste momento analisar em termos do nosso processador.

Prosseguindo com a nossa analogia, temos que montar um carro completo, consideremos, demora 60 minutos, considerando que cada fase de montagem demora 10 minutos. A este tempo total damos o nome de **latência**. No final, a linha de montagem congratula-se pela montagem de 1 carro por cada 10 minutos, razão a qual damos o nome de **throughput**. À relação entre ambas as dimensões designadas damos o nome de **speedup**, sendo, neste caso, o *speedup* entre pipeline e single-cycle igual a 6. De um ponto de vista mais gráfico, a Figura 100 mostra o conceito de pipeline aplicado à unidade em estudo.



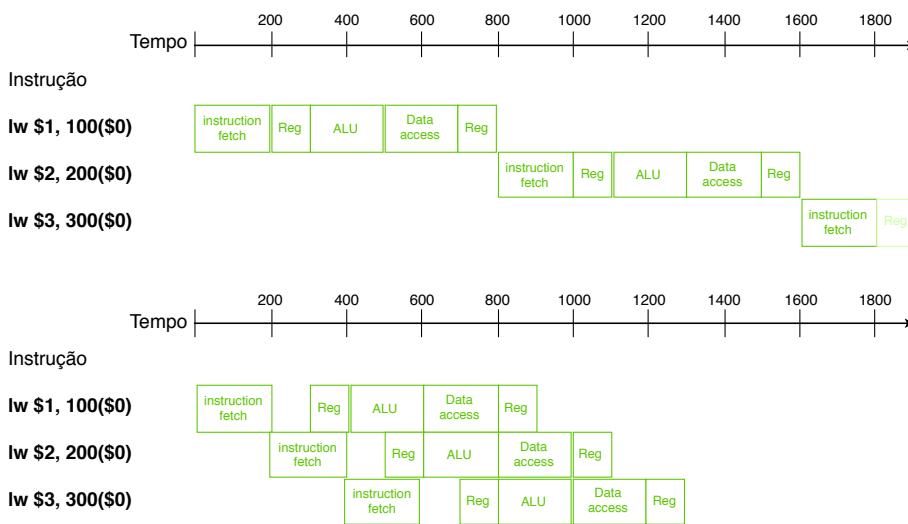
**pipeline**

**latência**

**throughput**

**speedup**

Tendo apenas em questão a comparação entre tempos pela implementação multi-cycle e pela implementação pipeline vejamos a Figura 101.



**figura 101**  
tempos em single-cycle  
e pipeline

Interpretando a Figura 100 temos que uma instrução completa em multi-cycle demora cerca de 800 ps (admitindo que a unidade é **picosegundo** (ps)). Em comparação com os tempos da implementação em pipeline (imagem de baixo) temos que cada instrução, completa, demora 200 ps.

**picosegundo**

## ISA com pipelining no MIPS

Como já sabemos de normas do repertório de instruções do MIPS, todas as instruções têm um tamanho de 32 bits. Este procedimento facilita a execução da fase de *instruction fetch*, sendo que logo após a sua execução num ciclo, ela passa para a

fase de *instruction decode*. Também o facto de haver poucos formatos de instruções do repertório do MIPS facilita o processo, dado que os registos dos operandos estão sempre na mesma posição, por conseguinte, a leitura dos registos pode ser feita logo após a fase de *instruction fetch*, pelo que caso tal não se pudesse fazer, teríamos o grande problema de poder fazer apenas após a descodificação da instrução, obrigando a uma implementação pipeline de seis andares.

O facto dos operandos estarem em memória só nas operações de *load* e de *store* é um critério também importante a ter em conta - a fase de *execute* será, assim, a seguir à fase de leitura dos registos, permitindo executar a operação ou calcular o endereço de memória. Já em memória, os operandos estão todos alinhados, isto é, os dados podem ser transferidos num único acesso à memória.

## Hazards em implementações pipeline

Como podemos prever pela Figura 99, o speedup deve ser igual ao número de andares do pipeline, embora, na verdade, ele corresponda ao número de obstáculos ao aumento do desempenho por parte do pipeline. A estes obstáculos damos o nome de **hazards**. Os hazards podem ser de dois tipos: estruturais ou de dados. No caso dos **hazards estruturais** o que acontece é que o hardware não permite a combinação de algumas instruções no mesmo ciclo de relógio. Por outro lado, os **hazards de dados** (em inglês *data hazards*), ocorrem quando a admissão de novas instruções no pipeline tem de ser parada porque a execução numa das fases tem de ser parada à espera que outra fase seja completada, de forma a poderem estar disponíveis (os dados), necessários para outras instruções (como o exemplo do Código 59).

```
add    $s0, $t0, $t1
sub    $t2, $s0, $t3
```

**hazards**

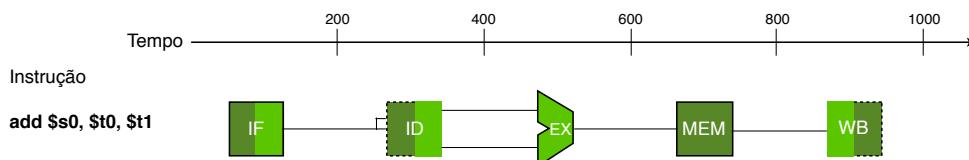
**hazards estruturais**

**hazards de dados**

**código 59**

instruções a experimentar

Podemos representar uma implementação pipeline como a Figura 102.



**figura 102**

tempo por pipeline

**nota!!** na Figura 101 a verde claro pretende-se representar a parte ativa dos componentes, sendo a leitura dos registos ou da memória feita na segunda parte do ciclo, a escrita na primeira parte do ciclo e o acesso à memória inativo, dada a desnecessidade de aceder à memória do registo.

Uma solução para o nosso grave problema do hazard de dados é através de **forwarding**. Esta solução provém da analogia de uma bolha. A bolha aqui será vista como uma simulação de instrução a ser cumprida. No exemplo do Código 60 haverá a necessidade de ser realizado o forwarding (Figura 103).

```
lw      $s0, 20($t1)
sub    $t2, $s0, $t3
```

**nota**

**forwarding**

**código 60**

instruções a experimentar

Em termos de software, a melhor solução que se pode apresentar é uma reordenação do código, de forma a que os cruzamentos de requisitos mínimos para a execução de dadas instruções se evitem.

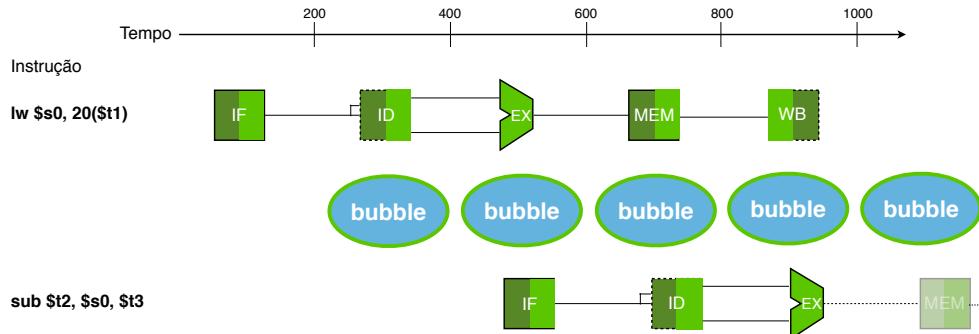


figura 103

tempo por pipeline (hazard)

## Control Hazards

Para fazer uma operação branch, é primeiro necessário fazer o *instruction fetch* da seguinte instrução, quando o resultado do branch ainda não está calculado. Mas como é que tal se pode processar? Para tal usa-se o poder da **previsão**. Fazer uma previsão de salto. Na Figura 104 podemos ver dois diagramas temporais nos quais, no primeiro encontramos um branch não tomado (*branch not taken*) com uma previsão correta, e no segundo com uma previsão errada.

