Organização de Computadores II (GRAD/DCC)
Arquitetura de Computadores (PÓS/DCC)
Universidade Federal de Minas Gerais

Trabalho Prático: MIPS32 Pipeline

Testes com instruções 1

Omar Paranaíba
Antônio Otávio Fernandes
Claudionor Nunes Coelho Jr.
Celina Gomes do Val
José Augusto Nacif
Thiago Sousa F. Silva

1 Procedimentos para o teste com instruções

A realização dos testes com instruções envolve uma seqüência de procedimentos que devem ser tomados para que um código escrito em linguagem *assembly* de MIPS possa ser transformado em um arquivo de entrada para simulação em nossa implementação do MIPS. Na figura 1 são ilustrados estes procedimentos, que serão detalhados adiante.

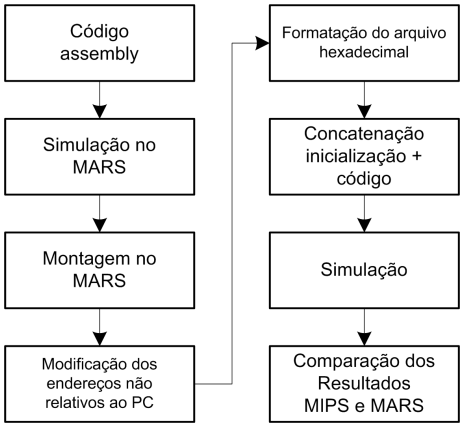

Figura 1: Procedimentos para o teste com instruções

Para os testes que poderão ser realizados, deve-se seguir os procedimentos descritos acima. Junto a este tutorial será disponiblizado alguns códigos de exemplo que serão utilizados para o teste de nossa implementação. Como ferramenta de apoio foi escolhido o simulador MARS devido a fácil utilização do mesmo, e também pelo recurso de montagem, fundamental para a geração dos arquivos de entrada necessários para cada simulação.

Para cada código *assembly* que vocês implementarem para teste, recomenda-se que os mesmos sejam simulados antes no MARS. Desta maneira, erros de implementação do código poderão ser verificados e corrigidos. Em seguida, deve ser realizado o processo de montagem do código, tranformando as instruções em um arquivo hexadecimal base para entrada no simulador escrito em **Verilog**. Após o processo de montagem, deve ser observado uma diferença entre a divisão de memória implementada no simulador MARS, e a divisão de memória implementada em nosso trabalho. A figura 2 apresenta a divisão utilizada no MARS. A figura 3 apresenta a divisão utilizada em nossa implementação.

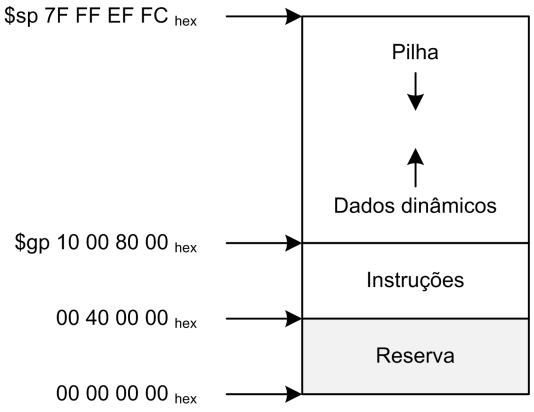

Figura 2: Divisão RAM no MARS

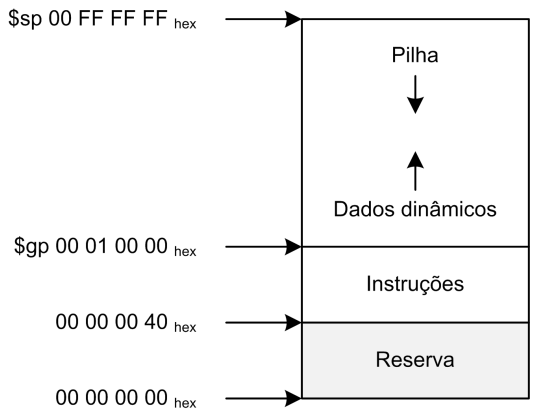

Figura 3: Divisão RAM nossa implementação

2 Instruções para os testes no MIPS

- **Carga inicial nos registradores:** executa instruções necessárias para modificar os registradores `$gp` e `$sp`. Este código é armazenado na área de reserva do MIPS descrito na figura 3. Estes registradores serão utilizados em alguns dos testes disponibilizados.

