**Arquitetura de Computadores – Antes de Single Cycle.**

**Aula 1**

**Computador: The big Picture**

**CPU:** ou microprocessador, executa sequencialmente instruções. Responsável pelo processamento de informação.

**Memória:** Armazena o programa (conjunto de instruções) e dados e resultados

**IO Devices:** Comunicação com o exterior: Unidades de entrada e de Saída

**System Bus:** Interliga os subsistemas:

* Data Bus: Barramento de transferência de informação CPU <--> Memória, CPU <--> Input/Output.
* Adress Bus: identifica a origem e o destino da informação.
* Control Bus: sinais de protocolo que especificam o modo como a transferência deve ser feita.

**Visão simplificada do CPU**

**ALU:** (Unidade Aritmética e Lógica), realiza as operações aritméticas e lógicas mais comuns (Soma, Mult, Div, AND, NOR, XOR; OR)

**Registos:** elementos de armazenamento (memória) localizados dentro do CPU. Um registo armazena uma única unidade de informação: por exemplo 1 byte.

**Unidade de Controlo:** responsável pela coordenação de vários blocos do CPU, durante a execução de uma instrução.

**Registos mais importantes:**

**Program Counter (PC):** Usado para guardar o endereço da memória onde se situa a próxima instrução a ser executada. Após a leitura do código da instrução, o PC é incrementado para a instrução seguinte.

**Registos de utilização geral**

**Programa**

É um conjunto de instruções que vão ser executadas sequencialmente pelo CPU.

Envolve 3 passos fundamentais:

**Leitura da Instrução:** as instruções estão armazenadas na memória externa

**Descodificação da instrução:** qual operação a realizar, onde residem os operandos e onde deve ser colocado o resultado

**Realização da Operação:** Associada á instrução

Após a execução da instrução, o CPU avança para a instrução seguinte.

**Assembleia**

Linguagem básica de programação de microprocessadores, legível por humanos

Conjunto de instruções que realizam operações simples como somar o conteúdo de 2 registos, subtrair, inicializar um registo com um valor, transferir um registo interno para a memória.

Add $1, $5, $7

Sub $3, $4, $2

Ori $6, $0, 0x1234

**Código Máquina**

Sequência de bits que codifica cada uma das instruções assembly.

Exemplo:

Add $1, $5, $7 -> 0x00A70820

Sub $3, $4, $7 -> 0x00821822

Ori $6, $0, 0x1234 0x340 -> 0x34061234

É gerado por um compilador quando é escrito numa linguagem, por exemplo assembler para assembly.

**MIPS**

É um **microprocessador de 32 bits**, ou seja cada registo interno armazena uma word (de 32 bits) e a ALU opera sobre 32 bits.

Tem 32 registos internos de uso geral.

Podemos fazer **operações aritméticas** (add, sub , addi, etc...), **operações lógicas bitwise** (and, or, ori, etc...), **operações de deslocamento** (sll, srl, sra)

**System calls**

Como funciona uma syscall?

1. O SO verifica o valor de $v0 para saber qual é a tarefa a realizar
2. (Se necessário) o SO lê os valores de entrada dos registos $a0 a $a3 (argumentos)
3. O SO executa a tarefa
4. O SO coloca o resultado no registo $v0

Ex.:

Ori $v0, $0, 11 # syscall print char

Ori $a0, $0, 0x31 # argumento = 0x31 = ‘1’ caractere 1

Syscall

**Aula 2**

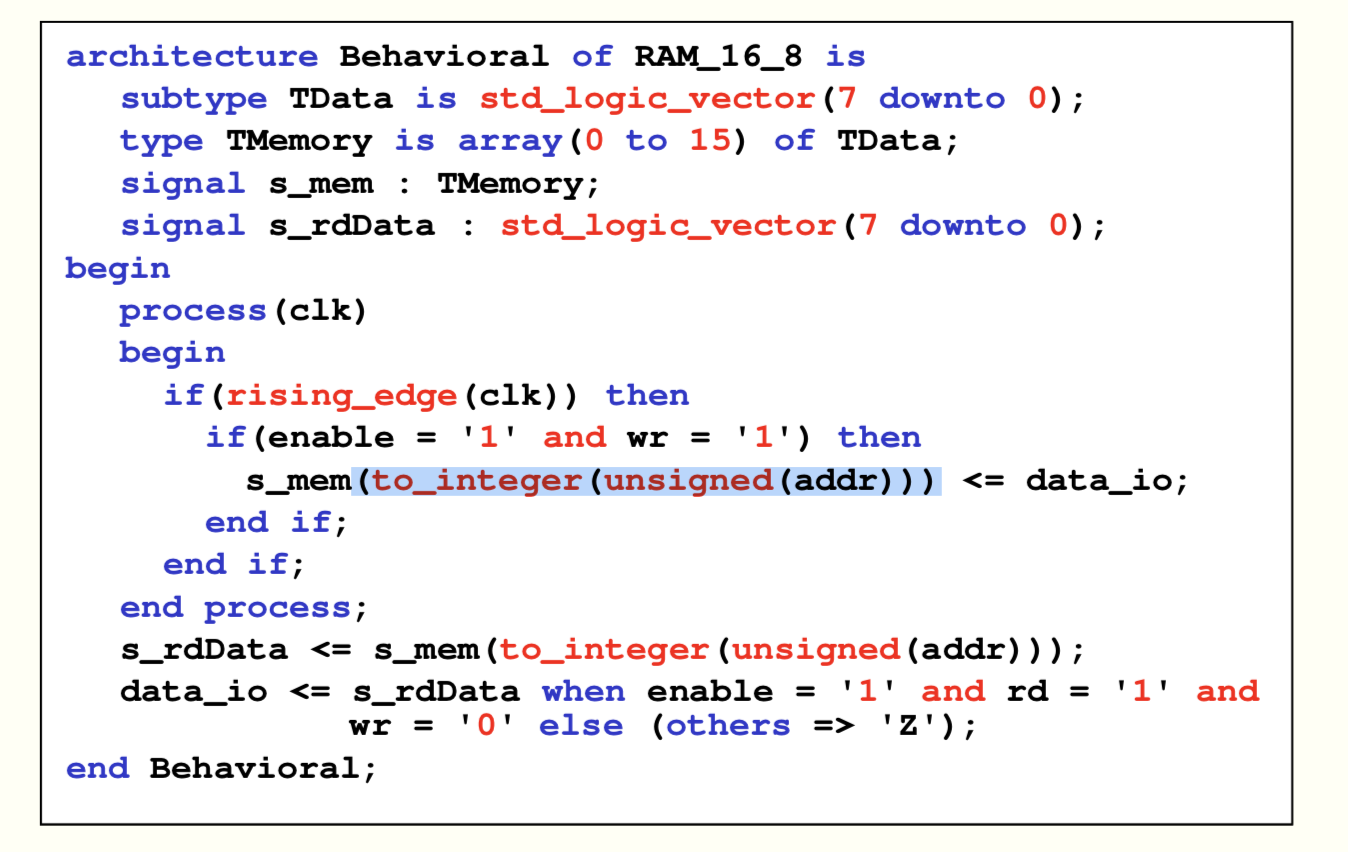
**Memória RAM**

**Memória:** Dispositivo capaz de armazenar informação digital binária

São construídos à custa de flip-flops.

Portos de entrada: clk, addr (vector de 4 bits), enable, wr (write), rd (read)

Portos de entrada e saída: data\_io (vector de 8 bits)



Para compreendermos o algoritmo:

**type** type\_name **is array** (range) **of** element\_type;

* **Addr:** endereço de 4 bits porque dá para 16 combinações.
* **TData** é o elemento atómico, o byte que é 8 bits.
* **TMemory** é um array de 16 posições de memória do tipo TData (byte)
* **s\_mem**: signal interno com TMemory
* **s\_rdData**: signal interno vector 8 bits
* Se o **enable** e o **write** estiverem a um (1), passa o valor do **data\_io** (em byte) para o array**s\_mem** na posição de memória do valor do addr inserido. Depois disso passa a data do **s\_mem** para **s\_rdData** que **passa** para para a saída **data\_io** se o **enable** estiver a um (1), o **read** tiver a um (1) e o **write** tiver a **zero(0).** Se não fica em alta impedância.

Exemplo:

Clk = 1, addr = 1001, enable = 1, wr = 1, rd = 0, data\_io = 11110000

Logo:

to\_integer(unsigned(1001) == 9

Addr s\_mem

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | Null | Null | Null | Null | Null | Null | Null | Null |
| 1 | Null | Null | Null | Null | Null | Null | Null | Null |
| 2 | Null | Null | Null | Null | Null | Null | Null | Null |
| 3 | … |  |  |  |  |  |  |  |
| 4 |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |
| 6 |  |  |  |  |  |  |  |  |
| 7 |  |  |  |  |  |  |  |  |
| 8 |  |  |  |  |  |  |  |  |
| 9 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 10 |  |  |  |  |  |  |  |  |
| 11 |  |  |  |  |  |  |  |  |
| 12 |  |  |  |  |  |  |  |  |
| 13 |  |  |  |  |  |  |  |  |
| 14 |  |  |  |  |  |  |  |  |
| 15 |  |  |  |  |  |  |  |  |

**Arquitectura básica de um SO**

Modelo de von Neumann: mesma memória para os dados e para as instruções.

Modelo de Havard: duas memórias diferentes.

**Endereço:** Número único que identifica cada registo de memória.

**Espaço de endereçamento**: a gama total de endereços que o CPU consegue referênciar, o que depende da dimensão do barramento de endereços. Número de endereços gerados: 2^16 se o barramento for de 16 bits.

Exemplo: se um CPU com um barramento de endereços de 16 bits pode gerar de 0 a 2^16-1 endereços!!!!

Gama: 0x0000 a 0xFFFF

1 Endereço em binário: \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_

No lugar de cada \_ pode ser 0 ou 1, 2 hipóteses para 16 lugares.

**Secção de dados / datapath:** elementos de encaminhamento (multiplexers p.e.), processamento(ALU), armazenação(registos) de informação.

**Unidade de controlo:** coordena os elementos do datapath. Dependendo da arquitectura, pode ser um elemento combinatório ou uma máquina de estados!!!!!

Independentemente da Unidade de controlo, o **o CPU é SEMPRE uma máquina de estados SÍNCRONA!!!!**

**Ciclo base de execução de uma instrução:**

**Instruction fetch**: leitura do Código máquina da instrução residente em memória.

**Instruction decode:** descodificação da instrução pela unidade de controlo.

**Operand fetch:** leitura dos operandos

**Execute:** Execução da operação específica pela instrução.

**Store Result:** armazena o resultado da operação no destino especificado na instrução.

*Next instruction.*

A Cinza azul é a fase fetch e a laranja carregado é a fase executer...

**Codificação de instruções:**

**Arquitectura do Conjunto de Instruções:** ISA (Instruction Set Architecture)

**Instruction set:** coleção de todas as operações/instruções que o processador pode executar

O ISA também é designado por “modelo de programação.”