previsão

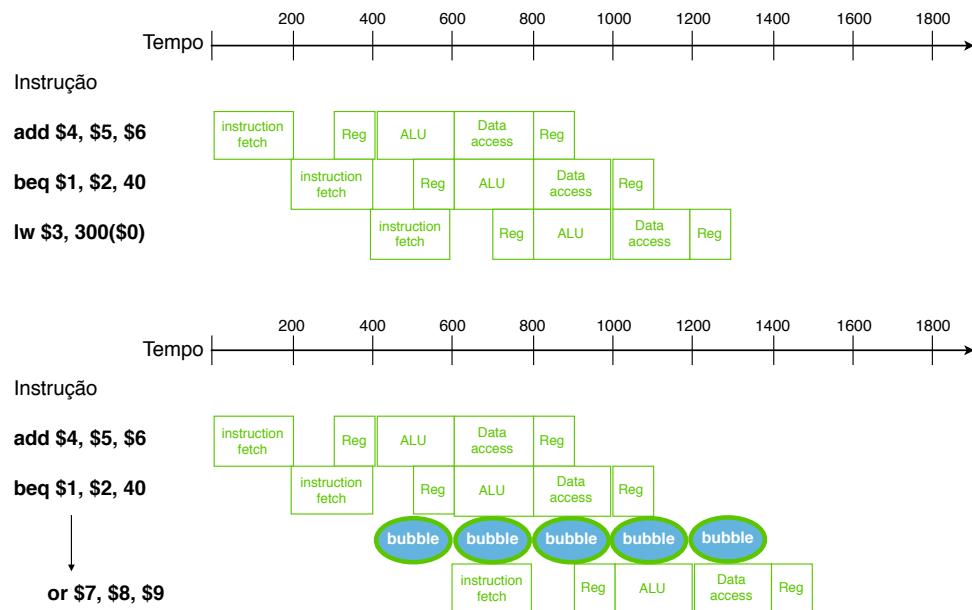


figura 104

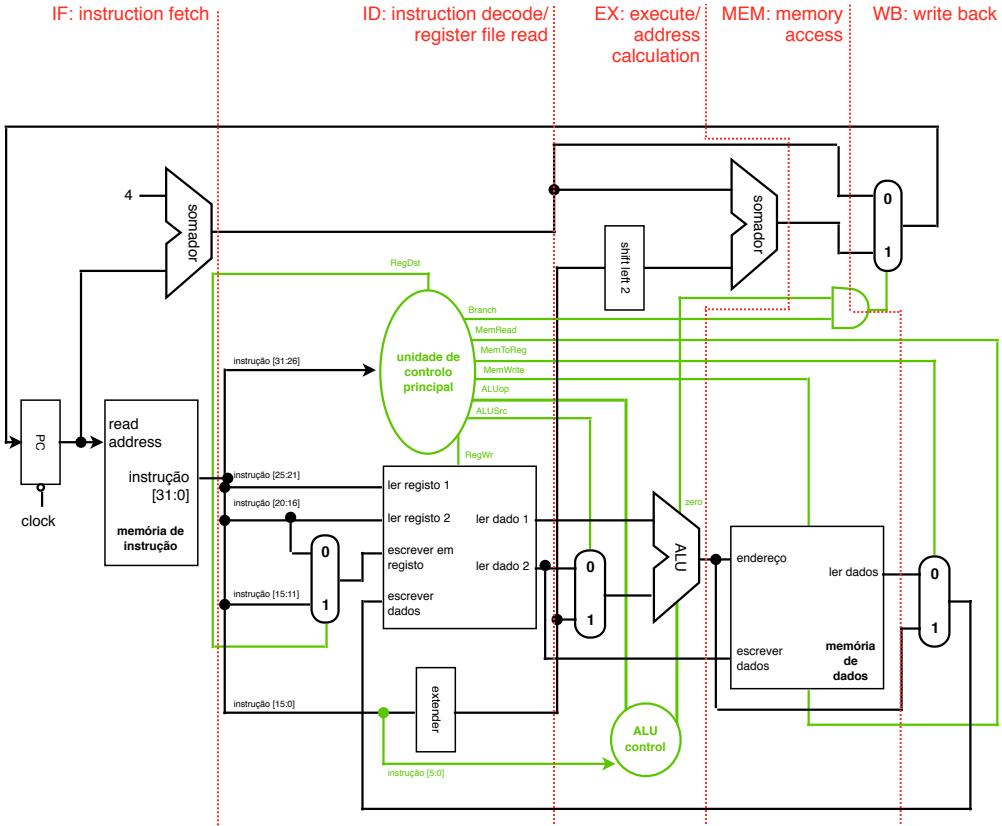
branch prediction

As formas mais elaboradas de previsão de branches (em inglês **branch prediction**) são, por exemplo, a *dynamic branch prediction*, a qual se baseia no historial do comportamento do branch, chegando a acertar em cerca de 90% das suas previsões. Uma alternativa é um atraso gerado no branch, usado no MIPS, onde a instrução a seguir ao branch é sempre executada, efetuando o salto com uma instrução de atraso - o *assembler* reordena as instruções de modo completamente transparente ao utilizador.

branch prediction

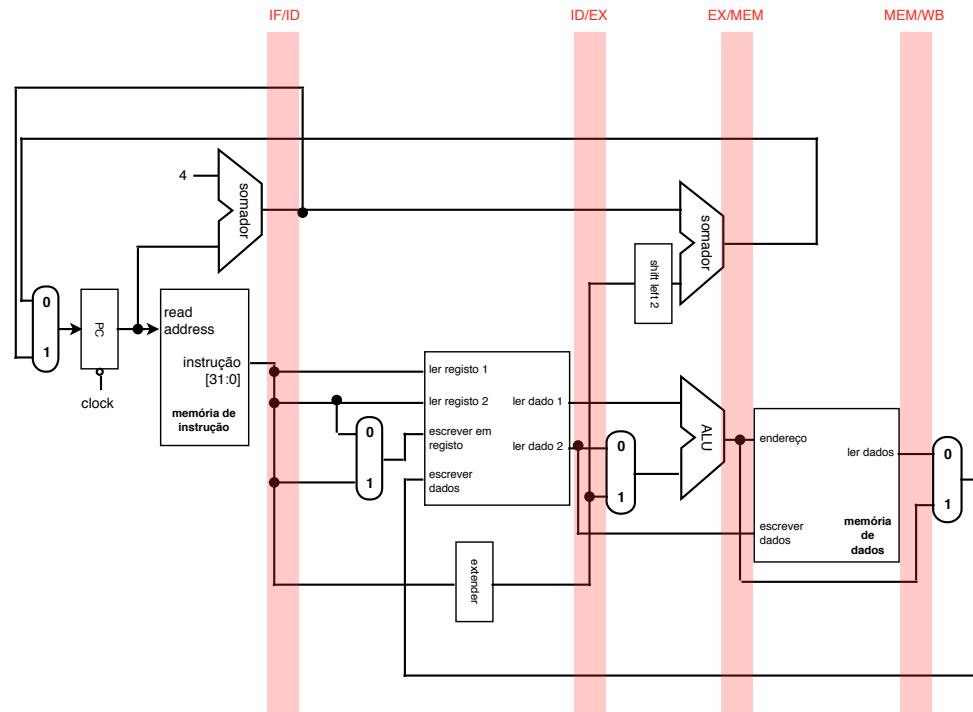
## Implementação do datapath em pipeline

Voltemos atrás no programa e vejamos a implementação single-cycle já estudada (Figura 78). Na Figura 105 temos a mesma implementação com a divisão de estados de execução, pelos próprios componentes da máquina.



**figura 105**  
implementação single-cycle  
segmentada

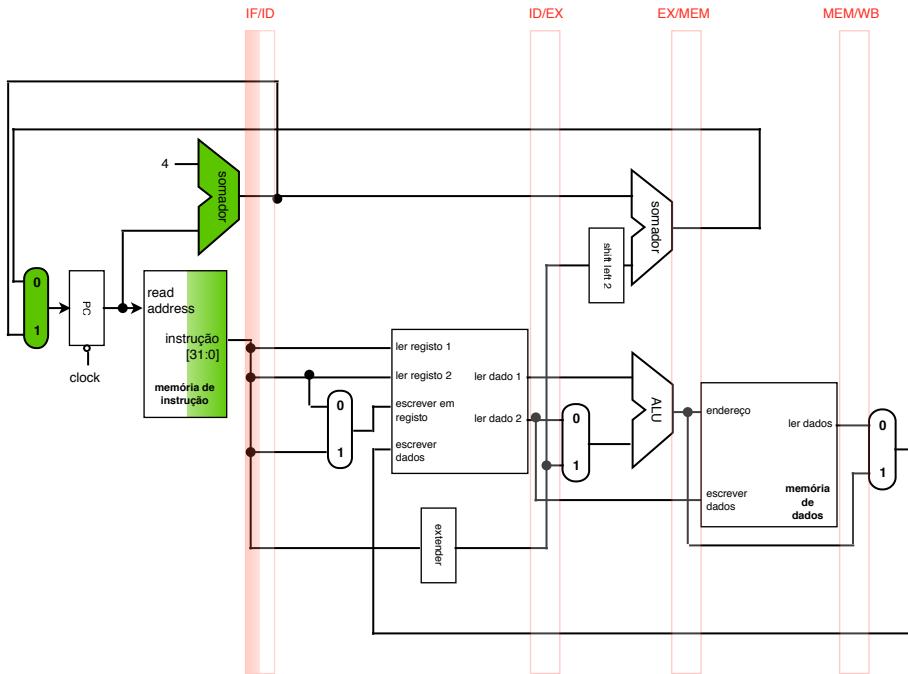
Tendo esta divisão feita, é a partir desta que vamos criar a implementação pipeline, gerando blocos representativos das transições entre fases, como podemos ver na Figura 106.



