ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

Fernando Karchiloff Gouveia de Amorim - 10387644

Lucas Pereira Castelo Branco - 10258772

Exercício Programa - Organização de Computadores Digitais

São Paulo-SP
2018

ÍNDICE

ORGANIZAÇÃO E ARQUITETURA MIPS

HISTÓRIA

MIPS é o acrônimo para Microprocessor without Interlocked Pipeline Stages. Trata-se de uma arquitetura de microprocessadores, onde o seu conjunto de instruções se concentra em fazer uma comunicação rápida e eficiente com o processador, ao mesmo tempo que permite ao programador um nível de abstração razoável. Sua origem se dá em 1981, através de um projeto de pesquisa realizado na Universidade de Stanford, uma das mais importantes nos Estados Unidos, iniciada por John L. Hennessy.

Hennessy viria a iniciar suas pesquisas para o desenvolvimento do MIPS, onde outras duas frentes de pesquisas na área já estavam em andamento: a IBM, com o microcomputador experimental 801, num projeto liderado por John Cocke; e a Universidade de Berkeley, na Califórnia, com a iniciativa de Patterson e Séquin. É notável que, de fato, havia uma característica em comum entre os três projetos: possuíam um conjunto de instruções reduto, mas que eram de simples execução e portanto possuíam um um tempo de execução baixíssimo a partir do processador.

A esta arquitetura, na qual os três citados foram pioneiros, foi-se dado o nome de *Reduced Instruction Set Computer*, ou seja “Computadores de Conjunto de Instruções Reduzido”, normalmente chamada de RISC. Tal nomenclatura se popularizou a partir do nome dado ao projeto da Berkeley, *RISC PC I,* em 1980. Mais detalhes sobre o que é o RISC serão dados nos próximos tópicos.

O projeto de pesquisa e desenvolvimento em Stanford durou de 1981 a 1984 com o auxílio de alunos. Neste mesmo ano, a partir do que fora produzido, Hennessey fundaria a MIPS Computer Systems, com o intuito de vender a solução comercialmente. Em 1985, um ano depois, já anunciaria uma nova arquitetura de conjunto de instruções, a qual também chamou de MIPS, juntamente com o seu primeiro processador, o R2000.

A partir de então, a arquitetura comercial MIPS passaria a ser um sucesso, com constantes evoluções e variações com o passar dos anos, chegando a possuir uma versão de 64 bits (ao contrário dos seus usuais 32 bits), em 1999. Entre as várias empresas que viriam a utilizá-la, temos nomes de grande magnitude, como Cisco, Nintendo, Sony e Toshiba.

A MIPS Computer Systems viria a ser renomeada para MIPS Technologies em 1992, quando fora adquirida pela empresa Silicon Graphics, Inc., e se tornou um subgrupo. Por fim, em 13 de Junho de 2018, a MIPS Technologies viria a ser vendida novamente, desta vez para a Wave Computing, uma *startup* focada no desenvolvimento de tecnologias de Inteligência Artificial, localizada no Vale do Silício, e continua a oferecer o suporte à arquitetura que a consagrou no mercado.

O MODELO RISC

A arquitetura de processadores MIPS possui algumas peculiaridades que a fizeram se tornar tão utilizada em sistemas nos mais diversos ramos para a programação de seus microprocessadores, e assim, um sucesso comercial. A princípio, é interessante ressaltar o que define um RISC por si só, e que é a base desta arquitetura.

RISC, ou *Reduced Instruction Set Computers*, foi um modelo que, apesar de não haver um consenso sobre o seu surgimento, seu maior desenvolvimento se deu na década de 1980, em contraposição aos sistemas que estavam sendo produzidos na época (que posteriormente, viriam a serem chamados de CISC, ou seja, *Complex Instruction Set Computers*, que nada mais são que máquinas que não seguem a filosofia de arquitetura RISC). Ao fazer esse paralelo entre estes dois modelos, podemos dizer que toda estrutura RISC vale-se de três atributos pontuais: *conjunto simples e limitado de instruções com um um formato fixo;* um *número grande de registradores de uso geral;* e *ênfase na otimização do pipeline de instruções.*

Motivação

Com o barateamento dos hardwares, a produção de softwares, em contraponto, se tornou cada vez mais caro, e com isso, a confiabilidade do sistema era uma preocupação. O caminho natural fora que as pesquisas focalizaram-se em construir linguagens de programação que permitissem softwares mais complexos e mais concisos, ao mesmo tempo que tratasse mais erros. Porém, com isto, as linguagens dependiam cada vez mais de compiladores potentes, além de causar tamanhos excessivos de programas e a consequente ineficiência na sua conversão para a linguagem de máquina. A partir disto, surgiu a ideia de criar um suporte às linguagens de alto nível de forma a ser simplificada, e não sofisticada.

De acordo com Stallings(2010), diversas pesquisas foram realizadas com a intenção de identificar os fatores que impactam o processo de instruções. Podemos dizer que de forma geral, temos dois fatores principais: as *operações* e os *operandos*. Vale citar que muitas dessas pesquisas foram encabeçadas justamente por aqueles que se dispuseram a produzir as arquiteturas RISC.

No que se refere às operações, o objetivo seria identificar os tipos mais solicitados, mas mais do que isso, as operações em alto nível que se refletem em mais operações em linguagens de máquina, para permitir observar quais poderiam ser otimizados em nível de processador. A partir disso, foi-se constatado que a operação de atribuição é a mais comum, porém o gasto computacional para as operações de retorno é significativamente maior do que o número de usos.

Em outras poucas pesquisas, desta vez relacionando-se aos operandos que formam as instruções em alto nível, chegou-se ao consenso que mais de 90% das

operações não utilizam mais do que três operandos no alto nível (TANENBAUM, 1978; KATEVENIS, 1983), e que o acesso a estes operandos, ainda que poucos, eram muito recorrentes e portanto, muito impactantes para a instrução.

A partir disso, criou-se os pilares das instruções destas novas arquiteturas: primeiramente, percebeu-se que, como há uma grande solicitação de operações de atribuição, então houve uma preocupação com a forma que estes dados eram manipulados. Portanto, há o uso de *registradores*, muito disso devido ao Princípio da Localidade de Referência.

Ou seja, a escolha do uso dos registradores ao invés do uso de memórias como a *cache* tem total ligação com a velocidade de acesso, apesar de seu tamanho reduzido. Com a constatação de que a maioria significativa das variáveis são locais (PATTERSON e SEQUIN, 1982), e geralmente em número reduzido por instrução, os registradores são mais que suficientes para lidar com o número de operandos, e privilegia o acesso constante a eles.