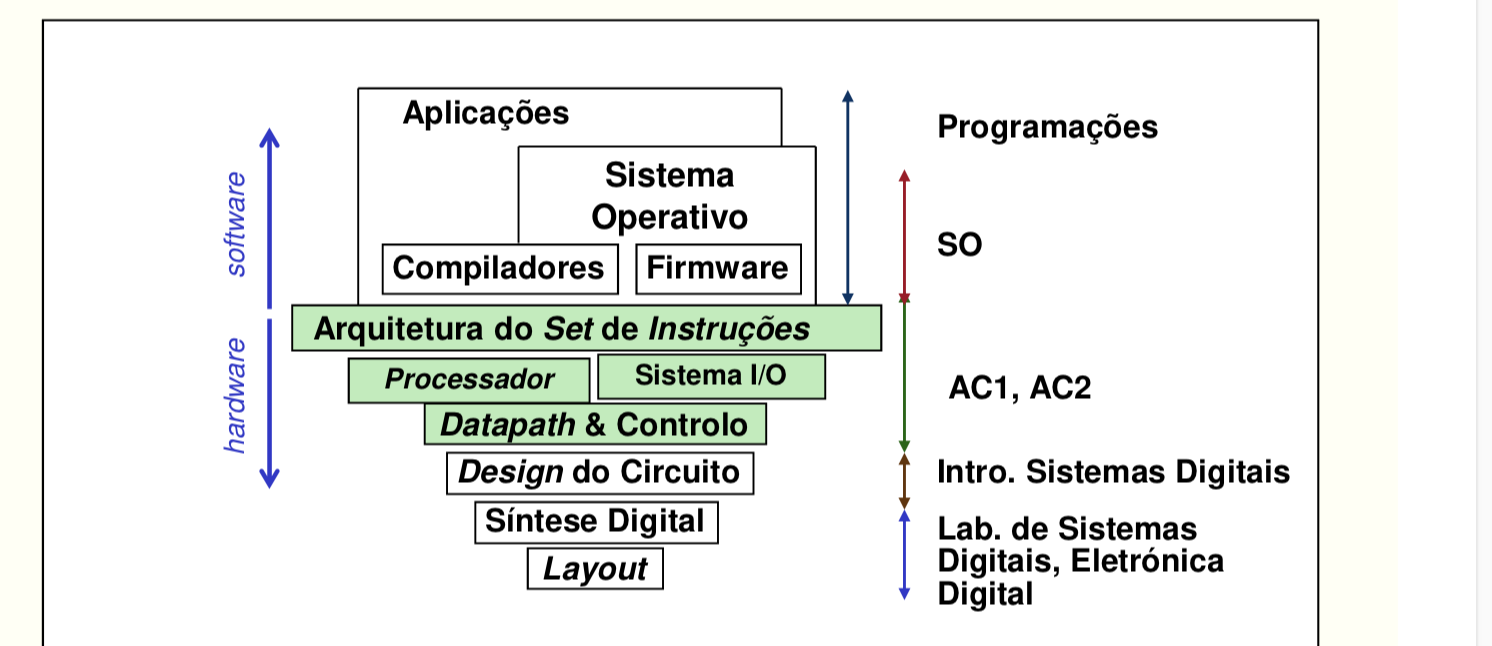
Camada de abstracção que representa a interface entre o hardware e o nível mais básico de software.

Descreve tudo o que o programador necessita de saber para programar correctamente, em linguagem máquina, um determinado processador.

**Exemplos de ISAs:**

* Mips
* ARM (Nintendo, Smartphones, etc...)
* Intel x86 (PCs, MACS...)
* PowerPC
* Cell (PlayStation 3).

Requesitos básicos: fácil de entender, e programar desenvolver compiladores eficientes e implementação simples em hardware.



**Aula 3 – MIPS e ISA**

**Stored program:** Implica que na memória possa residir ao mesmo tempo o código fonte de um programa em C, um editor de texto, um compilador, e o próprio programa resultante da compilação.

**Instruções e classes de instruções:**

É possível considerar que um grupo limitado de classes de instruções possam ser comuns a todas ou à maior parte das arquitecturas, como por exemplo, haver classes de processamento aritmético e lógico, transferência de informação e controlo de fluxo dse execução.

**Princípos de bom design de hardware:**

1. **Mais simples = melhor para usos regulares**
2. **Mais pequeno = mais rápido**
3. **Mais comum = mais rápido**
4. **Bom design = mais compromissos com o cliente.**
5. Todas as instruções são codificadas com o mesmo número de bits, instruções aritméticas operam sempre sobre registos internos e depositam o resultado também num registo interno.

**(3)** quando o operando é uma constante, ela deve fazer parte da instrução (vulgar em mais de 50% das instruções).

**(4)** por exemplo a possibilidade de codificar constantes de maior dimensão mantendo o tamanho das instruçõ~ees.

**Instruções de tamanho variável vs tamanho fixo:**

Instruções de tamanho variável: Codigo mais pequeno, maior flexibilidade, instruction fetc em vários passos.

Instruções de tamanho fixo: instruction fetch e decode mais simples, mais simples de implementar em pipeline.

**Números de registos internos:**

Menos registos:

* Menos hardware
* Acesso mais rápido
* Menos bits de representação de registo.
* Mudança de contexto mais rápida

Mais registo:

* Menos acessos à memória
* Variáveis em registos
* Certos registos podem ter restrições sde utilização

**Localização dos operandos das instruções:**

**Acumulador:** resultado das operações armazenado num registo especial chamado de acumulador.

**Stack:** operandos e resultado armazenados numa pilha de registos.

**Register-memory:** operandos residem nos registos internos do CPU ou em memória.

**Load-Store**: operandos das instruções residem em registos internos mas nunca em memória. (register to register operation).

**MIPS**

* 32 registos de uso geral de 32 bits cada um (1 word == 32bits)
* ISA Baseado em instruções de dimensão fixa 32bits.
* Memória organizada em bytes (8 bits) (byte addressable).
* Espaço de endereçamento (com um barramento) de 32 bits ( 2^32 endereços possíveis == 4GB de memória)
* Barramento de dados externo de 32 bits (????
* Arquitectura load-store

**Codificação das instruções no MIPS**

**Formato R**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| OP (6 bits) | RS (5 bits) | RT (5bits) | RD(5bits) | Shamt(5) | Funct(6) |

**op:** opcode, é sempre zero nas instruções do tipo R

**rs:** endereço do registo que contem o primeiro operando fonte

**rt:** endereço do registo que contem o segundo operando fonte

**td:** endereço do registo onde o resultado vai ser armazenado

**shamt:** shift amount (útil apenas em operações de deslocamento)

**funct:** código da operação a realizar

**EXEMPLO:**

Add $8, $17, $18

Op=0. Rs = $17 rt = $18. Rd = $8 shamt = 0Funct = 100000

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

Código máquina: 000000 10001 10010 01000 00000 100000 ==

== 0000 0010 0011 0010 0100 0000 0010 0000 ==

== 0x02324020

EXEMPLO 2:

sll $10, $16, 4

op null $16 $10 4 funct

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 000000 | 00000 | 10000 | 01010 | 00100 | 100000 |

**Instrução: 0x00000000:**

“Stands for sll r0,r0,0, meaning: Logically shift register 0 zero bits to the left and store the result in register 0”

**Nop: no operation.**

**Instrução move:**

Instrução virtual:

**sTR**Reansferência entre registos:

or $r1 $0 $r2

= move $r1, $r2

**Aula 4 - Branches**

**Instruções de controlo de fluxo de execução (beq, bne, etc...)**

**Branches:** saltos / jumps condicionais

Se a condição for verdadeira, então o valor do program counter (PC) é substituído pelo endereço correspondente do label. E a instrução que é executada de seguida é a que está no endereço alvo.

