Questões de Operações

Lucas

8 de setembro de 2015

1 Operações, alguns comentários

Por que o MARS faz essa conversão?

```
addi $s1, $s2, 0xfffd   # quando coloca 0xfffd, ele faz essa:
=>
  lui $1, 0             (cod: 0x3c01 0000)
  ori $1, $1, 0xfffd    (cod: 0x3421 fffd)
  add $17, $18, $1      (cod: 0x0241 8820)
```

Ao invés dessa?

```
addi $s1, $s2, -3       # quando coloca -3, ele faz essa:
=>
  addi $s1, $s2, 0xfffd (cod: 0x2251 fffd)
```

pg. 161 compiladores modernos fazem o levantamento pesado, por exemplo você hoje não precisa mais ficar se preocupando em fazer tudo em código usando ponteiros, pois o compilador irá trabalhar para deixar seu código que usa arrays, listas, etc, bem optimizado para atingir boa performance. E ARM não tem um registrador contendo estilo $zero no mips, para operações aritméticas, etc. Mips tem 3 modos de endereçamento simples, ARM tem 9 modos mais complexos.

pg. 162 ARM inclui shifts como parte de cada instruções que opera em dados, os shifts lsl, lsr, asr são uma variação da instrução move no MIPS, ARM não tem instrução diz, diu já no MIPS essas instruções existem.

pg. 163 ARM usa flags para suas instruções de condições, tais como negative, zero, carry, overflow. Essas flags podem ser setadas em qualquer instruções lógica ou aritmética.

pg. 164 ARM tem um campo chamado Opx que aparece em todos os formatos e serve tanto para diferenciar se precisa ser lido mais informação para saber que instrução é, quanto para sinalizar que aquela instrução deve ser anulada (tornada um nop) caso necessário, então todas as instruções ARM podem virar nops. A figura na página 164 mostra bem isso, juntamente com a comparação com o formato das instruções MIPS. O campo com 12-bits imediato do ARM faz a seguinte operação binárias, ele estende com zeros para 32 bits, o valor é então rotacionado para direita o número de bits especificado nos primeiro 4 bits do campo multiplicado por 2. Essa operação faz com que se possa representar todas as potencias de 2 numa palavra de 32 bits.

pg. 165 ARM tem instruções para fazer `LOAD` e `STORE` de blocos de dados, MIPS não contém isso.

pg. 166 Pode-se perceber um aumento no número de instruções das arquiteturas da timeline dessas páginas, aumento de instruções especializadas e consideravelmente mais complexas para executar tarefas específicas de nichos específicos, instruções especiais sum of absolute differences, dot products for arrays of structures, sign or zero extension of narrow data to wider sizes, population count, etc (intel, 2006). Porém note que apesar disso sempre foram mantidas as instruções simples de sempre, e o compilador mapeia para essas instruções mais simples e estrincha sobre elas, com uma porcentagem baixa de uso das instruções complexadas, ou seja, RISC sempre esteve contido no núcleo de instruções.

pg. 168 **x86** provê suporte para ambos 8 bit (byte) e 16 bit (word). O 80386 adiciona endereçamento de 32 bits (double word). (AMD64 adiciona 64 endereçamento de 64 bits e dados de 64 bits, chamados quad words). x86 refere ao processador 80386 da Intel.

pg. 169 **x86** tem 8 registradores de propósito geral. começam com 'E' de **extended**.

pg. 170 A operação com inteiro do x86 pode ser dividida em quatro principais classes:

(a) Instruções de transações de dados, incluindo move, push e pop.

(b) Aritmética e lógica, incluindo operações aritméticas de teste, inteiros, e decimais.

(c) Controle de fluxo, incluindo branchs condicionais, jump incondicionais, chamadas de função e retornos.

(d) Instruções de strings, incluindo 'string move' e 'string compare'.

pg. 171 Contrário de ARM and MIPS, 80386 não tem todas suas instruções no formato 4 bytes, mas sim formatos muito variados em tamanho, desde 1 byte, quando não tem operadores, até 15 bytes.

pg. 172 Se olharmos na tabela 2.40 da página 172 vemos que muitas instruções são realizadas com a RISC. E instruções como `cmp`, `test`, que são realizadas usando `slt`, `sltu`, `sltiu`, etc.

pg. 173 Muitas instruções `x86` contém um campo de 1-bit que determina se a operação em questão é do tamanho de um byte ou de uma palavra dupla (8, 16 respectivamente). O campo `MOV` é usado em instruções que podem mover para memória ou trazer da memória, o campo `MOV` mostra a direção da transação. AMD e Intel tem recursos financeiros para sobreviver à adições de grandes complexidades aos seus respectivos conjunto de instruções. Na página 172 mostra só a superfície da arquitetura **x86**.

pg. 179 SPEC CPU2006 Benchmarks

add, sub, addi	16%
lw, sw, lb, lbu, lh, lhu,sub, lui	36%
and, or, nor, andi, ori, sll, srl	12%
beq, bne, slt, slti, sltiu	34%
j, jr, jal	2%

Nessa tabela da SPEC CPU2006 mostra 99% das instruções sendo mapeadas para RISC, o que me da 1% de instruções sendo mapeadas para o conjunto complexo do x86, AMD64.