Outro ponto está relacionado com a forma de se manipular a ordem de execução destas instruções. De acordo com Stallings (2010):

> uma atenção cuidadosa precisa ser dedicada ao projeto de pipelines de instruções. Por causa de alta proporção de instruções de desvio e chamadas de procedimentos, um pipeline de instruções direto não será eficiente. Isso se manifesta pela grande proporção de instruções que são obtidas, mas nunca são executadas (STALLINGS, 2010)

Decidiu-se, por fim, utilizar de um número reduto de instruções para realizar o suporte necessário às linguagens de alto nível. Apesar de o motivo para tal estar ligado com os fatores levantados acima, existem diversos outros fatores que o motivam a ser modelado dessa forma, que serão melhores definidos através do entendimento de como o MIPS utiliza disso.

CONJUNTO DE INSTRUÇÕES MIPS

Introdução

Como vimos, a arquitetura MIPS parte diretamente do RISC, e portanto, possui um conjunto de instruções simples e reduzido. Num primeiro momento, talvez seja interessante notar que, assim como todo o resto, o conjunto de instruções possuem algumas características.

As instruções num modelo RISC seguem um modelo, onde cada instrução deve obrigatoriamente ser *executada em um ciclo de máquina*. Dessa forma, evita-se que seja necessária algum tipo de controlador para as instruções conforme ocorrem os ciclos, como ocorrem em modelos CISC citados anteriormente.

Outra característica é a presença de *modos de endereçamento simples*. Este fator está de certa forma ligada ao fato de estarmos lidando sempre com registradores, e assim, a forma como os referenciamos tende a simplificar. Porém, é possível usarmos de outros modos de endereçamento caso necessário, desde que sejam tratados via *software* em algum ponto.

Por fim, o *formato de instruções* deve ser simples. Mais especificamente, a base estrutural das instruções costuma ser fixa, com pequenas variações no tamanho de cada campo, desde que a soma destes não ultrapasse o tamanho também fixo de cada instrução.

Assim, sabendo destes atributos gerais aos modelos RISC, é correto afirmar que o MIPS possuem estas mesmas características. Porém, isto não o resume, de fato, conforme veremos a seguir.

Registradores

Primeiramente, é importante ressaltar que, como uma derivação do que o define como pertencente a filosofia RISC, a arquitetura é baseada unicamente na manipulação de registradores, ou seja, a arquitetura MIPS segue o modelo *registrador-para-registrador*. Por isso, a forma como estes registradores são distribuídos, e mais ainda, como estes são organizados como operandos das instruções é importante para o seu funcionamento.

A arquitetura MIPS possui 32 registradores básicos, numerados de 0 a 31 dentro do sistema. Destes, 31 são modificáveis durante a execução, sendo alguns acessíveis ao programador, enquanto um deles (zero) é uma constante que representa o valor zero. Vale dizer que em nível de implementação, toda referência a um desses registradores deve ser precedido por um cifrão ($). Todos os registradores citados são de “propósito geral”, porém, existe uma subdivisão destes registradores, de forma a terem determinada funcionalidade ligada a ele. A divisão destes registradores é algo exclusivo do MIPS, apesar de não haver, de fato, uma limitação do uso destes do ponto de vista prático. De acordo com Ellard (1994):

> Even though any of the registers can theoretically be used for any purpose, MIPS programmers have agreed upon a set of guidelines that specify how each of the registers should be used. Programmers (and compilers) know that as long as they follow these guidelines, their code will work properly with other MIPS code. (ELLARD, 1994)

Assim, os registradores podem ser divididos em grupos de funções específicas para facilitar a divisão de responsabilidades dentro do programa, e dessa forma, padronizar o seu uso pelos programadores. Podemos dividi-los nas seguintes categorias:

- **at:** registrador reservado unicamente para o uso do montador da linguagem, e portanto, não pode ter seu valor adulterador por chamadas do programador.
- **Registradores de Parâmetro**: numerados de $a0 a $a3, são os registradores usados para passar os quatro principais argumentos a uma rotina específica determinada.
- **Registradores de Resultado:** usados unicamente para retornar os valores das funções. São chamados por $v0 e $v1.
- **Registradores Temporários:** representados de $t0 a $t9, são aqueles que são também dito *salvos pelo caller*, ou seja, na qual sua funcionalidade é temporária, e não precisam ser preservadas entre as chamadas.
- **Registradores salvos:** também chamados de *salvos pelo callee*, são os que mantêm valores de longa duração, e portanto devem ser preservadas entre as chamadas feitas.
- **Registradores de Kernel**: reservadas para o sistema operacional, também não podem ser acessados e modificado pelo programador.
- **gp:** sigla para “global pointer”, nada mais é que um ponteiro que aponta para o meio de um bloco de dados estático de tamanho 64K.
- **sp:** siga para “stack pointer”, aponta o último local (word) passado da pilha. Normalmente, usamos a pilha para passar mais argumentos caso necessário.
- **fp:** sigla para “frame pointer”, é aquele que aponta para a primeira parte (*word*) da pilha. A distância entre o $fp e o $sp nada mais é que o espaço de memória reservado para a pilha.
- **ra:** sigla para “endereço de retorno”. Como o nome sugere, é responsável por salvar o endereço da instrução que será executada após a execução de um dado procedimento ou subrotina.

A organização destes registradores, seus números dentro do sistema e suas respectivas funções assimiladas pode ser claramente vista através da tabela abaixo (ELLARD, 2005)

Symbolic Name	Number	Usage
zero	0	Constant 0.
at	1	Reserved for the assembler.
v0 - v1	2 - 3	Result Registers.
a0 - a3	4 - 7	Argument Registers $1 \cdots 4$.
t0 - t9	8 - 15, 24 - 25	Temporary Registers $0 \cdots 9$.
s0 - s7	16 - 23	Saved Registers $0 \cdots 7$.
k0 - k1	26 - 27	Kernel Registers $0 \cdots 1$.
gp	28	Global Data Pointer.
sp	29	Stack Pointer.
fp	30	Frame Pointer.
ra	31	Return Address.

Estrutura da instrução

Uma vez definido os registradores, é mais simples de entendermos a estrutura geral que forma o conjunto de instruções disponível. Toda instrução MIPS possui exatamente 32 bits de tamanho. Destes, sendo 6 bits reservados para o *operation code*, que define qual tipo de instrução será realizada. Dessa forma, as variações encontradas são em como os operandos necessários estão divididos entre os 26 bits remanescentes.

A organização dos operandos dentro dele importa. No caso do MIPS, toda instrução segue o modelo *little-endian,* o que significa que, quanto menor o índice do bit, menos significante ele será. Portanto, os “últimos” bits são os mais importantes para a instrução.