**Instruções reais:**

* beq
* bne

**Instruções Virtuais:**

* bgt
* blt
* bge
* ble
* li
* la
* move
* ...

**Instrução SLT**

slt $r1, $r2, $r3

set r1 if r2 < r3

* r1 = 1 se r2 < r3 é verdadeiro
* r1 = 0 se r2 < r3 é falso

A utilização das instruções bne, beq, slti e slt em conjunto com o registo $0, **permitem a implementação de todas as condições de comparação entre dois registos.**

**Exemplo:**

A instrução virtual “bge”:

**bge $4, $7, exit**

é decomposta em:

**slt $5, $4, $7,**

**beq $0, $5, exit**

A instrução virtual “bgt”

**bgt $4, $7, exit**

é decomposta em:

**slt $5, $7, $4,**

**bne $5, $0, exit**

Modo de endereçamento: relativo ao PC

**Aula 5 – Acesso à memória**

**Instruções de transferência entre registos e memória externa:**

**Modo de endereçamento:** método (instrução) usado para aceder ao elemento que contém a informação.

**Endereçamento do tipo registo:** os endereços dos registos internos envolvidos são diretamente especificados na própria instrução.

**Como é feito o acesso a informação residente na memória externa?**

Como um endereço de memória ocupa 32 bits, isso ocupava a totalidade do código máquina da instrução, então a instrução indica um registo que contém o endereço de memória a aceder. Endereçamento indireto por registo.

**Leitura da memória (lw):**

Transfere uma word de 32 bits da memória para um registo interno.

lw $r\_destino, const($r\_endereço)

**const:** constante de 16 bits em complemento para 2

**Escrita na memória (sw):**

Transfere uma word de 32 bits do registo para a memória internal.

sw $r\_origem, const($r\_endereço\_destino)

**const:** constante de 16 bits em complemento para 2

**Codificação das instruções do acesso à memoria no MIPS:**

**Formato I (immediate)**

|  |  |  |  |
| --- | --- | --- | --- |
| Opcode | rs | rt | offset |

Offset: 16 bits em complemento para 2: -32768, 32767

EXEMPLO:  
  
A instrução:

lw $8, 20($5)

“load word in $5 to $8”

Ou seja, o rs vai ser o $5, e o rt vai ser o $8:

|  |  |  |  |
| --- | --- | --- | --- |
| 100011 | 00101 ($5) | 01000($8) | 0000 0000 0001 0100 (20) |

No store word sw também é igual.

**É nestas instruções (lw e sw) que vemos a arquitectura load-store.**

**NOTA:** o registo $5 é equivalentea dizer que o endereço de memória dele é 0x10010006 na memory stack. ($5 == 0x10010006).

**Restrições no acesso aos endereços do MIPS:**

Externamente, só podemos aceder aos endereços do MIPS memória em múltiplos de 4, porque só tem 30 bits dos 32 disponíveis:

**\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_**

\_\_\_\_1 byte\_\_\_\_ \_\_\_\_1 byte\_\_\_\_ \_\_\_\_1 byte\_\_\_\_ \_\_\_\_1 byte\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 1 word \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Os dois tracinhos a vermelho estão indisponíveis e são as combinações dnos números 00 -> 0, 01 -> 1, 10 -> 2, 11 -> 3

**Logo o acesso a words só é possível em múltiplos de 4.**

Se numa instrução de leitura ou escrita de uma word for especificado um **endereço não múltiplo de 4, o MIPS gera uma excepção. Contornamos o problema com a directiva “align n”** que vai alinhar num endereço com valor múltiplo de 2^n.

No entanto no MIPS a memória é byte addressable, ou seja**, conseguimos aceder a cada byte fazendo a leitura e escrita de um só byte (lb ou sb).**

Como?

O MIPS gera o endereço múltiplo de 4 (uma word) que inclui o endereço pretendido.

No caso de leitura:

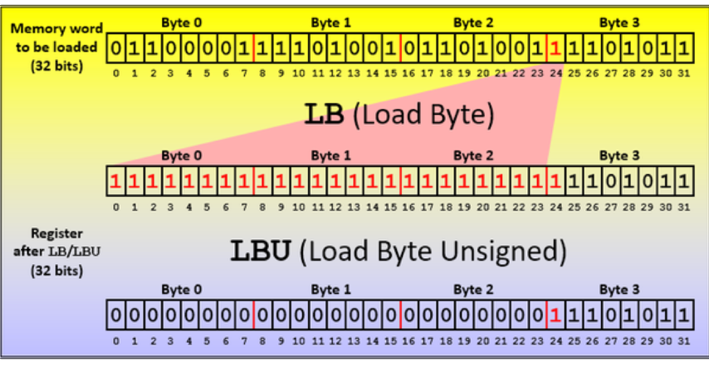
* executa uma instrução de leitura de 1 word desse endereço múltiplo de 4
* dos 32 bits lids. retira os 8 correspondentes.

No caso de escrita:

* executa uma instrução de leitura de 1 word desse endereço mult. De 4.
* de entre os 32 bits substitui os 8 bits que são para modificar / escrever
* escreve a word nesse endereço

Isto é conhecido como “read-modify-write”

**LUBU: LOAD BYTE UNSIGNED:**



Mete o valor do registo de destino tudo a zero 24 bits mais signiticativos a 0:

**lbu $5, -4 ($2)** # o offset continua a ser comoplemento para dois. O valor byte a ser recebido pelo $5 é que vai ser positivo.

