Design of Digital Circuits: Lab Report		
LAB 9 – The Performance of MIPS		
Date	29. May 28, 2019	Grade
Names	David Zollikofer	
	Sven Pfiffner	Lab session / lab room
		Fri 10-12 / HG E26.1

You have to submit this report via Moodle.

Use a zip file or tarball that contains the report and any other required material. Only one member from each group should submit the report. All members of the group will get the same grade.

The name of the submitted file should be *LabN_LastName1_LastName2.zip* (or *.tar*), where *LastName1* and *LastName2* are the last names of the members of the group.

Note 1: Please include all the required material. No links/shortcuts are accepted.

Note 2: The deadline for the report is a hard deadline and it will not be extended.

Exercise 1

For the following values of A and B, how many clock cycles are needed to execute your first program from Lab 7 on your baseline MIPS processor, before adding optimizations of Lab 9? Assuming that we run the MIPS processor at 20 MHz, how much time (in seconds) would that take?

Value of A	Value of B	Number of cycles	Time in seconds
0	8	37	37*1/20MHz = 1.85*10^(-6) s
6	8	13	13*1/20MHz = 6.5*10^(-7) s
0	250’000’000	1’000’000’006[1]	1’000’000’006*1/20MHz = 50s
249’999’996	250’000’002	31[2]	31*1/20MHz = 1.55*10^(-6) s

We will use the following code:

```
main:
        # initialize the two registers
        addi $t0, $zero, 0
        addi $t1, $zero, 1000
        add $t2, $zero, $zero

loop:   add $t2, $t2, $t0
        beq $t0, $t1, end
        addi $t0, $t0, 1
        j loop

end:
        j       end     # Infinite loop at the end of the program.
```

We note that initialization takes 3 steps if the numbers fit into the immediate of length 16 bits. If not, it will take two steps per load (which we have accounted for in the calculation above). (Please refer to exercise 2 where we will discuss this in detail). After initialization we will increment the counter a total of (B-A) times which will result in the execution of (B-A) times the loop body (4 instructions). Furthermore, we will have another two instructions in the final loop pass (after the B-A passes) where we execute the first add of the loop as well as the beq, after which we will have the result of our calculation. Hence the formula for normal sized numbers (fitting into the immediate) is 3 + 4(B-A) + 2.

We are now running x instructions at 20 MHz, as a result it will take us x*1/20MHz seconds to execute the program.

[1] +1 cycle since B doesn’t fit into immediate

[2] +2 cycles since A & B don’t fit into immediate → see Ex 2 for more details on why we need the extra cycle

Exercise 2

Fill in the new values for the Table in Exercise 1 when using the modified MIPS architecture running the optimized code, as discussed in the manual for Lab 9.

Value of A	Value of B	Number of cycles	Time in seconds
0	8	11	11*1/20MHz = 5.5*10^(-7) s
6	8	11	11*1/20MHz = 5.5*10^(-7) s
0	250’000’000	11+1 = 12	12*1/20MHz = 6*10^(-7) s
249’999’996	250’000’002	11 + 1 + 1 = 13	13*1/20MHz = 6.5*10^(-7) s

We note that the code now looks as follows: (we have opted for the code you have given us since this makes correcting easier for you):

```
.text

main:
        addi  $t0, $0, 0         # $t0 = A
        addi  $t1, $0, 200       # $t1 = B
        addi $t2, $t0, -1 # A-1
        multu $t0, $t2    # A (A-1)
        mflo $t0          # mult result in t0
        srl $t0, $t0, 1   # divide by two

        addi $t2, $t1, 1  # B+1
        multu $t1, $t2    # B (B+1)
        mflo $t1          # mult result in t1
        srl $t1, $t1, 1   # divide by two

        sub $t2, $t1, $t0 # end result is the difference

 end:
        j end             # loop t2 is the result
```

We note that there are a total of 11 instructions until the result is available in $t2 after the sub. The first one being the first add instruction initializing value A (in this case with 0) and the last one calculating the difference between the sum of 0 to (A-1) and 0 to B giving us the sum of all numbers between A and B including endpoints, which is our desired result. Since changing either A or B just changes the initialization this program’s length does not depend on the input (as long as A and B can be given using add immediate, if they are longer then we will need to initialize in a more sophisticated manner using load upper immediate etc.)

This is the case in the third and fourth example: The numbers 250’000’000, 249’999’996 and 250’000’002 do not fit in the 16 bits that we have to store an immediate value. Hence, we must initialize them as follows:

```
lui $t0, upper16Bits
ori $t0, $t0, lower16Bits
```

This adds 1 cycle per oversized initialization.