2.34.1a Ao traduzir as instruções precisamos dar um jeito de simular o que acontece por trás da instruções ADD no ARM. Ela é uma instruções aritmética que seta flags (Z, N, O, C). Flag Zero, Flag Negative, Flag Overflow, Flag Carry.

a.

```
slt $t1, $s1, $s2 # em $t1 nós temos o mesmo valor que teríamos em
                  # Z, N

add $s0, $s1, $s2 # como não é uma adição condicional podemos fazer
                  # assim, porém ela seta antes ou depois as flags
```

b.

```
li $t0, 0x4            # carrega a constante 4 no regs t0
beq  $s0, $t0,  NOT    # s0 = 0x4 ? NOT else "ADDNE"
add  $t0, $s1, $s0     # seta dinovo a flag por causa de add
slt  $s1, $s1, $s0     # executa o add esperado
NOT: ...               # ADDNE contém uma branch implícita
                       # além disso, contém setação de flag.
```

2.35.2a Lembrar de gerar código como se você fosse um compilador, e você não sabe criar lógicas mirabolantes, apenas você segue o padrão e gera instruções, e depois você verifica se existem possíveis otimizações que também são padrões, nada de lógica mirabolante.

```
a. LDR r0, [r1, #4]  ; r0 = memory[r1+4], r1 += 4

lw $s0, 0x4($s1)     ; r0 recebe M[r1 + 4]
addi $s1, $s1, 0x4   ; r1 recebe r1 + 4
```

2.36.1a Nesse eu entendo que ele começa dá direita para esquerda.[a]

```
a. ADD, r3, r2, r1, LSR #4  ; r3 = r2 + (r1 >> 4)
                            ; lembrar também que a instrução aritmética
                            ; no armv7 seta flag, e no caso do ADD
                            ; pelo menos pode haver o caso dele setar
                            ; Flag de Carry, Flag de Overflow. É possível.
                            ; Então na verdade traduzir esse código para mips
                            ; não seria apenas duas instruções, pois você
                            ; teria que simular o esquema que o arm faz
                            ; setando as 4 flags de controle.
                            ; no mips isso daria algumas instruções
                            ; envolvendo stlu, stl, ou sub direto mesmo.

srl $s1, $s1, 4             ; r1 recebe r1 >> 4
add $s3, $s2, $s1           ; r3 recebe r2 + r1
sltu $t0, $s2, $s1          ; determina carry out em t0 = 0 ou 1, carry out 0
                            ; ou carry 1 respectivamente.

                            ; It coloca 0 ou 1 no registrador $t0 se ocorrer
                            ; carry out 0 ou 1, respectivamente.
                            ; Isso funciona por que $s3 == $s2 + $s1 se não
                            ; ocorrer carry out (ou adição sem sinal com
                            ; overflow equivalente). Então se não acontecer
                            ; carry out $s3 vai ser >= $s1 (ou $s3 >= $s2).
                            ; Se violar? então -> if $s3 for ao final da
                            ; operação < $s2 um carry out deve ter acontecido
                            ; Quando o carry out acontecer nós temos que
                            ; $s3 + 2^32 = $s2 + $s1.
```

2.37.2a

```
a. START: mov  eax, 3
          push eax
          mov  eax, 4           ; x86 não é uma máquina loadstore
          mov  ecx, 4           ; o que permite essa arquitetura
          add  eax, ecx         ; trabalhar com a memória diferenciadamente
          pop  ecx
```

[a] http://staff.science.nus.edu.sg/ phywjs/CZ101/labs-tutorials/tut5answr.html

```
          add  eax, ecx

(eax, ecx) -> ($s1, $s2)

STARTMIPS: addi $s1, $0, 3
           addi $sp, $sp, -4     ; Mips é uma máquina load store
           sw   $s1, 0($sp)      ; então você vai ter que fazer as
           addi $s1, $0, 4       ; transações com a memória, abrindo
           addi $s2, $0, 4       ; espaço na pilha e fechando depois.
           add  $s1, $s1, $s2
           lw   $s2, 0($sp)
           add  $sp, $sp, 4
           add  $s1, $s1, $s2
```

2.37.4b

```
b. test eax, 0x00200010

(eax) -> ($s1)
lui $at, 0x0020
ori $t0, $at, 0x0010
slt $t1, $s1, $t0
```