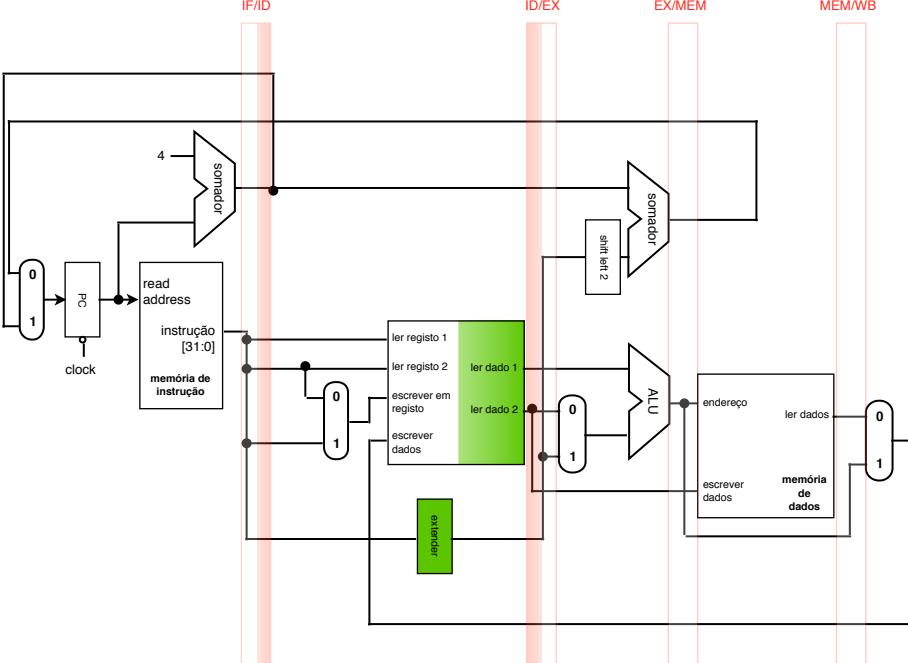
**figura 106**  
implementação pipeline

Tendo a nossa implementação concretizada, e tendo em consideração que os dados fluem sempre da esquerda para a direita, na Figura 105 - com exceção do estado *write back* e modificação do conteúdo do registo \$pc nos branches -, vejamos como é que se processam algumas instruções possíveis, em todas as fases (instruções load e store).

A instrução de *load word* (*lw*) começa por ser executada pela fase de *instruction fetch*, representada na Figura 107 por IF.

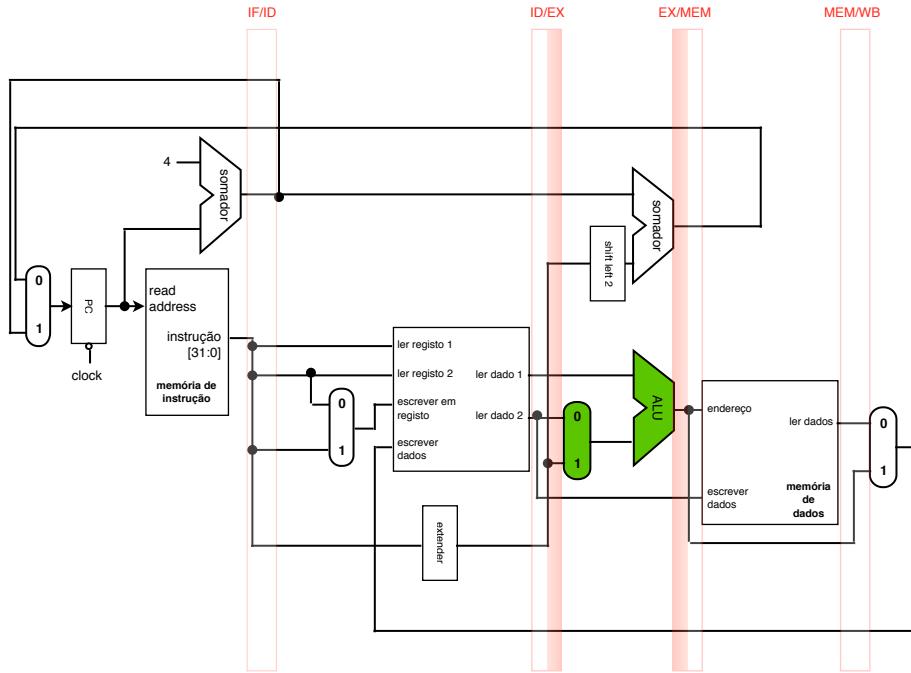


**figura 107**  
instrução *lw*



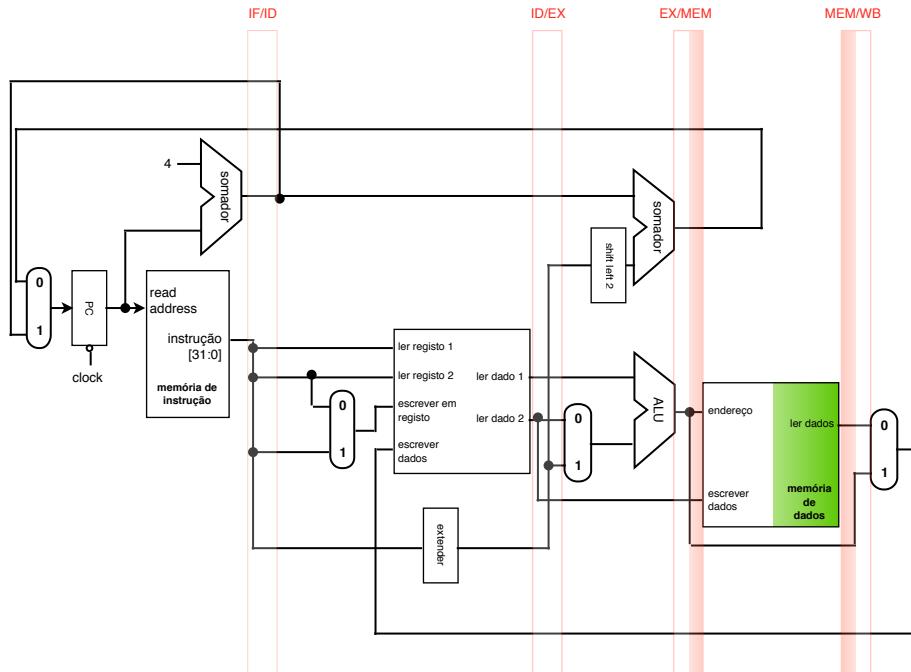
**figura 108**  
instrução *lw*

Na terceira fase (fase EX) a máquina estará no estado da Figura 109.



**figura 109**  
instrução lw

Depois de efetuar o cálculo do endereço de memória, a máquina deve entrar na última fase, a de *memory access*, representada por MEM na Figura 110.



**figura 110**  
instrução lw

Na última fase (Figura 111), com a instrução de *load word* (*lw*), há uma ativação do sinal proveniente da memória de instrução, sendo que pelo significado da fase em causa, a ativação deveria ser provocada pela memória de dados, ao contrário da memória de instrução. Mais à frente iremos tentar procurar por uma correção deste pormenor, pela sua implementação.