We are now running x instructions at 20 MHz, as a result it will take us x*1/20MHz seconds to execute the program.

Exercise 3

Compare the size/device utilization of the two implementations (before and after the modifications in Lab manual 9). What differences do you see? Briefly comment on them. *Hint: Look into the synthesis report.*

We would like to point out that a direct comparison is not possible since Lab 9 had a different structure, it had no traditional top module that could be "mounted" with a constraints file onto the FPGA but was rather designed to be tested on the testbench. As a result, we have added the modification of lab 9 to the files of lab 8 (→ the new ALU as well as support for the new operations) which we will compare to the "normal" lab 8. This will make a comparison more useful since metaphorically speaking comparing apples to oranges would not be useful. Hence we can compare two very similar CPU's where one implements MIPS like multiplication and the other doesn't.

The results are the following (after synthesis):

Resource	Estimation	Available	Utilization %
LUT	611	20800	2.94
LUTRAM	288	9600	3.00
FF	181	41600	0.44
IO	15	106	14.15
BUFG	2	32	6.25

no multiplication

Resource	Estimation	Available	Utilization %
LUT	669	20800	3.22
LUTRAM	288	9600	3.00
FF	181	41600	0.44
IO	15	106	14.15
BUFG	2	32	6.25

with multiplication

As expected the implementation does not the number of inputs. We see a big increase in the number of LUT's which has been caused by the addition of the multiplication logic (since this is the only thing that changed between the two designs).

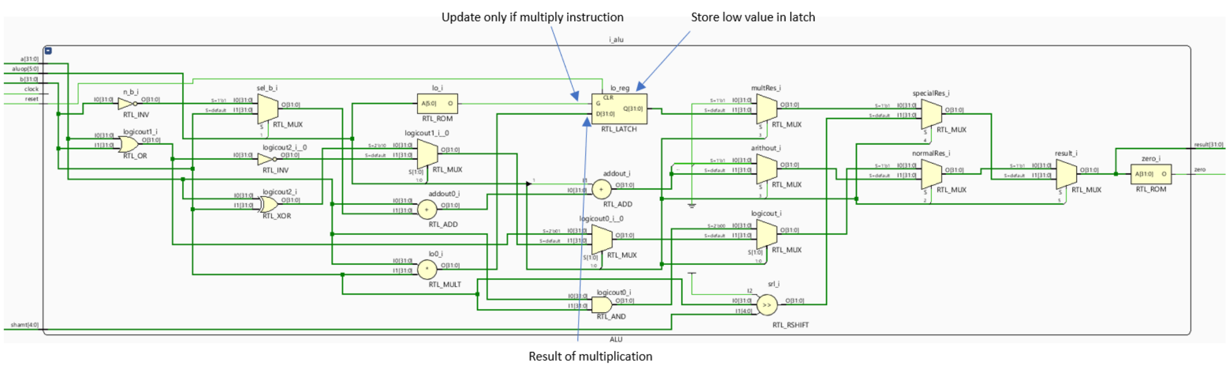
The increase in the number of LUT’s can be further investigated in the Synthesis reports where we see the following picture for our ALU.v file[3]:

Component	**Without Multiplication**	**With Multiplication**
Adders	1	1
XORs	1	1
Multipliers	0	1
Muxes	5	9

We can see that the inclusion of the multiplication (obviously) needs a multiplier as well as more muxes (since we now have more complex control logic).

One question that remains is how do the registers that store the low value get implemented in the code. We first thought that this would be done using flipflops, which would increase the number of flipflops used in the synthesis report (see picture on previous page).

However, since we don’t have an increase usage something else must be happening. After looking at the schematics it becomes clear. We are storing the value of low in a latch which can be implemented using LUT’s as we learned in lecture 6 (see the slides on latches lecture 6 → they can be implemented using gates). This also contributes to the usage of LUT’s but does not increase the number of flipflops needed.



[3] This is only the ALU.v file and not the whole processor.

Feedback

If you have any comments about the exercise please add them here: mistakes in the text, difficulty level of the exercise, or anything that will help us improve it for the next time.

We didn't really know if we had to handle the case where the numbers wouldn't fit into the immediate 16 bits. As a result, we have put a lot of time into getting this right, perhaps it could be a bit clearer if this was necessary or not.

We have put a lot of effort into this lab report. If you have any questions, feel free to reach out to us.

Our email addresses are: svenp@student.ethz.ch & zdavid@student.ethz.ch