Arquitetura de Conjunto de Instruções MIPS

*Assembly ou linguagem de montagem é uma notação legível por humanos para o código de máquina que uma arquitetura de computador específica usa, utilizada para programar códigos entendidos por dispositivos computacionais, como microprocessadores e microcontroladores. O código de máquina torna-se legível pela substituição dos valores em bruto por símbolos chamados mnemónicos.
Por exemplo, enquanto um computador sabe o que a instrução-máquina IA-32 (B0 61) faz, para os programadores é mais fácil recordar a representação equivalente em instruções mnemónicas MOV AL, 61h. Tal instrução ordena que o valor hexadecimal 61 (97, em decimal) seja movido para o registrador 'AL'. Embora muitas pessoas pensem no código de máquina como valores em binário, ele é normalmente representado por valores em hexadecimal.
A tradução do código Assembly para o código de máquina é feita pelo montador ou assembler. Ele converte os mnemónicos em seus respectivos opcodes, calcula os endereços de referências de memória e faz algumas outras operações para gerar o código de máquina que será executado pelo computador.

Não podemos confundir Arquitetura do Conjunto de Instruções MIPS com MIPS, *Milhões de Instruções por Segundo*. O MIPS que tratamos aqui é uma Arquitetura de Conjunto de Instruções (Instruction Set Architecture - ISA), desenvolvida pela empresa MIPS Computer Systems, que hoje é chamada de MIPS Technologies. MIPS significa Microprocessor Without Interlocked Pipeline Stages (**Microprocessador Sem Estágios Intertravados de Pipeline**).

Antes de falarmos especificamente sobre o MIPS, vamos discutir um pouco sobre **Conjunto de Instruções.** Como bem sabemos, todo Sistema Computacional é composto, muito basicamente, por Entrada, Saída, Processamento e Armazenamento. Cada um desses subsistemas pode ser organizado de formas diferentes (**organização**) no sistema, e cada elemento que faz parte desses subsistemas pode ser projetado também de formas diferentes (**arquitetura**).

O **Conjunto de Instruções** é um dos elementos desse grande sistema, e é de extrema importância para a construção de um sistema computacional. Um Processador não é exatamente um dispositivo único, ele é um conjunto de sistemas, cada um responsável por executar determinadas ações. O que temos, na verdade, é uma CPU - Unidade Central de Processamento - composta pela Unidade de Controle, Unidade Lógica Aritmética, entre muitas outras **unidades funcionais** necessárias para realizar o processamento de qualquer tipo de dados que precisamos.

Por exemplo, se um microprocessador não é capaz de executar uma soma em ponto flutuante, então este computador não poderá processar determinados tipos de dados, programas, etc. O computador ficará limitado, o que nos dias atuais não é nada interessante. Portanto, quando se projeta um novo microprocessador, primeiro é necessário definir que tipo de instruções, dados e programas ele será capaz de executar.

Além disso, precisa-se manter a compatibilidade com microprocessadores anteriores. O novo microprocessador deve ser capaz de continuar executando os seus softwares. Não é uma ideia interessante lançar um microprocessador com muitas inovações se os usuários não puderem mais usar os softwares que estão acostumados. É claro que, um dia, logo mais à frente, a tecnologia vai mudar e, de certa forma, nos veremos na obrigação de evoluir. Os transistores, a tecnologia atual de fabricação de computadores, está em seu limite, e muitos pesquisadores estão buscando novas matérias primas para construção de processadores cada vez mais rápidos. Quando uma nova tecnologia surgir, nos encontraremos exatamente nesse ponto de evolução.

Os nossos computadores, notebooks e celulares atuais são dispositivos de uso geral. Isso significa que eles precisam ter capacidade de processamento para diversos tipos de dados diferentes, que variam desde um texto simples até um vídeo em três dimensões. É diferente de um controle remoto de TV ou um Microondas, que são projetados para um fim específico. Dessa forma, as instruções que o microprocessador é capaz de executar formam o conjunto de instruções. Algumas instruções que o microprocessador de uso geral pode executar são:

operações aritméticas; operações lógicas; operações relacionais; operações de ponto flutuante; transferência de dados; desvios condicionais; desvios incondicionais; controle;

A **Arquitetura do Conjunto de Instruções** define os tipos de instruções que serão executadas por um processador, assim como o formato de cada instrução, quantidade de bits, a forma como essas instruções acessarão registradores e memórias (modos de endereçamento), a forma como conversarão com outras unidades funcionais, etc.

O **tamanho**, em bits, de uma instrução. Conjuntos de instruções dos processadores X86 costumam não ter um tamanho fixo de Bits. Nessas arquiteturas, uma operação aritmética pode ter um tamanho diferente de uma operação de transferência, o que não é muito interessante. Hennessey e Patterson demonstraram com a Arquitetura MIPS que instruções de tamanho fixo de bits são melhores de se manipular, particularmente no processamento paralelo das instruções (Pipeline). Assim, em um projeto de arquitetura de conjunto de instruções, o tamanho da instrução, em bits, pode ser FIXO ou VARIÁVEL. A arquitetura MIPS atual tem tamanho **fixo de 64 bits** e é chamada de MIPS 64 Bits.

