Modos

Lucas

30 de agosto de 2015

1. Sinalização contra não-sinalização se aplica a instruções 'loads' assim como para instruções que mexem com aritmética (Patterson pg 124).

2. Caracteres são normalmente combinados em strings, essas mesmas tem um número variado de caracteres. Existem três possibilidades para representar uma string: (1) a primeira posição da string é reservada para conter o comprimento da string, (2) outro variável parametrizada contém o comprimento (como em uma estrutura de lista, pilha, fila, etc), ou (3) a última posição da string é indicada por um caractere usado para marcar o fim da string. C usa a terceira opção, terminando a string com um byte do qual o valor é 0' (nomeado null em ASCII). Por exemplo a string "Cal"é representada em C por 4 bytes =¿ `67, 97, 108, 0`. Em laboratório o java usa a primeira opção.

3. Os modos de endereçamento da figura página 133 retratam bem, da forma que:

4. O modo de endereçamento dos registradores utiliza os 5 bits para identificar um dos 32 registradores no caso do mips 32. Mas por exemplo se eu tivesse um banco de 43 registradores, quantos bits eu precisaria nos campos RS, RT, RD, respectivamente ? ($log_2 43$)

5. O modo de endereçamento "base+deslocamento"você soma o valor do campo de imediato ¡¡ 2 e com sinal estendido, ao valor contido no registrador base. Você pode capturar da com lb, lbu, lh, lhu, lw, ou também pode salvar na memória usando sw, sh, sb. A forma como você vai somar um deslocamento em relação à capturar ou salvar na memória usando esse modo de endereçamento, vai da lógica de programação. E cuidado com essa ideia, pois deslocar para bytes é diferente de deslocar para palavras inteiras, é óbvio mas muita gente esquece e eu também ._.

6. O PC-relativo é o modo dos branchs eles movem o fluxo de execução de código baseando-se no valor completo do PC_atual. E isso dá uma flexibilidade boa para que seja possível a implementação de if then else (que são usado extensivamente em alto nível). O salvo do branch é menor que o do jump devido ao campo de imediato ser menor, 16 bits em relação ao campo de um tipo-j que é de 26 bits. Lembrar que é PC_{atual} + 4 + (SignExt :: BranchAddress :: 00). Pois o branch mexe no fluxo de execução de instruções do MIPS32 que é alinhado de 4 em 4 bytes, outras arquiteturas como x86 tem um mecanismo de branching mais complexo, pois o tamanho das instruções varia e isso tem que ser cuidado pelo sistema. O x86 não perde desempenho por causa disso.

7. Pseudo-Direct Adressing pega o valor direto do imediato como endereço, por exemplo jump. Porém saiba que o jump depende do último byte do valor do PC_{atual} que em nem sempre é 0000, por vezes será 0001. Pois o último valor do espaço alocado no layout de memória para .text (instruções, seus programas, etc) é 10000000_{hex} (basta ver no mips greencard).

Exercise 2.24 Assume that the register $t1 contains the address 0x10000000 and the register $t2 contains the address 0x10000010. Note the MIPS architecture utilizes big- endian addressing.

```
a. lbu  $t0, 0($t1)
   sw   $t0, 0($t2)

b. lb   $t0, 0($t1)
   sh   $t0, 0($t2)
```

2.24.1 [5] **2.9** Assume that the data (in hexadecimal) at address 0x10000000 is:

```
[10000000][12][34][56][78]
```

What value is stored at the address pointed to by register $t2 ? Assume that the memory location pointed to $t2 is initialized to 0xFFFFFFFF.