Código assembly do arquivo `registradores.s`:

```
1  .text
2  .globl __start
3
4  __start:
5
6  addi $s1, $zero, 0x0001
7  sll $s2, $s1, 16
8  add $gp, $zero, $s2
9  addi $s1, $zero, 0x00ff
10 sll $s2, $s1, 16
11 add $s3, $s2, $zero
12 sll $s2, $s1, 8
13 add $s3, $s3, $s2
14 #add $s3, $s2, $zero
15 add $s3, $s3, $s1
16 #addi $s3, $s2, 0xff00
17 add $sp, $zero, $s3
18 nop
19 nop
20 nop
21 nop
```

Comando para montagem do código acima:

```
java -jar Mars.jar a dump 0x400000-0x10000000 HexText registradores.hex registradores.s
```

Formato de saída do arquivo `registradores.hex`:

```
20110001
00119400
0012e020
201100ff
00119400
02409820
00119200
02729820
02719820
0013e820
00000000
00000000
00000000
00000000
```

Espaçamento entre bytes necessários para leitura no `cver`.

```
20 11 00 01
00 11 94 00
00 12 e0 20
20 11 00 ff
00 11 94 00
02 40 98 20
00 11 92 00
02 72 98 20
02 71 98 20
00 13 e8 20
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
```

- **Teste 1:** testa as instruções `addi`, `stl`, `sw` e `add` no pipeline e o tratamento de conflito de dados feito pela unidade de repasse.

Código assembly do arquivo `teste1.s`:

```
1  .text
2  .globl __start
3
4  __start:
5
6  addi $a0, $zero, 100
7  slt  $a1, $zero, $a0
8  sw   $a0, 0($gp)
9  add  $a2, $a1, $a0
10 nop
11 nop
12 nop
13 nop
14 nop
```

Comando para montagem do código acima:

```
java -jar Mars.jar a dump 0x400000-0x10000000 HexText teste1.hex teste1.s
```

Formato de saída do arquivo `teste1.hex`:

```
20040064
0004282a
af840000
00a43020
00000000
00000000
00000000
00000000
00000000
```

Espaçamento entre bytes necessários para leitura no `cver`.

```
20 04 00 64
00 04 28 2a
af 84 00 00
00 a4 30 20
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
```

Para simulação do programa `teste1.s`, deve ser gerado um arquivo `memoria.txt` com as instruções de inicialização geradas em `registradores.hex`, seguido pelas instruções do programa `teste1.hex` que será simulado. A seqüência de comandos necessários para a criação do arquivo `memoria.txt` são descritos a seguir:

```
cat registradores.hex > memoria.txt
cat teste1.hex >> memoria.txt
```

A simulação é realizada pelo comando:

```
cver tb_mips.v | less
```

No final da simulação será gerado uma saída com os registradores e as posições de memória modificadas.

```
...

Reg[ 0] : 00000000      Reg[16] : 00000000
Reg[ 1] : 00000000      Reg[17] : 000000ff
Reg[ 2] : 00000000      Reg[18] : 0000ff00
Reg[ 3] : 00000000      Reg[19] : 00ffffff
Reg[ 4] : 00000064      Reg[20] : 00000000
Reg[ 5] : 00000001      Reg[21] : 00000000
Reg[ 6] : 00000065      Reg[22] : 00000000
Reg[ 7] : 00000000      Reg[23] : 00000000
Reg[ 8] : 00000000      Reg[24] : 00000000
Reg[ 9] : 00000000      Reg[25] : 00000000
Reg[10] : 00000000      Reg[26] : 00000000
Reg[11] : 00000000      Reg[27] : 00000000
Reg[12] : 00000000      Reg[28] : 00010000
Reg[13] : 00000000      Reg[29] : 00ffffff
Reg[14] : 00000000      Reg[30] : 00000000
Reg[15] : 00000000      Reg[31] : 00000000


Memoria[65536+  0] : 0000 0064
Memoria[65536+  4] : 0000 0000
Memoria[65536+  8] : 0000 0000
Memoria[65536+ 12] : 0000 0000
Memoria[65536+ 16] : 0000 0000
Memoria[65536+ 20] : 0000 0000
Memoria[65536+ 24] : 0000 0000
Memoria[65536+ 28] : 0000 0000
Memoria[65536+ 32] : 0000 0000
Memoria[65536+ 36] : 0000 0000
Memoria[65536+ 40] : 0000 0000
Memoria[65536+ 44] : 0000 0000
Memoria[65536+ 48] : 0000 0000
Memoria[65536+ 52] : 0000 0000
Memoria[65536+ 56] : 0000 0000
Memoria[65536+ 60] : 0000 0000
Memoria[65536+ 64] : 0000 0000
Memoria[65536+ 68] : 0000 0000
Memoria[65536+ 72] : 0000 0000
Memoria[65536+ 76] : 0000 0000
Memoria[65536+ 80] : 0000 0000
Memoria[65536+ 84] : 0000 0000
Memoria[65536+ 88] : 0000 0000
Memoria[65536+ 92] : 0000 0000
Memoria[65536+ 96] : 0000 0000
Memoria[65536+100] : 0000 0000
Memoria[65536+104] : 0000 0000
Memoria[65536+108] : 0000 0000
Memoria[65536+112] : 0000 0000
Memoria[65536+116] : 0000 0000
Memoria[65536+120] : 0000 0000
```