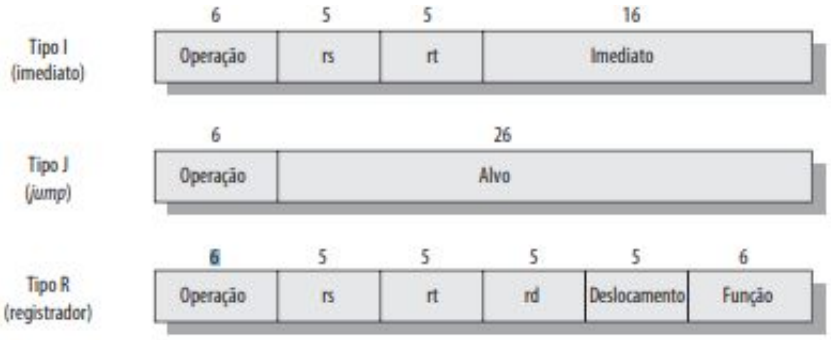
Comumente, as instruções do MIPS apresentam três *operandos* como parâmetros, apesar de esse número variar, como veremos a seguir. Este operandos podem ser divididos como os registradores de *origem* e os registradores de *destino.*

Podemos enquadrar qualquer instrução da arquitetura, de acordo com um dos três tipos possíveis:

I. instrução de *registro*: operações lógicas e/ou aritméticas que manipulam registradores. Neste modelo, o *opcode* deve ser 0, e o código que define a função se encontra nos últimos 6 bits da instrução. Neste formato, há a presença de um campo de *deslocamento*, que pode ser dinamicamente utilizado.
II. instrução *imediata*: ocorre quando há uma necessidade de um deslocamento ou desvio direto, ou alguma operação que referencia um endereço a ser acessado. Neste caso, os últimos 16 bits são reservados para tal.

III. instrução de *salto*: o tipo mais simples, onde todos os 26 bits são usados para passar um endereço de referência para onde o *jump* deve ser realizado.

Podemos visualizar as divisões dos 32 bits da instrução em cada um dos tipos na figura abaixo:



Operação = código da operação
rs = especificador de registrador de origem
rt = especificador de registrador de origem/destino
Imediato = deslocamento imediato, desvio ou endereço
Alvo = endereço alvo do salto
rd = especificador de registrador de destino
Deslocamento = quantidade de deslocamento
Função = especificador de função de ALU/deslocamento

Tipos de Instruções

Como já visto, as instruções que podem ser chamadas no MIPS podem variar em sua estrutura base de acordo com a necessidade. Mas mais do que isso, existem subdivisões de instruções que valem ser citados como os grupos que definem as funções suportadas pela arquitetura.

Antes de tudo, é importante entender que há duas formas de instrução: as *instruções nativas* são aquelas que são “absolutas”, ou seja, representam uma exata instrução para ser convertida para a linguagem de máquina e realizar uma operação; e há as *pseudo-instruções.* Apesar de o nome, elas são instruções válidas, porém são traduzidas pelo montador em questão para que esta instrução se reflita em uma sequência de instruções nativas, de forma a simplificar a chamada de tais operações pelo programador.

Os tipos operações das instruções disponíveis são:

Instruções Aritméticas

Instrução	Descrição
abs *rd, r1*	O registrador *rd* passou o valor absoluto de *r1*
add *rd, r1, r2*	O registrador *rd* recebe o resultado da soma entre os valores dos registradores *r1* e *r2,* que pode ser tanto um registrador quanto um inteiro imediato de 32-bits.
div *rd, r1, r2*	O registrador *rd* recebe o resultado da divisão entre os valores dos registradores *r1* e *r2,* que pode ser tanto um registrador quanto um inteiro imediato de 32-bits.
mul *rd, r1, r2*	O registrador *rd* recebe o resultado da multiplicação entre os valores dos registradores *r1* e *r2*, que pode ser tanto um registrador quanto um inteiro imediato de 32-bits.
neg *rd, r1*	O registrador *rd* recebe o resultado do negativo do valor do registrador *r1*, que pode ser tanto um registrador quanto um inteiro imediato de 32-bits.
nor *rd, r1, r2*	O registrador *rd* recebe o resultado da operação lógica **nor** entre os valores dos registradores *r1* e *r2*, que pode tanto ser um registrador quanto um inteiro imediato de 32-bits.
or *rd, r1, r2*	O registrador *rd* recebe o resultado da operação lógica **or** entre os valores dos registradores *r1* e *r2*, que pode tanto ser um registrador quanto um inteiro imediato de 32-bits.
sll *rd, r1, r2*	O registrador rd recebe o valor do registrador *r1* à esquerda por *r2* bits, que pode tanto ser um registrador quanto um inteiro imediato de 32-bits.

sra *rd, r1, r2*	O registrador *rd* recebe o valor do registrador *r1* deslocado por *r2* bits aritméticamente, que pode ser tanto um registrador quanto um inteiro imediato de 32-bits.
srl *rd, r1, r2*	O registrador *rd* recebe o resultado do deslocamento entre os valores dos registradores *r1* e *r2,* que pode ser tanto um registrador quanto um inteiro imediato de 32-bits.
sub *rd, r1, r2*	O registrador *rd* recebe o resultado da subtração entre os valores dos registradores *r1* e *r2*, que pode ser tanto um registrador quanto um inteiro imediato de 32-bits.
xor *rd, r1, r2*	O registrador *rd* recebe o resultado da operação lógica exclusiva **or** entre os valores dos registradores *r1* e *r2*, que pode ser tanto um registrador quanto um inteiro imediato de 32-bits.

Instruções Condicionais

Instrução	Descrição
seq *rd, r1, r2*	O registrador *rd* recebe o resultado da comparação entre *r1* e *r2*, caso sejam iguais, recebe 1, caso contrário 0. Os registradores *r1* e *r2*, que pode ser um inteiro imediato de 32-bits.
sne *rd, r1, r2*	O registrador *rd* recebe o resultado da comparação entre *r1* e *r2*, caso sejam diferentes, recebe 1, caso contrário 0. Os registradores *r1* e *r2*, que pode ser um inteiro imediato de 32-bits.
sge *rd, r1, r2*	O registrador *rd* recebe o resultado da comparação entre *r1* e *r2*, caso *r1* seja maior ou igual a *r2*, recebe 1, caso contrário 0. Os registradores *r1* e *r2*, que pode ser um inteiro imediato de 32-bits.

sgt *rd, r1, r2*	O registrador *rd* recebe o resultado da comparação entre *r1* e *r2*, caso *r1* seja maior que *r2*, recebe 1, caso contrário 0. Os registradores *r1* e *r2*, que pode ser um inteiro imediato de 32-bits.
sle *rd, r1, r2*	O registrador *rd* recebe o resultado da comparação entre *r1* e *r2*, caso *r1* seja menor ou igual a *r2*, recebe 1, caso contrário 0. Os registradores *r1* e *r2*, que pode ser um inteiro imediato de 32-bits.
slt *rd, r1, r2*	O registrador *rd* recebe o resultado da comparação entre *r1* e *r2*, caso *r1* seja menor que *r2*, recebe 1, caso contrário 0. Os registradores *r1* e *r2*, que pode ser um inteiro imediato de 32-bits.