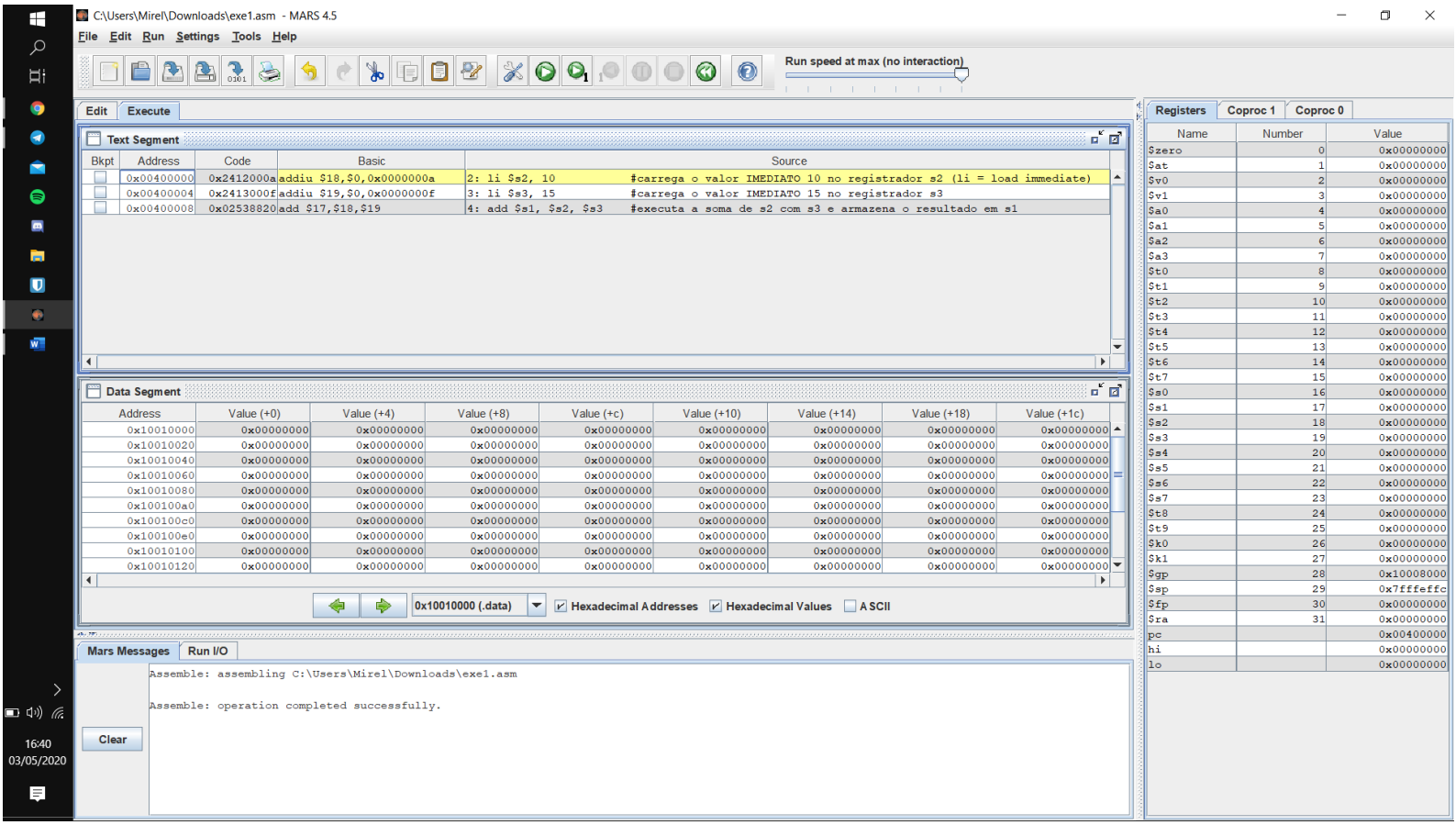
Normalmente uma instrução tem um campo reservado para o código de operação, chamado de OPCODE, campos reservados para os operandos da operação, entre outros campos, como, por exemplo, o endereço de um ponteiro. Novamente reforço que isso é definido pelos projetistas da arquitetura. O OPCODE indica qual é a operação que deverá ser executada, exemplo:

ADD $s1, $s2, $3.

ADD é o mnemônico que identifica o código da operação, neste exemplo, *uma adição com dois operandos, que são os registradores de uso geral $s2, $s3. O resultado da soma será armazenado no registrador $s1.*

Da mesma forma que a Linguagem Java precisa da máquina virtual Java para interpretar e compilar os programas desenvolvidos, uma ISA também precisa de um compilador. Todas as instruções Assembly são traduzidas para a linguagem binária (0 e 1), por isso é necessário um Compilador Assembly nos processadores. De fato, tudo o que é executado em nosso computador é traduzido para Assembly e depois para linguagem binária, de forma que possa ser processado pelo grande sistema digital que é o processador.

```
.text
li $s2, 10          #carrega o valor IMEDIATO 10 no registrador s2 (li = load immediate)
li $s3, 15          #carrega o valor IMEDIATO 15 no registrador s3
add $s1, $s2, $s3   #executa a soma de s2 com s3 e armazena o resultado em s1
```



MARS (simulador MIPS):

Execute: Apresenta uma tabela com todas as linhas do seu código com o respectivo endereço de memória, e código de máquina, em hexadecimal, assim como o código Basic e o Source, que é o código fonte. Note que em Basic, o código fonte que digitamos em MIPS fica da seguinte forma:

addiu $18, $0, 0x00000000a #$18 = 0 + 10
addiu $19, $0, 0x00000000f #$19 = 0 + 15
add $17, $18, $19 #$17 = $18 + $19 = 10 + 15 = 25

Conforme a referência MIPS, ADDIU é uma instrução de adição imediata sem overflow, que coloca a soma do registrador RS e o imediato com sinal estendido no registrador RT. Basicamente, aqui os valores imediatos que determinamos, 10 e 15, foram convertidos para o sistema de numeração hexadecimal, A e F, respectivamente, e então somados com ZERO ($0). O número 25 convertido em hexadecimal é 19, em binário é 00011001 e em octal é 31.

Observe que na coluna ADDRESS os endereços estão sempre como múltiplos de 4: 0x00400000, 0x00400004 e 0x00400008. Note também que a primeira linha do código está destacada em amarelo.
Data Segment: Essa é a área de segmento de dados (DS), mais conhecida como **memória**. É possível escolher a forma que quer ver os endereços de memória selecionando-os nos checks boxes logo abaixo: endereços hexadecimais, valores hexadecimais e ainda ASCII. Também pode escolher a região de memória a partir das setinhas para a direita e esquerda e do combobox.

Mars Messages: Apresenta mensagens produzidas pelo menu RUN.
Run I/O: É um console de entrada e saída do simulador.
Registers: Referente aos registradores do MIPS.
Coproc1: Referente aos registradores da unidade de ponto flutuante.
Coproc0: Referente aos registradores de interrupções e exceções.

Todo software é traduzido para a linguagem binária (0 e 1) pra poder ser executado pelo processador. Quando você está com o computador ligado, usando quaisquer programas, todos eles estão sendo traduzidos e executados pelo seu processador. Então é necessário existir um tradutor entre o software e o hardware.

Considere a seguinte instrução em linguagem C:

a = b + c;

Essa instrução, para ser executada no processador, deve ser convertida para MIPS e depois para 0 e 1. Para poder manipular as variáveis é necessário utilizar os registradores do MIPS. A CPU é composta por registradores, que é a memória mais rápida e mais cara do projeto de um microprocessador, justamente porque está na CPU, para ajudar a executar as instruções.

Para as conversões usamos os registradores de uso geral do MIPS. Normalmente, nas nossas conversões, as variáveis que armazenam valores usam os registradores de nome t e as variáveis que contêm os operandos usam os registradores de nome s.

Registradores de uso geral do MIPS32

Nome dos registradores	$Número_{10}$	$Número_{2}$	Uso
$zero	0	000 000	constante zero
$at	1	000 001	reservado para o montador
$v0	2	000 010	avaliação de expressão e resultador de uma função
$v1	2	000 011	avaliação de expressão e resultador de uma função
$a0	4	000 100	argumento 1 (passam argumentos para as rotinas)
$a1	5	000 101	argumento 2

$a2	6	000 110	argumento 3
$a3	7	000 111	argumento 4
$t0	8	001 000	temporário (valores que não precisam ser preservados entre chamadas)
$t1	9	001 001	temporário
$t2	10	001 010	temporário
$t3	11	001 011	temporário
$t4	12	001 100	temporário
$t5	13	001 101	temporário
$t6	14	001 110	temporário
$t7	15	001 111	temporário
$s0	16	010 000	temporário salvo (valores que precisam ser preservados entre chamadas)
$s1	17	010 001	temporário salvo
$s2	18	010 010	temporário salvo
$s3	19	010 011	temporário salvo
$s4	20	010 100	temporário salvo
$s5	21	010 101	temporário salvo

$s6	22	010 110	temporário salvo
$s7	23	010 111	temporário salvo
$t8	24	011 000	temporário
$t9	25	011 001	temporário
$k0	26	011 010	reservado para o Kernel do sistema operacional
$k1	27	011 011	reservado para o Kernel do sistema operacional
$gp	28	011 100	ponteiro para área global
$sp	29	011 101	stack pointer (aponta para o último local pilha)
$fp	30	011 110	frame pointer (aponta para a primeira palavra do frame de pilha do procedimento)
$ra	31	011 111	endereço de retorno de uma chamada de procedimento

De acordo com a tabela, usaremos então os registradores que vão de $t0 à $t7 e de $s0 à $s7. A instrução a = b + c ficará da seguinte forma:

ADD $t0, $s0, $s1

Os nomes dos registradores usados aqui foram escolhidos arbitrariamente, mas durante a execução real são usados os registradores que estiverem livres naquele instante.

