# “Arquitetura e Organização de Computadores I – Aula_02”

Prof. Dr. Emerson Carlos Pedrino

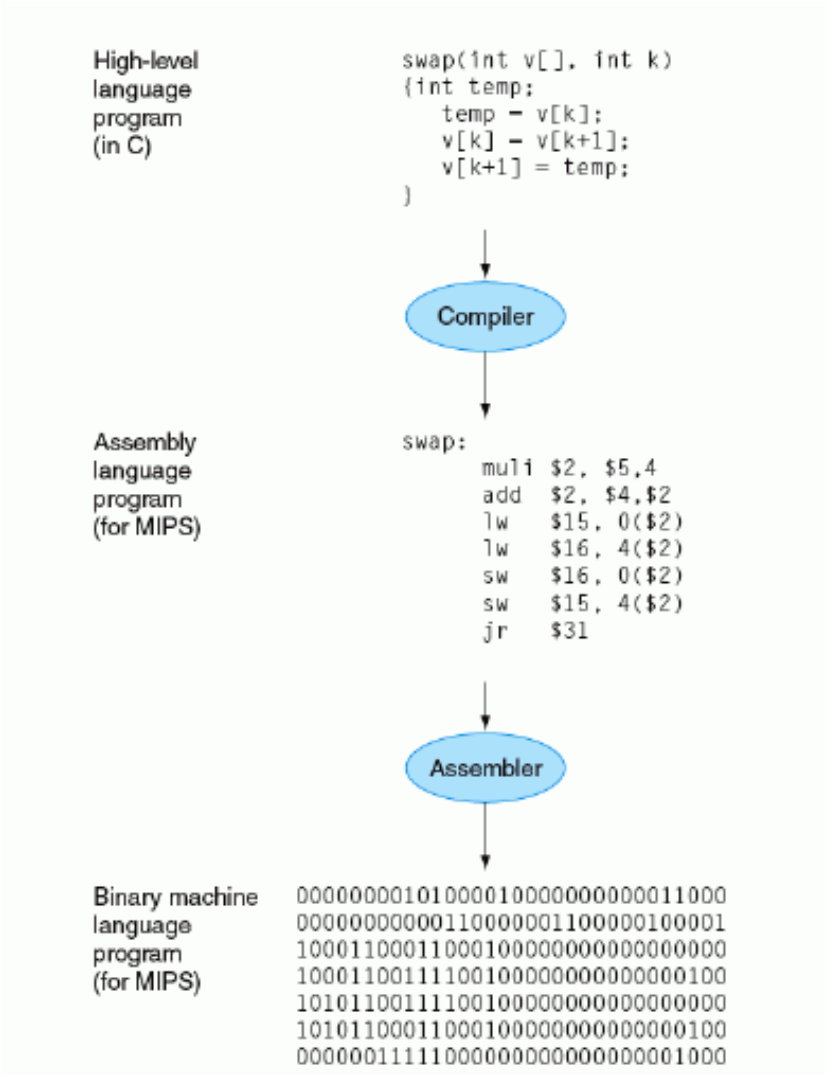
DC/UFSCar

São Carlos

# Instruções: A linguagem de Máquina

- Controle do hardware -> Através de sua linguagem.
- Palavras da linguagem -> Instruções.
- Vocabulário -> Conjunto de instruções.
- Linguagens de diferentes computadores -> Muito semelhantes.
- Objetivos de projeto: Encontrar uma linguagem que facilite o projeto do hardware, do compilador e maximize o desempenho e minimize o custo.

# Abordagem Top-Down




↗
- Linguagem mais natural;
- Maior produtividade;
- Independem de plataforma.

⟶ 32 bits

# MIPS

- Criado na década de 80 – John L. Hennessy – Stanford.
- Após 1984: MIPS Computer Systems – Desenvolveu um dos primeiros Microprocessadores RISC comerciais.
- 1991: Comprada pela Silicon Graphics – MIPS Technologies.
- 1998: Fica independente e foca em Microprocessadores para sistemas embutidos.
- 2004: Aproximadamente 300 milhões de microprocessadores MIPS foram entregues em dispositivos variando desde *videogames* e computadores *palmtop* até impressoras laser e *switches* de rede.
- Encontrado em produtos da ATI Technologies, Broadcom, Cisco, NEC, Nintendo, Silicon Graphics, Sony, Texas Instruments e Toshiba.

