Questões de Operações

Lucas

30 de agosto de 2015

1 Operações, alguns comentários

- O princípio desse capítulo é: "relationship between high-level programming languages and this more primitive one."(página 76 - Organização de Computadores 4ª). Relacionamento entre programação de alto nível e sua linguagem mais primitiva (ou à nível mais primitivo).
- **stored-program concept**: The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored-program computer. (Conceito de programa-armazenado, é a ideia que instruções e dados de muitos tipos podem ser armazenados em memória como valores).
- Na página 78 existem um quadro com as instruções da arquitetura mips, desse quadro podemos retirar alguns conceitos irados, de que 2^{30} palavras de memória são a sequência `Memory[0]`, `Memory[4]`, ..., `Memory[4294967292]`. `4294967292 bytes` são **4 GiB**.
- No final da tabela tem os jumps, j 2500, e go to 10000. Esse 2500 é em hexadecimal $(2500)_{base16}$ << 2 == $(10000)_{base10}$ == $(10011100010000)_{base2}$.
- Com o **nor** $s1 = ($s2 — $s3), você pode trabalhar ele zerando $s3 ou simplesmente colocar $zero no lugar, para negar $s2 e "armazenar no registrador $s1".
- O load linked: ll $s1,20($s2) e o Store Conditional: sc $s1,20($s2) são bastante importantes para se entender. Pois ambos trabalham 1st and 2nd half of atomic swap. O load linked carrega em $s1 = Memória[$s2 + Deslocamento] e o score conditional faz o seguinte Memória[$s2 + Deslocamento]=$s1;$s1=0 or 1. Isso quer dizer que o Load Linked pegou o que tinha em $s1 e armazenou na pilha algum valor e acionou o 1st half of atomic swap, engatilhou uma ação atômica, isto é, se ninguém armazenar nada no mesmo endereço de [$s2 + Deslocamento] até que o próximo store conditional complete a ação 2nd half of atomic swap, então o store conditional irá armazenar seja lá o que ele tiver em $s1 na posição Memória[$2 + Deslocamento] nisso ele sinaliza seu próprio registrador $s1 com o valor '1' (dizendo: "eu completei atomic exchange"); **caso contrário** se alguém mexeu no mesmo endereço de memória antes do store conditional (p.e: outro store conditional, store word, store half, etc..) então ele não vai mais completar o seu 2nd half of atomic swap o resultado disso é que ele sinaliza o próprio registrador $s1 como sendo 0 (dizendo: "eu não completei minha atomic exchange");

- Na página 84 diz que se deve cuidar com esquema o de alinhamento 4 por 4 que aparece no MIPS, e em outras arquiteturas parecidas com o MIPS, ou seja, um endereço apontando para uma instrução deve ser multiplicado por 4, para se ter certeza que você esta pegando um endereço válido nos termos naturais de alinhamento de instruções 4 por 4 bytes. Na página 84 do livro-texto de Organização de Computadores diz: "Computers divide into those that use the address of the leftmost or "big end" byte as the word address versus those that use the rightmost or "little end" byte. MIPS is in the big-endian camp.". Você pode testar essa afirmação dele no MARS, Simulador de MIPS. No apêndice

```
lui         $t0, 0x1DE              # $t0 = 0x01DE0000
ori         $t0, $t0, 0xCADE        # $t0 = 0x01DECADE
lui         $t1, 0x1001             # $t1 = 0x10010000
sb          $t0, 200 ($t1)          # $t1 + 200 bytes = 0x01DECADE
lb          $t2, 200 ($t1)          # $t2 = 0x01DECADE
```

  Se o byte 'DE' for para memória na posição do byte menos significativo, o MARS é little endian, quando reordenando na memória. Se você carregasse a palavra inteira 01DECADE com 'sw' a palavra apareceria nessa mesma ordem em memória.

- Quando você utilizar load byte signed (lb) para armazenar alguma coisa em registrador, saiba que você esta sujeito a estender o sinal (só isso). Por exemplo:

```
M($t0+20)=0x000000de
lb $s1, 0($t0+20)
$s1 <= 0xffffffde
```

- Portando te você precisar puxar um **dado**, use lbu, para ter certeza que seu byte não vai estragar o resto dos dados. Se você sabe que byte que você esta puxando é "positivo"(pe: 0x2), não vai causar grande estrago.