O próximo agora é converter a instrução de Linguagem de Montagem para Linguagem de Máquina. Cada registrador tem um número, conforme apresenta a Tabela 1. A Linguagem de Máquina corresponde a trocar o NOME DO REGISTRADOR pelo seu NÚMERO. A instrução ficará da seguinte forma:

ADD $8, $16, $17

Formatos de instrução:
As instruções estão divididas em três tipos: R, I e J. Toda instrução começa com um opcode de 6 bits. Além do opcode, as instruções do tipo **R** especificam três registradores, um campo de valor de deslocamento e um campo de função;
Instruções do tipo **I** especificam dois registradores e um valor imediato de 16 bits;
As instruções do tipo **J** seguem o código de operação com um alvo de salto de 26 bits.

Tipo	-31- formato(bits) -0-					
R	opcode(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)
I	opcode(6)	rs(5)	rt(5)	imediato(16)		
J	opcode(6)	endereço(26)				

- op - operação básica da instrução (opcode).
- rs - o primeiro registrador fonte.
- rt - o segundo registrador fonte.
- rd - o registrador destino.
- shamt - shift amount, para instruções de deslocamento.
- funct - function. Seleciona variações das operação especificada pelo opcode.

Todas as instruções no MIPS 32 têm exatamente 32 bits e cada formato de instrução tem campos diferentes. O formato R é dividido em seis campos, sendo os campos OP e FUNCT com 6 bits e o restante com 5 bits, totalizando 32 bits.

O primeiro campo, OP, é destinado para identificação do código de operação que será realizada. O último campo seleciona uma operação secundária, por exemplo, a operação principal é a aritmética, mas a secundária é uma soma. Nem todas as operações têm dois códigos (op-funct) e nesse caso o campo FUNCT é setado com o valor zero.

Mnemonic	Meaning	Type	Opcode	Funct
`add`	Add	R	0x00	0x20
`addi`	Add Immediate	I	0x08	NA
`addiu`	Add Unsigned Immediate	I	0x09	NA
`addu`	Add Unsigned	R	0x00	0x21

Mnemonic	Meaning	Type	Opcode	Funct
`and`	Bitwise AND	R	0x00	0x24
`andi`	Bitwise AND Immediate	I	0x0C	NA
`beq`	Branch if Equal	I	0x04	NA
`blez`	Branch if Less Than or Equal to Zero	I	0x06	NA
`bne`	Branch if Not Equal	I	0x05	NA
`bgtz`	Branch on Greater Than Zero	I	0x07	NA
`div`	Divide	R	0x00	0x1A
`divu`	Unsigned Divide	R	0x00	0x1B
`j`	Jump to Address	J	0x02	NA
`jal`	Jump and Link	J	0x03	NA
`jr`	Jump to Address in Register	R	0x00	0x08
`lb`	Load Byte	I	0x20	NA
`lbu`	Load Byte Unsigned	I	0x24	NA
`lhu`	Load Halfword Unsigned	I	0x25	NA

Mnemonic	Meaning	Type	Opcode	Funct
`lui`	Load Upper Immediate	I	0x0F	NA
`lw`	Load Word	I	0x23	NA
`mfhi`	Move from HI Register	R	0x00	0x10
`mthi`	Move to HI Register	R	0x00	0x11
`mflo`	Move from LO Register	R	0x00	0x12
`mtlo`	Move to LO Register	R	0x00	0x13
`mfc0`	Move from Coprocessor 0	R	0x10	NA
`mult`	Multiply	R	0x00	0x18
`multu`	Unsigned Multiply	R	0x00	0x19
`nor`	Bitwise NOR (NOT-OR)	R	0x00	0x27
`xor`	Bitwise XOR (Exclusive-OR)	R	0x00	0x26
`or`	Bitwise OR	R	0x00	0x25
`ori`	Bitwise OR Immediate	I	0x0D	NA
`sb`	Store Byte	I	0x28	NA

Mnemonic	Meaning	Type	Opcode	Funct
`sh`	Store Halfword	I	0x29	NA
`slt`	Set to 1 if Less Than	R	0x00	0x2A
`slti`	Set to 1 if Less Than Immediate	I	0x0A	NA
`sltiu`	Set to 1 if Less Than Unsigned Immediate	I	0x0B	NA
`sltu`	Set to 1 if Less Than Unsigned	R	0x00	0x2B
`sll`	Logical Shift Left	R	0x00	0x00
`srl`	Logical Shift Right (0-extended)	R	0x00	0x02
`sra`	Arithmetic Shift Right (sign-extended)	R	0x00	0x03
`sub`	Subtract	R	0x00	0x22
`subu`	Unsigned Subtract	R	0x00	0x23
`sw`	Store Word	I	0x2B	NA

Os campos RS e RT são destinados aos operandos fontes, em ordem, da esquerda para a direita, conforme aparecem na operação. Já o campo RD é destinado para o armazenamento do resultado da operação, portanto, operando destino. Shamt é um campo usado para setar uma quantidade de deslocamento, mas, por hora, não o usaremos, por isso será setado com o valor zero.

OPCODE	DECIMAL	BINÁRIO
ADD	32	100 000
SUB	34	100 010
OR	36	100 100
AND	37	100 101
SLT	42	101 010
BNE	4	000 100
BEQ	5	000 101
J	2	000 010
JR	8	001 000
LW	35	100 011
SW	43	101 011

Portanto:

op	rs	rt	rd	shamt	funct
0	$16	$17	$8	0	32

op	rs	rt	rd	shamt	funct
000000	10000	10001	01000	00000	100000

O código binário 00000010000100010100000000100000 é a instrução a = b + c em MIPS 32 bits. Diante disso, podemos fazer um algoritmo passo a passo para realizar a conversão:

1. Converter a instrução de alto nível para Linguagem de Montagem;

2. Converter a instrução na Linguagem de Montagem para Linguagem de Máquina;

3. Fazer a representação correspondente da Linguagem de Máquina;

4. Converter a representação da Linguagem de Máquina para Código de Máquina.

Como converter uma instrução com Array no MIPS:

Como exemplo, suponha a seguinte instrução em linguagem C:

a = b + c[10];

Vamos usar o registrador $s0 para a variável a, $s1 para variável b e $s2 para a variável c. c é um vetor e não sabemos quantas posições ele tem, só sabemos que iremos somar o valor da variável b com o valor que está armazenado na posição 10 do vetor c.