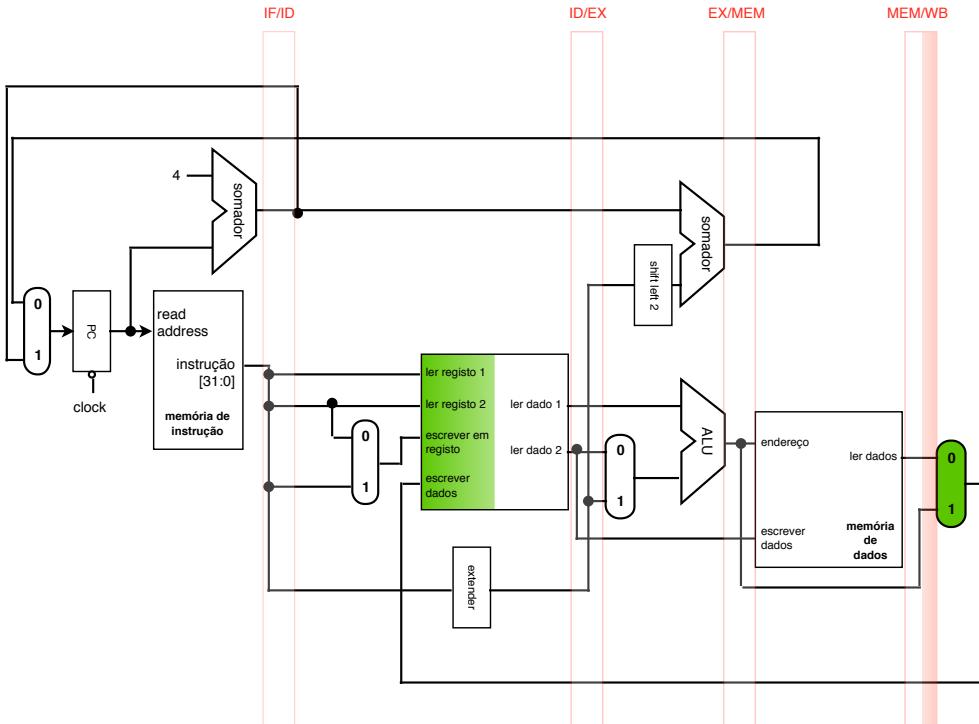


figura 111  
instrução `lw`

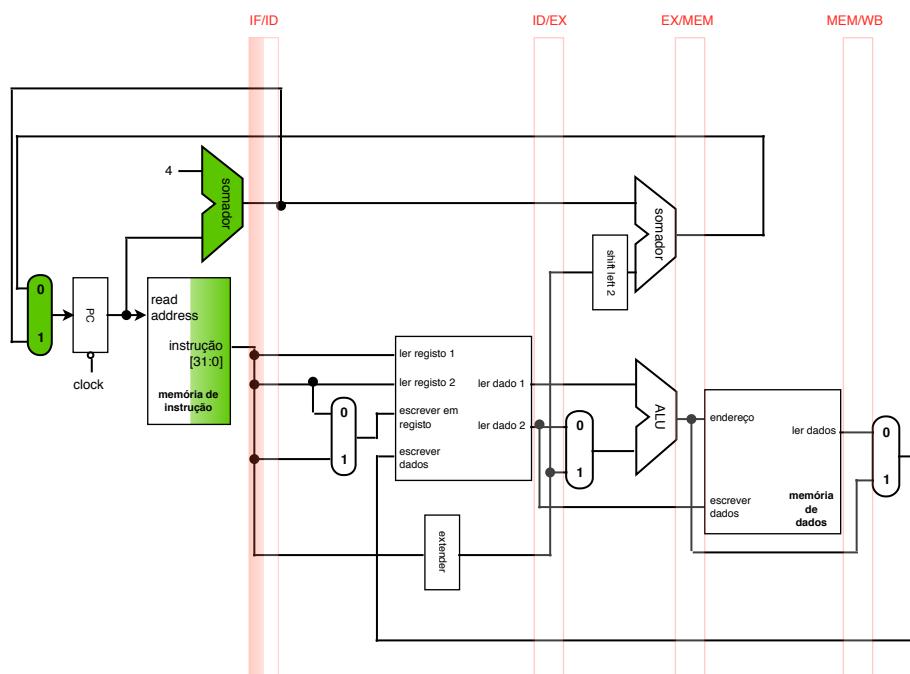
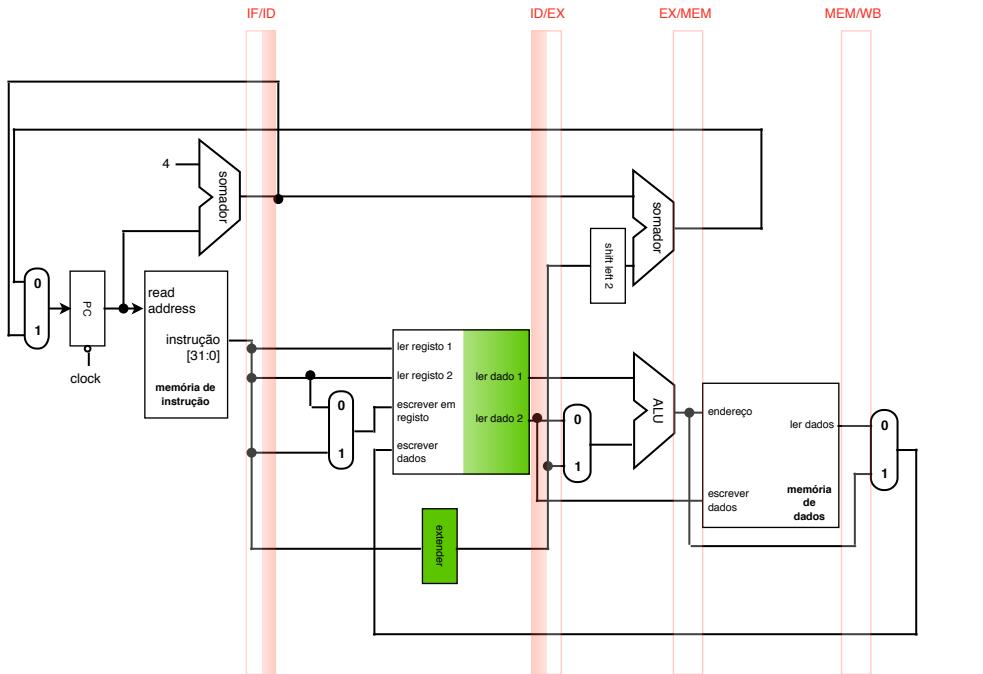


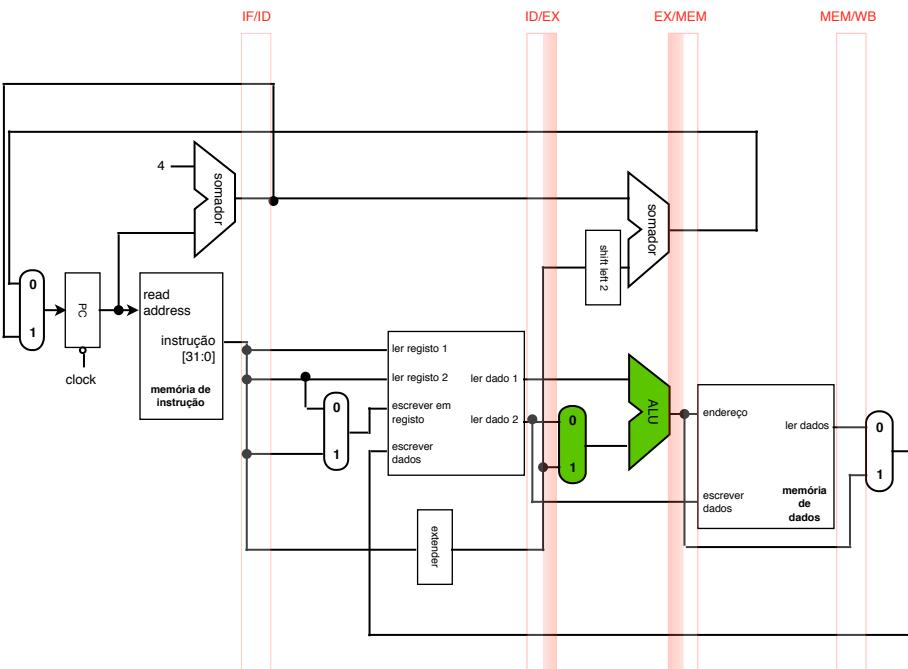
figura 112  
instrução `sw`

Passando para o estado seguinte, de *instruction decode*, a máquina apresentará-se à forma visível na Figura 113.



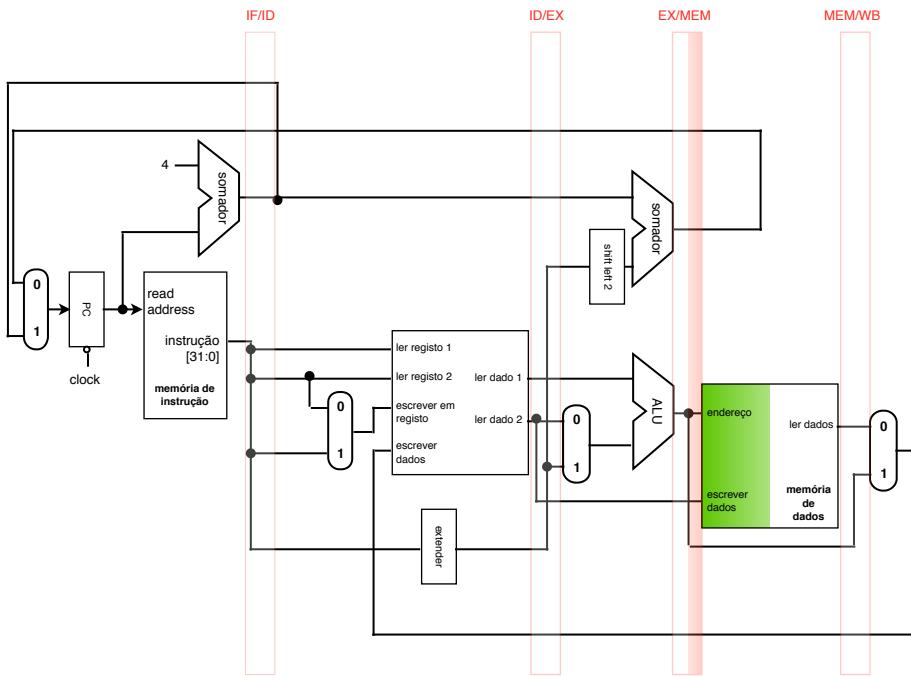
**figura 113**  
instrução sw

A terceira fase da instrução de *store word* apresenta a necessidade de calcular o endereço de memória, de forma a poder saber para que espaço é que o dado deve ser registado. Sendo assim, a nossa máquina deve apresentar o estado representado pela Figura 114.



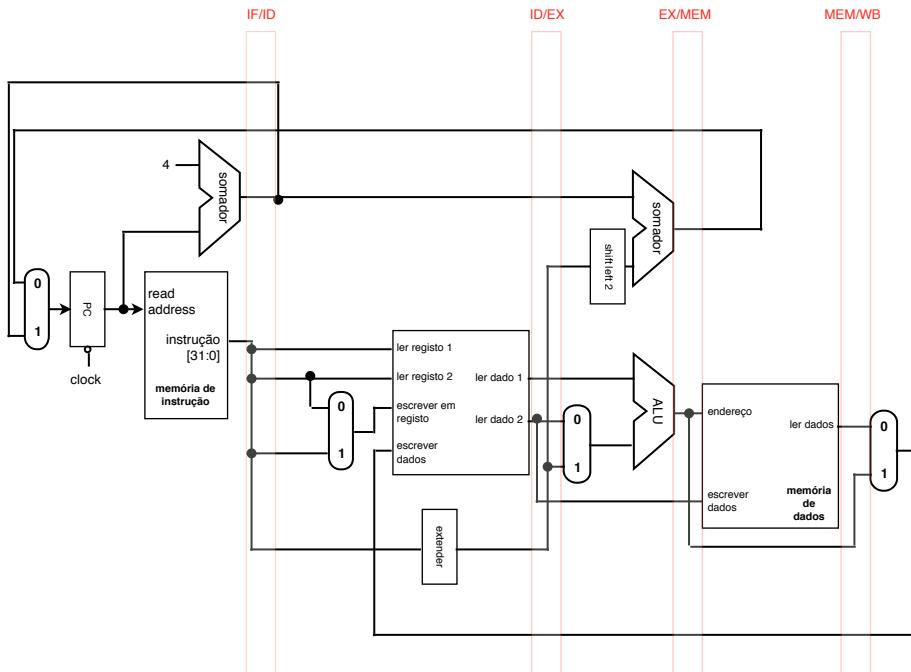
**figura 114**  
instrução sw

Na quarta fase de execução o processador deve aceder à memória efetuando a devida alteração pedida em instrução. Escrevendo os dados em memória, no espaço já calculado anteriormente, a máquina apresenta o estado em Figura 115.