# Operações do *Hardware* do Computador

- Exemplo de notação em *assembly* do MIPS:
  - Exemplo de instrução aritmética:
    - add a, b, c obs.: a soma de *b*+*c* será armazenada em *a* após a execução da instrução *add*.
    - notação rígida: cada instrução aritmética no MIPS realiza apenas uma operação e deve ter exatamente 3 variáveis.
    - obs.: o *hardware* se torna mais complexo para um número variável de operandos.

## *Exercício: escrever um programa em *Assembly* do MIPS para efetuar a operação dada a seguir:

- a=b+c+d+e
  - sol.:
    - add a,b,c  # a <- b+c+d+e.
    - add a,a,d
    - add a,a,e

# Instrução de Subtração

- Exemplo: sub d,a,e  # d <- a – e.
- *Exercício (Pesquisar): como seria gerado o código *asm* do MIPS por um dado compilador para avaliar a seguinte expressão:
  - f=(g+h)-(i+j);

# Operandos do Hardware: Registradores

- Registrador MIPS: 32 bits (word).
- Há 32 registradores na arquitetura estudada.
- Menor número de registradores -> Sistema mais rápido.
  - Uma quantidade muito grande de registradores pode aumentar o ciclo de *clk* simplesmente porque os sinais levam mais tempo quando precisam atravessar uma distância maior.

# Compilando uma atribuição em C

- Compilador -> Associa variáveis do programa aos registradores $\$s_n$. $\$t_n$: temporários.
- Exemplo: f=(g+h)-(i+j).
- Código MIPS compilado:
  - add $t0,$s1,$s2
  - add $t1,$s3,$s4
  - sub $s0,$t0,$t1  # f=$t0-$t1

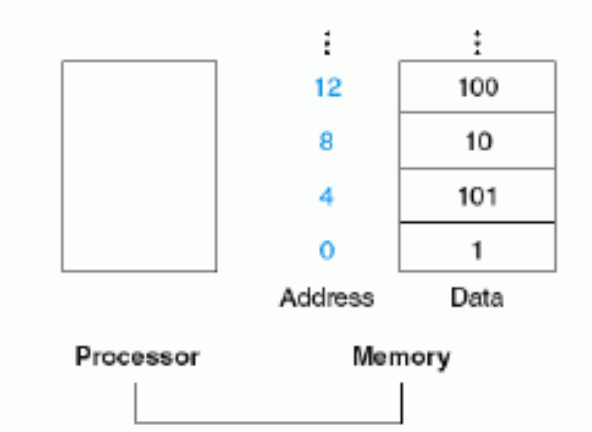
# Operandos de Memória

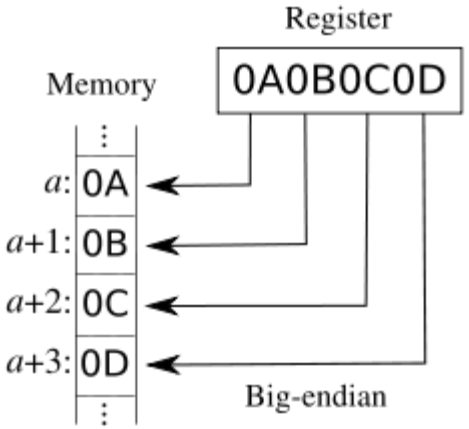
- *Load* -> Instrução de transferência de dados da memória para o registrador. Em MIPS: lw (*load word*).
- Exemplo: g=h+A[8], após a compilação:
  - lw $t0,8($s3)  # $s3 -> Endereço base de A. $t0 <- A[8]. Constante 8=*offset*.
  - add $s1,$s2,$t0 # resultado armazenado em $s1=g.

Obs.: g: s1 e h: s2.