**O MIPS é do tipo Little endian**

**Ou seja, a word: 0x012387A5 ( 0x 01 23 87 A5) é guandada como:**

|  |  |
| --- | --- |
| **Endereço** | **Memória** |
|  | **...** |
| 0x10010003 | 01 |
| 0x10010002 | 23 |
| 0x10010001 | 87 |
| 0x10010000 | A5 |
|  | ... |
|  |  |

**Aula 6 – Directivas e ponteiros em C.**

**Diretivas do Assembler**

* Não são instruções da linguagem do assembly.
* Servem para reservar e inicializar espaço de memória para variáveis
* Controlar os endereços reservados para variáveis de memória
* Especificar endereços de colocação de código e dados na memória
* Definir valores

**.DATA** # 0x10010000

Significa que as directivas / dados que ficam na memória podem ser escritas

**.ASCIIZ str**

Reserva espaço e armazena uma string em sucessivas posições de memória e acrescenta o terminador nulo ‘\0’

Exemplo:

.asciiz “ola mundo!”

**.SPACE n**

Reserva n posições consecutivas de memória sem inicialização.

EXEMPLO:

// para reservar um array de 8 inteiros: 8\*4=32

.space 32

// para reservar um array de caracteres: 8\*1= 8

.space 8

**.BYTE b1, b2, b3, b4**

Reserva espaço em bytes e armazena os bytes que inserimos.

EXEMPLO:

.byte ‘a’, ‘b’, ‘c’, ‘d’

**.WORD w1, w2, w3, w4**

Reserva espaço e armazena as words na memória que inserimos

EXEMPLO:

.word 1, 2, 3, 4

**.ALIGN n**

Alinha o próximo item num endereço múltiplo de 2^n

EXEMPLO:

.Asciiz “ob”

.align 2

.word 0x1234

|  |  |
| --- | --- |
| 0x10010007 | 0x00 |
| 0x10010006 | 0x00 |
| 0x10010005 | 0x12 |
| 0x10010004 (Multiplo de 4) | 0x34 |
| 0x10010003 |  |
| 0x10010002 |  |
| 0x10010001 | b |
| 0x10010000 | o |
|  |  |

(neste caso não precisamos de meter align porque a word já vai meter automaticamente num endereço múltiplo de 4., mas num byte já precisamos de meter).

**.EQV symbol, val**

Substitui o sumbol pelo val.

Exemplo:

.eqv str, **2**

**.TEXT** #0x00400000

Significa que começa a parte do código. A Parte das assembly language instructions.

**.globl main**

Nome do simbolo global aka simbolo que vai ser visível no código de outros ficheiros. Neste caso a função main.

**Ponteiros**

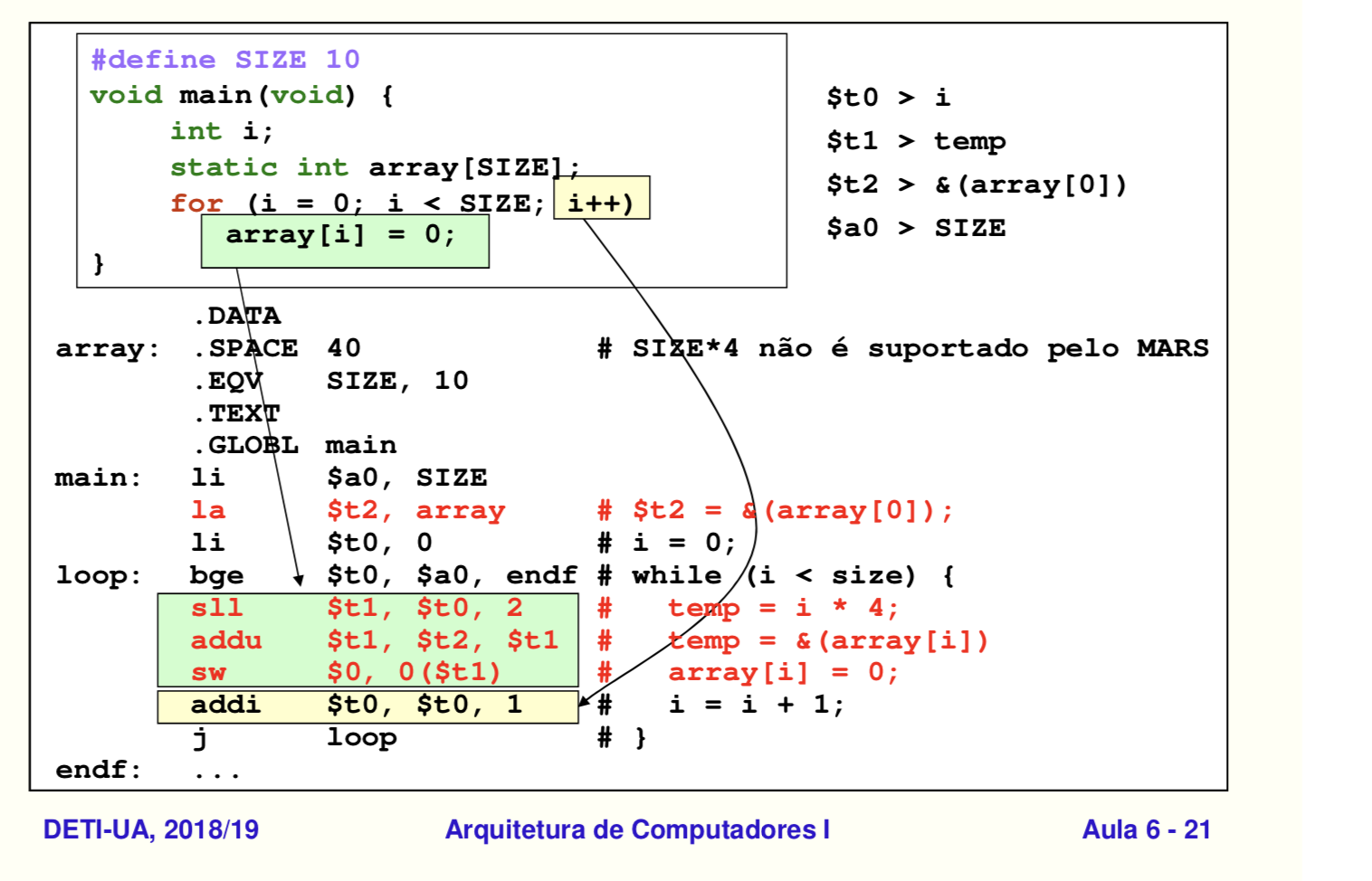
int \*px;