Instruções de Desvio

Instrução	Descrição
b *label*	Executa um desvio incondicional para o *label*.
beq *r1, r2, label*	Executa um desvio condicional para o *label* caso o registrador *r1* seja igual ao registrador *r2*, que pode ser um inteiro imediato de 32-bits.
bne *r1, r2, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja diferente do valor do registrador *r2*, que pode ser um inteiro imediato de 32-bits.
bge *r1, r2, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja maior ou igual ao valor do registrador *r2*, que pode ser um inteiro imediato de 32-bits.
bgt *r1, r2, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja maior que do valor do registrador *r2*, que pode ser um inteiro imediato de 32-bits.

ble *r1, r2, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja menor ou igual ao valor do registrador *r2*, que pode ser um inteiro imediato de 32-bits.
blt *r1, r2, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja menor que do valor do registrador *r2*, que pode ser um inteiro imediato de 32-bits.
beqz *r1, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja igual à 0.
bnez *r1, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja diferente de 0.
bgez *r1, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja maior ou igual à 0.
bgtz *r1, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja maior que 0.
blez *r1, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja menor ou igual à 0.
bltz *r1, label*	Executa um desvio condicional para o *label* caso o valor do registrador *r1* seja menor que 0.
bgezal *r1, label*	Executa um desvio condicional para o *label* salvando o endereço da próxima instrução no registrador *ra* caso o registrador *r1* seja maior ou igual à 0.
bgtzal *r1, label*	Executa um desvio condicional para o *label* salvando o endereço da próxima instrução no registrador *ra* caso o registrador *r1* seja maior do que 0.
bltzal *r1, label*	Executa um desvio condicional para o *label* salvando o endereço da próxima instrução no registrador *ra* caso o registrador *r1* seja menor do que 0.

Instruções de Salto

Instrução	Descrição
j *label*	Executa um salto para o *label*.
jr *ra*	Executa um salto para a instrução contida no registrador *ra*.
jal *label*	Executa um salto para o *label*, e salva o endereço da próxima instrução no registrador *ra*.
jalr *r1*	Executa um salto para o endereço no registrador *r1* e salva o endereço da próxima instrução no registrador *ra*.

Formato de endereçamento

As instruções de carregamento, armazenamento e movimentação de dados exigem suportam algumas formas de endereçamento:

Formato	Descrição
(reg)	Conteúdo do registrador *reg.*
const	Constante de um endereço.
const(reg)	A constante de um endereço pegando seu conteúdo.
symbol	O endereço de um dado símbolo.
symbol+const	O endereço de um dado símbolo acrescido de uma constante
symbol+const(reg)	O endereço de um dado símbolo acrescido da constante de um endereço, pegando seu conteúdo

Instruções de Carregamento

Instrução	Descrição

la *rd, addr*	Carrega o endereço de uma label.
lb *rd, addr*	Carrega no registrador *rd* um dado *byte* presente em *addr*
lh *rd, addr*	Carrega no registrador *rd* uma *halfword* presente em *addr*
li *rd, const*	Carrega no registrador *rd* uma constante *const*
lw *des, addr*	Carrega no registrador *rd* uma *word* presente em *addr*
ulw *des, addr*	Carrega no registrador *rd* uma *word (*talvez desalinhada) iniciada em *addr*

Instruções de Armazenamento

Instrução	Descrição
sb *r1, addr*	Armazena no registrador *rd* um dado *byte* presente em *addr*
sh *r1, addr*	Armazena no registrador *rd* uma *halfword* presente em *addr*
sw *r1, addr*	Armazena no registrador *rd* uma *word* presente em *addr*
usw *r1, addr*	Armazena no registrador *rd* uma *word (*talvez desalinhada) iniciada em *addr*

Instruções de Movimentação de Dados

Instrução	Descrição
move *rd, r1*	Copia o conteúdo do registrador *r1* para o registrador *rd*

Instruções de Tratamento de Exceção

Instrução	Descrição
rfe	Retorna de uma exceção
syscall	Faz uma chamada de sistema
break *const*	Usado pelo debugador para interromper a execução
nop	Instrução sem efeito, para que um ciclo seja executado.

Chamadas de Sistema

A instrução de chamada de sistema *syscall* é uma das mais importantes para que seja possível manipular dados, além de controlar a execução do programa. Seu parâmetro é passado através do registrador $v0. Os possíveis valores são:

Serviço	Código	Argumento	Resultado
print_int	1	$a0	nenhum
print_float	2	$f12	nenhum
print_double	3	$f12	nenhum
print_string	4	$a0	nenhum
read_int	5	nenhum	$v0
read_float	6	nenhum	$v0
read_double	7	nenhum	$f0
read_string	8	$a0 (endereço), $a1 (largura)	nenhum
sbrk	9	$a0 (largura)	$v0
exit	10	nenhum	nenhum

SISTEMA DE PIPELINING DAS INSTRUÇÕES

Através das seções passadas, pudemos observar como a estrutura da arquitetura MIPS preza por alguns padrões. Talvez não ficasse clara a motivação disso olhando do ponto de vista do programador, porém tudo é motivado pela questão da execução deste programa. Por isso, o MIPS utiliza de um sistema chamado de *pipelining* (que inclusive, o cunha em sua sigla na letra “P”).

De acordo com Patterson (2005), “*Pipelining* é uma técnica que explora o paralelismo entre as instruções em um fluxos de instruções sequenciais”. Ou seja, através deste sistema, é possível executarmos diferentes partes das instruções em um mesmo ciclo, de forma que otimizamos o tempo de execução. É importante notar que esta técnica não diminui o tempo de execução de uma instrução, mas sim diminui o *vazão* de tempo de execução entre elas.

Motivação

É interessante entender que a arquitetura MIPS, de fato, possui diversos fatores que a torna suscetível a uma implementação eficiente de *pipelining,* e por isso é importante que entendamos o impacto disto em seu uso.

Primeiramente, todas as instruções possuem o mesmo tamanho. Por isso, é simples delimitar as instruções, de forma que cada estágio do *pipeline* pode ser executado dentro de uma mesma margem. A presença de poucos formatos de instrução (que como já visto, são basicamente três) também são úteis, pois é fácil de saber a localização do operando que referencia o operando de origem, por exemplo, ao descobrir-se de qual tipo estamos usando.