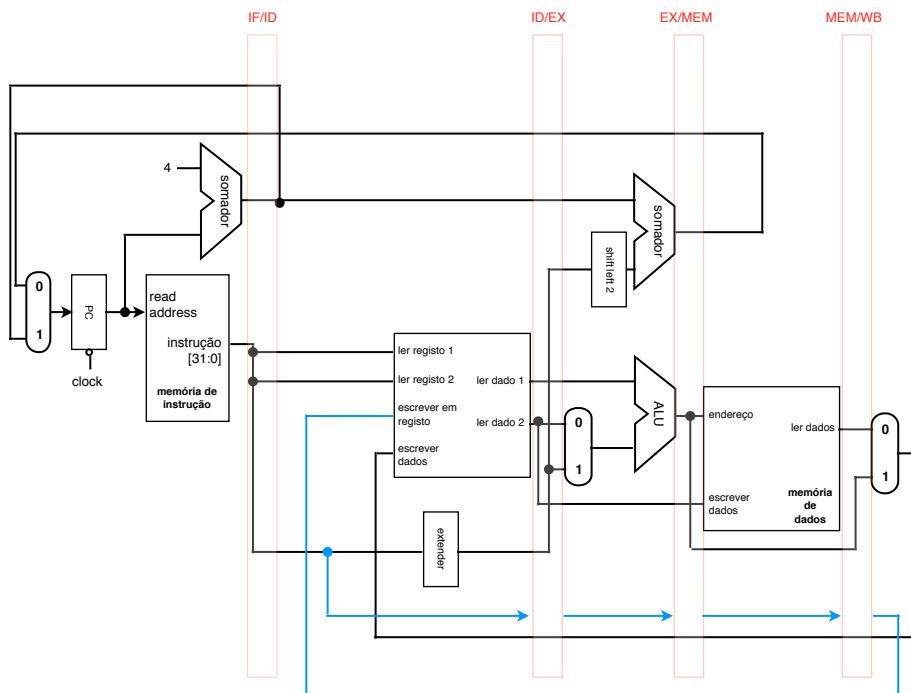
**figura 115**  
instrução *sw*

A “última fase” acresce de aspas porque nenhum componente do datapath estaria ativa, dado que com a instrução de *store word* não carece de necessidade de gravar dados em registos. Fica então a máquina no estado visível em Figura 116.



**figura 116**  
instrução *sw*

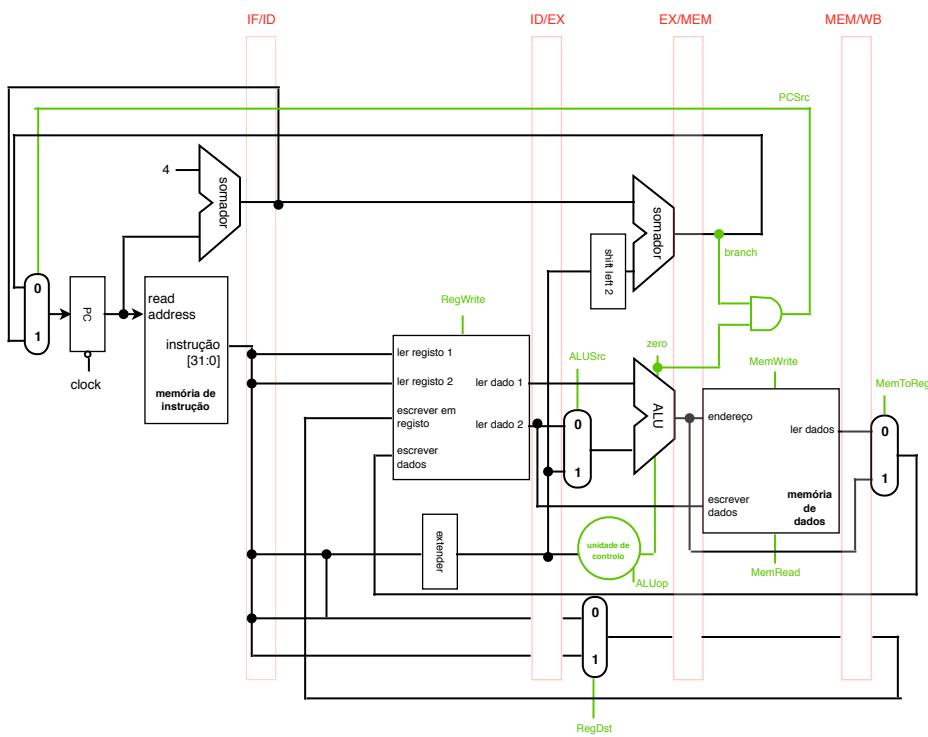
Como referimos atrás, o nosso datapath carece de uma pequena correção, de forma a que se trate uma ativação do sinal proveniente da memória de instrução, sendo que pelo significado da fase de *write back*, a ativação deveria ser provocada pela memória de dados, ao contrário da memória de instrução, com a instrução *load word*. Na Figura 117 mostra-se uma possível correção (a azul).



**figura 117**  
correção de implementação

### Implementação de controlo da unidade em pipeline

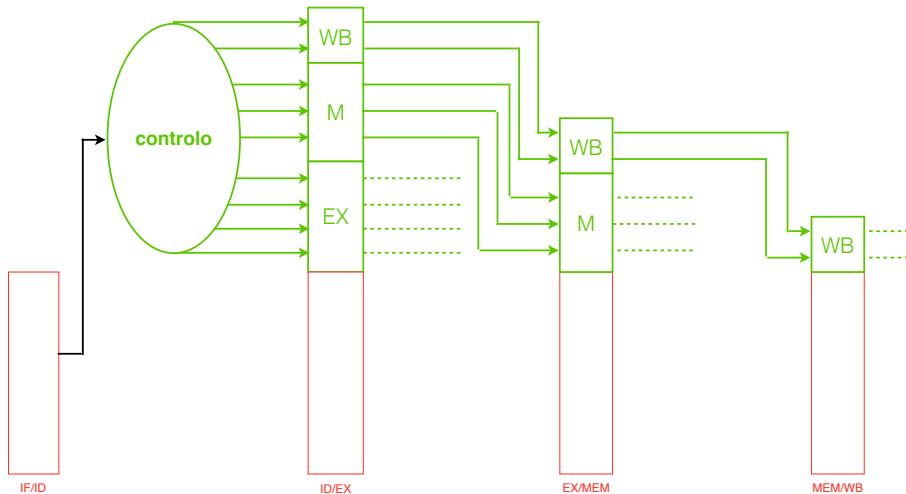
Como já fizemos nas implementações anteriores, para implementar a unidade de controlo na nossa unidade em estudo, primeiro identifiquemos os locais que necessitam de sinais de controlo e denominemos cada entrada (Figura 118).



**figura 118**  
sinais de controlo

Ao longo da execução de instruções pela máquina em estudo, de fase em fase há informação que tem de ficar salvaguardada em registos inter-andares, isto é, de transição de fase em transição de fase. Na transição IF/ID tanto o *instruction code* como o *program counter* deve ficar guardado, tal como na transição ID/EX os campos de *rt*, *rs* e *func* - de 16 bits -, o *program counter* e os dados de instrução dos bits 0 a 15, com extensão de sinal. Por fim, na transição EX/MEM, o código de instrução, campo *rt/rd* - 5 bits -, tal como o sinal de *ALUResult* ou o campo *rt* devem ser também guardados.

Nas diversas fases de processamento e execução da nossa máquina em estudo, os sinais de controlo devem apresentar diferentes valores. Na primeira fase de *instruction fetch* é sempre escrito no registo \$pc, em todos os ciclos, o seu valor, e a memória de registos deve ser lida em todos os ciclos, pelo que é desnecessário sinais de controlo nesta fase. Já no andar de *instruction decode/read register file* é exercida a mesma ação em todos os ciclos de relógio, tal como pudemos comprovar com a análise das instruções *lw* e *sw*, pelo que também não é necessário sinais de controlo neste andar. Estes sinais passam a ser importantes a partir da fase de execução ou cálculo de endereço de memória - os sinais importantes para esta fase são o *ALUSrc*, *ALUop* e *RegDst*. Na fase de *memory access* são importantes os sinais de controlo *branch*, *MemRead* e *MemWrite*, tal como no andar de *write back* é necessário o resultado dos sinais de controlo restantes: *MemToRead* e *RegWrite*. Dada esta associação de sinais de controlo com fases de processamento do nosso datapath, podemos sintetizar a nossa implementação - principalmente a nossa representação - da forma visível na Figura 119.



