Aritmética e Sincronia de Processadores

Lucas

19 de setembro de 2015

1 Aritmética e sincronia de processadores, alguns comentários

87-93 Essa seção fala sobre complemento de dois, números com sinal e sem sinal. Importante para load words, branchs e operações aritméticas com sinal. Ver por exemplo que o subi para o MIPS não é nativo e se ele trabalhar com um subi com uma constante mais que 16 bits, maior do que o campo imediato poderá representar ele vai, inserir instruções para carregar a constante grande em um registrador e vai depois realizar um sub em cima dos registradores alvos, e colocar por fim o valor no registrador destino. Na página 94 ele explica que o complemento de um número x é $2^n - x - 1$ genericamente falando. Que 320's é zero, 1::310's é o maior valor negativo em 32 bits que você poderá ter nessa faixa de representação, assim 0::311's é o maior valor positivo que você poderá ter. Isso é útil para entender a faixa de deslocamento de instruções tipo I, branchs, loads, e stores.

109-110 Compiladores MIPS usam `slt, slti, beq, bne` e o valor fixado de 0 (sempre disponível em $zero) para criar todas as condições relativas: iguais, não-iguais, menor-ou-igual, maior-que, maior-ou-igual-que. Por causa do conselho de Neumann's sobre simplicidade do equipamento, a arquitetura MIPS não inclui 'branch on less then' por ser muito complicada, ou iria alongar demais em ciclos de relógio ou iria aumentar ciclos de relógio por instruções CPI. Duas instruções rápidas é mais útil. `slt $t0, $s0, $s1` é uma comparação sinalizada. e `sltu $t1, $s0, $s1` é uma comparação **não** sinalizada. Tratar números sinalizados como se eles não fossem, nos dá uma técnica baixo custo de checar $0 <= x < y$ e isso é muito útil por exemplo em Java que testa tudo sempre automaticamente quando se fala de Java Arrays.

```
Bounds Check Shortcut

                                  # Pula para IndexOutOfBounds se $s1 >= $t2
                                  # ou se $s1 é negativo ($1 < 0).
sltu $t0, $s1, $t2                # $t0 = 0 if $s1 >= length or $s1 < 0
beq  $t0, $zero, IndexOutOfBounds # if bad, goto ERROR
```

224-227 Essas páginas exploram a noção de subtração de valores usando adição de valores em complemento de dois.

```
  Subtraindo 6 decimal de 7 dec pode ser feito diretamente como:

- 0000 0000 0000 0000 0000 0000 0000 0111 = 7 dec
  0000 0000 0000 0000 0000 0000 0000 0110 = 6 dec
= 0000 0000 0000 0000 0000 0000 0000 0001 = 1 dec

  ou via adição usando complemento de 2 representando o -6

+ 0000 0000 0000 0000 0000 0000 0000 0111 =  7 dec
  1111 1111 1111 1111 1111 1111 1111 1010 = -6 dec
= 0000 0000 0000 0000 0000 0000 0000 0001 =  1 dec
```

Ou seja, $x - y = x + (-y)$ para a máquina. Na página 226 tem uma tabela muito interessante:

Operação	Operando A	Operando B	Resultado com Overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Tabela 1: Overflow conditions for addition and subtraction.

Na página 227 o autor explora uma ideia de Aritmética para Multimídia. Desde que cada microprocessador desktop por definição tem seu próprio display gráfico, e o custo dos transistores aumentou então foi inevitável que suporte deveria ser adicionado para operações gráficas. Muitos sistemas gráficos originalmente usam 8 bits para representar cada uma das 3 cores primárias mais 8 bits para a localização de um pixel. A adições de auto-falantes e caixa de som, para teleconferência e video games sugere suporte sonoro. Gravações de Áudio requerem 8 bits de precisão, mas 16 é o suficiente. Cada microprocessador tem suporte especial para que bytes e halfwords ocupem menos espaço em memória. Mas dado a não frequência de operações aritméticas nesses tamanhos de dados em programas tipicamente 32 bits. Existe pouco suporte em transferência de dados.

243 Tabela de instruções MIPS. Da tabela podemos lembrar que:

```
NOT pode ser implementado assim $s1 = ~ ($s2 | $zero)
```

Branchs são relativos ao PC+4 (ver tabela). Load Linked de fato trás da memória algum valor e armazena em `$alvo`, mas ele ativa um flag especial de operação atômica = 1, em seguida se um próximo store conditional avaliar que esse flag permanece = 1 ele armazena o que tem em seu registrador `$fonte` na posição de memória em que você ligou atomicamente, se ele conseguir armazenar lá, quer dizer que ele avaliou que aconteceu uma operação atômica corretamente, então ele sinaliza o seu registrador com o valor 1 nele. O store conditional usa o mesmo registrador fonte como destino depois de colocar o valor do fonte na posição da memória desejada.

