Estudo, Questões Resolvidas de Branchs, e Dicas para o Teste de ORG

eu recomendo você testar tudo que eu fiz, para garantir...

Lucas

27 de setembro de 2015

Comparação entre instruções exige que você lide com o dilema de número com sinal e sem sinal (+/-). Algumas vezes você encontra o padrão do bit mais significativo igual a **1**, isso quer dizer que esse valor é negativo, e claro ele será menor que qualquer número positivo que começa com 0 no seu bit mais significativo.

máquina 32 bits você teria o seguinte exemplo:
0xF000CCCC < 0x700C8000 == 11110000000000001100110011001100 < 01110000000011001000000000000000, pensando em números sinalizados.

Por outro lado, números sem sinal, o 1 no bit mais significativo representa um número maior que qualquer outro que começa com 0 no seu bit mais significativo.

Agora com números sem sinal, o exemplo acima, se inverte:
0xF000CCCC > 0x700C8000 == 11110000000000001100110011001100 > 01110000000011001000000000000000, pensando em números sem sinal.

(Existe uma maneira de reduzir o custo de checagem de limitantes em arrays de dados que se utiliza de checadores do tipo unsigned em mips. Em java isso acontece bastante, visto que ele checa os limitantes do array automaticamente para você—facilidade do Java. Já em C/C++ não existe essa checagem automatica dos limitantes da estrutura de dados, e você tem que fazer isso na mão, quando está criando seus programas em C/C++—Pg 110 - Chapter 2 Instructions: Language of the Computer).

valores nas posições	24	5	14	13
array index (length = 4)	3	2	1	0

```
sltiu $t1, $s1, 4  # $t0=0 if $s1>=length or $s1<0
                   # pois você estará eliminando a possibilidade de
                   # s0 ser por algum motivo um valor negativo
```

Mips oferece duas versões de comparação **set on less than**:

1. Set on less than (slt) # trabalha com números com sinal
2. Set on less than immediate (slti) # trabalha com números com sinal
3. Set on less than unsigned (sltu) # trabalha com números sem sinal
4. Set on less than immediate unsigned (sltiu) # trabalha com números sem sinal

2.15.2

```
a. not $t1, $t2 // bit-wise invert
b. orn $t1, $t2, $t3 // bit-wise OR of $t2, !$t3
```

2.15.2 [10] ¡2.6¿ The logical instructions above are not included in the MIPS instruction set, but can be synthesized using one or more MIPS assembly instruc- tions. Provide a minimal set of MIPS instructions that may be used in place of the instructions in the table above.

```
a. not $t1, $t2 // bit-wise invert

   NOT: A NOR 0 = NOT (A OR 0) = NOT (A)
   nor $t1, $t2, $zero # reg $t1 = ~ (reg $t1 | reg $zero)

b. orn $t1, $t2, $t3 // bit-wise OR of $t2, !$t3

   nor $t3, $t3, $zero # reg $t3 = ~ (reg $t3 | reg $zero)
   or $t1, $t2, $t3 # reg $t1 = (reg $t2 | reg $t3)
```

2.16 For these problems, the table holds various binary values for register $t0 . Given the value of $t0 , you will be asked to evaluate the outcome of different branches.

```
a. 0010 0100 1001 0010 0100 1001 0010 0100 = 0x24924924
b. 0101 1111 1011 1110 0100 0000 0000 0000 = 0x5FBE4000
```

2.16.1 [5] 2.7 Suppose that register $t0 contains a value from above and $t1 has the value 0011 1111 1111 1000 0000 0000 0000 0000 = 0x3FF80000. Note the result of executing these instructions on particular registers. What is the value of $t2 after the following instructions?

```
slt $t2, $t0, $t1       # $t2 = 1 se $t0 < $t1, 0 caso contrário.
beq $t2, $0, ELSE       # desvia para ELSE se $t2 == 0, PC+4 caso contrário.
j DONE                  # desvia para DONE
ELSE: addi $t2, $0, 2   # $t2 recebe o valor 2
DONE:                   # ...
```

Comentário: slt é um comparador **com sinal**, quer dizer que ele considera se o número é sinalizado ou não. nenhuma dos números das letras a) e b) são negativos. são dois valores de 32 bits, ambos positivos. o valor de $t1 fornecido pela questão 2.16.1 é positivo também.