2.1.1 Uma dica que eu dou é a seguinte. Existirão questões, na prova inclusive que exigem que você pense como a máquina pensa algebricamente. Máquina não sabe subtrair algebricamente, ela só sabe somar, complementar, etc. Então ela vai trabalhar as componentes para equalizar o seu resultado desejado. A complementação de constantes que serão adicionadas com endereços base que estão em registradores e constantes de branchs que levam para endereços menores que o do PC_{atual} também seguem a mesma lógica.

```
a. sub f, g, h      -> sub $s0, $s1, $s2

b. addi h, h, FFFB -> addi $s0, $s0, fffb
   add f, g, h      -> add $s1, $s2, $s0
```

2.2.1

```
a. sub f, g, f      -> sub $s0, $s1, $s0

b. addi h, h, FFFE  -> addi $s0, $s0, fffe
   add f, g, h      -> add $s1, $s2, $s0
```

2.3.1 Nessa questão minha dica é você tentar bolar um jeito de gerar código mips de uma maneira padrão. Sabemos que você pode criar estratégias para efetuar a mesma situação que o código c apresenta. Mas a máquina não tem a capacidade que nós seres humanos temos de raciocinar, ela apenas tem mais resistência física para cálculos repetitivos. Pensa em código mips gerado padrão.

```
a.1. f = $zero - f;   -> sub $s0, $zero, $s0
a.2. f = f - g;       -> sub $s0, $s0, $s1

b. f = g + (- f - 5); -> addi $s1, $s1, 0xfffb
                      -> sub $s1, $s2, $s1
sub ->          -
               / \
            $s2   +          <- addi
                 / \
               $s1  0xfffb
```

2.2.4 Na versão 4 estendida do livro de organização de computadores. Essa questão pede para transformar em código c.

2.3.4

```
a. addi f, f, -4

addi $s0, $s0, 0xfffc

b. add i, g, h
   add f, i, f

add $s0, $s1, $s2
add $s3, $s0, $s3
```

2.3.5 [5] 2.2 If the variables f , g , h , and i have values 1, 2, 3, and 4, respectively, what is the end value of f ?

```
addi $s0, $s0, 0xfffc

$s0 recebe 0x0001 + 0xfffc = 0xfffd (=> -3)

b. add i, g, h
   add f, i, f

add $s0, $s1, $s2
add $s3, $s0, $s3

$s0 recebe 0x2 + 0x3 (=> 0x5)
$s3 recebe 0x5 + 0x1 (=> 0x6)
```

2.4.1 $\rightarrow$ **2.4.2** Nesse exercício você vai elaborar suas skills com push e pop e também e eventualmente sacar que o lw e sw são um modo de endereçamento chamado base + deslocamento.

```
f , g , h , i , j -> $s0, $s1, $s2, $s3, $s4,
Assume that the base address of the arrays A and B are in registers $s6
and $s7, respectively.

a. f = -g -A[4];

lw $t0, 0x10($s6)   #      A ->  0*4=0x00000000 [   data   ]
sub $t0, $zero, $t0 #            1*4=0x00000004 [   data   ]
sub $s0, $s1, $s0   #            2*4=0x00000008 [   data   ]
                    #            3*4=0x0000000C [   data   ]
                    #            4*4=0x00000010 [   data   ]

b. B[8] = A[i{j];

sub $t0, $s3, $s4
sll $t1, $t1, 2
add $t2, $t2, $s6
lw $t3, 0($t0)

sw $t3, 0x20($s7)
```

2.11.5 Na prova de organização de computadores é exigido que você massantemente traduza código mips para binário, e além disso calcule o deslocamento do branch equal ou branch not equal.

Sempre olhar opcode primeiro (op) e depois funct. Pois eles são decodificados primeiro no datapath. Depois vc se preocupa com constantes em complemento de dois, se houverem, deslocamentos em relação a base em complemento de dois, etc... Já coloca em binário, no tamanho do comprimento do campo no formato que você identificou para o opcode+funct. O registrador em base 10 te ajuda a identificar qual é o registrador no cartão verde do mips.

```
a. op=0, rs=3, rt=2, rd=3, shamt=0, funct=34

    op = 0 base10 -> 000000
    rs = 3 base10 -> 00011
    rt = 2 base10 -> 00010
    rd = 3 base10 -> 00011
 shamt = 0 base10 -> 00000
funct = 34 base10 -> 22 base16 -> 100010

sub $v1, $v0, $v1 # return x - y; em 'c/java'

Subtract sub R R[rd] = R[rs] - R[rt] (1) 0 / 22 hex
(1) May cause overflow exception

b. op=0x23, rs=1, rt=2, const=0x4

   op = 0x23 base16 -> 00100011
      rs = 1 base10 -> 00001
      rt = 2 base10 -> 00001
const  = 0x4 base16 -> 0000000000000100

lw $v0, 0x4($at)         # geralmente o $at quando
                         # vem como source assim foi carregado
                         # nele, automaticamente, um endereço
                         # lui $at, 0x1001
                         # lw $v0, 0x0004($at)
                         # M[10010004] <- $v0

Load Word lw I R[rt] = M[R[rs]+SignExtImm] (2) 23 hex
(2) SignExtImm = { 16{immediate[15]}, immediate }
```

2.15.4, questão de prova. 2.15.4 [5] 2.6 The table above shows different C statements that use logical operators. If the memory location at C[0] contains the integer values 0x00001234, and the initial integer values of A and B are 0x00000000 and 0x00002222, what is the result value of A ?

```
a. A = B | !A;
A recebe 0xFFFFDDDD

b. A = C[0] << 4;
temp recebe C[0*4=0]
temp = 0x00001234 << 4
temp = 0000 | 0000 | 0000 | 0001 | 0010 | 0011 | 0100 | 0000
temp = 0x00012340 = 0x12340
```