# Endereçamento de *bytes* do MIPS

- Endereço de uma *word* -> Combina endereços de 4 bytes. No exemplo o endereço em bytes da 3ª *word* é 8.
- Restrição de alinhamento -> *words* começam com endereços múltiplos de 4. (+ eficiente).
- MIPS -> *Big Endian.*



# Instrução *Store*

- *Store* -> Copia dados de um registrador para a memória.
- No MIPS: *sw* -> *store word*.
- Exemplo: A[12]=h+A[8].
  - Obs.: variável h associada a $s2; end. base do vetor associado a $s3.
  - lw $t0,32($s3) # $t0 <- A[8]
  - add $t0,$s2,$t0 # $t0 <- h+A[8]
  - sw $t0,48($s3) # A[12] <- h+A[8]

# *Spilling registers*

- Processo de colocar variáveis menos utilizadas na memória pelo compilador.
  - Muitos programas possuem mais variáveis do que os computadores possuem registradores.

# Observações

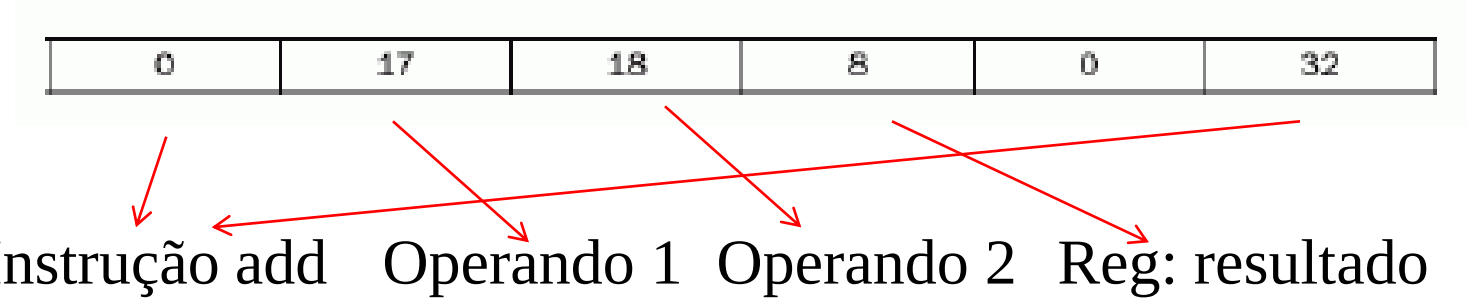
- Acessos a registradores são mais rápidos que acessos à memória.
  - Uma instrução aritmética MIPS pode ler 2 registradores, atuar sobre eles e escrever o resultado. Uma instrução de transferência de dados MIPS só lê um operando ou escreve um operando, sem atuar sobre ele.

# Operandos imediatos

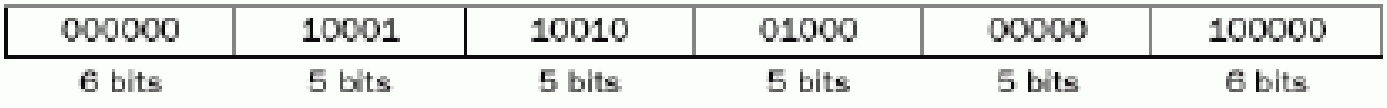
- addi -> *add imediate*.
- Exemplo: addi $s3,$s3,4
- # $s3<-$s3+4.
- Obs.: Operandos constantes ocorrem com frequência e incluindo estes nas instruções aritméticas, estas serão mais rápidas do que se as constantes fossem lidas da memória.

# Representando Instruções no Computador

- Exemplo: add $t0,$s1,$s2
- Obs.: $s0 - $s7 -> registradores: 16 – 23; $t0 - $t7: 8 – 15.
- Em linguagem de máquina (decimal):



# Formato de instrução (32 bits)



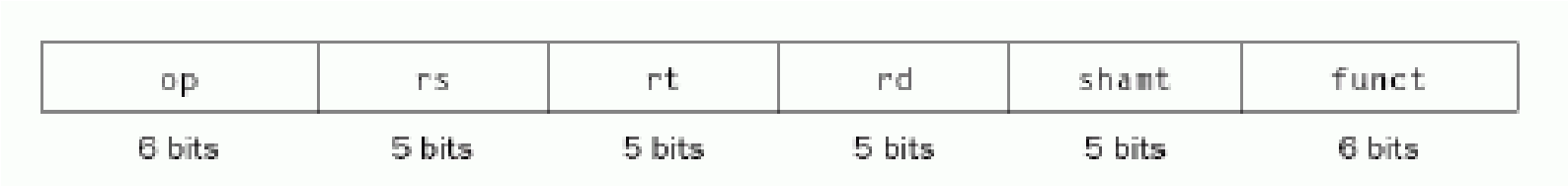

↑ ...