a. $0x5FBE4000 - 0x3FF80000 =$ positivo pois $0x5FBE4000 > 0x3FF80000$.

```
slt $t2, $t0, $t1       # $t2 = 0 pois 0x5FBE4000 > 0x3FF80000
beq $t2, $0, ELSE       # desvia para ELSE pois $t2 == 0
j DONE                  # --------
ELSE: addi $t2, $0, 2   # $t2 recebe o valor 2
DONE:                   # ...
```

b. $0x24924924 - 0x3FF80000 =$ negativo pois $0x24924924 < 0x3FF80000$.

```
slt $t2, $t0, $t1       # $t2 = 1 pois 0x24924924 < 0x3FF80000
beq $t2, $0, ELSE       # PC+4 e vai próxima instrução, pois t2 = 1 e não 0
j DONE                  # pula para DONE e se esquiva do ELSE
ELSE: addi $t2, $0, 2   # --------
DONE:                   # ...
```

Comentário: Basta você olhar para o bit mais significativo, assim você pode determinar qual valor é maior que o outro, apenas olhando para os bits mais significativos. O slt é sinalizado, se tivesse algum número com bit 1 na posição 31, contando de 0-31. Esse valor seria sinalizado, com negativo, e portanto seria menor. Nessa questão você tem que sacar o uso de números sinalizados. O sltu pegaria um valor de 32 bits e mesmo que o bit no índice 31 dele fosse 1, ele consideraria esse valor como sendo positivo e faria o calculo normalmente, comparando dois valores.

2.16.2 [5] 2.7 Suppose that register $t0 contains a value from the table above and is compared against the value X , as used in the MIPS instruction below. Note the format of the slti instruction. For what values of X , if any, will $t2 be equal to 1?

```
slti $t2, $t0, X
```

Comentário: Quando o valor de $t0 é 0x5FBE4000 e X é negativo então o valor de $t2 será 0 pois um valor positivo é maior que um valor negativo.

Se o valor de $t0 é 0x5FBE4000 e X é positivo, vai depender de quem é maior ou menor, se o resultado da subtração entre eles for negativo, 0x5FBE4000 será menor que X, caso contrário não.

O raciocínio é o mesmo para a letra b.

For these problems, the table holds MIPS assembly code fragments. You will be asked to evaluate each of the code fragments, familiarizing you with the different MIPS branch instructions.

```
      addi $t1, $0, 50   # t1 recebe 50
LOOP: lw $s1, 0($s0)     # s1 recebe M[s0]
      add $s2, $s2, $s1  # s2 recebe s2+s1
      lw $s1, 4($s0)     # s1 recebe M[s0+4]
      add $s2, $s2, $s1  # s2 recebe s2+s1
      addi $s0, $s0, 8   # s0 recebe s0+8
      subi $t1, $t1, 1   # t1 recebe t1-1
      bne $t1, $0, LOOP  # desvia para LOOP se t1 != zero
```

2.18.5 [5] 2.7 Translate the loops above into C. Assume that the C-level integer i is held in register $t1, $s2 holds the C-level integer called result, and $s0 holds the base address of the integer MemArray.

```
int t = 0;
int a = 0;
int b = 0;

for (i = 50; i > 0; i--) {
  a = MemArray[t];
  b = b + a;
  a = MemArray[t + 1];
  b = b + a;
  t = t + 2;
}
```

2.18.6 [5] 2.7 Rewrite the loop to reduce the number of MIPS instructions executed.

```
int t = 0;

for (i = 50; i > 0; i--) {
  int a = MemArray[t];
  int b = MemArray[t + 1];
  c += a + b;
  t = t + 2;
}
```

```
      addi $t1, $0, 50   # t1 recebe 50
LOOP: lw $s1, 0($s0)     # s1 recebe M[s0]
      lw $s2, 4($s0)     # s1 recebe M[s0+4]
      add $s3, $s1, $s2  # s2 recebe s2+s1
      addi $s0, $s0, 8   # s0 recebe s0+8
      addi $t1, $t1, -1   # t1 recebe t1-1
      bne $t1, $0, LOOP  # desvia para LOOP se t1 != zero
```

Comentário: o que eu consegui mudar eliminar foi 1 instrução. Ao invés de somar acumular à $s2 os valores puxados da memória. Puxa primeiro os dois valores e depois acumula somando os dois a uma terceira variáveis $s3. Isso acarreta em queda de desempenho pois `add $s3, $s1, $s2`. Podemos mover os dois addi para cima, para que sobre tempo para os loads carregarem completamente os valores em $s1, $s2.

```
      addi $t1, $0, 50   # t1 recebe 50
LOOP: lw $s1, 0($s0)     # s1 recebe M[s0]
      lw $s2, 4($s0)     # s1 recebe M[s0+4]
      addi $s0, $s0, 8   # s0 recebe s0+8
      addi $t1, $t1, -1  # t1 recebe t1-1
      add $s3, $s1, $s2  # s2 recebe s2+s1
      bne $t1, $0, LOOP  # desvia para LOOP se t1 != zero
```

1 Extra

Suponha que você tem seu PC_{atual} setado com o valor `0x0040054`. E pedem para você medir o alcance de um branch ou um jump a partir desse valor de PC_{atual}.

