Universiteit van Amsterdam

Lab 5

Architectuur en Computerorganisatie

18 oktober 2015

Student:
Sander Hansen
10995080

Practicumgroep:
Assembly

1 Lab report

Data wordt van en naar de *cache* getransporteerd door *cache lines*. De *cache lines* in de gegeven architectuur bestaat uit 128_{10} bits ofwel 16_{10} bytes. Dat betekent dat er maximaal 16_{10} bytes aan data uit het cache geheugen kan worden getransporteerd.

Het adres van de *cache* bestaat uit een *offset*, *index* en *tag*. De *offset* is 4_{10} bits want $2^4 = 16_{10}$. We maken hier gebruik van *two way set associative cache* dus hebben we 8_{10} mogelijke plekken waar de data zich kan bevinden. $^2\log(8) = 3$. Dit betekent dat de *index* uit 3 bits bestaat. Dit betekent dus dat de tag uit 32_{10} - 3_{10} - 4_{10} = 25_{10} bits moet bestaan.

Als je in de *Two's Complement* 128_{10} en het adres 'AND'. Dan worden de laatste 7_{10} bits 0. Op deze manier hou je alleen de tag over en je kan je dus met de tag werken. De drie XNOR elementen in de architectuur vergelijken of de twee input's het zelfde zijn. De meest rechter XNOR zou ook vervangen kunnen worden door een NOR port. Het is namelijk niet mogelijk dat er twee keer een 1 uit de eerste twee XNOR gates komt. Dus hoeft er alleen maar 1 uit te komen als beide XNOR gates 0 resulteren. Dit is precies wat een XOR gate.

Een matrix is een twee dimensionale *array*. Zo een matrix is met de volgende code te vermenigvuldigen.

```
1  for (int i = 0; i < 16; i++)
2  {
3      for (int j = 0; j < 16; j++)
4      {
5          int sum = 0;
6          for (int k = 0; k < 16; k++)
7              sum += a[i * 16 + k] * b[k * 16 + j];
8          res[i * 16 + j] = sum;
9      }
10 }
```

Dit kan uiteraard ook in *assembly* worden omgezet. Er zijn dan een aantal dingen waar we rekening mee moeten houden; zo kunnen er *Data hazards* optreden en moet er rekening gehouden worden met de *offset* van de onderdelen in de Array

Om een array in te laden kan de volgende code worden gebruikt, hierbij wordt eerst de *offset* bepaald.

```
1 ADD $9, $0, $0
2 SLL $9, $3, 4
3 ADD $9, $9 ,$6
4 SLL $9, $9, 2
5
6 LW $10, a, $9
```

De *offset* wordt hier berekend voor [i * 16 + k]. Hierin is i register 3 en k register 6. Na dat de *offset* op 0 is gezet wordt deze door middel van een *shift* instructie met 16 vermenigvuldigt en vervolgens wordt hier k bij opgeteld. Bij de *load word* instructie wordt in register 10 array a met de zojuist berkende *offset* geladen.

Als beide array's met de juist *offset* zijn geladen, moeten deze onderdelen met elkaar vermenigvuldigt worden. Als we dit echter direct zouden doen zou er een *data hazard* op treden. De *load word* instructie heeft tijd nodig voordat het register waarin hij geladen wordt beschikbaar is. Dit kunnen we simpel op lossen door twee NOP instructies toe te voegen.

```
1 LW $11, b, $9
2
3 NOP
4 NOP
5 MUL $7, $10, $11
```

Nadat b is ingeladen, wordt er hier dus met twee NOP instructies de tijd gegeven om de LW instructie te voltooien. Helaas zaten er in het uiteindelijke programma alsnog een aantal fouten, het is echter niet gelukt om te ontdekken of dit aan de architectuur lag of aan de code. Dit omdat er aangegeven is dat er wellicht een fout in de architectuur kan zitten. De code compileerde en was volledig uit te voeren, de uiteindelijke uitkomst klopte echter niet.

Een matrix kan je transponeren. Dit betekent dat de kolommen van een matrix rijen worden, en vice versa. Dit kunnen we implementeren door de berekening anders te maken. Hiervoor moet b[k * 16 + j] veranderd worden in b[j * 16 + k]. We gaan dit met de bovenstaande assembly code doen. Hiervoor passen we de berekening van de *offset* aan van de onderstaande code naar de daarop volgende. (Hierin is register 9 de offset, 4 staat voor j en 6 voor k)

```
1 ADD $9, $0, $0
2 SLL $9, $6, 4
3 ADD $9, $9 ,$4
4 SLL $9, $9, 2
```

```
1 ADD $9, $0, $0
2 SLL $9, $4, 4
3 ADD $9, $9 ,$6
4 SLL $9, $9, 2
```

Als we dit doen zien we ook een performance verbetering. Het eerste programma doet er 61472 *clock cycles* over om te voltooien. Bij het tweede programma ligt dit ongeveer 8000 clock cycles lager op 52664. Er is dus een grote performance verbetering zichtbaar. Op deze manier komen er namelijk minder *cache misses* voor dit is te wijten aan het feit dat hij bij de tweede manier *cache* direct na elkaar laad in plaats van telkens een ruimte van 16 ertussen. Bij de tweede manier kunnen we alles dus direct opvragen.

