Laboratório 3 Utilidades, Representação de Constantes

Chamadas de Sistema

O assembly MIPS possui incorporada a instrução `syscall`, projetada para acessar as funções básicas do sistema (no PC, estaríamos falando das funções da BIOS), tais como acesso a tela, teclado e arquivos. O MARS simula várias funções úteis com *syscalls*. Por exemplo, o trecho a seguir lê um inteiro do teclado e o coloca em $t0.

```
li $v0,5        # v0 é carregado para executar a chamada 5
syscall         # faz a chamada (lê o inteiro da interface)
move $t0,$v0    # o inteiro foi colocado em v0; copiamos em t0
```

Veja a lista completa de chamadas de sistema na ajuda do MARS, incluindo um exemplo complexo.

Constantes String e Lista de Dados

Para inserirmos constantes no nosso programa de forma elegante, utilizamos *diretivas assembly*, que são linhas de código que não resultam em instruções MIPS quando compiladas. Começam com ponto decimal, como abaixo.

- `.asciiz` serve para inserir uma string terminada em zero (caracteres de *um byte cada*);
- `.word` inclui uma lista de valores de 32 bits, a partir do próximo endereço múltiplo de 4.

Note que precisamos definir seções no *assembly* para o montador se achar, usando:

- `.data` inicia uma seção de dados, ou seja, indica que as linhas a seguir são apenas dados (constantes ou espaço reservado);
- `.text` inicia uma seção de programa, ou seja, indica as linhas a seguir são instruções MIPS. É o *default*.

Por exemplo:

```
        .data                       # início do segmento (trecho) dados
frase:  .asciiz "Frase a imprimir" # \0 inserido automaticamente
dados:  .word 34,-12,0,0x5FFE0D23  # 4 words de 32 bits

        .text           # início de segmento de programa
        la $s1,dados    # carrega o endereço dos dados em $s1
        lw $s2,12($s1) # carrega da memória o valor 0x5FFE0D23 em $s2
        la $a0,frase
        li $v0,4
        syscall         # imprime a frase na tela
```

O comando `la` carrega o endereço de um *label* no registrador indicado.

▶	Faça um programa que obtém vinte números do teclado e os imprime na tela na ordem inversa à de entrada. Pode usar a diretiva `.space` se quiser, mas `.word` serve também.

Tamanho de Operandos – Constantes de 32 bits

Uma vez que as instruções têm no máximo 32 bits e às vezes precisamos trabalhar com constantes deste tamanho, um artifício é usado: o montador quebra estas constantes ao meio, utilizando duas instruções para efetivar a carga (*load upper immediate* e *or immediate*):

```
 li $s0,0x12345678      =>   lui $s0,0x1234    (carrega parte alta)
(instrução original)         ori $s0,$s0,0x5678 (seta parte baixa)
```

Para *labels*, é o montador que decide onde colocá-los na memória (etapa de *linkagem* ou linkedição), portanto ele sabe o valor final exato, algo difícil para o programador fazer sozinho.

Contador de Programa e Saltos

O endereço da *próxima* instrução a ser executada por um processador é sempre guardado em um registrador chamado PC – *Program Counter*. Durante a execução de uma instrução comum, o PC é somado com 4 para apontar para a instrução seguinte (lembre: cada instrução de 32 bits ocupa 4 posições de memória no MIPS).

Temos três tipos de saltos no MIPS:

- com endereço absoluto (instruções *j* e *jal*);
- com endereço relativo (todos os *branches*, como *beq*, *bne* etc.);
- destino via registrador (instrução *jr*).

O endereço final de destino para um desvio deve ter, obrigatoriamente, 32 bits, mas isso não cabe dentro de uma instrução de 32 bits, pois no mínimo ela precisa de um *opcode,* que ocupa espaço.

Note-se que, no MIPS, o PC é incrementado logo após a instrução ser lida (*opcode fetch*) e antes dela ser executada (*instruction decode* e *instruction execute*), portanto o valor atual de referência do PC para ser usado nos desvios é somado de 4.

Codificação de *branches*

Os *branches* são codificados como instruções do tipo I, o que nos dá 16 bits para especificar o endereço. Simplesmente iremos *somar* o valor imediato *sinalizado* ao valor do PC. *O MIPS força que todas as instruções estejam em endereços múltiplos de 4* (dados alinhados em 32 bits, que é o tamanho das instruções), portanto simplesmente consideram-se os dois bits LSB do destino como fixos em 00_2, totalizando 18 bits. Um exemplo bobo:

```
        addi $s1,$zero,33
        bne  $s1,$zero,pulei
        nop
        nop
pulei:  addi $s1,$s1,1
```

O *bne* será codificado como:

000101	10001	00000	0000000000000010
opcode	rs	rt	constante 16 bits

A constante é 2, expressa em 16 bits, pois o *bne* vai pular duas instruções à frente; o MIPS coloca dois bits LSB extra 00_2 e obtém um número de 18 bits: 000000000000001000_2=8. Ou seja, o *PC será somado em 8*. Como toda instrução soma 4 no PC para avançar o programa, isso totaliza os 12 endereços entre o *bne* e o *label* "pulei:". Teste no MARS pra entender direito.[1]

Codificação de *jumps*

Com endereço absoluto usamos o formato J, visto a seguir. O MIPS simplesmente salta para o endereço especificado na constante. Infelizmente, temos apenas 26 bits disponíveis; somados aos dois zeros LSB, como nos *branches*, ficamos com 28 bits para indicar o destino do salto. Os 4 bits faltantes (MSB) são copiados dos MSBs do valor atual do PC. O opcode da instrução *j* é 0x02.