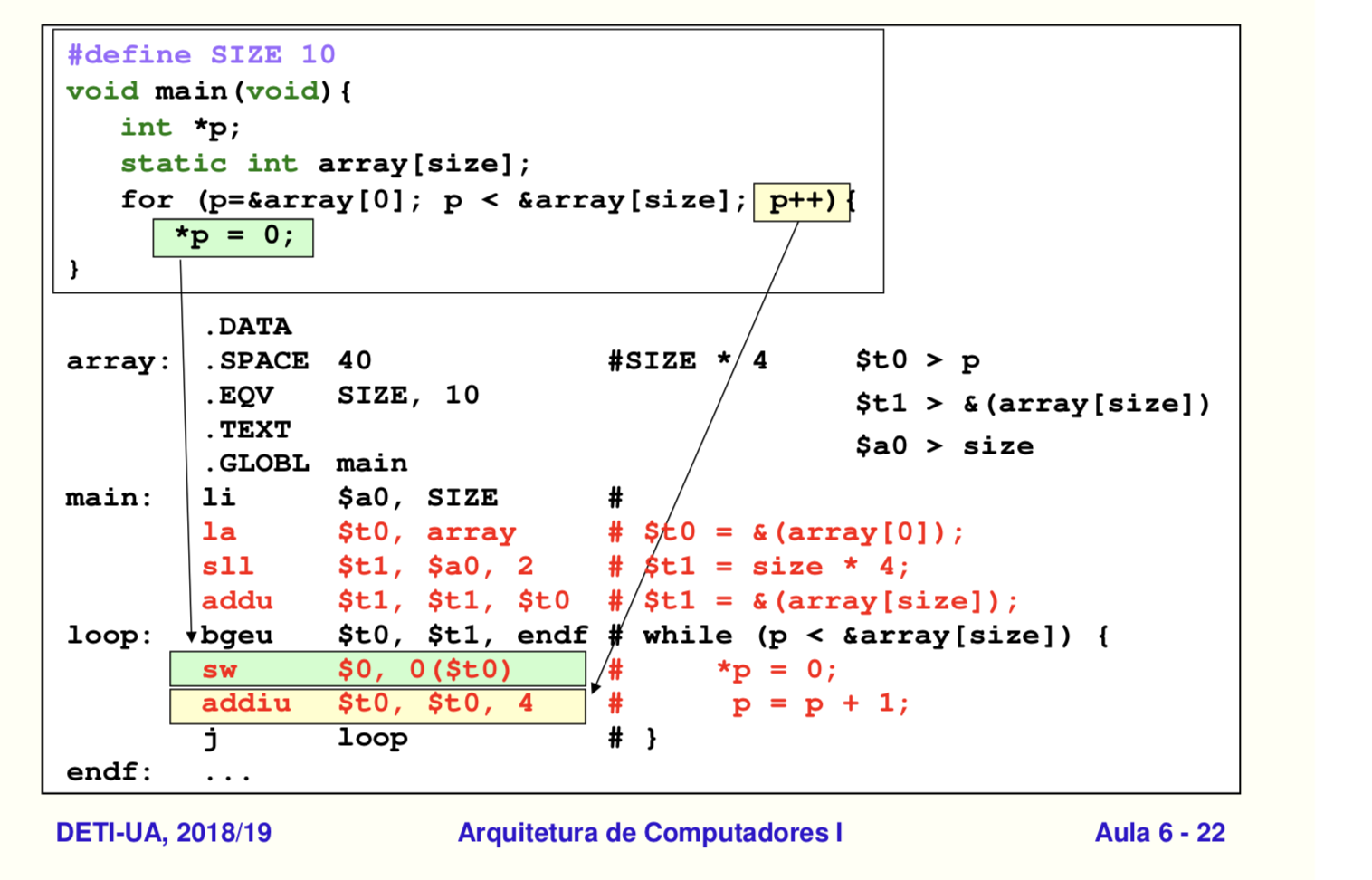
\*px = \*px +1; // igual a ter x = x + 1

\*px++; // igual a ter x = x + 1 (devolve o valor antigo do x e mas avança para próxima posição / endereço)

(\*px)++; // igual a ter x = x + 1

\*(px)++; // igual a ter x = x + 1 (devolve o valor antigo do x e mas avança para próxima posição / endereço)





**Aula 7**

**Codificação de Branches no MIPS – RELATIVO AO PC.**

**É do formato do tipo I**

Exemplo:

Bne $8, $21, Exit

|  |  |  |  |
| --- | --- | --- | --- |
| 5 (Op) | 8 (rs) | 21 (rt) | Exit (offset) |

No offset, o exit não é o valor do endereço porque o offset ocupa 16 bits. Se fosse, a combinação máxima de endereços seria 2^16, em que só teríamos 64kb de armazenamento na memoria para o programa. Isso é muito pouco.

O offset para o exit, neste caso **é um valor para ser somado ao valor de endereço atual, pelo PC.**

**PC = PC\_ATUAL + offset**

Assim já poderia ser de 2^32 = 4GB RAM.

**No entanto...** a maior parte das instruções de salto condicional realizam-se na vizinhança da própria instrução (normalmente entre +/- 16 instruções) e com 16 bits é possível endereçar 2^16 endereços distintos **(64k endereços distintos).**

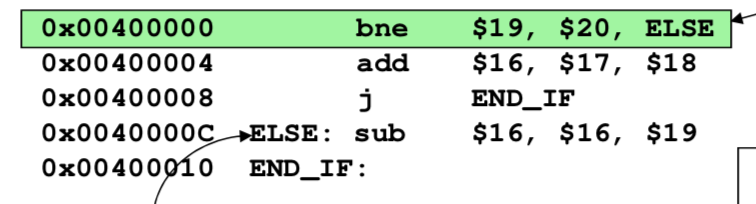
No MIPS todas as instruções são armazenadas em endereços múltiplos de 4:

0x00400000, 0x00400004, 0x00400008, etc...)