Campos em números binários

# Campos do MIPS

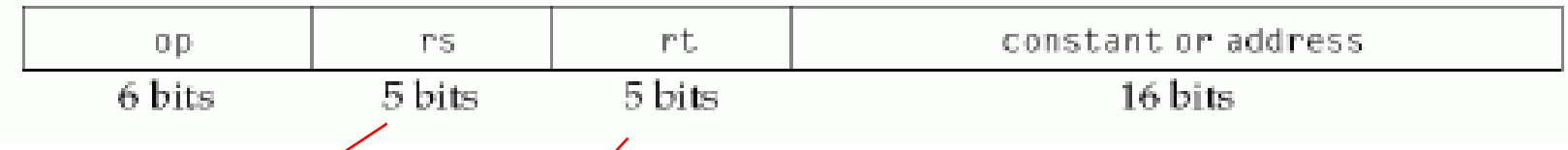
- op: *opcode*.
- rs: registrador do primeiro operando de origem.
- rt: registrador do segundo operando de origem.
- rd: registrador do operando de destino.
- *shamt*: *shift* amount.
- *funct*: função. Código de função. Variante da operação em *op*.

# Exemplo



# Formatos (diferenciados por *op*)

- Formato R (exemplo anterior)
- Formato I (instruções imediatas e de transferência de dados).




End_Reg_Base

Reg_Destino

Pode carregar qualquer *word* dentro de uma região de +- 2^15 bytes relativa a *rs*.

# *Exercício: mostre a representação de lw $t0,32($s3) no formato I. Ends. (dec): op=35, s3=19 e t0=8.

- Sol.: # $t0 <- A[8]
  - $s3 =rs=19.
  - $t0=rt=8.
  - End=32.

| | | | |
|---|---|---|---|
| 35 | 19 | 8 | 32 |

# Codificação de Instruções MIPS

| Instruction | Format | op | rs | rt | rd | shamt | funct | address |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | reg | reg | reg | 0 | $32_{ten}$ | n.a. |
| sub (subtract) | R | 0 | reg | reg | reg | 0 | $34_{ten}$ | n.a. |
| add immediate | I | $8_{ten}$ | reg | reg | n.a. | n.a. | n.a. | constant |
| lw (load word) | I | $35_{ten}$ | reg | reg | n.a. | n.a. | n.a. | address |
| sw (store word) | I | $43_{ten}$ | reg | reg | n.a. | n.a. | n.a. | address |

Obs.: reg -> registrador de 0 a 31; n.a. -> não se aplica; *address* -> endereço de 16 bits.

# *Exercício

- Dados: $t1: end. base de A; $s2=h; A[300]=h+A[300] -> compilada em:
  - lw $t0,1200($t1)  # $t0<-A[300]
  - add $t0, $s2,$t0  # $t0<-h+A[300]
  - sw  $t0,1200($t1)  # A[300]<-h+A[300],
  - Qual é o código em linguagem de máquina MIPS para as 3 instruções anteriores?