O Branch seguindo a mips greensheet, pega seu valor imediato e estende o bit de sinal 14x e também ele concatena '00' como bits menos significativos, para multiplicar por 4, para manter assegurar o alinhamento das instruções mips que tem todas o mesmo tamanho de 32 bits (4 bytes).

O valor imediato tem 16 bits complemento de 2, então a distância ele pega de baixo $-2^{15} == (32768)_{10} == (8000)_{16}$ pra cima até $2^{15} - 1 == (32767)_{10} == (7FFF)_{16}$. Perceba que o computador só soma algebricamente, então ele vai pegar o valor complementado de $-2^{15} == (32768)_{10} == (8000)_{16}$ e somar. Esses são os valores que cabem dentro do campo de imediato. Mas no final das contas ele vai aumentar esse comprimento porque ele vai concatenar '00', multiplicando por 2, isso significa que de 2^{15} vai para 2^{17}, porém note que desses 2^{17} valores possíveis apenas os múltiplos de 4 servirão para o branch, pois isso esta na norma do mips greensheet e as instruções são todas de mesmo tamanho 32 bits (4 bytes).

opcode	rs	rt	immed
B_{31-26}	B_{25-21}	B_{20-16}	B_{15-0}
000000	sssss	ttttt	iiii iiii iiii iiii

iiii iiii iiii iiii → extende o bit na posição 15ª (MSB do imediato no formato I: beq, bne, bgtz, bgez, bltz, blez, etc) → ee eeee eeee eeee iiii iiii iiii iiii → contatena '00' no LSB → ee eeee eeee eeee iiii iiii iiii iiii 00. Percebeu como o endereço do branch esta válido e agora ele pode ser somado a $PC_{atual} + 4$? Percebeu como a distância aumentou de 2^{15} para 2^{17}?

Se para subir o máximo possível a partir do endereço de $PC_{atual} + 4$ basta pegar o comprimento do campo de imediato, contar quantos bits tem e estabelecer o comprimento possível, lembrando que é complemento de 2, ou seja, ele considera o bit de sinal, saiba que se o branch enxergar 1 no bit da posição 15ª ele vai estender esse bit, considerando que o valor ali é negativo.

upwards(branch)	0x400058 + 0x1FFFC = 0x420054
	↑
$PC_{atual} + 4$	0x400058
	↓
downards(branch)	**0x400058 - 0x20000** = 0x3E0058

Note que: 0x420054 - 0x3E0058 = $2^{17} + (2^{17} + 4)$ = 0x3FFFC. Que é o comprimento da regra toda para o branch. Tem que dar certo 0x420054 - 0x3E0058 pois o PC esta deslocado. E o branch usa o modo de endereçamento relativo ao $PC_{atual} + 4$, isto é, ele depende do PC para saltar. E os alcances finais são múltiplos de 4.

O negrito e vermelho essa operação **0x400058 - 0x20000** ? ... Por que ela não existe para máquina. A máquina não sabe subtrair, ela só sabe somar. O que ela faz é somar $0x400058 + (-0x20000)$, isto é, $a + \bar{b} = c$.

```
0x 0002 0000 é 0000 0000 0000 0010 0000 0000 0000 0000
inverte tudo
0x FFFD FFFF é 1111 1111 1111 1101 1111 1111 1111 1111
soma 1
0x FFFE 0000 é 1111 1111 1111 1110 0000 0000 0000 0000
```

Pronto, o valor que a máquina vai somar com 0x400058 é 0x FFFE 0000 = 0x 003E 0058 (o resultado já sai do jeito que você quer, não precisa refazer o complemento de 2 em cima do resultado.)

O Jump é mais simples, ele usa o modo de endereçamento pseudo-direto.

opcode	target address
B_{31-26}	B_{25-0}
000000	tt tttt tttt tttt tttt tttt tttt

Ele depende dos 4 bits mais significativos do valor de $PC_{atual} + 4$. Ou seja, ele vai fazer o seguinte: digamos que 'pppp' são os 4 bits do $PC_{atual} + 4$ → concatena os 4 bits na parte MSB do campo target do formato **J** → pppp tt tttt tttt tttt tttt tttt tttt → concatena '00' na parte LSB do campo target do formato **J** → pppp tt tttt tttt tttt tttt tttt tttt 00. É importante lembrar do +4, pois pode ser que falte apenas +4 para que na parte dos 4 bits mais significativos do PC fique, por exemplo, 0001. E isso pode fazer uma diferença monstruosa em questão de endereços.