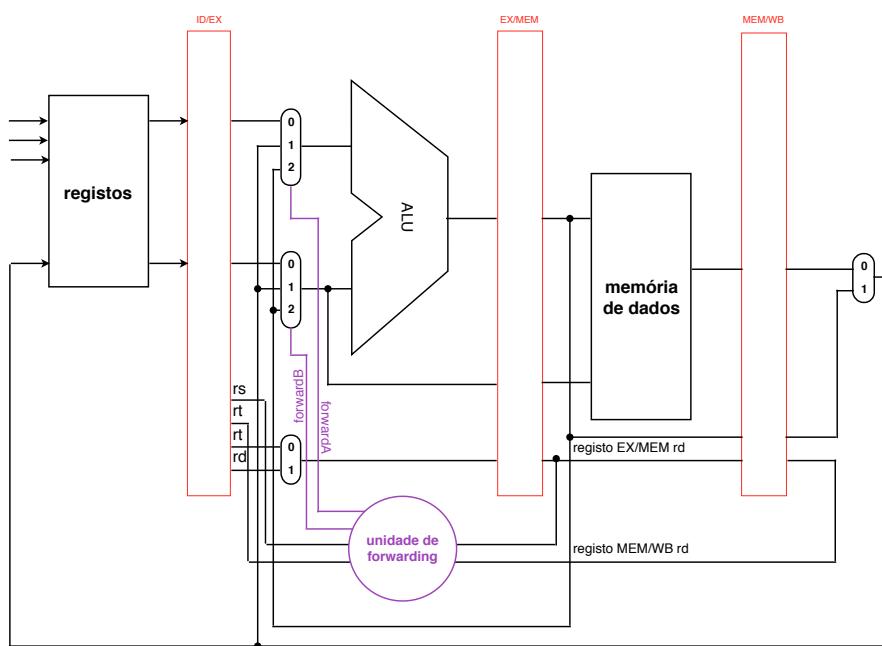
**figura 119**  
sinais de controlo

## Hazards em implementações pipeline (continuação)

Mais atrás referimos uma forma de tratar problemas de hazard de dados (*data hazarding*) - referimos a técnica de **forwarding**. Mas o que é o forwarding? O forwarding é uma técnica de aproveitamento de recursos por *bypass*, pelo que é criada uma unidade de forwarding (encaminhamento) que encaminha valores precisos por caminhos de espera descontínua, isto é, valores “novos” - aqueles que ainda não se encontram prontos-a-usar, eles são produzidos em dois locais no pipeline do processador - no estado EX e no estado MEM, registo o qual se designa por EX/MEM. A unidade de encaminhamento, no nosso pipeline, deve então ter dependências da área da execução da fase EX e da MEM.

**forwarding**

Um pipeline com unidade de forwarding é uma igual à da Figura 120.



**figura 120**  
unidade de forwarding

Com uma unidade de forwarding há a necessidade de estabelecer algumas condições de hazarding, entre elas  $registro\ EX/MEM\ rd = ID/EX.\underline{registro\_rs}$ ,  $registro\ EX/MEM\ rd = ID/EX.\underline{registro\_rt}$ ,  $registro\ MEM/WB\ rd = ID/EX.\underline{registro\_rs}$ ,  $registro\ MEM/WB\ rd = ID/EX.\underline{registro\_rt}$ .

Temos então dois grupos de hazarding: do bloco EX e do bloco MEM. Mas em que condições, pelas já designadas, é que acontece hazard em cada um dos blocos? No bloco EX acontece hazard quando:

```
se (EX/MEM.RegWrite e (EX/MEM.RegisterRd ≠ 0)
e (EX/MEM.RegisterRd = ID/EX.RegisterRs)):
ForwardA = 10
```

```
se (EX/MEM.RegWrite e (EX/MEM.RegisterRd ≠ 0)
e (EX/MEM.RegisterRd = ID/EX.RegisterRt)):
ForwardB = 10
```

Já no bloco MEM acontece hazard quando:

```
se (MEM/WB.RegWrite e (MEM/WB.RegisterRd ≠ 0)
e (MEM/WB.RegisterRd = ID/EX.RegisterRs)):
ForwardA = 01
```

```
se (MEM/WB.RegWrite e (MEM/WB.RegisterRd ≠ 0)
e (MEM/WB.RegisterRd = ID/EX.RegisterRt)):
ForwardB = 01
```

De forma mais resumida e sintetizada, a Figura 121 mostra uma tabela na qual se sumaria o controlo dos multiplexeres.

controlo dos multiplexeres	fonte	explicação
ForwardA = 00	ID/EX	o primeiro operando da ALU provém do registo ( <i>register file</i> )
ForwardA = 10	EX/MEM	o primeiro operando da ALU é encaminhado do resultado anterior da ALU
ForwardA = 01	MEM/WB	o primeiro operando da ALU é encaminhado da memória de um resultado da ALU recente
ForwardB = 00	ID/EX	o segundo operando da ALU provém do registo ( <i>register file</i> )
ForwardB = 10	EX/MEM	o segundo operando da ALU é encaminhado do resultado anterior da ALU
ForwardB = 01	MEM/WB	o segundo operando da ALU é encaminhado da memória de um resultado da ALU recente

Se tentarmos executar, com esta implementação, o Código 61 iremos encontrar um problema, pelo que na execução da terceira instrução tem de se fazer o encaminhamento do resultado da segunda soma.

```
add $1, $1, $2
add $1, $1, $3
add $1, $1, $4    # (EX/MEM.RegisterRd = ID/EX.RegisterRs) & (MEM/WB.RegisterRd =
# = ID/EX.RegisterRs)
```

Para fazer o encaminhamento que precisamos, precisamos de fazer a seguinte alteração nos hazards do MEM:

```
se (MEM/WB.RegWrite e (MEM/WB.RegisterRd ≠ 0)
e (MEM/WB.RegisterRd = ID/EX.RegisterRs)) e
não (EX/MEM.RegWrite e (EX/MEM.RegisterRd ≠ 0) e
(EX/MEM.RegisterRd = ID/EX.RegisterRs)):
ForwardA = 01

se (MEM/WB.RegWrite e (MEM/WB.RegisterRd ≠ 0)
e (MEM/WB.RegisterRd = ID/EX.RegisterRt)) e
não (EX/MEM.RegWrite e (EX/MEM.RegisterRd ≠ 0) e
(EX/MEM.RegisterRd = ID/EX.RegisterRt)):
ForwardB = 01
```

Vejamos agora a implementação do nosso datapath com unidades de encaminhamento e de controlo completas (Figura 122).