# Solução

| op | rs | rt | rd | address/ shamt | funct |
|---|---|---|---|---|---|
| 35 | 9 | 8 | 1200 | | |
| 0 | 18 | 8 | 8 | 0 | 32 |
| 43 | 9 | 8 | 1200 | | |

| | | | | | |
|---|---|---|---|---|---|
| 100011 | 01001 | 01000 | 0000 0100 1011 0000 | | |
| 000000 | 10010 | 01000 | 01000 | 00000 | 100000 |
| 101011 | 01001 | 01000 | 0000 0100 1011 0000 | | |

# Resumo da Arquitetura Vista até Aqui.

| Name | Example | Comments |
|---|---|---|
| 32 registers | $s0, $s1, . . . , $s7<br>$t0, $t1, . . . , $t7 | Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16–23 and $t0-$t7 map to 8–15. |
| $2^{30}$ memory words | Memory[0],<br>Memory[4], . . . ,<br>Memory[4294967292] | Accessed only by data transfer instructions in MIPS. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers. |

**MIPS assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add $s1,$s2,$s3 | $s1 = $s2 + $s3 | Three operands; data in registers |
| | subtract | sub $s1,$s2,$s3 | $s1 = $s2 - $s3 | Three operands; data in registers |
| Data transfer | load word | lw $s1,100($s2) | $s1 = Memory[$s2 + 100] | Data from memory to register |
| | store word | sw $s1,100($s2) | Memory[$s2 + 100] = $s1 | Data from register to memory |

**MIPS machine language**

| Name | Format | Example | | | | | | Comments |
|---|---|---|---|---|---|---|---|---|
| add | R | 0 | 18 | 19 | 17 | 0 | 32 | add $s1,$s2,$s3 |
| sub | R | 0 | 18 | 19 | 17 | 0 | 34 | sub $s1,$s2,$s3 |
| addi | I | 8 | 18 | 17 | 100 | | | addi $s1,$s2,100 |
| lw | I | 35 | 18 | 17 | 100 | | | lw $s1,100($s2) |
| sw | I | 43 | 18 | 17 | 100 | | | sw $s1,100($s2) |
| Field size | | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits | All MIPS instructions 32 bits |
| R-format | R | op | rs | rt | rd | shamt | funct | Arithmetic instruction format |
| I-format | I | op | rs | rt | address | | | Data transfer format |

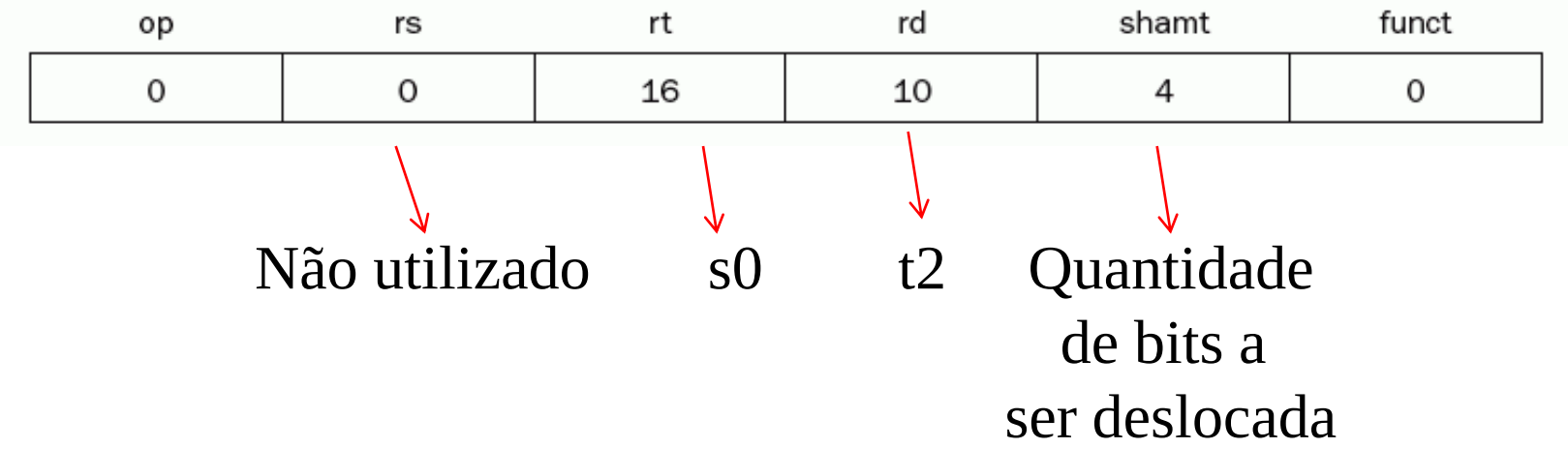
# Operações Lógicas

Geralmente é útil atuar sobre campos de bits dentro de uma *word* ou até mesmo sobre bits individuais. Examinar os caracteres dentro de uma *word*, cada um dos quais armazenados como 8 bits, é um exemplo dessa operação.