Além disso, a questão envolvendo a memória é importante para o funcionamento do *pipelining.* O fato de que apenas dois tipos de processos utilizam de operandos em memória (no caso, *load* e *store*). então, isto permite diminuirmos os estágios relacionados a isto, como um eventual estágio de obter o endereço, por exemplo. Por fim, a arquitetura MIPS prevê que os operandos precisam estar alinhados na memória. Dessa forma, sua busca é muito mais eficiente, e pode ser realizada em um único estágio, ao invés de realizar a busca de cada operando separadamente.

Estágios do Pipeline

No contexto de *pipeline*, chamamos de estágio a execução de uma etapa que poderá ser executada paralelamente conforme a sua instrução atual passar de estágio. Ou seja, quando dizemos que uma dada arquitetura possui três estágios, isto quer dizer que é possível executarmos três instruções paralelamente, desde que estas estejam em tempos diferentes.

No caso da arquitetura MIPS, a forma como os estágios se organizam evoluíram com o tempo, permitindo assim que mais estágios fossem adicionados, o que significou um maior paralelismo na execução de múltiplas instruções. Porém, é correto afirmar que existem *cinco estágios principais*, que podemos dizer que são generalizáveis para quaisquer arquitetura RISC, e de onde nos baseamos para nomeá-los (WEATHERSPOON, 2012)*:*

1. **Busca de Instrução (Instruction Fetch - IF):** Parte responsável por obter a instrução lida da memória através de PC (Program Counter). Após esse processo, o PC deve ser devidamente incrementado.
2. **Decodificação de Instrução (Instruction Decoder - ID):** De maneira geral, obtém a instrução e busca pelos seus operandos no arquivo de registradores (que como já vimos, será realizado em apenas uma etapa graças a estrutura proporcionada).
3. **Execução (EX):** etapa na qual, após obter os operandos necessários, são enviados para o ALU (ou Unidade Lógico-Aritmética) caso seja necessária alguma operação lógica ou aritmética. Neste mesmo estágio, é possível que seja gerado novos endereços de operando de dados, em casos de operações como um *branch,* por exemplo.
4. **Acesso a Memória (MEM):** realiza o acesso a memória de dados, caso seja necessário uma operação (por exemplo, caso seja necessário um *store* para um dado que fora manipulado).
5. **Reescrita (RW):** O resultado das operações anteriores é escrito no *register file,* ou banco de registradores.

Registradores de pipeline

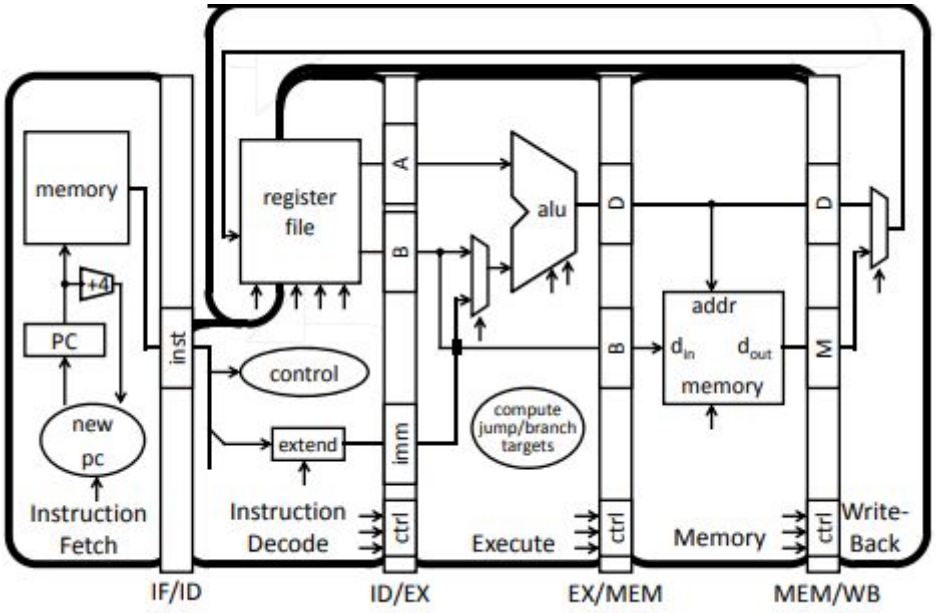
Como já vimos, a arquitetura MIPS prevê 32 registradores de uso geral para o programador durante a execução de suas rotinas. Porém, é importante dizer que, durante um dado estágio na execução, alguns dados são necessárias enquanto outros são gerados. Esses dados devem ser passados de um estágio para outro. Por isso, existe a presença de registradores entre esses estágios que permitem essa comunicação. São eles:

- **IF/ID:** registrador que intermedia a ligação entre a busca de instrução e a decodificação de instrução. Neste ponto, são passados *bits* de instrução que podem servir para a decodificação, além de passar a referência de PC+4, para eventuais casos de *branch*.
- **ID/EX:** registrador que intermedia a ligação entre a decodificação de instrução e a execução da instrução em si. Nesta etapa, passa-se informações de controles, imediatos, etc., sendo o principal os

conteúdos dos registradores a serem manipulados pelo ULA. Também deve passar a referência de PC+4.

- **EX/MEM:** intermediário entre a execução e o acesso a memória. Pode guardar informações de controle, mas principalmente, o resultado da eventual operação em ALU. Pode passar o valor em caso de termos uma instrução de acesso a memória.
- **MEM/WB:** guarda informações entre a operação de memória e a reescrita. Neste caso, além das informações de controle, o registrador guarda o resultado das operações de memória e da ALU, caso necessário.

O processo de execução dos estágios, e sua relação com os registradores citados pode ser observado na imagem abaixo:



DESCRIÇÃO DO PROBLEMA E CÓDIGO ALTO NÍVEL DA SOLUÇÃO

O problema abordado pela dupla é o de número 10, baseando-se na relação encontrada no arquivo “Lista de Problemas para o Exercício Programa - MIPS”. O problema é descrito da seguinte forma:

> (a) Escreva uma função com o protótipo void converte (char ch, int *tipo, char *valor); que recebe um caractere ch e devolve em *tipo 0, se o caractere for um número inteiro, 1 se for uma letra (maiúscula ou minúscula) e 2 caso contrário; e além disso, no caso de ser uma letra, converte para maiúscula, senão devolve ch inalterado.

> (b) Escreva um programa que leia uma seqüência de n caracteres e imprima a seqüência convertida para maiúscula, eliminando os caracteres que não forem letras ou números.