2 Appendix

De bestanden uit de appendix zijn ook bijgevoegd als bijlage.

2.1 a - Vermenigvuldiging met matrixen

```
1  # Sander Hansen 10995080
2  @include "Mips.wasm"
3
4  .data MyRegisters : REGISTERS
5  0x00 : WORD 0x00 # put $0 to zero (default assumption)
6
7  .data MyMemory : DATAMEM
8  @include "matrixes.wasm"
9  0x300 : WORD c 0 # you can put the result matrix starting at c
10
11 .code MyCode: MIPS, MyMemory
12
13 start:
14         ADDI $1, $1, 15 #zet controle op 15
15         ADD $3, $0, $0 #zet i op 0
16
17 loop:
18         BEQ $1, $3, exit #kijk of i < 16 is
19         ADD $4, $0, $0 #zet j op 0
20
21 loop2:
22         BEQ $1, $4, loopoptellen #kijk of j < 16 is
23
24         ADD $5, $0, $0 #zet sum op 0
25         ADD $6, $0, $0 #zet k op 0
26
27 loop3:
28         BEQ $1, $6, optellen #kijk of k < 16 is
29
30         ADD $9, $0, $0
31         SLL $9, $3, 4
32         ADD $9, $9 ,$6
33         SLL $9, $9, 2
34
35         LW $10, a, $9 #na het bepalen van offset matrix laden
36
37         ADD $9, $0, $0
38         SLL $9, $6, 4
39         ADD $9, $9 ,$4
40         SLL $9, $9, 2
41         LW $11, b, $9 #na het bepalen van offset matrix laden
42
43         NOP
44         NOP
45         MUL $7, $10, $11
46         ADD $5, $5, $7 #tel sum op bij sum + antwoord
47
```

```
48
49          ADDI $6, $6, 1 #tel bij k 1 op
50          BEQ $0, $0, loop3 #spring terug begin van loop3
51
52  optellen:
53
54          ADD $9, $0, $0
55          SLL $9, $6, 4
56          ADD $9, $9 ,$4
57          SLL $9, $9, 2
58          SW $5, c, $9
59
60          ADDI $4, $4, 1 #tel bij j 1 op
61          BEQ $0, $0, loop2 #spring terug begin loop2
62
63  loopoptellen:
64
65          ADDI $3, $3, 1 #tel bij i 1 op
66          BEQ $0, $0, loop #spring terug begin loop
67
68
69
70
71  NOP
72
73  exit:
```

2.2 b - Getransponeerde matrix vermenigvuldiging

```
1  # Sander Hansen 10995080
2  @include "Mips.wasm"
3
4  .data MyRegisters : REGISTERS
5  0x00 : WORD 0x00 # put $0 to zero (default assumption)
6
7  .data MyMemory : DATAMEM
8  @include "matrixes.wasm"
9  0x300 : WORD c 0 # you can put the result matrix starting at c
10
11 .code MyCode: MIPS, MyMemory
12
13 start:
14         ADDI $1, $1, 15 #zet controle op 15
15         ADD $3, $0, $0 #zet i op 0
16
17 loop:
18         BEQ $1, $3, exit #kijk of i < 16 is
19         ADD $4, $0, $0 #zet j op 0
20
21 loop2:
22         BEQ $1, $4, loopoptellen #kijk of j < 16 is
23
24         ADD $5, $0, $0 #zet sum op 0
25         ADD $6, $0, $0 #zet k op 0
26
27 loop3:
28         BEQ $1, $6, optellen #kijk of k < 16 is
29
30         ADD $9, $0, $0
31         SLL $9, $3, 4
32         ADD $9, $9 ,$6
33         SLL $9, $9, 2
34
35         LW $10, a, $9 #na het bepalen van offset matrix laden
36
37         ADD $9, $0, $0
38         SLL $9, $4, 4
39         ADD $9, $9 ,$6
40         SLL $9, $9, 2
41         LW $11, b, $9 #na het bepalen van offset matrix laden
42
43         NOP
44         NOP
45         MUL $7, $10, $11
46         ADD $5, $5, $7 #tel sum op bij sum + antwoord
47
48
49         ADDI $6, $6, 1 #tel bij k 1 op
50         BEQ $0, $0, loop3 #spring terug begin van loop3
51
52 optellen:
```

```
53
54          ADD $9, $0, $0
55          SLL $9, $6, 4
56          ADD $9, $9 ,$4
57          SLL $9, $9, 2
58          SW $5, c, $9
59
60          ADDI $4, $4, 1 #tel bij j 1 op
61          BEQ $0, $0, loop2 #spring terug begin loop2
62
63  loopoptellen:
64
65          ADDI $3, $3, 1 #tel bij i 1 op
66          BEQ $0, $0, loop #spring terug begin loop
67
68
69
70
71  NOP
72
73  exit:
```