- **Fibonacci(10):** gera a seqüência de Fibonacci para $n = 10$. Testa algumas instruções aritméticas, *store word*, e do tipo *jump* e *branch*.

Código assembly do arquivo `fib.s`:

```
1  .text
2  .globl __start
3
4  __start:
5
6          # n value
7          addi $a0, $zero, 10
8
9          # parameters a = $a1, b = $a2, sum = $a3 and i = $t0
10         addi $a1, $zero, 0 # a
11         addi $a2, $zero, 1 # b
12         addi $a3, $zero, 0 # sum
13         addi $t0, $zero, 0 # i
14
15         jal loop
16         nop
17
18         j end
19         nop
20
21 loop:
22
23         slt $t1, $t0, $a0
24         nop
25         nop
26         nop
27         nop
28
29         blez $t1, go_main
30         nop
31
32         sw $a1, 0($gp)
33
34         add $a3, $a1, $a2
35         add $a1, $zero, $a2
36         add $a2, $zero, $a3
37
38         addi $gp, $gp, 4
39         addi $t0, $t0, 1
40
41         j loop
42         nop
43
44 go_main:
45
46         jr $ra
47         nop
48
49 end:
50
51         nop
```

Comando para montagem do código acima:

```
java -jar Mars.jar a dump 0x400000-0x10000000 HexText fib.hex fib.s
```

Visualizando a transformação código *assembly* no código que será processado no MIPS.

```
java -jar Mars.jar a dump 0x400000-0x10000000 SegmentWindow fib.txt fib.s
```

```
1   Address    Code        Basic                    Source
2
3  0x00400000  0x2004000a  addi $4,$0,10       7          addi $a0, $zero, 10
4  0x00400004  0x20050000  addi $5,$0,0        10         addi $a1, $zero, 0 # a
5  0x00400008  0x20060001  addi $6,$0,1        11         addi $a2, $zero, 1 # b
6  0x0040000c  0x20070000  addi $7,$0,0        12         addi $a3, $zero, 0 # sum
7  0x00400010  0x20080000  addi $8,$0,0        13         addi $t0, $zero, 0 # i
8  0x00400014  0x0c100009  jal 4194340         15         jal loop
9  0x00400018  0x00000000  nop                 16         nop
10 0x0040001c  0x0810001a  j 4194408           18         j end
11 0x00400020  0x00000000  nop                 19         nop
12 0x00400024  0x0104482a  slt $9,$8,$4        23         slt $t1, $t0, $a0
13 0x00400028  0x00000000  nop                 24         nop
14 0x0040002c  0x00000000  nop                 25         nop
15 0x00400030  0x00000000  nop                 26         nop
16 0x00400034  0x00000000  nop                 27         nop
17 0x00400038  0x19200009  blez $9,9          29         blez $t1, go_main
18 0x0040003c  0x00000000  nop                 30         nop
19 0x00400040  0xaf850000  sw $5,0($28)        32         sw $a1, 0($gp)
20 0x00400044  0x00a63820  add $7,$5,$6        34         add $a3, $a1, $a2
21 0x00400048  0x00062820  add $5,$0,$6        35         add $a1, $zero, $a2
22 0x0040004c  0x00073020  add $6,$0,$7        36         add $a2, $zero, $a3
23 0x00400050  0x239c0004  addi $28,$28,4      38         addi $gp, $gp, 4
24 0x00400054  0x21080001  addi $8,$8,1        39         addi $t0, $t0, 1
25 0x00400058  0x08100009  j 4194340           41         j loop
26 0x0040005c  0x00000000  nop                 42         nop
27 0x00400060  0x03e00008  jr $31             46         jr $ra
28 0x00400064  0x00000000  nop                 47         nop
29 0x00400068  0x00000000  nop                 51         nop
```