Portanto, o problema em questão se baseia em coletar e manipular caracteres, onde é necessário que seja identificado o tipo específico que está sendo representado em cada caractere. Perceba que, para que o problema *(b)* seja resolvido, é necessário que seja conhecida uma forma de implementação correta de *(a)*, visto que é necessário uma identificação desta sequência de caracteres para que sejam feitas as devidas alterações.

A implementação em alto nível deste problema pode ser realizado em diversas linguagens. Optou-se pelo uso da linguagem C, visto que a assinatura do programa necessária para a resolução da letra *(a)* explicite o uso de ponteiros, e portanto, o uso de tal linguagem é adequada.

Podemos dizer que existem duas formas distintas de escrever o programa que soluciona este problema em C, partindo do princípio que o ponto focal seria o tratamento acerca dos *tipos de caracteres*, conforme descrito na letra *(a)*:

1. Através da *biblioteca* <*ctype.h*>, é possível fazer toda operação relacionado aos variados tipos de caracteres possíveis, através de funções pré-determinadas pelo C. Graças a essa biblioteca, conseguimos verificar seu tipo com os vários métodos que testam cada um dos possíveis, retornando um booleano, além de permitir a conversão destes caracteres, como é exigido no problema estipulado. Dessa forma, teremos o seguinte método:

```c
void converte(char ch, int *tipo, char *valor){
    //converte de char para inteiro
    int c = (int) ch;
    //garante que valor receba o valor de ch
    *valor = ch;
    //verifica se é alfanumérico
    if(isalnum(c)){
        if(isdigit(c)){
            //é um número inteiro, logo retorna o tipo 0
            *tipo = 0;
            return;
        }else{
            //é uma letra, logo, retorna o tipo 1
            *tipo = 1;
            if(islower(c)){
                *valor = toupper(c);
            }
            return;
        }
    }else{
        //não é um número nem uma letra, logo, retorna o tipo 2
        *tipo = 2;
        return;
    }
}
```

2. Através da *tabela ASCII:* quando estamos lidando com caracteres ocidentais, estamos falando de uma representação de *bits* baseado num padrão chamado ASCII (derivada da sigla em inglês, *American Standard Code for Information Interchange*, significando em tradução livre, "Código Padrão Americano para o Intercâmbio de Informação"). Neste modelo, cada caractere é uma representado por 7 bits, e portanto possuem um valor decimal atrelado a tal. Para sabermos o que simboliza cada um dos 127 valores possíveis, é comum o uso de uma tabela ASCII. É interessante notar que, para fazermos a distinção necessária, devemos observar que para cada tipo de caractere, neste caso, se dá por um intervalo de valores.

Tabela ASCII:

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Assim, para este caso usando a tabela, o algoritmo será:

A função se utiliza dos intervalos para verificar se o caráter (transformado em inteiro) pertence à certos intervalos determinados que delimitam letras, número e especiais. De 48 a 57 para números inteiros, 65 a 90 para letras maiúsculas e 97 a 122 para minúsculas e converte caso sejam letras minúsculas para maiúsculas.

```
void converte(char ch, int *tipo, char *valor){
    /* O tipo 'char' eh na verdade uma representacao de um
       inteiro, transformado para caracter, exemplos:
       57 -> '9'
       48 -> '0'
       97 -> 'a'*/
    int caracter = (int) ch;

    //if(ch) inteiro retorna tipo 0
    if(caracter >= 48 && caracter <= 57){ //Inteiro
        *tipo = 0;
        *valor = (char) caracter;
    }
    else{
        //if(ch) letra(M ou m) retorna tipo 1
        /*
            No caso de ser letra, converte para maiuscula,
            se nao devolve inalterado. */
        if((caracter >= 65 && caracter <= 90) || ( //Letra
            caracter >= 97 && caracter <= 122)){

            *tipo = 1;
            if(caracter >= 97 && caracter <= 122){ //Se for minuscula
                /*
                    32 eh o valor correspondente na Tabela ASCII
                    para transformar de minuscula para maiuscula
                */
                caracter = caracter - 32;
            }
            *valor = (char)caracter;
        }
        else{   //Outra coisa.
            //else retorna tipo 2
            *tipo = 2;
            *valor = ch;
        }

    }
}
```

A segunda parte parte do problema é aplicar a função implementada na letra *(a)* e transformar uma *string* ou *array* passado como parâmetro com tamanho *n*, de forma a eliminar caracteres que não forem letras ou números durante a sua impressão. Para obter o tamanho da *string* foi utilizado a biblioteca <*string.h*>, que permite o uso do método strlen(). Foi utilizado um loop aplicando a função da letra *(a)* para que fosse resolvido o problema. Assim para este caso o algoritmo será:

```
void imprimeSequencia(char array[]){
    /*
        Le uma sequencia de n caracteres e imprime
        a sequencia convertida para maiscula, eliminando
        caracteres que nao forem letras ou numeros.
    */
    printf("Imprime sequencia.\n");
    int n = strlen(array);  //uso de string requer <string.h>
    //Caso seja um array de caracteres separados, pode se usar sizeof().
    int tipo;
    char valor;
    for(int i=0 ;i < n; i++){
        converte(array[i],&tipo,&valor);  //Converte
        if(tipo == 1 || tipo == 0){  //Se for letra, imprime (ja convertido)
            printf("%c",valor);
        }
    }
}
```