| Logical operations | C operators | Java operators | MIPS instructions |
|---|---|---|---|
| Shift left | << | << | sll |
| Shift right | >> | >>> | srl |
| Bit-by-bit AND | & | & | and, andi |
| Bit-by-bit OR | \| | \| | or, ori |
| Bit-by-bit NOT | ~ | ~ | nor |

# sll e slr (*shift left logical* e *shift right logical*)

- Exemplo: sll $t2,$s0,4  #$t2<-$s0<<4 bits.
- Em linguagem de máquina:



# *Exercício

- Se $s0=9, mostre o novo valor do mesmo após o deslocamento de 4 bits à esquerda? Qual é o benefício oferecido por esta operação?
  - Sol.: 144. Multiplicação por $2^i$.

# Resumo Até Aqui

| Name | Example | Comments |
|---|---|---|
| 32 registers | $s0, $s1, . . . , $s7<br>$t0, $t1, . . . , $t7 | Fast locations for data. In MIPS, data must be in registers to perform arithmetic. Registers $s0-$s7 map to 16–23 and $t0-$t7 map to 8–15. |
| $2^{30}$ memory words | Memory[0],<br>Memory[4], . . . ,<br>Memory[4294967292] | Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers. |

**MIPS assembly language**

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | add $s1,$s2,$s3 | $s1 = $s2 + $s3 | Three operands; overflow detected |
| | subtract | sub $s1,$s2,$s3 | $s1 = $s2 - $s3 | Three operands; overflow detected |
| | add immediate | addi $s1,$s2,100 | $s1 = $s2 + 100 | + constant; overflow detected |
| Logical | and | and $s1,$s2,$s3 | $s1 = $s2 & $s3 | Three reg. operands; bit-by-bit AND |
| | or | or $s1,$s2,$s3 | $s1 = $s2 \| $s3 | Three reg. operands; bit-by-bit OR |
| | nor | nor $s1,$s2,$s3 | $s1 = ~ ($s2 \| $s3) | Three reg. operands; bit-by-bit NOR |
| | and immediate | andi $s1,$s2,100 | $s1 = $s2 & 100 | Bit-by-bit AND reg with constant |
| | or immediate | ori $s1,$s2,100 | $s1 = $s2 \| 100 | Bit-by-bit OR reg with constant |
| | shift left logical | sll $s1,$s2,10 | $s1 = $s2 << 10 | Shift left by constant |
| | shift right logical | srl $$s1,$s2,10 | $s1 = $s2 >> 10 | Shift right by constant |
| Data transfer | load word | lw $s1,100($s2) | $s1 = Memory[$s2 + 100] | Word from memory to register |
| | store word | sw $s1,100($s2) | Memory[$s2 + 100] = $s1 | Word from register to memory |

Obs.: verificar as instruções *and*, *or* e *nor*.

# Instruções para Tomada de Decisões (desvios condicionais)

- beq -> *branch if equal.*
- Exemplo: beq reg1,reg2,L1  # se reg1=reg2 vá para L1.
- bne -> *branch if not equal.*
- Exemplo: bne reg1,reg2,L1  # se reg1 diferente de reg2 vá para L1.
- j (*jump*) -> desvio incondicional.

# Compilando um *Loop While* em C

- Exemplo: while (save[i]==k) i+=1;
- Dados: i=$s3; k=$s5; end. base de save=$s6.
- Código *assembly* gerado:

```
loop: sll $t1,$s3,2          #$t1<-4*I
add $t1,$t1,$s6    #$t1=end_save[i]
lw $t0,0($t1)                #$t0=save[i]
bne $t0,$s5,Exit   #Exit se save[i]<>k
addi $s3,$s3,1               #i=i+1
j loop
Exit:
```

# Instruções slt e slti

- slt -> *set on less than*.
- Exemplo: slt $t0,$s3,$s4 # se $s3<$s4, $t0<-1 senão $t0<-0.
- slti -> versão imediata de slt.
- Exemplo: slti $t0,$s2,10 #$t0<-1 se $s2<10.