- Esse "inicializado para 0xffffffff"talvez queira dizer que tem 4GiB de memória, não tenho certeza. Mas basta você entender que little endian começa endereçar pelo byte menos significativo e vai em direção ao MSB. E o big endian começa a endereçar pelo byte mais significativo e vai em direção ao LSB. Agora a resposta se dá da seguinte forma. Lembre que o lb ele estende sinal, por isso existe lbu. E também saiba que após o bit oitavo do byte os '1' estendidos não tem valor significativo, mas representativo, representam um valor negativo.
- O valor que esta na memória tem o 0x12 como byte mais significativo. Então é ele o candidato a ser trazido primeiro da memória. E não o byte 0x78.
- 0x12 = 00010010, então não vai acontecer de estender o sinal negativo.
- Após 0x12 ter cima carregado no registrador $t1, a saber, 0x00000012. Essa mesma palavra 0x00000012 vai ser inteiramente colocada na memória na posição 0x10000010, deverá aparecer do jeito que esta 0x 00 00 00 12.
- Sim, por incrível que pareça, quando você coloca byte a byte, a sua percepção de que é byte endian ou little endian melhora. Mas quando você coloca uma palavra inteira de uma vez, a sua percepção diminui, pois você não está vendo os "passos intermediários"de que ele esta colocando byte a byte, e endereçando automaticamente para você da forma big endian addressing.

- Se a máquina fosse little endian? Aí sim a palavra sendo colocada inteiramente apareceria 0x 78 00 00 00. Aqui 0x78, pois no passo anterior ao 'sw' ele carregaria 0x78, pois é little endian.

```
a. lbu  $t0, 0($t1) #
   sw   $t0, 0($t2) #

b. lb   $t0, 0($t1)
   sh   $t0, 0($t2)
```

Exercise 2.25 In this exercise, you will explore 32-bit constants in MIPS. For the following problems, you will be using the binary data in the table below.

```
a. 0010 0000 0000 0001 0100 1001 0010 0100 two
b. 0000 1111 1011 1110 0100 0000 0000 0000 two
```

2.25.1 [10] 2.10 Write the MIPS assembly code that creates the 32-bit constants listed above and stores that value to register $t1 .

```
a. lui $at, 0x2001
   ori $t1, $at, 0x4924

b. lui $at, 0x0fbe
   ori $t1, $at, 0x4000
```

2.25.2 [5] 2.6, 2.10 If the current value of the PC is 0x00000000, can you use a single jump instruction to get to the PC address as shown in the table above?

```
a. 0010 0000 0000 0001 0100 1001 0010 0100 two = 0x20014924

max(pc) -> pc+4[31-28] == 0000 :: 1111 1111 1111 1111 1111 1111 11 :: 00

           pc atual +4 == 0000    0000 0000 0000 0000 0000 0000 01    00

min(pc) -> pc+4[31-28] == 0000 :: 0000 0000 0000 0000 0000 0000 00 :: 00

fffffc < 20014924, portanto não alcança.
```

```
b. 0000 1111 1011 1110 0100 0000 0000 0000 two = 0xFBE4000

max(pc) -> pc+4[31-28] == 0000 :: 1111 1111 1111 1111 1111 1111 11 :: 00

           pc atual +4 == 0000    0000 0000 0000 0000 0000 0000 01    00

min(pc) -> pc+4[31-28] == 0000 :: 0000 0000 0000 0000 0000 0000 00 :: 00

ffffffc > fbe4000, portanto alcança.
```

2.25.3 [5] **2.6, 2.10** If the current value of the PC is 0x00000600, can you use a single branch instruction to get to the PC address as shown in the table above?

```
a. 0010 0000 0000 0001 0100 1001 0010 0100 two = 0x20014924

max(pc) -> pc+4[31-28] == 0000 :: 1111 1111 1111 1111 1111 1111 11 :: 00

           pc atual +4 == 0000    0000 0000 0000 0000 0100 0000 01    00

min(pc) -> pc+4[31-28] == 0000 :: 0000 0000 0000 0000 0000 0000 00 :: 00

fffffc < 20014924, portanto não alcança.

b. 0000 1111 1011 1110 0100 0000 0000 0000 two = 0x0fbe4000

max(pc) -> pc+4[31-28] == 0000 :: 1111 1111 1111 1111 1111 1111 11 :: 00

           pc atual +4 == 0000    0000 0000 0000 0000 0100 0000 01    00

min(pc) -> pc+4[31-28] == 0000 :: 0000 0000 0000 0000 0000 0000 00 :: 00

ffffffc > fbe4000, portanto alcança.
```