Repare que nas linhas 8, 10, e 25 são utilizadas instruções do tipo *jump*. Estas instruções foram montadas pelo MARS, e possuem respectivamente os valores hexadecimais de saída `0x0c100009`, `0x0810001a`, e `0x08100009`. Como o segmento de instruções do simulador MARS começa em `0x00400000` (valor inicial do PC no simulador), e nossa implementação do MIPS possui o segmento de instruções começando em `0x00000040`, as posições de memória endereçadas pelas instruções não relativas ao PC deverão ser modificadas. A modificação é feita observando a maneira como os endereços foram montados pelas instruções de desvio incondicional (*jump*). Sabe-se que estas instruções são processadas da seguinte maneira: os 4 bits mais significativos do próximo PC serão concatenados com os 26 bits do campo de índice da instrução, que são concatenados com mais 2 bits zeros do alinhamento, formando o valor do PC para o desvio. Contudo para que estes valores sejam gerados corretamente no processamento destas instruções, as linhas 8, 10 e 25 do arquivo `fib.hex` devem ser alteradas recebendo os valores: `0c000019`, `0800002a`, e `08000019`.

Formato de saída do arquivo `fib.hex` modificado:

```
2004000a
20050000
20060001
20070000
20080000
0c000019
00000000
0800002a
00000000
0104482a
00000000
00000000
00000000
00000000
19200009
00000000
af850000
00a63820
00062820
00073020
239c0004
21080001
08000019
```

```
00000000
03e00008
00000000
00000000
```

Espaçamento entre bytes necessários para leitura no `cver`.

```
20 04 00 0a
20 05 00 00
20 06 00 01
20 07 00 00
20 08 00 00
0c 00 00 19
00 00 00 00
08 00 00 2a
00 00 00 00
01 04 48 2a
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
19 20 00 09
00 00 00 00
af 85 00 00
00 a6 38 20
00 06 28 20
00 07 30 20
23 9c 00 04
21 08 00 01
08 00 00 19
00 00 00 00
03 e0 00 08
00 00 00 00
00 00 00 00
```

Da mesma maneira como foi gerado o arquivo `memoria.txt` para o teste do programa `teste1.s`, a seqüência de comandos abaixo é necessária para geração do código de entrada para simulação do programa `fib.s`.

```
cat registradores.hex > memoria.txt
cat fib.hex >> memoria.txt
```

A simulação é realizada pelo comando:

```
cver tb_mips.v | less
```

O resultado da simulação do programa **Fibonacci** segue abaixo. Repare que a seqüência é armazenada nas posições 65536+0 até 65536+36 na memória. Repare que o valor 65536 é o valor em decimal que foi atribuído ao registrador `$gp`, pelo código de inicialização dos registradores.

```
...

Reg[ 0] : 00000000      Reg[16] : 00000000
Reg[ 1] : 00000000      Reg[17] : 000000ff
Reg[ 2] : 00000000      Reg[18] : 0000ff00
Reg[ 3] : 00000000      Reg[19] : 00ffffff
Reg[ 4] : 0000000a      Reg[20] : 00000000
Reg[ 5] : 00000037      Reg[21] : 00000000
Reg[ 6] : 00000059      Reg[22] : 00000000
Reg[ 7] : 00000059      Reg[23] : 00000000
Reg[ 8] : 0000000a      Reg[24] : 00000000
Reg[ 9] : 00000000      Reg[25] : 00000000
Reg[10] : 00000000      Reg[26] : 00000000
Reg[11] : 00000000      Reg[27] : 00000000
Reg[12] : 00000000      Reg[28] : 00010028
Reg[13] : 00000000      Reg[29] : 00ffffff
Reg[14] : 00000000      Reg[30] : 00000000
Reg[15] : 00000000      Reg[31] : 00000058


Memoria[65536+  0] : 0000 0000
Memoria[65536+  4] : 0000 0001
Memoria[65536+  8] : 0000 0001
Memoria[65536+ 12] : 0000 0002
Memoria[65536+ 16] : 0000 0003
Memoria[65536+ 20] : 0000 0005
Memoria[65536+ 24] : 0000 0008
Memoria[65536+ 28] : 0000 000d
Memoria[65536+ 32] : 0000 0015
Memoria[65536+ 36] : 0000 0022
Memoria[65536+ 40] : 0000 0000
```