TDTS08: Lab Report

Lab 4: VLIW Processors

Name	PIN	Email
Alexander Yngve	930320-6651	aleyn573@student.liu.se
Pål Kastman	851212-7575	palka285@student.liu.se

Contents

1 Introduction **3**

2 Method **3**
2.1 Basic Block 3
2.2 Dependencies 3
2.3 VLIW 4

3 Result **4**

4 Discussion **7**

A Graph file **8**

B VLIW files **9**

1 Introduction

The purpose of this lab was to convert normal sequential code to VLIW instructions, so that we would get a greater performance. Basically we do what the VLIW compiler does during compilation time.

2 Method

The approach for this lab was the following:

1. Choose a basic block, and disassemble the block.
2. Find dependencies between the instruction in the block.
3. We pack the instruction into VLIWs.

2.1 Basic Block

To view all the basic blocks within the program (go.ss) the following command is issued:

```
vliwc /home/TDTS08/bin/go.ss | sort -nr +3 -4 | less
```

A basic block with at least 15 instructions is needed for the lab.

The next step is to view the disassemble the program to view the code for the basic block. This is done with the command below:

```
sslittle-na-sstrix-objdump -d /home/TDTS08/bin/go.ss | less
```

Which gives output similiar to this:

```
41b528:     28 00 00 00     lw $3,16($29)
41b52c:     10 00 03 1d
41b530:     28 00 00 00     lw $4,-31444($28)
41b534:     2c 85 04 1c
41b538:     a2 00 00 00     lui $6,4100
41b53c:     04 10 06 00
41b540:     43 00 00 00     addiu $6,$6,-6208
41b544:     c0 e7 06 06
```

2.2 Dependencies

To pack the instructions into Very Long Instruction Words, the dependencies between the instructions must be resolved. The dependencies of interest are the true data dependencies, read after write, where the output of one instruction is required as an input to one of the following instructions.

Output dependencies (write after write) and anti-dependencies (write after read) are not considered since the are artificial dependencies which can be resolved in preprocessing.

The true data dependencies should be visualized in a directed graph where the address of each instruction is a node. The edges between the nodes symbolizes a dependency. This graph should also be described in a textual representation which will be used by the *vliwc* program. An example of the graph file looks like this:

```
0x00000001
0x00000001 0x00000002
```

This file describes two instructions, *0x00000001* which is independent, and *0x00000002* which is dependent on *0x00000001*.

2.3 VLIW

The last step is to pack the sequential instructions obtained in section 2.1 into VLIW format with the help of the dependency graph from section 2.2.

The text format for the VLIW file begins with a line which specifies *alu_no*, *mul_no*, *fpu_no* and *bau_no* - how many ALUs, MULs, FPUs and BAUs the VLIW processor will have.

The next lines are the Very Long Instruction Words in the form of addresses to the sequential instructions. The first *alu_no* instructions will be ALU instructions, the next *mul_no* instructions will be MUL instrustions and so on.

An example VLIW file can look like this:

```
1           1   1   2
nop         nop nop 0x0041b528 0x0041b538
0x0041b540 nop nop 0x0041b530 nop
```

3 Result

The chosen basic block is number 2292 of the benchmark *go.ss* and consists of 16 instructions. The disassembly is shown below:

```
41b528:     28 00 00 00     lw $3,16($29)
41b52c:     10 00 03 1d
41b530:     28 00 00 00     lw $4,-31444($28)
41b534:     2c 85 04 1c
41b538:     a2 00 00 00     lui $6,4100
41b53c:     04 10 06 00
41b540:     43 00 00 00     addiu $6,$6,-6208
41b544:     c0 e7 06 06
41b548:     a2 00 00 00     lui $2,4097
41b54c:     01 10 02 00
```

```
41b550:       28 00 00 00     lw $2,5960($2)
41b554:       48 17 02 02
41b558:       a2 00 00 00     lui $5,4100
41b55c:       04 10 05 00
41b560:       43 00 00 00     addiu $5,$5,30352
41b564:       90 76 05 05
41b568:       55 00 00 00     sll $3,$3,0x2
41b56c:       02 03 03 00
41b570:       42 00 00 00     addu $6,$3,$6
41b574:       00 06 06 03
41b578:       28 00 00 00     lw $7,0($6)
41b57c:       00 00 07 06
41b580:       42 00 00 00     addu $5,$3,$5
41b584:       00 05 05 03
41b588:       42 00 00 00     addu $2,$16,$2
41b58c:       00 02 02 10
41b590:       42 00 00 00     addu $2,$2,$7
41b594:       00 02 07 02
41b598:       34 00 00 00     sw $2,0($6)
41b59c:       00 00 02 06
41b5a0:       01 00 00 00     j 41b758 <try_connect+0x750>
41b5a4:       d6 6d 10 00
```