Pra isso, será usada a instrução de formato I (transf. De dados)

Load Word (instrução no formato I): Essa instrução transfere dados da memória para os registradores e, sempre que tivermos um Array, deveremos utilizá-la pois, antes de manipularmos o valor de uma determinada posição do Array, devemos tê-lo disponível para isso. Sua sintaxe é a seguinte:

LW registrador_destino, valor (registrador_fonte)

LW $t0, 30 ($s0) # $t0 = memória [$s0 + 30]

O registrador $t0 receberá o valor que está no endereço de memória que é calculado pela própria instrução: $s0 + 30. Então, toda vez que você usar a instrução LW, você está transferindo para um registrador, um valor que está no endereço de memória calculado pela soma do registrador fonte com um valor. Neste exemplo é um valor dado (30), ou seja, é a posição 30 do Vetor que aqui é representado por $s0.

O primeiro passo é converter c[10] que ficará assim:

LW $t0, 10 ($s2) # $t0 = memória [$s2 + 10]

Observe que $s2 é o vetor c, 10 é a posição do Vetor e $t0 é um registrador temporário que armazenará o valor que está em c[10]. O segundo passo é fazer b + c[10]:

ADD $s0, $s1, $t0 # $s0 = $s1 + $t0

Em que $t0 é o valor de c[10], $s0 é a variável a e $s1 é a variável b. Assim, o código final para a = b + c [10] fica da seguinte forma:

LW $t0, 10 ($s2)

ADD $s0, $s1, $t0

A Linguagem de Máquina para a = b + c [10] ficará da seguinte forma:

LW $8, 10 ($18)

ADD $16, $17, $8

opcode	rs	rt	rd	shamt	funct
35	8	18	10		
0	17	8	16	0	32

Código de máquina:

opcode	rs	rt	rd	shamt	funct
100 011	01000	10010	0000 0000 0000 1010		
000 000	10001	01000	10000	00000	100 000

Store Word: tem como objetivo armazenar um valor, que está em um registrador, na memória. Store Word significa Armazenar Palavra, ao pé da letra, assim como Load Word significa Carregar Palavra.

Sintaxe da instrução:

SW registrador_fonte, valor (registrador_destino)

SW $t0, 30 ($s0) # memória [$s0 + 30] = $t0

a conversão da seguinte instrução:

a[15] = b + c

Usar $s0 para a, $s1 para b e $s2 para c. O primeiro passo é converter b + c, que ficará assim:

ADD $t0, $s1, $s2 # $t0 = $s1 + $s2

A instrução acima realiza a soma de b com c, armazenando o resultado em $t0. Agora, vamos armazenar $t0 no endereço de memória, que é o nosso array a[15]:

SW $t0, 15 ($s0) # memória [15 + $s0] = $t0

a[15] é correspondente ao código 15 ($s0) e $t0 é o valor da soma. Portanto, o código final é:

ADD $t0, $s1, $2
SW $t0, 15 ($s0)

A Linguagem de Máquina para a[15] = b + c ficará da seguinte forma:

ADD $8, $17, $18

SW $8, 15 ($16)

RS e RT são os registradores fonte e RD o registrador destino, para as instruções do tipo R. Para as instruções STORE WORD, RS é o registrador fonte e RT é o registrador destino. Finalizando, para as instruções LOAD WORD, RS é o registrador destino e RT é o registrador fonte.

opcode	rs	rt	rd	shamt	funct
0	17	18	8	0	32
43	8	16	15		

Código de máquina:

opcode	rs	rt	rd	shamt	funct
000 000	10001	10010	01000	00000	100 000
101 011	01000	10000	0000 0000 0000 1111		

Converter a instrução a seguir:
a[22] = b[1] – c

Observe que na instrução de LW o array está no final e, na instrução que apresento hoje, o array está no início. A ordem em que o array é apresentado ao longo da instrução também deve ser respeitada. A ordem de leitura da esquerda para a direita tem de ser mantida na instrução original e na instrução resultante em MIPS.

Não esquecer
O processo de conversão envolve quatro passos:

- Linguagem de Montagem
- Linguagem de Máquina
- Representação da Linguagem de Máquina
- Código de Máquina

Primeiro resolvemos b[1] - c e o resultado disso armazenamos em a[22]. Consideremos *a = $s0, b = $s1 e c = $s2*. O primeiro passo é carregar o valor de b[1] em um registrador temporário usando a instrução de LW (load word):
LW $t0, 1 ($s1) # $t0 = memória [1 + $s1]
Agora vamos fazer a subtração usando um registrador temporário, $t1, para armazenar o resultado:
SUB $t1, $t0, $s2 # $t1 = $t0 - $s2
O último passo é transferir o resultado que está em $t1 para o array:
SW $t1, 22 ($s0) # memória [22 + $s0] = $t1
Assim, a Linguagem de Montagem completa, para essa instrução é:
LW $t0, 1 ($s1)
SUB $t1, $t0, $s2
SW $t1, 22 ($s0)
E a Linguagem de Máquina:
LW $8, 1 ($17)
SUB $9, $8, $18
SW $9, 22 ($16)
Código de Máquina:
10001101000011110000000000000001
00000001000100100100100000100010
10101101001100000000000000010110

Instrução BEQ (tipo I): significa Branch On Equal, em português, desvie se for igual. Essa instrução força um desvio para o comando com o LABEL (nome de desvio) se o valor no registrador 1 for igual ao valor no registrador 2, portanto, é uma instrução de comparação.

BEQ registrador1, registrador2, endereço de desvio

Basicamente, se r1 = r2 então, desvia para o endereço que está sendo apontado pelo LABEL e execute as instruções que ali estão. Caso contrário, o programa continuará executando as instruções seguintes. exemplo:

if (x == y) go to L2;
a = b + c;
L2 : a = b - c;

BEQ $s0, $s1, L2 #desvia para L2 se x = y
ADD $s2, $s3, $s4 # Executa se x != y
L2 : SUB $s2, $s3, $s4 # Executa se x = y

Instrução BNE (tipo I): significa Branch On Not Equal, ou seja, desvie se não for igual. É basicamente igual a BEQ, com a diferença que o desvio ocorre caso os valores sejam diferentes.

IF COMPOSTO:
Considere f = $s0; g = $s1; h = $s2; i = $s3 e j = $s4.

if(i==j) f= g+h

else f=g-h

De acordo com os criadores do MIPS, o código será mais eficiente se for testada a desigualdade, isto é, se testarmos o THEN primeiro. Qual a probabilidade de i ser igual a j? Provavelmente ela é muito menor do que i ser diferente de j. E mais, muitas arquiteturas de conjuntos de instruções, incluindo o MIPS, consideram que o desvio sempre ocorre.
Se usar BEQ funcionará? BEQ forçará a entrada para ELSE se e somente se i for igual a i.
Quando no código em alto nível vai entrar no ELSE? Entrará no ELSE quando i for diferente de j e não o contrário. É por isso que se usa BNE ao invés de BEQ, pois queremos tratar o desvio condicional primeiro.

```
bne $s3, $s4, Else      # desvia para ELSE se i != j
add $s0, $s1, $s2       # f = g + h (salta esta instrução se i != j)
```

precisamos adicionar uma instrução MIPS que se chama JUMP e é abreviada como J. Temos de adicionar essa instrução para indicar o fim da execução desse bloco de instruções e, também, para que o processador continue a executar o programa. Essa instrução é um desvio INCONDICIONAL e EXIT indica o fim desse bloco de programa.

```
j Exit          # desvia para exit
```