Formato J	opcode (6)	endereço destino (26)

Se precisarmos de um salto que cubra necessariamente toda a memória, deveremos carregar o seu destino num registrador de 32 bits e usar a instrução *jr* (jump register), que saltará para o valor especificado pelo registrador em questão.

1 No livro do Hennessy, o segundo exemplo da seção 2.9 da 3ª edição (ou da 2.10 na 4ª) contém estas explicações.

Teste no MARS o programa bobo abaixo e explique *cada bit* da codificação binária resultante:

```
start:    j final
          beq $zero,$zero,final
          la $t6,final
          jr $t6
final:    bne $0,$0,final
```

Assuma que o endereço de `start` é 0x0040 0000. A codificação de *jr* está no Lab #2.

Constantes Negativas

Podemos assumir, para todos os efeitos, que todos os números que lidamos no MIPS são sinalizados, o que significa dizer que seus valores são representados em complemento de 2.

Obviamente, é preciso fazer *extensão de sinal:* `addi $s0,$zero,-1000` deve carregar 0xFFFF FC18 em $s0 (isso é -1000 em 32 bits), mas a constante de 16 bits na instrução formato I é 0xFC18; a "extensão de sinal" é apenas repetir o bit MSB para todos os bits à esquerda.

Deve-se notar que a instrução *addiu* (*add immediate unsigned*) na verdade não quer dizer que o número não é sinalizado; apenas está indicando que o processador não deve se preocupar caso haja um estouro de 32 bits na soma. A constante imediata terá seu sinal estendido.

Exercícios

1. Percebe-se que temos limites para o destino de saltos, dados conforme a instrução usada. Quais são estes limites, ou seja, quão longe pode estar nosso destino em relação à instrução de salto? Exprima este valor tanto em *endereços* quanto em *número de instruções*.
2. Prove que estender o sinal é equivalente a copiar o bit MSB para todos os bits à esquerda.
3. Um assim chamado *modo de endereçamento* indica uma maneira possível para o processador acessar a memória. No MIPS, temos endereçamento imediato (constantes, como em addi), endereçamento de registradores (como em add) e o indexado de lw e sw.
   É usual termos, em microprocessadores simples, o modo direto (colocamos o endereço a acessar como constante na instrução), o indireto simples (o endereço está num registrador) e o indexado (uma constante é somada a um valor indireto).
   Tanto o x86 quanto o ARM possuem muitos outros modos. Cite vantagens e desvantagens de termos modos diversificados de acesso à memória.
4. No 8051, é o programador que decide se o número de 8 bits é sinalizado ou não. Dada a conta: $11001110_2+11000111_2$ com resultado num registrador de 8 bits, quais são os valores decimais envolvidos? Houve estouro da capacidade de 8 bits ou não? Veja o apêndice se precisar.
5. O comando addiu $s0,$zero,0x8000 é interpretado como uma pseudoinstrução. Por quê?
6. Dê os opcodes hexadecimais para o seguinte programa:

```
inicio:              # considere o programa iniciado em 0x00400000
     add  $17,$0,$0
loop:addi $17,$17,-1
     bne  $17,$0,loop
     j inicio
```

7. Faça um programa que descobre e imprime *x* tal que o fatorial de *x* estoura 31 bits.

Apêndice: Complemento de 2

Há várias representações possíveis para números negativos em binário, dentre elas complemento de 1 e sinal-magnitude; estas duas fazem os circuitos ficarem mais complicados, tanto por possuírem ambos +0 e -0 quanto para os ajustes de operações pelos sinais.

Operações feitas em complemento de dois utilizam *os mesmos circuitos* do que as operações de números não-sinalizados (exclusivamente positivos ou zero), o que é bastante conveniente. Isso funciona porque a ULA opera em módulo, e os resultados são idênticos: numa ULA de 8 bits, podemos trabalhar com números de 0 a 255 ou de -128 a +127, de forma transparente.

Baseado na matemática de módulos de divisão, podemos descrever estes números

negativos de várias formas:

- Para obter um negativo, tomamos o número, invertemos seus bits e somamos um;
- Ou então consideramos que o bit MSB tem seu peso multiplicado por menos um;
- Ou ainda calculamos o quanto falta (complemento) para a próxima potência de 2.

Por exemplo: como representar o número -78 em complemento de 2 de 8 bits?

- $78_{10} = 01001110_2$; invertemos e incrementamos para $10110001_2+1 = 10110010_{Compl2}$;
- Perceba: $10110010_{Compl2} = 1*\mathbf{(-2^7)}+0*2^6+1*2^5+1*2^4+0*2^3+0*2^2+1*2^1+0*2^0 = \mathbf{-128}+32+16+2 = -78_{10}$;
- E ainda: a próxima potência é $2^7 = 256$; calculamos 256-78 = 178.
  Observe que $178_{10} = 10110010_2$, que são os mesmos bits determinados antes.

Observe que o *carry*, que é o vai-um da soma ou empresta-um da subtração, pode ser usado como estouro de operações não-sinalizadas (o *unsigned* do C). Quando há estouro sinalizado (com o *signed* do C) isso se chama *overflow* e pode ser detectado quando o sinal do resultado é inconsistente. Por exemplo: em oito bits, -123-123 resulta em +10: isto é um *overflow*.