Logo o valor de offset pode ser numa gama de -32k até 32k porque está em complemento para dois.

O valor PC\_ATUAL é o valor da instrução seguinte a ser executada.

Exemplo:



Queremos saltar para a instrução else.

* O valor do PC é 0x00400004
* Queremos ir para 0x0040000C
* Logo: 0x0040000C – 0x00400004 = 8
* Como cada instrução ocupa 4 bytes: 8/4 = 2
* Offset = 0x0002

Código máquina: op = 5, rs = 19, rt = 20, offset = 2

000101 10011 10100 0000000000000010

**Interpretação de uma instrução de branch pelo CPU:**

Exemplo:

* 0x1509FFFD (está no endereço 0x00400030)
* 0001 0101 0000 1001 1111 1111 1111 1101
* 000101 01000 01001 1111111111111101
* bne $8 $9 -3 (complemento para 2)

Endereço alvo = PC + offset\*4 = 0x00400030 + (-3\*4) = 0x00400030 – 0x0C = 0x00400028

**Codificação de saltos incondicionais de MIPS**

**Instrução do tipo J – PSEUDO-DIRECTO.**

|  |  |
| --- | --- |
| Op code (6 bits) | R (26 bits) |

**As instruções começam sempre em múltiplos de 4**, logo a instrução começa sempre em “00”, logo a **parte R não tem esses dois “00” implícitos.**

A máquina ao ler a instrução do tipo R, dá um shift de 2 para meter esses dois “00” finais passando de 26 para 28 bits:

Onde a máquina vai buscar os 4 bits finais (da esquerda) para completar os 32 bits do endereço, da instrução?

**Ao program counter, aos 4 primeiros bits do endereço do program counter.**

**Então fica:**

**\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_**

4 primeiros bits do endereço do PC

valor R

00 derivado ao shift pois derivado às instruções serem múltiplos de 4

**Instrução JR – INDIRECTO POR REGISTO.**

É do tipo R

Jump On Register

jr $ra # salta para o endereço que está armazenado no registo $ra

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Op (000000) | Rs | Rt (0) | Rd (0) | shmt (0) | Funct(8) |

Funct é a função que jump on register

Rs é onde contem o registo com o endereço.

**Chama-se a isto salto de endereçamento indirecto por registo.**

**Modos de endereçamento, MIPS:**

**Endereçamento tipo registo:** instruções aritméticas e lógicas

**Endereçamento indirecto por registo com deslocamento:** acesso á memória

**Endereçamento relativo ao PC:** instruções de salto condicional

**Endereçamento indirecto por registo:** instruções de salto incondicional através do jr

**Endereçamento directo ou pseudo directo.:** salto incondicional j

**Endereçamento imediato:**

Um número muito grande de instruções são de operações aritméticas com constantes num dos seus operandos, onde são armazenadas na própria instrução (no caso dos addi, subi, andi, ori, slti, etc...)

É do formato I, logo os immediates vão ter 16 bits, 2^16 combinações. Grandes o suficientes para guardar as mais utilizadas constantes

Gama de representações:

Casos gerais (addi, stli, etc) é de 32k a -32k, complemento para 2.

Instruções lógicas: 65k

**Instrução lui (load upper immediate)**

Imaginemos, temos um número 0x1234:

Se fizermos um **ori** com zero:

ori $12, $0, 0x1234 (16 bits)

o registo $12 fica com o valor x00001234 (32 bits)

no entanto, se fizermos com o **lui**:

lui $12, 0x1234 (16 bits)

o registo **$12** fica com o **valor 0x12340000 (32 bits).**

Ou seja, a instrução lui é uma maneira de facilitar a manipulação de inteiros com mais de 16 bits, passando um valor de 16 bits para 32 bits em que o valor passa para os 16 bits mais significativos enquanto o valor dos 16 bits menos significativos é zero.

**Casos onde é usado a lui:**

**Na instrução load address:**

EXEMPLO:

temos um array declarado como MyData em que o array começa no segmento de dados em 0x10010034.

Instrução:

la $16, MyData

é traduzida como:

lui $1, 0x1001 #0x10010000

ori $16, $1, 0x0034 #0x10010000 | 0x00000034

$16 = 0x10010034

Notas:

* O registo $1 é o $at que é reservado pelo assembler para permitir a tradução de instruções virtuais para instruções nativas.
* A instrução li é decomposta de igual forma à la.

**Aula 8 – Subrotinas**

**Instrução JAL**

Jump and link

Exemplo:

Endereço do jal func1 = 0x0040002c

Endereço da função func1 = 0x00400070

Endereço da instrução seguinte a jal func1= 0x00400030

código:

**jal func1** # PC = PC(inst. seguinte) + Const(jal func1 -> funct1)

\*salta- para o endereço da função func1\*

\*está na função func1\*

\*ela chega ao fim e encontra o segunte trecho de código\*

**jr $ra**

\*o $ra guarda-te o endereço do PC antes de ser incrementado para a função func1, ou seja o pc era o endereço posterior ao endereço do “jal func1” que era 0x00400030\*

\* código remota a função original do jal, mas na instrução seguinte.\*

Para mais que duas funções não terminais, **tem que se guardar os valores do $ra e das variáveis numa stack.**

**Podemos passar argumentos antes do jal para os registos $a0 a $a3**

**O valor a receber é no $v0**

**Se o valor for de 64 bits os 32 mais significativos são recebidos no $v1 e os restantes nos revistos $v0.**

**Estratégias para salvaguarda de registos:**

**Caller-Saved:**

A função intermédia que chama uma outra função é ela que fica encarregue de guardar os registos antes de evocar a sub-rotina e depois de repor também.