CÓDIGO EM ASSEMBLY DESENVOLVIDO

```
#Problema 10 - Conversao de caracteres e impressao de uma sequencia de caracteres sendo convertido
#Dupla: Lucas Pereira e Fernando Karchiloff
#
#Tipos de registradores utilizados:
#       $a0   - recebe caracter a ser convertido
#       $a1   - contador que incrementa ao percorrer a string
#       $a2   - armazena o tamanho da string
#       $t0   - usado para os testes logicos
#       $t1   - usada para os testes logicos
#       $t2   - recebe o tipo do caracter  (0, 1, 2)
#       $t3   - recebe o byte do caracter
```

```
.text
.globl main
main:
    li $a1,0      #carrega contador apartir do 0
    lb $a2,$tam  #carrega o tamanho da string
    j imprime_sequencia   #vai para a funcao de imprimir sequencia

imprime_sequencia:   #funcao que imprime uma sequencia de caracteres (convertendo)
    lb $t3,$sequencia($a1)      #pega o byte segundo o indice apresentado
    move $a0, $t3               #passa o caracter para o $a0 como argumento
    addi $a1,$a1,1               #adiciona 1 no contador
    jal converte                    #funcao de conversao
    jal printa_caracter      #funcao de printar caracter
    blt $a1,$a2, imprime_sequencia   #se a1 menor que a2 faz loop
    j fim_programa            #chama o encerramento do programa

fim_programa:   #funcao que imprime uma sequencia de caracteres
    li $v0, 10    #codigo para sair do programa
    syscall   #chama o sistema operacional
```

```
converte:  #funcao que converte um unico caracter

    #numero
    sgeu $t0, $a0, 48   #intervalo de ascii
    sleu $t1, $a0, 57   #intervalo de ascii
    and $t0, $t0, $t1   #se esta dentro desse intervalo  $t0 = 1 / c.c. $t0 = 0
    beq $t0, 1, numero  #se eh numero entra na funcao

    #letra maiuscula
    sgeu $t0, $a0, 65   #intervalo de ascii
    sleu $t1, $a0, 90   #intervalo de ascii
    and $t0, $t0, $t1   #se esta dentro desse intervalo $t0 = 1 / c.c. $t0 = 0
    beq $t0, 1, letra_maiuscula  #se eh letra maiuscula entra na funcao

    #letra minuscula
    sgeu $t0, $a0, 97     #intervalo de ascii
    sleu $t1, $a0, 122    #intervalo de ascii
    and $t0, $t0, $t1     #se entra dentro desse intervalo $t0 = 1 / c.c. $t0 = 0
    beq $t0, 1, letra_minuscula  #se eh letra minuscula entra na funcao

    #se passou direto entra para caracter especial
    j carac_especial  #funcao de caracter especial

converte_retorno:   #funcao que entra para dar jump ao endereco de retorno
    jr $ra     #da jump para a instrucao onde o registro de retorno aponta
```

```
printa_caracter:   #funcao que faz o print dos caracteres
    beq $t2, 2, printa_carac_retorno #se nao for letra ou numero, ja retorna sem printar
    li $v0, 11   #printa caracter
    syscall    #chama o sistema para fazer a operacao
    j printa_carac_retorno   #chama o retorno ao final para voltar ao loop

printa_carac_retorno:  #funcao que entra para dar jump ao endereco de retorno
    jr $ra     #da jump para a instrucao onde o registro de retorno aponta

numero:      #funcao que coloca o tipo do caracter no registrador $t2
    li $t2, 0    #coloca 0 em $t2 para numeros
    j converte_retorno     #chama o retorno da funcao de conversao

letra_minuscula:    #funcao que coloca o tipo do caracter no registrador $t2 e converte em maiuscula
    li $t2, 1     #coloca 1 em $t2 para letras
    sub $a0, $a0, 32   #transforma para maiuscula subtraindo 32
    j converte_retorno   #chama o retorno da funcao de conversao

letra_maiuscula:   #funcao que coloca o tipo do caracter no registrador $t2
    li $t2, 1    #coloca 1 em $t2 para letras
    j converte_retorno   #chama o retorno da funcao de conversao

carac_especial:    #funcao que coloca o tipo do caracter no registrador $t2
    li $t2, 2    #coloca 2 em $t2 para caracteres especiais
    j converte_retorno    #chama o retorno da funcao de conversao

.data
#deve ser especificado o tamanho da sequencia de caracteres em .asciiz e a sequencia em si
$tam: .word 14        #tamanho da sequencia de caracteres
$sequencia: .asciiz "C0oNv3eRS 4Ao"
```

EXPLICAÇÃO DETALHADA DAS INSTRUÇÕES UTILIZADAS NO CÓDIGO FEITO PELA EQUIPE

Nesta parte do trabalho será dada uma explicação para cada instrução utilizada no código feito pela equipe em Assembly na arquitetura MIPS, será feito um detalhamento em como a instrução é executada em micro-operações no sistema.

Antes de qualquer instrução ser executada, o processador busca a instrução e a decodifica para saber o que ele deverá fazer para executá-la, isso envolve acessar a memória através de um ciclo para que ele sempre tenha a próxima instrução a executar, só sabendo qual instrução ele deve executar que pode começar o procedimento de execução para a instrução.

Instruções utilizadas no código:

(Legenda: **R** é um registrador; **mem** é um endereço na memória; **target** é uma label de instrução(nome de função); **X** é uma constante numérica; **|** (barra em pé) significa OU; **I** é o tamanho de uma instrução)

- **li R, X** - Load Immediate
- **lb R, mem** - Load Byte
- **j target** - Jump Unconditionally
- **move R1, R2** - Move
- **addi R1, R2, X** - Add Immediate
- **jal target** - Jump and Link
- **blt R1, X | R2, target** - Branch Less than

- **sgeu R1, R2, X | R3** - Set Greater or Equal Unsigned
- **sleu R1, R2, X | R3** - Set Less or Equal Unsigned
- **and R1, R2, X | R3** - Bitwise AND
- **beq R1, X | R2, target** - Branch if Equal
- **sub R1, R2, X | R3** - Subtraction
- **jr R** - Jump Register

LOAD IMMEDIATE - (Carregamento imediato)

Formato da instrução: **li R, X**

Carrega o valor de uma constante X de 16 bits (*signed* ou *unsigned*) ou 32 bits para um registrador R especificado na instrução como operando.

Instrução em micro-operações:

t1: R ← X

t2: PC ← PC + I

LOAD BYTE - (Carregamento de byte)

Formato da instrução: **lb R, mem**

Carrega o valor de um byte de um endereço de memória como operando (podendo ser direto ou indireto) e coloca no registrador R especificado na instrução como operando.

Instrução em micro-operações:

t1: MAR ← (IR(Endereço))

t2: MBR ← Memória

t3: R ← MBR

JUMP UNCONDITIONALLY - (Pulo incondicional)

Formato da instrução: **j target**

Modifica o fluxo de execução do programa para a instrução especificada no operando.

Instrução em micro-operações:

t1: MAR ← (IR(Endereço))

t2: MBR ← Memória

t3: PC ← MBR

MOVE - (Mover)

Formato da instrução: **move R1, R2**

Copia o valor de um registrador para outro especificado nos operandos.

Instrução em micro-operações:

t1: R1 ← R2

t2: PC ← PC + I

ADD IMMEDIATE - (Adição imediata)

Formato da instrução: **addi R1, R2, X**

Faz a adição de uma constante X com um registrador R2 e coloca no registrador R1.

Instrução em micro-operações:

t1: R1 ← (R2) + (R)

t2: PC ← PC + I

JUMP AND LINK - (Pulo e ligação)

Formato da instrução: **jal target**

Salva o endereço de retorno no registrador RA por padrão e vai para a endereço especificado pela target.

Instrução em micro-operações:

t1: RA ← PC + I

t2: PC ← PC + TARGET + I

BRANCH NOT EQUAL - (Desvio se não igual)

Formato da instrução: **bne R_{aux}, R1, target**

Faz um desvio condicional caso R_{aux} seja diferente de R1 para o *target* e só continua caso seja igual.