247 IEEE 754 tem um símbolo para o resultado de operações invalida, tal como 0/0 ou subtraindo infinito de infinito. Esse simbolo é NaN, que é Not a Number.

137-139 Aqui nessas páginas ele trata o problema de semáforos em MIPS. Como eu expliquei acima, no item 243 essas instruções ll e sc são usadas para garantir troca de dados de maneira consistente, isto é, sem que ninguém se intrometa no meio, esse ninguém que eu falo são outros processadores fazendo data racing. Vamos analisar o código:

```
ll $t1, 0($s0)              ; vincula um flag de operação atômica

sc $t0, 0 ($s0)             ; verifica se a operação é atômica
                            ; se sim, então armazena o valor de $t0
                            ; na memória e retorna dentro do registrador
                            ; que é ao mesmo tempo fonte/destino
                            ; esse mesmo registrador é usado para testar caso
                            ; precise travar até concluir a operação.

lock: la $s0, sem           ; pega o endereço base da label sem
try:  ll $t1, 0($s0)        ; liga atômicamente com o endereço 0($s0)
      bne $t1, $0, try      ; se para esse endereço ainda não estiver disponível
                            ; operações atômicas, então tenta novamente o try:
                            ; até algum outro processador liberar esse 0($s0)

      addi $t1, $0, 1       ; esse addi representa operações complexas
                            ; que são opcionais, depende do código de cada um

      sc $t1, 0($s0)        ; armazena o valor de $t1 (que é um valor qualquer)
                            ; depende da lógica de cada pedaço de programa, etc.
                            ; armazena na mesma posição de memória a qual você
                            ; deu load linked, senão você pode estar fazendo
                            ; operações estranhas para os outros processadores.

      beq $t1, $0, try      ; se ainda sim, quando você tentou dar 'sc' na
                            ; na posição desejada, e ela estava ocupada por
                            ; outro processador, você devera tentar tudo dinovo.

      jr $ra                ; caso contrário, você conseguiu terminar sua op
                            ; atômica. parabéns você não está mexendo com
                            ; atomos. retorna ao procedimento chamador.
```

Para destravar o semáforo basta você setar 0 na posição de memória dedicada ao label 'sem'.

```
unlock: la $t1, sem         ; pega a posição de memória correta de 'sem'
        sw $zero, 0($t1)    ; armazena 32{0}'s lá, zera tudo, sinal verde.
        jr $ra              ; retorna ao procedimento chamador.
```

não fazer ~~2.9.1~~ → fazer 2.10.1 da edição 4 revisada.

```
a. 0000 0010 0001 0000 1000 0000 0010 0000 two

opcode rs    rt    rd    shamt funct
000000 10000 10000 10000 00000 100000

Então é tipo R pois opcode é 0x0. Agora veja no campo funct que vale 0x20.
Então a instruções no cartão verde é 0/20 hex.

É um add $rd, $rs, $rt (fiz dessa forma para você ver que é trocado a ordem dos
registradores em relação ao formato, mas fica alegre isso é o de menos.). A
instruções do tipo add podem causar overflow.

add $16, $16, $16 ou ainda de forma simbólica add $s0, $s0, $s0 lembrando você
que $s0 é preservado pela função chamada, $s[0-9] são preservador através da
chamada eu quero dizer.

b. 0000 0001 0100 1011 0100 1000 0010 0010 two

opcode 10    11    9     shamt funct
000000 01010 01011 01001 00000 100010

Opcode é 0x0 novamente então é do tipo R, agora saibamos qual é o campo funct.
E ele vale 0x22, basta buscar a instrução do 0/22 hex no mips green card. E é
um subtract.

sub $9, $10, $11 ou simbolicamente falando podemos colocar sub $t1, $t2, $t3.
```

não fazer ~~2.9.2~~ → fazer 2.10.2 da edição 4 rev. Ambas são tipo R. (Patterson & Hennesy - Computer Organization and Design, Edição 4 Revisada)

não fazer ~~2.9.4~~ → fazer 2.10.4 da edição 4 rev. In the following problems, the data table contains MIPS instructions. You will be asked to translate the entries into the bits of the opcode and determine the MIPS instruction format.

```
a.   addi $t0, $t0, 0
            \ /
             .                                    ; ele troca os campos de
            / \                                   ; lugar no formato I
           /   \
opcode   rs      rt      immed                    ; uma coisa experta de se
[001000] [01000] [01000] [0000000000000000]       ; fazer é contar os bits
31    26  25  21 20   16 15               0       ; para cada campo e depois
                                                  ; contar para ver se tem 32
                                                  ; bits no formato.

b.   sw $t1, 32($t2)
          \   /
           \ /                                    ; ele troca os campos de
            .                                     ; lugar no formato I
           / \
          |   \
          |    \
opcode   rs      rt      immed
[101011] [01010] [01001] [0000000000100000]
31    26  25  21 20   16 15               0
```