Dependency analysis of the above code results in the graph shown in figure 1. The graph file is also included in appndix A. Note that the unconditional jump instruction at address *41b5a0* doesn't have any data dependencies but still needs to be run last due to procedural dependencies.

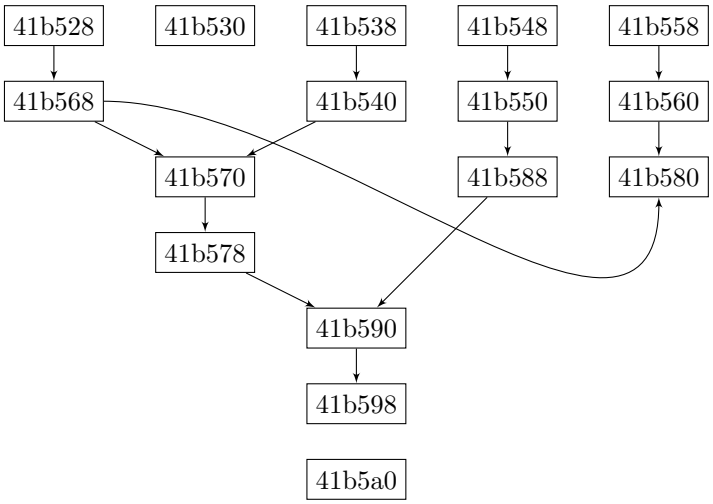

Figure 1 – Data dependencies between the instructions.

We first choose to make our architecture with no limit, this design can be seen in table 1.

Table 1 – When using 3 ALU, 1 MUL, 1 FPU and 5 BAU units.

ALU			MUL	FPU	BAU				
					41b528	41b530	41b538	41b548	41b558
41b540	41b568	41b560			41b560				
41b570	41b588	41b580							
					41b578				
41b590									
					41b598	41b5a0			

We then tried to reduce some of the units without loosing any performance, we first by tried by removing one of the BAUs (figure 2), since those are the most expensive units. In table 5 we can see that this design (VLIW2) doesn't perform any worse than the first design (VLIW1), so we continued to reduce more units to try and get an even cheaper design for this block.

Table 2 – When using 3 ALU, 1 MUL, 1 FPU and 3 BAU units.

ALU			MUL	FPU	BAU		
					41b528	41b558	41b538
41b568	41b540	41b560			41b548	41b530	
41b570	41b580				41b550		
41b588					41b578		
41b590							
					41b598	41b5a0	

We tried to remove one more BAU, and one ALU (table 3). In table 5 we can see that the performance is still the same as in the previous design, even though we have reduced it a lot.

Table 3 – When using 2 ALU, 1 MUL, 1 FPU and 2 BAU units.

ALU		MUL	FPU	BAU	
				41b528	41b538
41b568	41b540			41b548	41b558
41b570	41b560			41b550	41b530
41b588	41b580			41b578	
41b590					
				41b598	41b5a0

If we take a look in table 6 we can see that the by far most expensive unit is the BAU, and by reducing this from 5 to 2 units we have reduced the price by more than half of the first deigns price. Because of this, we wanted to make sure we weren't able to reduce the design even further, and so we tried using only 1 BAU unit (table 4).

Table 4 – When using 2 ALU, 1 MUL, 1 FPU and 1 BAU.