Continuando a compilação para MIPS, a próxima linha em C é o ELSE e dentro dele temos a instrução de subtração. Ficará assim:

```
Else: sub $s0, $s1, $s2      # f = g – h (salta esta instrução se i = j)
Exit:
```

Linguagem final:

```
      bne $s3, $s4, Else        # desvia para ELSE se i != j
      add $s0, $s1, $s2         # f = g + h (salta esta instrução se i != j)
      j Exit                    # desvia para exit
Else: sub $s0, $s1, $s2         # f = g – h (salta esta instrução se i = j)
Exit:
```

Linguaguem de máquina:

```
      bne $19, $20, Else      # desvia para ELSE se i < > j
      add $16, $17, $18       # f = g + h (salta esta instrução se i <> j)
      j Exit                  # desvia para exit
```

Else: sub $16, $17, $18 # f = g – h (salta esta instrução se i = j
Exit:

Código de máquina:
00010010011101000010011100100000
00000010001100101000000000100000
00001000000000000010011100100100
00000010001100101000000000100010

Instrução SLT: Significa Set on Less Than, ou seja comparar menor que. A função desta instrução é comparar dois valores de dois registradores diferentes e atribuir o valor 1 a um terceiro registrador se o valor do primeiro registrador for menor que o valor do segundo registrador. Caso contrário, atribuir zero. A sintaxe é:

SLT registrador_temporário, registrador1, registrador2

if (i<j) a=b+c

else a=b-c

Considere a = $s0, b = $s1, c = $s2, i = $s3, j = $s4.

linha	código
1	slt $t0, $s3, $s4
2	bne $t0, $zero, ELSE
3	add $s0, $s1, $s2 *#a = b + c; (se $t0 != 0)*
4	j Exit *#desvia para exit*
5	ELSE: sub $s0, $s3, $s4 *#a = b – c; (se $t0 = 0)*
6	Exit:

Linguagem de máquina:

slt $8, $19, $20
bne $8, $zero, ELSE
add $17, $18, $16
j Exit
ELSE: sub $19, $20, $16
Exit:

Agora vamos entender algo importante sobre os valores que devem ser inseridos no lugar dos labels ELSE e EXIT. Sabemos que o endereço 80004 pula para o endereço 80016, que é a nossa quinta linha de código, ou seja, o ELSE.
Precisamos lembrar que as instruções MIPS possuem endereços em bytes, de modo que os endereços das palavras sequenciais diferem em 4 bytes. Começamos esse bloco de comando no endereço 80000 e terminamos no endereço 80020 (exit).
Cada instrução está inserida no seu endereço correspondente que difere em quatro bytes. O cálculo que devemos fazer deve considerar o endereço SEGUINTE e NÃO o endereço atual da instrução.

Assim, a instrução BNE, na segunda linha, acrescenta duas palavras, ou oito bytes, ao endereço da instrução SEGUINTE, especificando o destino do desvio em relação à instrução seguinte, e não em relação à instrução de desvio.
O endereço da instrução seguinte é 80008. Agora, veja que curioso, se você fizer 80016 - 80008, tem-se como resultado o número 8, isto é, oito bytes, o que significa que devemos "pular" duas posições na memória, portanto, o número 2 é quem deve ir no campo endereço da instrução BNE. Se você fizer 80008 + 8, o resultado será 80016, que é exatamente o endereço para onde queremos ir. Ou seja, o valor deverá ser colocado conforme a quantidade de instruções entre os comandos.

Operações lógicas:

Operações Lógicas	Instruções MIPS
Shift à direita	SLL
Shift à esquerda	SRL
And bit a bit	AND, ANDI
Or bit a bit	OR, ORI
Not bit a bit	NOR

SLL significa **Shift Left Logical** (deslocamento lógico à esquerda) e SRL significa **Shift Right Logical** (deslocamento lógico à direita), que são os mnemônicos e nomes das instruções MIPS.

0000 0000 0000 0000 0000 0000 1001
0000 0000 0000 0000 0000 1001 0000

SLL $t2, $s0, 4 # $t2 = $s0 << 4 bits

SLL $10, $16, 4 (lg. máquina)

000000 00000 10000 01010 00100 000000

Assim, 4 é o número de bits que deve ser deslocado, portanto, ele deve ser colocado no campo **SHAMT**, que significa Shift Amount. Esse campo armazena a quantidade de deslocamento e é usado apenas para instruções de deslocamento. $t2 ($10) é o registrador que armazenará o resultado, portanto, registrador destino e, por isso, é colocado no campo RD. $s0 ($16) deve ser inserido no campo RT. O campo RS não é utilizado, por isso colocamos zero. A codificação padrão para SLL é zero pra o campo de código e para funct. Portanto, SLL é uma instrução do tipo R.

opcode	rs	rt	rd	shamt	funct
0	0	16	10	4	0
6 bit	5 bits	5 bits	5 bits	5 bits	6 bits

Em nosso exemplo o número decimal inicial era 9 e o resultado do deslocamento em 4 bits foi 144. Se vocês fizerem 9 x 24 o resultado será 144, ou seja, a instrução SLL é equivalente a uma multiplicação por 2i.

A instrução SRL é muito parecida com a SLL, a diferença básica é que a SRL desloca os bits para a direita. O código de FUNCT para SLL é 2 e o opcode é 0. Vejamos um exemplo:

0000 0000 0000 0000 0000 1001 0000 = 144

Deslocando 4 bits a direita:

0000 0000 0000 0000 0000 0000 1001 = 9

Assim a instrução na linguagem de montagem fica da seguinte forma:

SRL $t2, $s0, 4 # $t2 = $s0 >> 4 bits

Na linguagem de máquina fica:

SRL $10, $16, 4

A representação fica da seguinte forma:

opcode	rs	rt	rd	shamt	funct
0	0	16	10	4	2
6 bit	5 bits	5 bits	5 bits	5 bits	6 bits

Finalizando a compilação, o código de máquina:

000000 00000 10000 01010 00100 000010

A operação lógica **AND** é uma operação bit a bit com dois operandos que resulta em 1 somente se os dois bits dos operandos também forem 1. É o mesmo comportamento esperado de uma porta lógica AND (multiplicação).

A instrução tem a seguinte sintaxe:

AND registrador_destino, registrador_fonte, registrador_fonte

O registrador destino armazena o resultado da operação que é operada em cima dos dois operandos que estão armazenados cada um em um registrador fonte diferente. Considere o seguinte exemplo: o registrador $t3 possui o valor decimal 3328, que em binário é

0000 0000 0000 0000 0000 1101 0000 0000

e o registrador $t4 possui o valor decimal 15.360 que em binário é

0000 0000 0000 0000 0011 1100 0000 0000

Assim temos a instrução

AND $t0, $t3, $t4 #$t0 = $t3 & $t4

O que acontece nesta operação pode ser representado como a tabela abaixo:

$t3	0	0	**1**	**1**	0	**1**	0	0	0	0	0	0	0	0
$t4	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
$t0	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Cada bit que compõe o número binário será verificado individualmente, começando pelo bit mais à direita e terminando no bit mais à esquerda, como acontece quando você resolve uma conta aritmética no caderno.