2.28.2

```
try: mov  r3, r4        ; move o conteúdo de r4 para r3
     ll   r2, 0(r2)     ; salva em r2 o conteúdo de 0(r2) e inic. op. atômica
     addi r2, r2, 1     ; incrementa r2 de 1
     sc   r3, 0(r1)     ; tenta finalizar op. atômica porém em outro endereço
     beqz r3, try       ; se não conseguir realizar op. atômica tenta novamente
                        ; isto é, se r3 retornar com 0 ele falhou na operação
                        ; atômica e deverá tentar novamente, se r3 retornar com
                        ; o valor 1 então deu certo e o branch não desloca para
                        ; try: "tentar novamente".
     mov  r4, r2        ; move o conteúdo de r2 para r4
```

O store conditional pode não estár apontando para a mesma posição de memória que o load linked. Isto é, pode ser que 0(r2) seja diferente de 0(r1). Mas o código irá executar normalmente. Digamos que os endereços são diferentes, então nesse caso, se não tiver outra thread fazendo load linked na posição 0(r1) igual a posição 0(r1) do store conditional aqui, então esse código vai ficar eternamente tentando pois r3 vai ser sempre 0. Mas se tem outra thread que faz load linked a tempo

do código acima verificar operação atômica então o store conditional aqui vai conseguir retornar 1 em seu registrador e vai conseguir terminar a operação atômica, e colocar o valor de r3 em 0(r1). Outro caso é se 0(r2) é igual a 0(r1), nesse caso então temos que o código executará no mínimo 6 instruções (todo o corpo), e no máximo $5n + 1$ instruções.

2.28.3

```
try:  mov  r3, r4        ; move o conteúdo de r4 para r3
      ll   r2, 0(r3)     ; salva em r2 o conteúdo de 0(r2) e inic. op. atômica

      bne  r2, $0, try   ; se não liberar o semáforo fica tentando

      addi r2, r2, 1     ; incrementa r2 de 1
      sc   r2, 0(r3)     ; tenta finalizar op. atômica porém em outro endereço
      beqz r2, try       ; se não conseguir realizar op. atômica tenta novamente
                         ; isto é, se r3 retornar com 0 ele falhou na operação
                         ; atômica e deverá tentar novamente, se r3 retornar
                         ; com o valor 1 então deu certo e o branch não desloca
                         ; para try: "tentar novamente".

      mov  r4, r2        ; move o conteúdo de r2 para r4
```

2.28.4a [5] 2.11 Fill out the table with the value of the registers for each given cycle. Each entry in the following table has code and also shows the contents of various registers. The notation "($s1)" shows the contents of a memory location pointed to by register $s1 . The assembly code in each table is executed in the cycle shown on parallel processors with a shared memory space.

Processador 1 (P1)	Processador 2 (P2)	Ciclo	P1 - $t1	P1 - $t0	($s1)	P2 - $t1	P2 - $t0
		0	1	2	99	30	40
	ll $t1, 0($s1)	1	1	2	99	99	40
ll $t1, 0($s1)		2	99	2	99	99	40
	sc $t0, 0($s1)	3	99	1	2	99	40
sc $t0, 0($s1)		4	99	1	2	99	0

Exemplo baseado no exercício anterior, mas com as instruções em lugares diferentes.

Processador 1 (P1)	Processador 2 (P2)	Ciclo	P1 - $t1	P1 - $t0	($s1)	P2 - $t1	P2 - $t0
		0	1	2	99	30	40
ll $t1, 0($s1)	ll $t1, 0($s1)	2	99	2	99	99	40
sc $t0, 0($s1)		3	99	1	2	99	40
	sc $t0, 0($s1)	4	99	1	2	99	0

2.28.4b Esse é um pouco mais complicado, mas da para fazer também, manda bala.

Processador 1 (P1)	Processador 2 (P2)	Ciclo	P1 - $t1	P1 - $t0	($s1)	P2 - $t1	P2 - $t0
		0	1	2	99	30	40
ll $t1, 0($s1)		1	99	2	99	30	40
	ll $t1, 0($s1)	2	99	2	99	99	40
	addi $t1, $t1, 1	3	99	2	99	100	40
	sc $t1, 0($s1)	4	99	2	100	1	40
sc $t0, 0($s1)		5	99	0	100	1	40