Instrução em micro-operações:

Se for constante:

t1: PC ← PC + I

t2: R_{aux} ← X

t3: IF (R_{aux} != R1):

PC ← PC + TARGET + I

ELSE:

PC ← PC + I

BRANCH LESS THAN - (Desvio se menor que)

Formato da instrução: **blt R1, X | R2, target**

Faz um desvio caso registrador R1 seja menor que a constante X ou o conteúdo do registrador R2, para o endereço alvo target usando a instrução com o registrador reservado R0.

Instrução em micro-operações:

Se for constante:

t1: R_{aux} ← X
t2: R_{aux} ← (R1) < (R_{aux})
t3: IF (R_{aux} != R0):
PC ← PC + TARGET + I
ELSE:
PC ← PC + I

Caso contrário:
t1: R_{aux} ← (R1) < (R2)
t2: IF (R_{aux} != R0):
PC ← PC + TARGET + I
ELSE:
PC ← PC + I

SET GREATER OR EQUAL UNSIGNED - (Coloque se maior ou menor sem sinal)

Formato da instrução: **sgeu R1, R2, X | R3**

Se o conteúdo de R2 é maior ou igual que a constante X ou o conteúdo de R3, colocar o bit 1 em R1, caso contrário coloca o bit 0.

Instrução em micro-operações:

Se for constante:
t1: R_{aux} ← X
t2: R1 ← (R2) > (R_{aux})
t3: PC ← PC + I

Caso contrário:
t1: R1 ← (R2) > (R3)
t2: PC ← PC + I

SET LESS OR EQUAL UNSIGNED - (Coloque se menor ou igual sem sinal)

Formato da instrução: **sleu R1, R2, X | R3**

Se R2 é menor ou igual que a constante X ou do conteúdo de R3 se colocar o bit 1 em R1, caso contrário coloca o bit 0.

Instrução em micro-operações:

Se for constante:
t1: R_{aux} ← X
t2: R1 ← (R2) < (R_{aux})
t3: PC ← PC + I
Caso contrário:
t1: R1 ← (R2) < (R3)

t2: PC ← PC + I

BITWISE AND - (Coloca se ambos valores são verdade)

Formato da instrução: **and R1, R2, X | R3**

Compara os conteúdos do registrador R2 e do registrador R3 ou da constante X sendo que R2, R3 ou X são valores bitwise (0 ou 1) para executar a operação lógica onde R2 e R3 ou X devem ser 1 para colocar em R1 o bit 1, caso contrário colocará em R1 o bit 0.

Instrução em micro-operações:

Se for constante:

t1: R_{aux} ← X
t2: AND ← (R2) && (R_{aux})
t3: R1 ← AND
t4: PC ← PC + I

Caso seja registrador:

t1: AND ← (R2) && (R3)
t2: R1 ← AND
t3: PC ← PC + I

BRANCH IF EQUAL - (Desvio se igual)

Formato da instrução: **beq R1, X | R2, target**

Executa um desvio para o *target* caso o valor do conteúdo do registrador R1 seja igual ao conteúdo do registrador R2 ou da constante X (usando R_{aux}). Geralmente a operação feita para comparação é subtrair ambos e verificar se resultou em 0 e colocar na flag EQUAL.

Instrução em micro-operações:

Se for constante:

t1: R_{aux} ← X
t3: EQUAL ← (R1) - (R_{aux})
t4: EQUAL == 0:
PC ← PC + TARGET + I
EQUAL != 0: #diferente de 0
PC ← PC + I

Caso seja registrador:

t3: EQUAL ← (R1) - (R2)
t4: EQUAL == 0:
PC ← PC + TARGET + I
EQUAL != 0:
PC ← PC + I

SUBTRACTION - (Subtração)

Formato da instrução: **sub R1, R2, X | R3**

Faz a subtração do conteúdo de um registrador R2 de uma constante X(onde ele coloca X em um registrador reservado do processador R_{aux}) ou um outro registrador R3, e coloca o resultado em R1.

Instrução em micro-operações:

Se for constante:

t1: $R_{aux} \leftarrow X$

t2: $R1 \leftarrow (R2) - (R_{aux})$

t3: $PC \leftarrow PC + I$

Caso seja registrador:

t1: $R1 \leftarrow (R2) - (R3)$

t2: $PC \leftarrow PC + I$

JUMP REGISTER - (Pulo para registro)

Formato da instrução: **jr R**

Faz o pulo para um registro especificado no operando.

Instrução em micro-operações:

t1: $PC \leftarrow (R) + I$

REFERÊNCIAS

STALLINGS, William. **Arquitetura e Organização de Computadores.** 8. ed. São Paulo: Pearson Pratice Hall, 2010.

TANENBAUM, Andrew S.; AUSTIN, Todd. **Organização Estruturada de Computadores.** 6. ed. São Paulo: Pearson Pratice Hall, 2014.

PATTERSON, David A.; HENNESSY, John L.. **Organização e Projeto de Computadores:** A Inteface Hardware/Software. Rio de Janeiro: Elsevier, 2005.

ELLARD, Daniel J.. **MIPS Assembly Language Programming:** CS50 Discussion and Project Book. Santa Barbara: 1994. Disponível em: <https://www.cs.ucsb.edu/~franklin/64/lectures/mipsassemblytutorial.pdf>. Acesso em: 21 jun. 2018.

STANFORD UNIVERSITY. **RISC Architecture.** Disponível em: <https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/mips/index.html>. Acesso em: 21 jun. 2018.

FREUND, Karl; FORBES. **AI Pioneer Wave Computing Acquires MIPS Technologies.** Disponível em: <https://www.forbes.com/sites/moorinsights/2018/06/13/ai-pioneer-wave-computing-acquires-mips-technologies/>. Acesso em: 21 jun. 2018.

WEATHERSPOON, Hakim. **MIPS Pipeline.** Ithaca: Cornell University, 2012. Disponível em: <https://www.cs.cornell.edu/courses/cs3410/2012sp/lecture/09-pipelined-cpu-i-g.pdf>. Acesso em: 21 jun. 2018.

GILLARD, Paul. **The MIPS instruction set architecture.** Saint John's: Memorial University Of Newfoundland, 2015. Color. Disponível em: <http://web.cs.mun.ca/~paul/cs3725/material/review.pdf>. Acesso em: 21 jun. 2018.

MIPS ISA and Single Cycle Datapath. Disponível em: <https://pdfs.semanticscholar.org/presentation/790f/ece4faad4471af608221c1da1e11eee03a43.pdf>. Acesso em: 21 jun. 2018.