# Resumo até Aqui

| Category | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | add | `add $s1,$s2,$s3` | $s1 = $s2 + $s3 | Three operands; data in registers |
| | subtract | `sub $s1,$s2,$s3` | $s1 = $s2 – $s3 | Three operands; data in registers |
| Data transfer | load word | `lw $s1,100($s2)` | $s1 = Memory[$s2 + 100] | Data from memory to register |
| | store word | `sw $s1,100($s2)` | Memory[$s2 + 100] = $s1 | Data from register to memory |
| Logical | and | `and $s1,$s2,$s3` | $s1 = $s2 & $s3 | Three reg. operands; bit-by-bit AND |
| | or | `or $s1,$s2,$s3` | $s1 = $s2 \| $s3 | Three reg. operands; bit-by-bit OR |
| | nor | `nor $s1,$s2,$s3` | $s1 = ~ ($s2 \|$s3) | Three reg. operands; bit-by-bit NOR |
| | and immediate | `andi $s1,$s2,100` | $s1 = $s2 & 100 | Bit-by-bit AND reg with constant |
| | or immediate | `ori $s1,$s2,100` | $s1 = $s2 \| 100 | Bit-by-bit OR reg with constant |
| | shift left logical | `sll $s1,$s2,10` | $s1 = $s2 << 10 | Shift left by constant |
| | shift right logical | `srl $$s1,$s2,10` | $s1 = $s2 >> 10 | Shift right by constant |
| Conditional branch | branch on equal | `beq $s1,$s2,L` | if ($s1 == $s2) go to L | Equal test and branch |
| | branch on not equal | `bne $s1,$s2,L` | if ($s1 != $s2) go to L | Not equal test and branch |
| | set on less than | `slt $s1,$s2,$s3` | if ($s2 < $s3) $s1 = 1; else $s1 = 0 | Compare less than; used with `beq`, `bne` |
| | set on less than immediate | `slt $s1,$s2,100` | if ($s2 < 100) $s1 = 1; else $s1 = 0 | Compare less than immediate; used with `beq`, `bne` |
| Unconditional jump | jump | `j L` | go to L | Jump to target address |

# *Exercício

- *Pesquisar como seria a linguagem de máquina para cada instrução lógica e de desvio vistas anteriormente.☺
- *Estudar o formato *J*.☺

# Registradores Especiais

- $a0 - $a3: registradores de argumento para passar parâmetros.
- $v0 - $v1: registradores para valores de retorno.
- $ra: registrador de endereço de retorno.
- pc: contador de programa. End. Da instrução que está sendo executada.
- Obs.: instrução jal: *jump and link;* desvia para um end. de procedimento e salva o endereço de pc+4 em $ra.
- Exemplo: jal end_proc.
- jr $ra: *jump register* ($ra=end_retorno).

# Pilha

- Exemplo: considere o código em C dado a seguir:

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

# Pilha

- g,h,i,j -> $a0, $a1, $a2 e $a3.
- f -> $s0.
- É preciso salvar os registradores temporários ($s0, $t0 e $t1) na pilha:

```
addi $sp,$sp,-12 # cria espaço para 3 itens na pilha.
sw $t1,8($sp)    # salva $t1.
sw $t0,4($sp)    # salva $t0.
sw $s0,0($sp)    # salva $s0.
```

Obs.: sp -> *stack pointer*.