AND $t0, $t3, $t4

AND $8, $11, $12

opcode	rs	rt	rd	shamt	funct
0	11	12	8	0	37

00000001011011000100000000100101

A operação lógica **OR** é uma operação bit a bit com dois operandos que resulta em 1 quando o valor 1 estiver presente em quaisquer dos dois operandos (soma).

A instrução tem a seguinte sintaxe:

OR registrador_destino, registrador_fonte, registrador_fonte

O registrador destino armazena o resultado da operação que é operada em cima dos dois operandos que estão armazenados, cada um, em um registrador fonte diferente. Vamos efetuar a operação OR nos mesmos números decimais do exemplo da operação AND:

OR $t0, $t3, $t4 #$t0 = $t3 | $t4

$t3	0	0	**1**	**1**	0	**1**	0	0	0	0	0	0	0	0
$t4	**1**	**1**	**1**	**1**	0	0	0	0	0	0	0	0	0	0
$t0	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

Cada bit que compõe o número binário será verificado individualmente, começando pelo bit mais à direita e terminando no bit mais à esquerda, como acontece quando você resolve uma conta aritmética no caderno. O bit resultante de cada posição resultará em 1 quando qualquer um dos operandos também for 1.

OR $t0, $t3, $t4

OR $8, $11, $12

opcode	rs	rt	rd	shamt	funct
0	11	12	8	0	36

00000001011011000100000000100100

A operação lógica **NOT** é uma operação bit a bit com um operando, diferente das operações AND e OR que precisam de dois operandos. O efeito desta operação é o de INVERTER os valores binários existentes em cada posição, significando que 0 se torna 1 e 1 se torna 0.

Entretanto, não é exatamente esta a operação que o MIPS executa. Para seguir o padrão de dois operandos, esta instrução de INVERSÃO foi projetada para atuar também em cima de dois

operandos, por isso, usa-se na verdade a instrução NOR (NOT OR), e não NOT, propriamente dito. A instrução funciona da seguinte forma:

A NOR 0

NOT (A OR 0)

NOT (A)

Substituindo A pelo valor binário zero (0):

0 NOR 0

NOT (0 OR 0)

NOT (0)

1

Substituindo A pelo valor binário um (1):

1 NOR 0

NOT (1 OR 0)

NOT (1)

0

Perceberam que o comportamento é igual ao da instrução NOT original? Quando um operando for zero (0) ele se torna um (1), e vice-versa. A sintaxe desta instrução no MIPS é:

NOR registrador_destino, registrador_fonte, registrador_fonte

O registrador destino armazena o resultado da operação que é operada em cima dos dois operandos que estão armazenados, cada um em um registrador fonte diferente. Vamos efetuar a operação NOR no número decimal 15.360 que está armazenado no registrador $t3 e o registrador $t5 armazenará apenas zeros.

NOR $t0, $t3, $t5 #$t0 = ~ ($t3 | $t5)

$t3	0	0	1	1	0	0	0	0	0	0	0	0	0	0
$t5	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$t3 OR 0	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
NOT	**1**	**1**	**0**	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

NOR $t0, $t3, $t5

NOR $8, $11, $13

opcode	rs	rt	rd	shamt	funct
0	11	13	8	0	39

000000 001011 01101 01000 00000 100111

Grande parte das Arquiteturas de Computadores são endereçáveis por Byte (8 bits) e o MIPS é uma delas. Assim, arquiteturas de computadores podem ser classificadas de acordo com a ordem dos bytes, o que é denominado **Endian.** Quando se projeta para armazenar os bytes menos significativos nos endereços mais baixos, tem-se uma arquitetura Little Endian. Já quando os bytes mais significativos são armazenados nos endereços mais baixos, tem-se uma arquitetura Big Endian.

Big	Byte 0	Byte 1	Byte 2	Byte 3
Little	Byte 3	Byte 2	Byte 1	Byte 0

Uma Arquitetura de Computadores de 32 bits possui 32 registradores, e assim por diante. Portanto, a quantidade de dados que pode ser manipulada por Linguagens de Alto/Médio nível é muito maior que a quantidade de registradores disponível no computador. Exemplificando: imagine um software de processamento de texto, como o Microsoft Word, executando em seu computador; um arquivo gerado por esse tipo de software gera muitos bytes, sendo impossível armazenar esses bytes nos registradores. Como um computador pode representar, e também acessar, essas estruturas de dados, que costumam ser realmente muito grandes?

Um Sistema Computacional é formado por vários outros sistemas, que conversam entre si, entre eles: sistema de entrada e saída, sistema de barramento, sistema de armazenamento, sistema de processamento, etc.

Então, concluímos que as Memórias RAM, e também as Memórias Cache, são aquelas que podem resolver o nosso problema com as estruturas. Assim, o Processador mantém uma pequena quantidade de dados nos registradores e, na Memória RAM/Cache, a maior parte deles. Quando trabalhamos com ARRAYS na compilação de instruções de alto/médio nível para Assembly MIPS, conforme já mencionado em artigos anteriores, devemos sempre usar as instruções LW e SW, para realizar a transferência dos dados entre os registradores e a memória RAM/Cache.

Com a restrição de alinhamento, o endereçamento em bytes acaba por afetar o índice do array, sendo necessário fazer um pequeno cálculo para que seja atribuído o valor de endereço correto ao índice. Por exemplo:

a = b + c [10]

que em Assembly MIPS é

LW $t0, 10 ($s2) **# $t0 = memória [$s2 + 10]**

LW $8, 10 ($18) **# $8 = memória [$18 + 10]**

O deslocamento correto aqui não é 10. O endereço correto é 10 * 4, pois na verdade o índice é:

a = b + c [40 / 4]

Assim garantimos que o índice correto será selecionado, isto é, o índice 10 será selecionado e não o índice 40. Em Assembly MIPS fica da seguinte forma:

LW $t0, 10 ($s2) **# $t0 = memória [$s2 + (10 * 4)] = memória [$s2 + 40]**
LW $8, 10 ($18) **# $8 = memória [$18 + (10 * 4)] = memória [$18 + 40]**

Spilled Registers, ou registradores derramados, é o termo dado ao processo de colocar as variáveis menos utilizadas na Memória. O compilador tenta manter as variáveis mais usadas nos registradores e o restante é posto, ou na Memória Cache, ou na Memória RAM e para isso são utilizadas as instruções de LW/SW.