2.25.4 [5] **2.6, 2.10** If the current value of the PC is 0x1FFFf000, can you use a single branch instruction to get to the PC address as shown in the table above?

```
a. 0010 0000 0000 0001 0100 1001 0010 0100 two = 0x20014924
```

```
max(pc) -> pc+4[31-28] == 0001 :: 1111 1111 1111 1111 1111 1111 11 :: 00

           pc atual +4 == 0001    ffff ffff ffff ffff 0000 0000 01    00

min(pc) -> pc+4[31-28] == 0001 :: 0000 0000 0000 0000 0000 0000 00 :: 00

ffffffc < 20014924, portanto não alcança, por poucos bytes, precisaria de mais
um jump para completar.

b. 0000 1111 1011 1110 0100 0000 0000 0000 two = 0x0fbe4000

max(pc) -> pc+4[31-28] == 0000 :: 1111 1111 1111 1111 1111 1111 11 :: 00

           pc atual +4 == 0000    0000 0000 0000 0000 0100 0000 01    00

min(pc) -> pc+4[31-28] == 0000 :: 0000 0000 0000 0000 0000 0000 00 :: 00

ffffffc > fbe4000, portanto alcança.
```

2.25.5 [10] 2.10 If the immediate field of an MIPS instruction was only 8 bits wide, write the MIPS code that creates the 32-bit constants listed above and stores that value to register $t1.

```
a. 0010 0000 0000 0001 0100 1001 0010 0100 two = 0x20014924

lui $t1, 0x20
ori $t1, $t1, 0x01  # 0x20010000

lui $t2, 0x49
ori $t2, $t2, 0x24  # 0x49240000

srl $t2, $t2, 16    # 0x00004924

add $t1, $t1, $t2   # 0x20014924

b. 0000 1111 1011 1110 0100 0000 0000 0000 two = 0x0fbe4000

lui $t1, 0x0f
ori $t1, $t1, 0xbe  # 0x0fbe0000

lui $t2, 0x40
ori $t2, $t2, 0x00  # 0x40000000
```

```
srl $t2, $t2, 16    # 0x00004000

add $t1, $t1, $t2   # 0x0fbe4000
```

```
a. 0x00400000       beq $s0, $0, FAR
   ...
   0x00403100 FAR:  addi $s0, $s0, 1

-----------------------------------

b. 0x00000100       j 3FFFC
   ...
   0x04000010 AWAY: addi $s0, $s0, 1
```

2.27.5 [10] 2.10, questão de prova By reducing the size of the immediate fields of the I-type and J-type instructions, we can save on the number of bits needed to represent these types of instructions. If the immediate field of I-type instructions were 8 bits and the immediate field of J-type instructions were 18 bits, rewrite the MIPS code above to reflect this change. Avoid using the lui instruction.

```
I-type [opcode 6 bits][rs 5 bits][rt 5 bits][immed 8 bits]
       [oooooo]       [rrrrr]    [ttttt]    [iiiiiiii]

J-type [opcode 6 bits][immed 18 bits]
       [oooooo]       [iiiiiiiiiiiiiiiiii]
```

Branch desloca em relação ao PC+4, então basta carrega a constante (0x3100 >> 2) == 0xC40 no campo imediato de 16 bits. Porém a questão pede para que esse campo seja reduzido a 8 bits no tipo I e 18 bits no campo imediato do tipo J.

0xC40 é tranquilamente representável em um imediato de 12 bits (ARM). Porém, com apenas 8 bits você tem que trabalhar um jump e depois um branch para atingir o seu objetivo final que é chegar em FAR através da condição, isto é, se a condição $s0 == 0x00000000 permitir.

```
a. 0x00400000       j 0x300            # vai ser multiplicado por 4 (ou << 2)
   ...
   0x00403000 L1:   beq $s0, $0, 0x3F  # PC+4+(SignExt::BranchAddress::00)
                                       # 0x403000 + 0x4 = 0x403004
                                       # BranchAddress = 000... 0011 1111 00
                                       # 0x3F << 2 -> FC
                                       # 0x403004 + 0xFC = 0x403100 ----+
                                                                        |
   ...                                                                  |
   0x00403100 FAR:  addi $s0, $s0, 1   <--------------------------------+
```