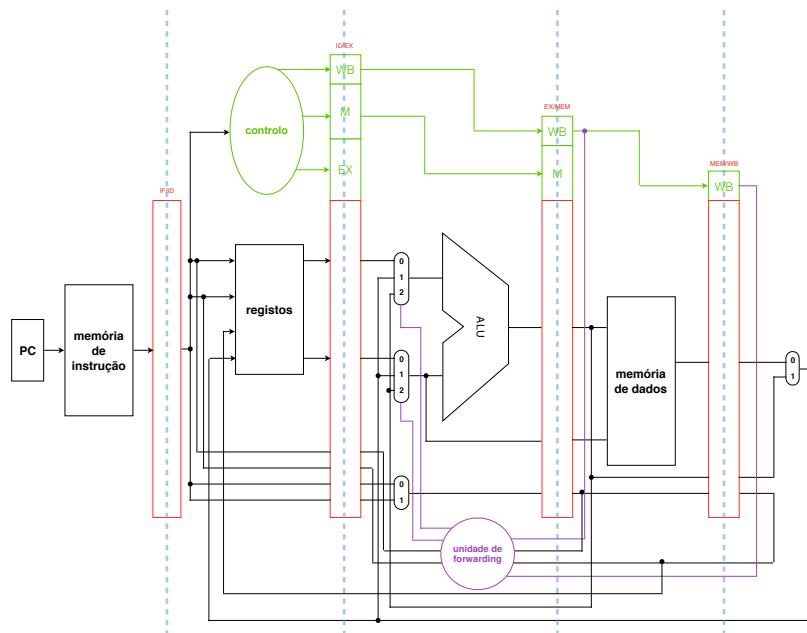


figura 121  
tabela de controlo

código 61  
instruções a experimentar

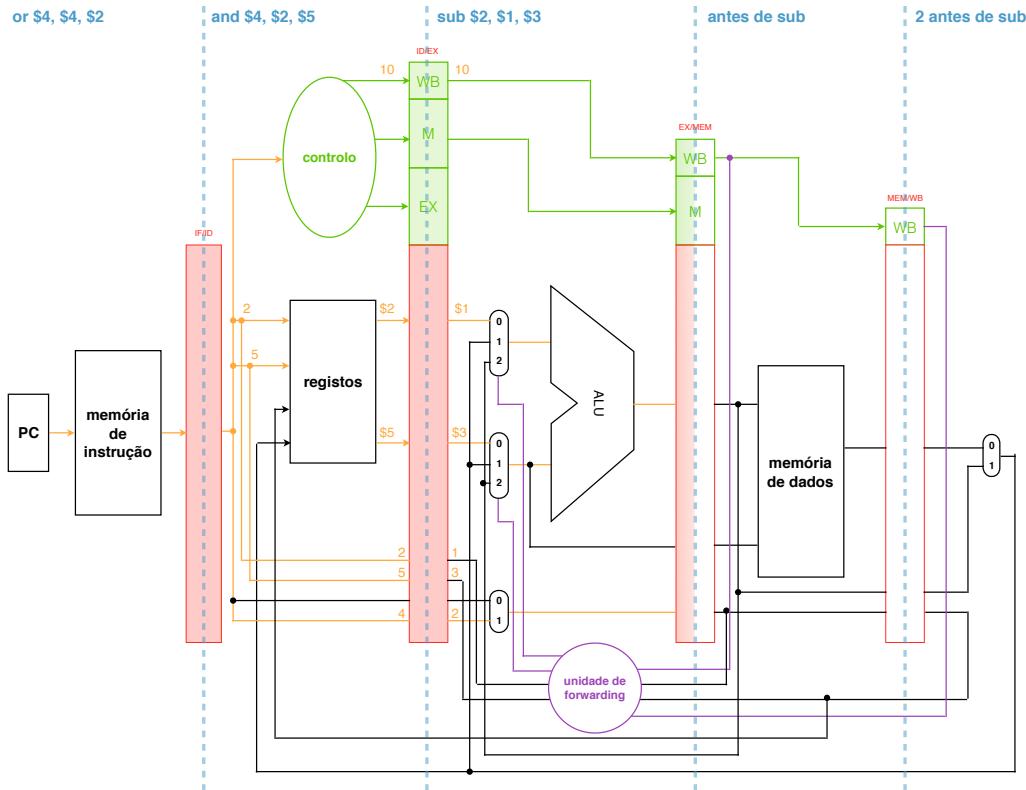
figura 122  
implementação de  
forwarding

Experimentemos simular o comportamento da máquina para a execução do Código 62.

```
sub      $2, $1, $3
and      $4, $2, $5
or       $4, $4, $2
add      $9, $4, $2
```

**código 62**  
instruções a experimentar

A execução do Código 60 na nossa implementação com encaminhamento no terceiro ciclo de relógio está representada na Figura 123.



**figura 123**  
implementação de  
forwarding

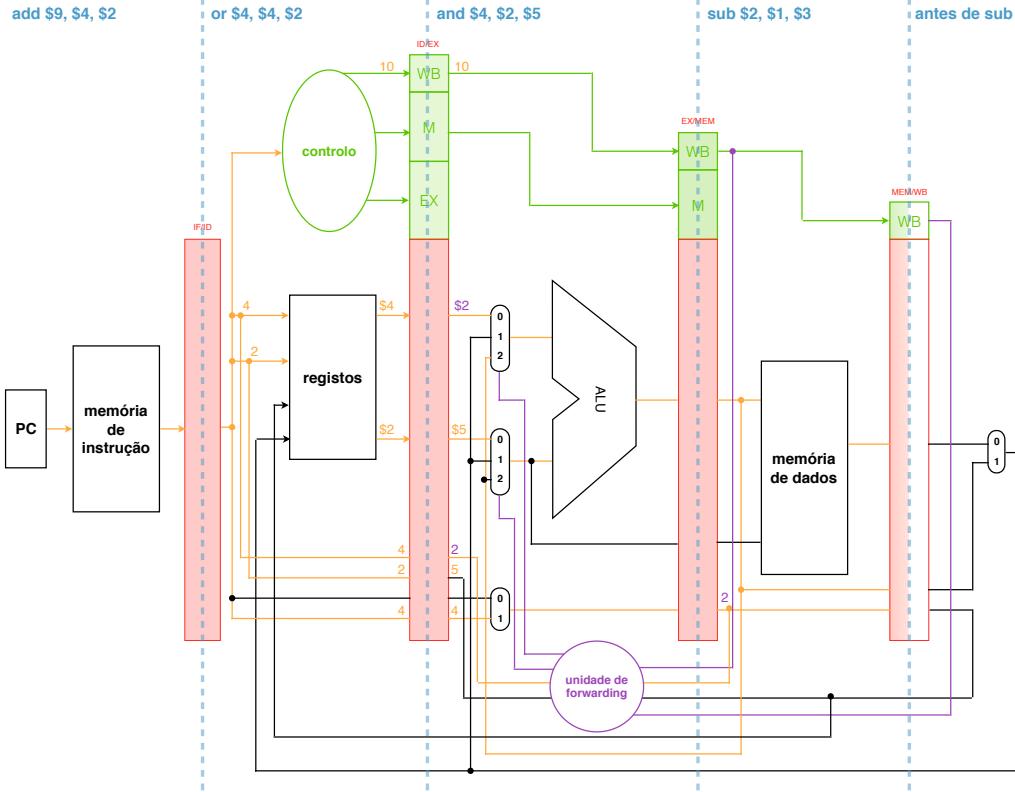
**nota!!** as linhas do circuito da Figura 122 ativas estão com cor alaranjada, tal como o valor dos seus sinais.

**nota**

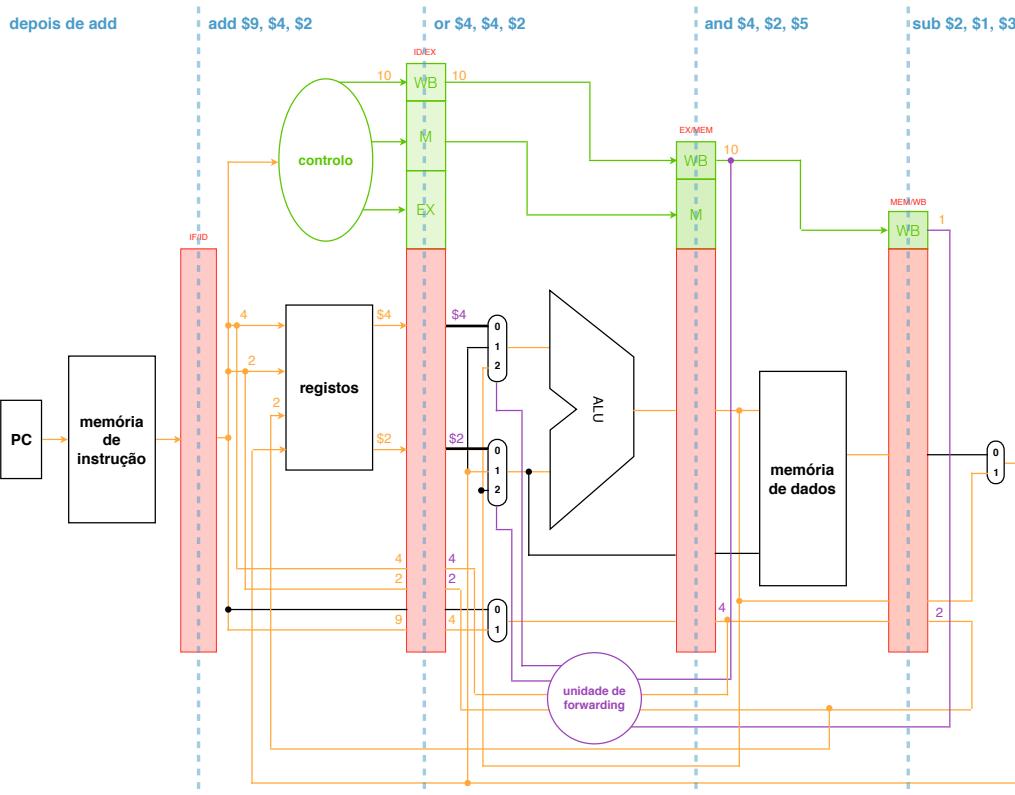
A execução do quarto ciclo de relógio no pipeline é exibida na Figura 124, na qual as instruções fluem da esquerda para a direita, entrando a instrução de adição.

Para o penúltimo ciclo de execução por parte do nosso processador com datapath projetado com lógica pipeline e com unidade de encaminhamento, é neste que há execução de retorno de valores para os registos. Este passo de execução pode ver-se na Figura 125.