Exemplo: 0x00400054 + 4 = 00000000010000000000000001011000. Finge que o target de 26 bits é o seguinte valor: **10011000000010010001110000**, concatena **00** no LSB e PC[31-28] no MSB.

Endereço alvo exemplo: **0000**1001100000001001000111000000, esse é o endereço efetivo final que o jump desviará o fluxo de execução.

Abaixo uma tabela esquematizando o que está acontecendo no nosso exemplo:

Static Data	↑
upwards(jump)	0000 :: $\underline{11111111111111111111111111 :: 00} = 0xFFFFFFC$
	↑
$PC_{atual}[31-28]$	[31-28]=[0000] 4 bits MSB do PC vai variando conforme o pc se movimenta (espaço text). para o nosso layout de memória do mips greensheet.
	↓
PC → $0x400000$	↓
	Espaço de Memória Reservado
downards(jump)	0000 :: 00000000000000000000000000 :: 00 = 0

Se você observar o layout de memória do mips greensheet do livro Computer Organization and Design (Patterson & Hennessy), você vai ver que a partir do endereço 1000000_{hex} é o campo no layout de dados estáticos, ou seja não tem instruções nessa

parte da memória. As instruções ficam num espaço de 400000_{hex} até 0xFFFFFFC$_{hex}$. Mas de 0_{hex} até 0x3FFFFC$_{hex}$ é um espaço reservado pela memória que você não vai querer pular com jumps para lá, sem saber perfeitamente o que está fazendo. Ou seja, os jumps ficam pulando num espaço entre onde o PC inicia seu contador em 0x400000$_{hex}$ até 0xFFFFFFC$_{hex}$ esse é o espaço de TEXT para o jump pular, são 268435452 bytes, 67108863 endereços válidos. O mais longe que ele pode pular se o `pc+4[31-28] == 0000` é FFFFFFC, são os 26 bits do jump iguais a 26{b'1'} assim ele desloca 2 para esquerda, e dá em FFFFFFC.

O loadword, storeword e derivados do mesmo tipo, usam outro modo de endereçamento chamado base+deslocamento.

O deslocamento não vai não vai ser multiplicado por 4, pois fica a cargo do programador que pedaço da memória ele quer acessar. Mas o valor do deslocamento vai ser estendido para completar um valor de 32 bits, para ser somado ao valor que o registrador base ($s0,1,2,3, etc.. $t0,1,2,3,4,etc...) contém.

16 × posição 15 do campo imediato (essa é a extensão de sinal) concatenado com o próprio valor do campo imediato, e não precisa multiplicar por 4 nesse caso, pois o programador pode querer acessar apenas bytes ímpares por exemplo, depende da lógica do programa.

```
(1) eeee eeee eeee eeee eiii iiii iiii iiii
(2) os bits 'e' poderão ser todos '0' ou todos '1'
```

opcode	rs	rt	immed
B_{31-26}	B_{25-21}	B_{20-16}	B_{15-0}
000000	sssss	ttttt	eiii iiii iiii iiii

Um exemplo simples é:

```
.text
.globl main

main:

addi $t0, $0, 0x50
la $s0, 0x10010080
sw $t0, -0x20($s0)
```

O Mars converte para:

```
addi $t0, $zero, 0x50
lui $at, 0x1001        # ele da um jeito de carregar grandes valores
ori $s0, $at, 0x80     # dentro do $s0
sw $t0, 0xffffffe0     # ele estendeu o sinal de 0xffe0
                       # 0xffe0 em complemento de 2 quer dizer -20 base 16
                       # quero dizer que o que veio no immediate do store
                       # word foi ffe0, ele então estende o sinal.
                       # e o mips sabe que é um valor complementado
                       # agora ele vai somar e deve voltar alguns endereços
```

```
# 0x1001 0080 + 0xFFFF FFE0 = 0x1001 0060
# a soma com o valor complementado já dá o endereço
# exatamente 0x10010060
# 0x50 aparecerá no byte menos significativo
# pois o mars implementa um mips tipo little endian
```

Load Upper Immediate (lui) é uma instrução do tipo **I** logo o que **chega** pra esse cara é um valor no campo imediato de 16 bits, esse 16 bits ele apenas insere na **parte alta** de algum alvo de 32 bits, só isso.

```
suponha que immed = aaaa aaaa aaaa aaaa
e suponha que alvo = tttt tttt tttt tttt tttt tttt tttt tttt

ao executar: lui alvo, immed

alvo recebe: aaaa aaaa aaaa aaaa :: 0000 0000 0000 0000

então alvo = aaaa aaaa aaaa aaaa 0000 0000 0000 0000
não preserva a parte baixa, na verdade ele zera a parte baixa.
```