ALU		MUL	FPU	BAU
				41b528
				41b538
41b568	41b540			41b558
41b570	41b560			41b548
41b580				41b550
41b588				41b578
41b590				41b530
				41b598
				41b5a0

Instead of 6 clock cycles, we now got 9. Which means the design is 50% worse than the other ones. thus we stopped here.

Table 5

Design	VLIW1	VLIW2	VLIW3	VLIW4
No. ALU	3	3	2	2
No. MUL	1	1	1	1
No. FPU	1	1	1	1
No. BAU	5	3	2	1
Total Cost	554	354	252	152
No. Cycles	6	6	6	9
Cost per. ratio	3324	2124	1512	1368

The prices for the different units can be seen in table 6.

Table 6

Pricelist	
ALU cost	2
MUL cost	16
FPU cost	32
BAU cost	100

4 Discussion

Of course our results are only for one block of code and therefore our design of course might not be a very good one, except here. We choose to use an FPU and an MUL even though they weren't needed in our block, We think that this can be justified by arguing that without these, the design would be completely useless for other code blocks.

Depending on if cost or performance is the most important factor designs 3 (table 3) and 4 (table 4) respectively, are the best. Design 3 gives optimal performance for this block at a reasonably low cost. Design 4 gives the lower performance but at an even lower cost.

A Graph file

```
0x0041b528
0x0041b530
0x0041b538
0x0041b548
0x0041b558

0x0041b528 0x0041b568
0x0041b538 0x0041b540
0x0041b548 0x0041b550
0x0041b558 0x0041b560

0x0041b568 0x0041b570
0x0041b540 0x0041b570

0x0041b568 0x0041b580
0x0041b560 0x0041b580

0x0041b570 0x0041b578

0x0041b550 0x0041b588

0x0041b578 0x0041b590
0x0041b588 0x0041b590

0x0041b590 0x0041b598

0x0041b5a0
```

B VLIW files

Listing 1 – VLIW 1

```
3                                   1   1   5
nop         nop         nop         nop nop 0x0041b528 0x0041b530 0x0041b538 0x0041b548 0x0041b558
0x0041b540 0x0041b568 0x0041b560 nop nop 0x0041b550 nop         nop         nop
nop
0x0041b570 0x0041b588 0x0041b580 nop nop nop         nop         nop         nop
nop
nop         nop         nop         nop nop 0x0041b578 nop         nop         nop
nop
0x0041b590 nop         nop         nop nop nop         nop         nop         nop
nop
nop         nop         nop         nop nop 0x0041b598 0x0041b5a0 nop         nop
nop
```

Listing 2 – VLIW 2

```
3                                   1   1   3
nop         nop         nop         nop nop 0x0041b528 0x0041b558 0x0041b538
0x0041b568 0x0041b540 0x0041b560 nop nop 0x0041b548 0x0041b530 nop
0x0041b570 nop         0x0041b580 nop nop 0x0041b550 nop         nop
0x0041b588 nop         nop         nop nop 0x0041b578 nop         nop
0x0041b590 nop         nop         nop nop nop         nop         nop
nop         nop         nop         nop nop 0x0041b598 0x0041b5a0 nop
```

Listing 3 – VLIW 3

```
2                   1   1   2
nop         nop         nop nop 0x0041b528 0x0041b538
0x0041b568 0x0041b540 nop nop 0x0041b548 0x0041b558
0x0041b570 0x0041b560 nop nop 0x0041b550 0x0041b530
0x0041b588 0x0041b580 nop nop 0x0041b578 nop
0x0041b590 nop         nop nop nop         nop
nop         nop         nop nop 0x0041b598 0x0041b5a0
```

Listing 4 – VLIW 4

```
2                   1   1   1
nop         nop         nop nop 0x0041b528
nop         nop         nop nop 0x0041b538
0x0041b568 0x0041b540 nop nop 0x0041b558
0x0041b570 0x0041b560 nop nop 0x0041b548
0x0041b580 nop         nop nop 0x0041b550
0x0041b588 nop         nop nop 0x0041b578
0x0041b590 nop         nop nop 0x0041b530
nop         nop         nop nop 0x0041b598
nop         nop         nop nop 0x0041b5a0
```