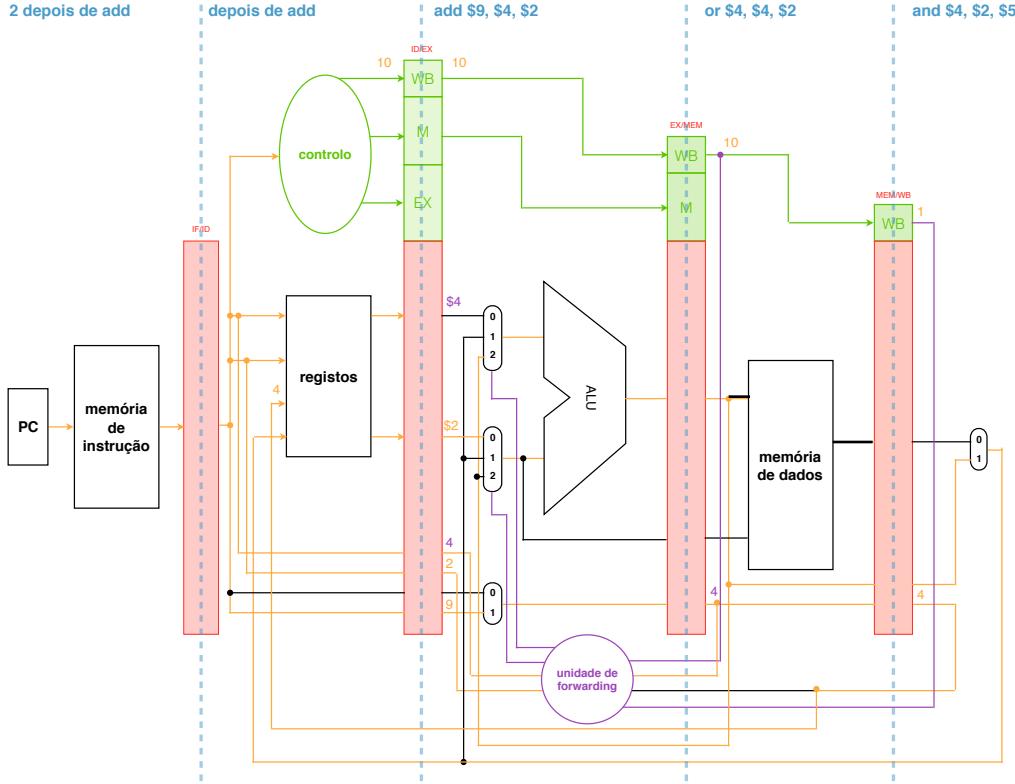
Para finalizar, na Figura 126, o sexto e último passo de processamento termina a lógica disponível para a expressão da instrução com que iniciámos a nossa análise - a operação disjunção lógica “ou” (*or \$4, \$4, \$2*).



**figura 124**  
implementação de  
forwarding



**figura 125**  
implementação de  
forwarding



**figura 126**  
implementação de  
forwarding

Estes exemplos de execução são exemplos de operações simples, do tipo R, mas mal começamos a tratar de instruções de acesso à memória (transferências de dados), o cenário muda por completo, porque estas instruções só calculam o endereço no passo EX (execução). Não é possível fazer o **stall** aqui. Para resolver esta situação usa-se uma unidade de deteção de hazards. Esta unidade deve funcionar segundo a seguinte condição:

```
se (ID/EX.MemRead e ((ID/EX.RegisterRt = IF/ID.RegisterRs) ou
(ID/EX.RegisterRt = IF/ID.RegisterRt)):
    stall pipeline
```

O passo de stall significa que o valor do registo \$pc deve ser mantido - repete-se a leitura de uma determinada instrução (a atual) - e mantém-se o conteúdo do registo IF/ID. Numa reorganização de código o que o *assembler* deve fazer é introduzir uma instrução *nop* (instrução “*no operation*”), pelo que coloca a zero os campos EX, MEM e WB do registo ID/EX do pipeline.

stall

## Control hazards (continuação)

Anteriormente ficámos por definir os *pipeline hazards* introduzidos pelas instruções branch. O problema, então, que se coloca é como determinar a próxima instrução para a qual se deve fazer a fase de *fetch*. Para lidar com este tipo de hazards, como já vimos por alto atrás, prevemos o resultado do branch (*branch prediction*).

A previsão de branches, estáticos, para o caso de *branches not taken*, isto é, branches que avaliam afirmações falsas, é feita do seguinte modo: primeiro há que assumir que a condição de branch não se verifica e só depois continuar a execução.

Caso a condição de branch se verificar, as instruções que estão nas fases de *instruction fetch* e *instruction decode*, têm de ser descartadas, de forma a poder continuar a execução na instrução alvo do branch. Por descartar instruções, pretende-se dizer colocar a zero os respetivos sinais de controlo - semelhante ao que foi feito para as instruções de *load*, só com a diferença de que é para as fases IF, ID e EX (e não apenas para a fase de ID).

Para modificar o nosso datapath há que antecipar a avaliação da condição de branch e o cálculo do endereço-alvo. A avaliação dessa condição de igualdade não exige a utilização da ALU, pelo que basta um componente que execute a operação XOR *bitwise* dos dois registos e fazer OR do resultado - o que pode ser feito na fase ID. Isto introduz a necessidade de encaminhamento para o estado ID e se a instrução imediatamente anterior escrever num dos registo da condição de branch ativa-se o estado de stall. Por fim deve ser feito o cálculo do endereço-alvo, através do somador dedicado, o qual pode ser colocado no andar ID.

Há pouco também referimos numa técnica de previsão de branch chamada de *dynamic branch prediction*. Esta previsão, embora também estática, é independente do comportamento real do branch. Uma alternativa a este método é um outro denominado de *branch prediction buffer* (*branch history table*), que se baseia numa pequena memória acedida pelos bits menos significativos do endereço da instrução de branch em que cada posição da memória tem um bit que indica se o branch foi ou não aceite por *taken* na execução anterior.

E assim termina o conteúdo programático da disciplina de Arquitetura de Computadores I (a2s1). Para dar continuidade a estes apontamentos, segue-se a disciplina de Arquitetura de Computadores II (a2s2).



## 1. Introdução à Arquitetura de Computadores

Modelo de von Neumann.....	2
Conjunto de instruções de uma arquitetura.....	3
Evolução dos computadores.....	4
Evolução do software.....	6
Tecnologia integrada.....	7

## 2. Introdução à Arquitetura MIPS

Relação entre endereço e conteúdo da memória.....	9
Execução de instruções.....	10
Conjunto de instruções do MIPS .....	10
Organização da memória .....	12
Representação de sistemas numéricos com e sem sinal.....	12
Codificação das instruções em MIPS .....	13
Operadores lógicos.....	15
Instruções de escolha e operações condicionais .....	16
Basic Blocks.....	22
Construção de instruções case/switch .....	22
Modos de endereçamento das instruções branch e jump.....	23
Arrays e Ponteiros .....	24
Utilização dos registos .....	27
Invocação de procedimentos/funcões .....	27
Introdução à Stack .....	28
Tipos de procedimento e processo de invocação .....	29
Preservação de valores sob invocações de procedimentos .....	30
Aplicação de procedimentos.....	30
O processo de assemblagem.....	32
Diretivas do MIPS.....	33
Pseudoinstruções .....	34
Assembler de duas passagens.....	34
Formato de um ficheiro.....	35
Linker .....	35
Loader .....	35

## 3. Aritmética Binária

Soma e subtração de quantidades binárias.....	36
Unidade aritmética e lógica (ALU) .....	36
Multiplicação binária .....	37
Algoritmo de Booth .....	39
Multiplicação no MIPS .....	40
Divisão binária.....	41
Divisão no MIPS .....	42

## 4. Representação de Vírgula Flutuante

Formato de representação .....	44
Representação de valores especiais .....	45
Gamas de representação por precisão .....	45
Conversão de quantidade decimal para vírgula flutuante .....	45
Conversão de quantidades em vírgula flutuante para decimal.....	46
Adição em vírgula flutuante .....	47
Multiplicação em vírgula flutuante .....	48
Processamento de vírgula flutuante em MIPS .....	48

## 5. Avaliação de Desempenho dos Sistemas de Computação

Medições de desempenho .....	51
Potência consumida.....	53
Desempenho relativo .....	53

## 6. Implementação Single-cycle de um Processador

Fases de construção de um processador .....	54
Análise do repertório de instruções (ISA) .....	54
Seleção dos componentes para o datapath.....	55

Clocking.....	56
Construção do datapath.....	56
Identificação de sinais de controlo .....	57
Realização da lógica de controlo .....	62
Caminho crítico.....	64
Desvantagens do processador single-cycle .....	65

## 7. Implementação Multi-Cycle de um Processador

Fases de Execução.....	67
Identificação dos sinais e realização da lógica de controlo .....	71

## 8. Implementação com Pipelining

ISA com pipelining no MIPS.....	75
Hazards em implementações pipeline .....	76
Control Hazards .....	77
Implementação do datapath em pipeline.....	77
Implementação de controlo da unidade em pipeline .....	84
Hazards em implementações pipeline (continuação) .....	85
Control hazards (continuação) .....	90

## Apontamentos de Arquitetura de Computadores I

1ª edição - janeiro de 2015

ac1

**Autor:** Rui Lopes

**Fontes bibliográficas:** Computer Organization and Design: The Hardware/Software Interface, Patterson, David A., Hennessy, John L., Fourth Edition (2012); Computer Architecture: A Quantitative Approach, John L. Hennessy, David A. Patterson, Third Edition; Digital Design: Principles and Practices, WAKERLY, J. F., Prentice-Hall, 2005; Fundamentals of Logic Design, ROTH, Jr. Charles H.. CL Engineering, 2009; Computer Science, An overview, BROOKSHEAR, James, Addison-Wesley, 2008.

**Outros recursos:** Notas das aulas de Arquitetura de Computadores I.

**Agradecimentos:** Professor Antônio de Brito Ferrari e Professor Adrego da Rocha.

Todas as ilustrações gráficas são obra de Rui Lopes e as imagens são provenientes das fontes bibliográficas divulgadas.



apontamentos

© Rui Lopes 2015 Copyright: Pela Creative Commons, não é permitida a cópia e a venda deste documento. Qualquer fraude será punida. Respeite os autores e as suas marcas. Original - This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit [http://creativecommons.org/licenses/by-nc-nd/4.0/deed.en\\_US](http://creativecommons.org/licenses/by-nc-nd/4.0/deed.en_US).