# Pilha

- Corpo do procedimento:

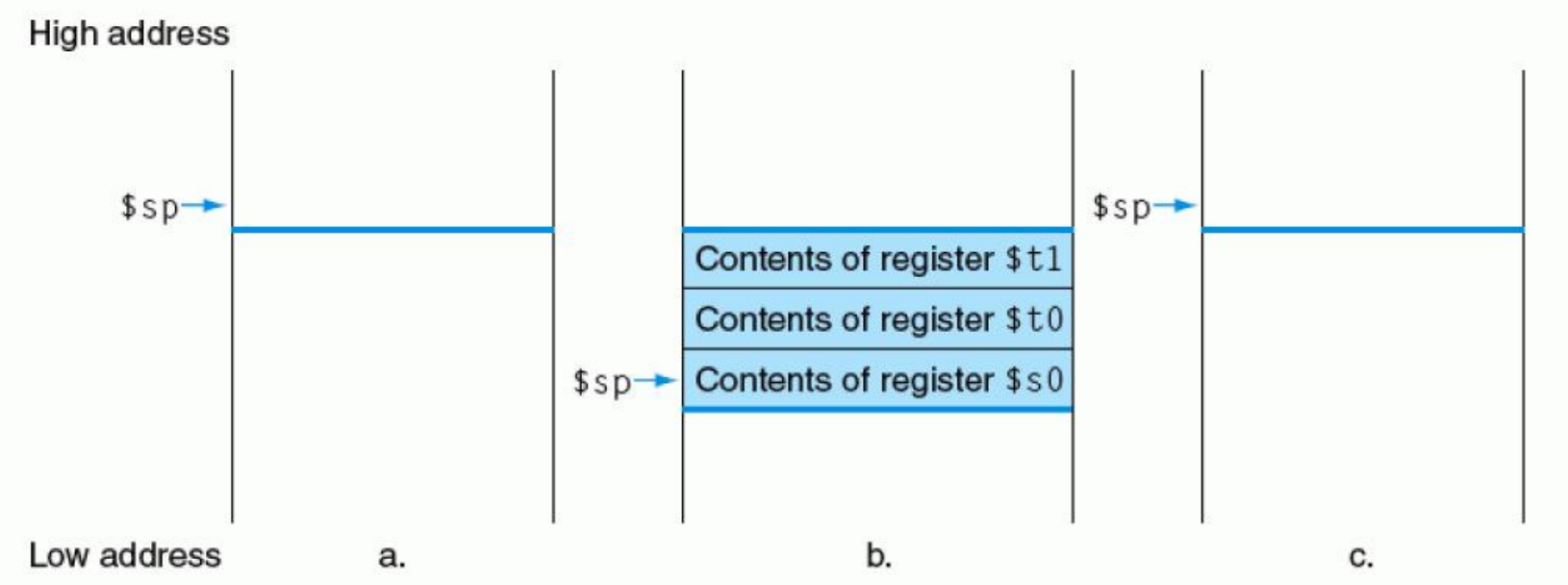
```
add $t0,$a0,$a1      # $t0<-g+h.
add $t1,$a2,$a3      # $t1<-i+j.
sub $s0,$t0,$t1      # f<-$t0-$t1.
add $v0,$s0,$zero    # retorna f.
```

# Pilha

- Antes de retornar é necessário restaurar os valores antigos dos registradores:

```
lw $s0,0($sp)
lw $t0,4($sp)
lw $t1,8($sp)
addi $sp,$sp,12  # exclusão dos itens da pilha.
jr $ra           # desvia novamente para a rotina que chamou.
```

# Pilha


High address
$sp
$sp
Contents of register $t1
Contents of register $t0
$sp
Contents of register $s0
Low address
a.
b.
c.

# Compilador SPIM

- http://users.ece.gatech.edu/~sudha/2030/temp/spim/spim-tutorial.html
- http://ece.ut.ac.ir/Classpages/F83/Computer%20Architecture/totorial/tutorial.pdf

# *Exercício

- Edite o seguinte trecho de código em algum editor de texto e verifique seu funcionamento no ambiente de simulação apresentado.

```
# tutorial.s
        .data 0x10000000
msg1:           .asciiz "Arquitetura e Organizacao de Computadores I"
        .text
main:
        li $v0, 4 # código para imprimir "string"
        la $a0, msg1
        syscall
```

# *Tarefa

- Desenvolva um programa em *assembly* do MIPS para ler uma *string* do teclado e imprimir a cadeia lida no console do simulador PCSPIM. Observe a ordem de inserção dos caracteres na memória.
- Desenvolver um programa em *assembly* do MIPS para calcular o fatorial de um número inteiro *n*. Dica: Utilize o simulador PCSPIM para testar o programa.