Arrays com índice variável:

g = h + a[i];

g = $s0 h = $s1 a = $s2 i = $s3

1 forma de se fazer:

ADD $t1, $s3, $s3 # $t1 = $s3 + $s3 $t1 = i + i = 2*i

ADD $t1, $t1, $t1 # $t1 = $t1 + $t1 $t1 = 2i + 2i = 4i

ADD $t1, $t1, $s2 # $t1 = ($t1 + $s2) = (4*i + $s2) que é o mesmo que a[i]

LW $t2, 0($t1) # $t1 = a[i]

ADD $s0, $s1, $t2 # g = h + a[i]

2 forma de se fazer:

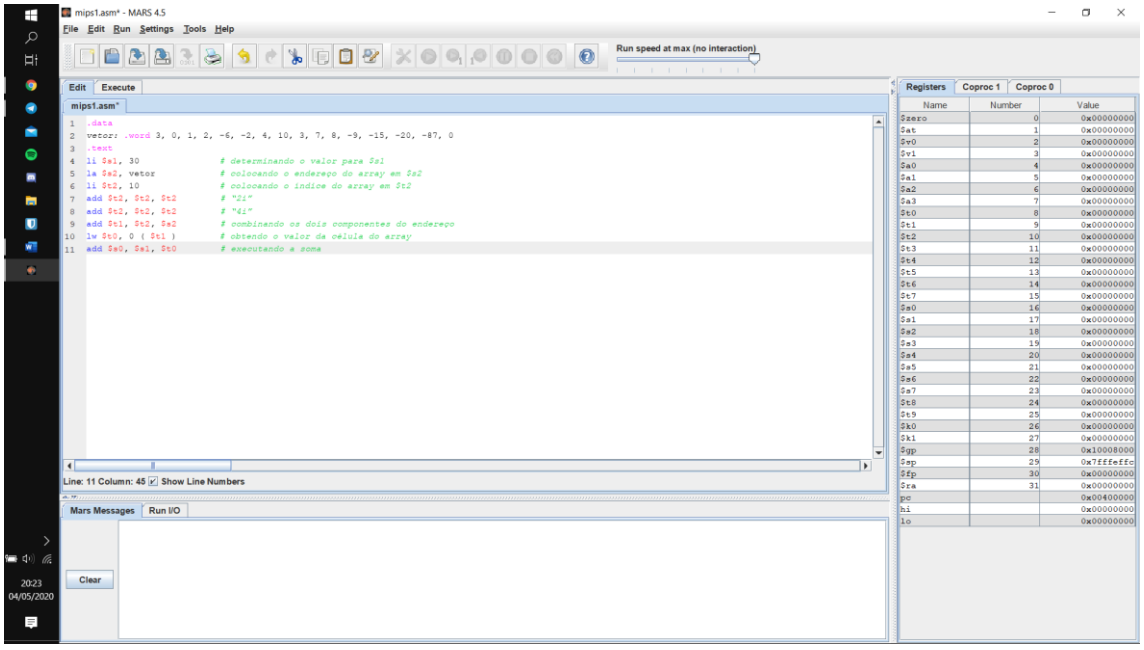
SLL $t1, $s3, 2 # $t1 = 4*i

ADD $t1, $t1, $s2 # $t1 = ($t1 + $s2) = (4*i + $s2) que é o mesmo que a[i]

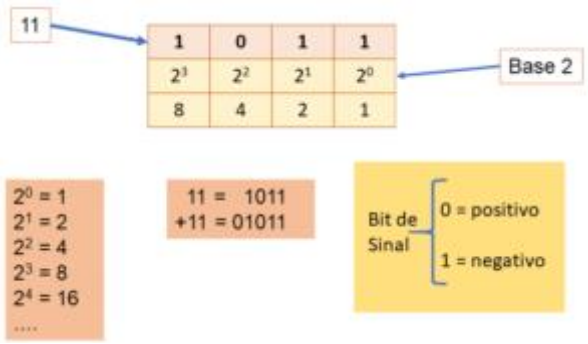
LW $t2, 0($t1) # $t1 = a[i]

ADD $s0, $s1, $t2 # g = h + a[i]

Teste no **MARS:**



O bit de sinal fica sempre no MSB (Most Significant Bit ou Bit Mais Significativo) do número, que fica mais à esquerda. LSB é o nome dado ao bit mais à direita do número, ou seja, o Least Significant Bit (Bit Menos Significativo). O MSB varia conforme a quantidade de bits que o sistema de computação trabalha. Se for 64 bits, então o bit de sinal fica no bit63, se for 32 bits, então o bit de sinal fica no bit31, e assim por diante.



a representação por Sinal e Magnitude não é utilizada na prática, em seu lugar é usado o **Complemento de 2**, mais funcional e simples.

Um número como o +11, quando o convertemos tem apenas 5 bits, isto é, ele pode ser representado com apenas 5 bits, mas uma máquina que trabalhe com 32 bits precisa de todos os 32 bits preenchidos para a manipulação do número. Assim, um número com o +11 ficaria assim:

0000 0000 0000 0000 0000 0000 0000 1011

e o -11 ficaria assim:

1111 1111 1111 1111 1111 1111 1111 0101

Onde o bit **roxo** é o bit de sinal quando o número é formado por 5 bits. O bit marcado em **azul** é o MSB, os bits em **verde** são o número propriamente dito, o **resto** disso tudo é a extensão do sinal, isto é, números 1s e 0s foram copiados do sinal (que está em roxo) para completar os 32 bits. O microprocessador agora consegue saber que esse número é positivo ou negativo analisando apenas o BIT DE SINAL, que será o bit31, sendo 0 = positivo e 1 = negativo. Em uma máquina 32 bits a representação dos números inteiros é dada de -231 até (231-1).

O **overflow** ocorre quando o hardware não é capaz de representar os números que normalmente são resultados de alguma operação aritmética.

CARRY	I	I				
+9		0	I	0	0	I
-4		I	I	I	0	0
+5	I	0	0	I	0	I
O CARRY é desconsiderado		BIT DE SINAL				

A estratégia usada aqui foi a de descartar o bit que ficou sobrando mais à esquerda, logo após o bit de sinal. Se esse bit extra for considerado, ele poderá alterar drasticamente todos os

valores, pois não corresponde à realidade. Neste exemplo o bit de sinal é zero e o bit de overflow é um, portanto, a extensão do sinal do número deve ser zero e não um. Portanto, quando precisar trabalhar de forma explícita com sinal e overflow, utilize as instruções projetadas para essas situações.

LOOPS:

compilação de um comando **while** para o MIPS

while(save[i]==k)

i +=1;

```
LOOP:    sll $t1, $s3, 2      # $t1 = 4 * i
add $t1, $t1, $s6
lw $t0, 0 ($t1)               # $t0 = save[i]
bne $t0, $s5, EXIT            # vá para EXIT se save[i] diferente de k
add $s3, $s3, 1               # $s3 = $s3 + 1      (ou i = i + 1)
j LOOP                        # volta para o LOOP (chaves)
```

compilação **switch/case:**

Cada case aqui no MIPS será tratado como um LABEL, que chamaremos de L0, L1, e assim por diante, sendo que cada um deles será correspondente a um endereço de memória, formando algo como uma Tabela de Endereços de Desvios. Não nos esqueçamos que switch/case é um comando de controle que desvia a execução do programa para o ponto desejado.

```
switch (k) {
case 0:
   f = i + j; // k = 0
   break;
case 1:
   f = g + h; // k=1
   break;
case 2:
   f = g – h; // k = 2
   break;
case 3:
   f = i – j; // k = 3
   break;
}
```

ENDEREÇO	LABEL	INSTRUÇÃO
0x0040003c	L0	f = i + j;
0x00400044	L1	f = g + h;
0x0040004c	L2	f = g – h;
0x00400054	L3	f = i – j;

Assim, precisamos criar no MIPS essa Tabela, que nada mais é que um Array:

.data

jTable: .word L0,L1,L2,L3 #jump table definition

agora precisamos atribuir o endereço base (ponteiro) dessa tabela para algum registrador, da seguinte forma:

la $t4, jTable # $t4 = base address of the jump table

é preciso verificar se K é válido, isto é, ele precisa ser igual a um dos valores presentes nos Labels, portanto, K deve estar entre 0 e 3 (4 valores diferentes). Se K não estiver dentro dessa faixa de valores, o switch não pode ser executado.

slt $t3, $s5, $zero # $t3 = ($s5 < 0)

bne $t3, $zero, EXIT # desvia para EXIT se $t3 != 0

testamos se K < 0, agora precisamos testar o outro limite, isto é, K precisa estar entre 0 e 3, portanto testa se K < 4:

slti $st3, $s5, 4 # $t3 = ($s5 < 4)

usamos agora a instrução SLTI pois precisamos comparar o valor de $s5 (K) com um imediato (4). Lembrando que o resultado da comparação entre o registrador $s5 (K) é armazenado em $t3, o qual será usado na instrução BEQ (Branch if Equal – "desvie se igual") para desviar ou não para EXIT. Se $t3 for igual a zero, então desviará para EXIT, caso contrário executará o bloco de comandos.

beq $t3, $zero, EXIT # desvia para EXIT se $t3 = 0

agora que a verificação de K está traduzida, precisamos fazer o cálculo do endereço dos labels, assim o primeiro passo é somar 4 ao endereço

sll $t1, $s5, 2 # calcula o endereçamento de 4 bytes

agora precisamos somar isso com o endereço da Tabela de Desvios ($t4):

add $t1, $t1, $t4 # $t1 será o endereço de jTable

de posse do endereço completo, podemos agora carregar o valor da Tabela. Não nos esqueçamos que a jTable começa no endereço que está em $t4, assim o endereço do desvio é carregado em um registrador temporário que neste exemplo será $t0:

lw $t0, 0($t1) # $t0 é onde está o label desejado

a execução de uma instrução de desvio para o conteúdo de um registrador faz com que o programa passe a executar a instrução apontada na tabela de endereços de desvio, assim precisamos usar a instrução JR para que o desvio ocorra:

```
jr $t0                    # desvia com base no conteúdo de $t0
```

essa instrução vai forçar o desvio para o Label escolhido, por exemplo, se o endereço é de L2, ele vai desviar para lá exatamente

```
L0:   add $s0, $s3, $s4    # Se K = 0 então f = i + j
j EXIT                     # fim deste case, desvia para EXIT
L1:   add $s0, $s1, $s2    # Se K = 1 então f = g + h
j EXIT                     # fim deste case, desvia para EXIT
L2:   sub $s0, $s1, $s2    # Se K = 2 então f = g - h
j EXIT                     # fim deste case, desvia para EXIT
L3:   sub $s0, $s3, $s4    # Se K = 3 então f = i – j
EXIT:                      # sai do Switch definitivamente
```

A instrução **LA (load address)** é uma pseudoinstrução, isso significa que ela é um tipo de instrução diferente das outras. Essas pseudoinstruções representam outas instruções e facilitam o trabalho de tradução, portanto, LA é uma combinação de duas outras instruções, LUI e ORI. A instrução LUI significa Load Upper Immediate (Carregar Superior Imediato) e é responsável por carregar a halfword (meia palavra, isto é, 16 bits) menos significativa do valor imediato na halfword mais significativa do registrador RT, sendo os bits menos significativos do registrador colocados em zero.

A instrução ORI significa OR Immediate (Ou imediato) e é responsável por colocar o OR lógico do registrador RS e o valor imediato estendido com zero no registrador RT. Outra instrução que vocês vão notar diferença é a **LI**, que também é uma pseudoinstrução representando a ADDIU (adição de imediato sem overflow), a qual é responsável por colocar a soma do registrador RS e o valor imediato com sinal estendido no registrador RT, portanto, a tradução da Linguagem de Máquina para a representação não ficará idêntica! Não se esqueça de que o registrador RT se comporta como o registrador RD (registrador destino) para algumas instruções, como é o caso das instruções envolvendo valores Imediatos.

Funções no MIPS:

$a0 até $a3: são os registradores de **argumentos** utilizados para a passagem de parâmetros;

$v0 e $v1: são os registradores de valor utilizados para o **retorno** do procedimento;

$ra: é o registrador de endereço do retorno do procedimento, utilizado na volta ao ponto de origem da chamada do procedimento.

JAL: é uma instrução de salto (jump) utilizada unicamente para os procedimentos (jump and link). Essa instrução desvia para um endereço e, ao mesmo tempo, salva o endereço da instrução seguinte no registrador de endereço de retorno. Nas palavras de Patterson: “Uma

instrução que salta para um endereço e simultaneamente salva o endereço da instrução seguinte em um registrador". Sintaxe da instrução:

jal endereco_procedimento

Exemplo:

jal fatorial

onde fatorial é o LABEL de um bloco de procedimentos e, relembrando, um LABEL é um nome dado a um **endereço de memória.**

JR: é uma instrução de salto, uma instrução de desvio incondicional para o endereço especificado em um registrador, ela volta ao endereço de retorno correto que é armazenado em $ra. Sintaxe da instrução:

jr $ra

Caller: é o programa que chama o procedimento, fornecendo os valores dos parâmetros. Faz uso dos registradores de $a0 a $a5 para armazenar os parâmetros e também de jal para pular para o procedimento.

Callee: é um procedimento que executa uma série de instruções armazenadas com base nos parâmetros fornecidos pelo Caller e depois retorna o controle para o Caller novamente.

Spilled Registers: é o processo de colocar variáveis menos utilizadas na memória (ou variáveis que serão necessárias mais adiante).

Pilha: da mesma forma que uma pilha de chamadas e retornos de procedimentos é criada para linguagens como C e Java, aqui no MIPS também faremos uso deste conceito para gerenciar os Spilled Registers. Isto é necessário pois os registradores usados pelo Caller devem ser restaurados aos seus valores anteriores a chamada do procedimento.

Stack Pointer: é um valor que indica o endereço alocado mais recentemente em uma pilha, mostrando onde devem ser localizados os valores antigos dos registradores e onde os Spilled Registers devem ser armazenados. O registrador 29 é usado para o $sp.

Push: coloca palavras para cada registrador salvo ou restaurando na pilha. Valores são colocados na pilha pela subtração do valor do stack pointer.

Pop: remove palavras da pilha. Valores são retirados da pilha pela soma do valor do stack pointer.

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

```
.text
li $a0, 15 # g = $a0 = 15
li $a1, 20 # h = $a1 = 20
li $a2, 10 # i = $a2 = 10
li $a3, 5 # j = $a3 = 5

exemplo:
addi $sp, $sp, -12
sw $t1, 8 ($sp)
sw $t0, 4 ($sp)
sw $s0, 0 ($sp)

add $t0, $a0, $a1 # (g + h) = 15 + 20 = 35 (hexa = 23)
add $t1, $a2, $a3 # (i + j) = 10 + 5 = 15 (hexa = F)
sub $s0, $t0, $t1 # (g + h ) - ( i + j) = 35 – 15 = 5 ( hexa = 5)
add $v0, $s0, $zero # f ($v0 = $s0 + 0 )

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

jr $ra
```