**Calle-Saved:**

É a sub-rotina chamada quem guarda os valores dos registos e depois repõe antes de regressar ao programa chamador.

**Um programa que chama outra função (intermédio) deverá guardar o seu conteúdo antes de evocar uma sub-rotina antes que esta altere algum registo dela.**

Caso alguma ronita queira utilizar algum valor do $sn, deverá passar esse valor para um registo temporário ($t(n) por exemplo) ou para a memória externa (0($tn)).

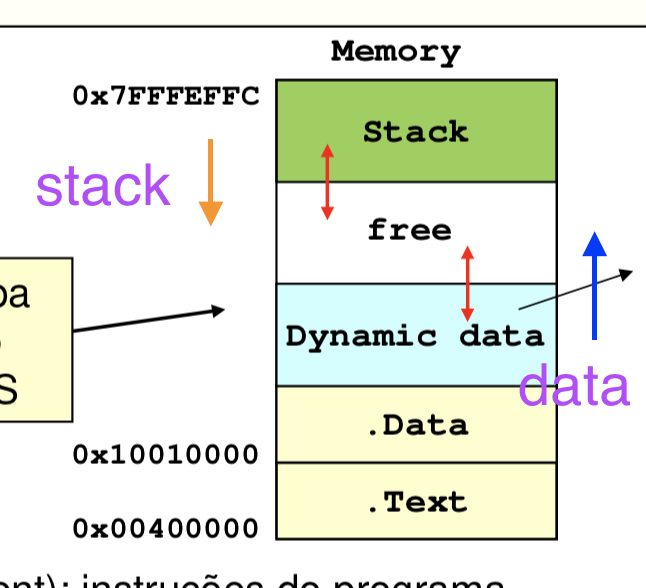
Aula 9 – Stacks

**Espaço de armazenamento temporário da memória:**

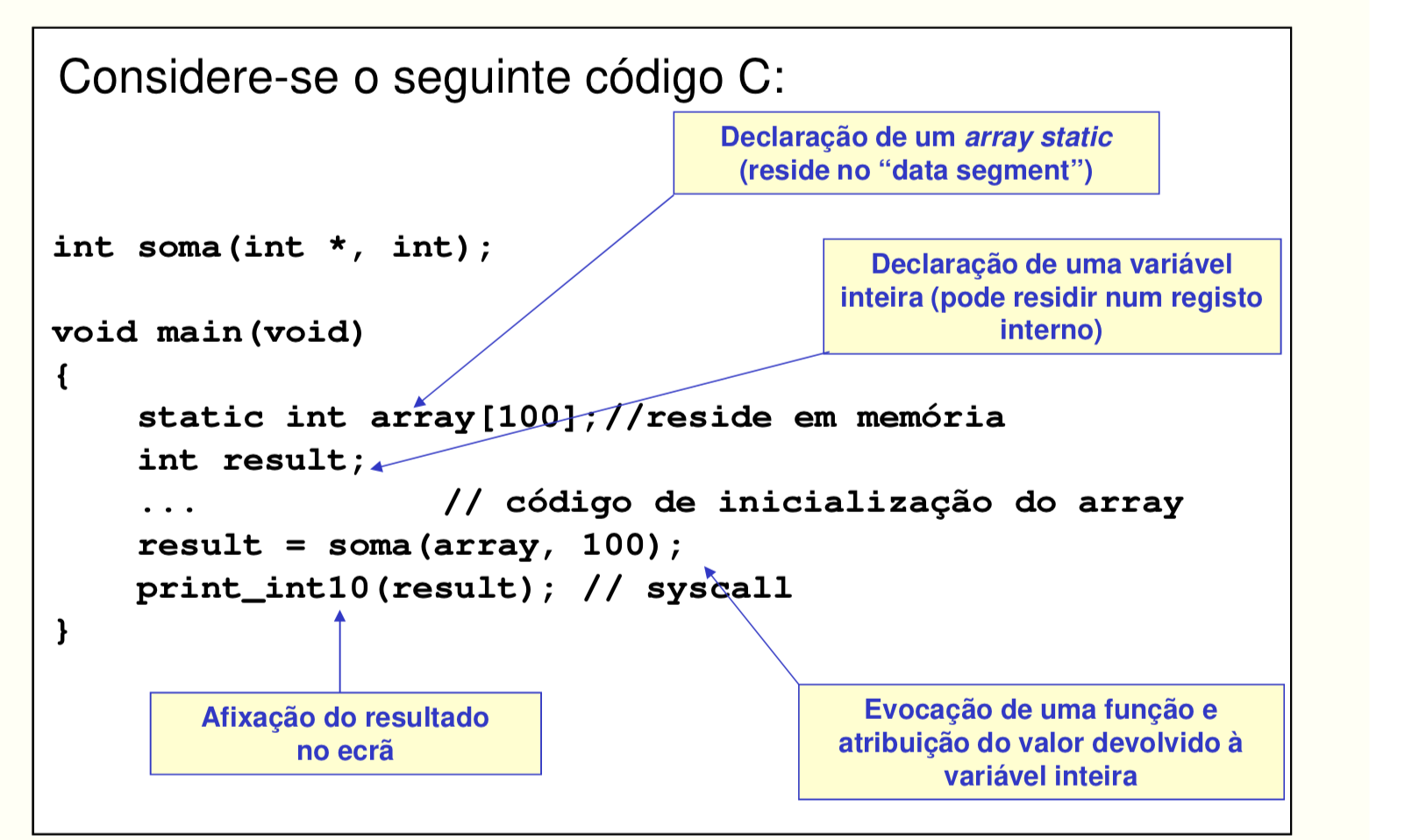
* Usa uma estratégia de LIFO ( Last in first out)
* Usa um registo SP (Stack Pointer) que aponta para o topo da stack (contém o endereço) e que permite fazer operações de push e de pop.

**Atualização do Stack pointer:**

A stack utiliza as posições de memória mais elevadas, então ao inserir-mos um elemento, a stack decrementa para ir para o endereço seguinte e vice-versa.



A dinamic data ta também é conhecida como heap.



**Recursividade:**

