Design of Digital Circuits: Lab Report		
Lab 8: Full System Integration (Session I)		
Date	May 17th	Grade
Names	Sven Pfiffner & David Zollikofer	
		Lab session / lab room
		Friday 10-12 | Room E 26.3

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

Which MIPS instructions do you think would produce wrong outputs if the ControlUnit signal *RegWrite* is *'stuck at 0'*, i.e., *RegWrite* always has value 0? In other words, which MIPS instructions depend on the control signal *RegWrite*?

This question is twofold:

We can look at all the instructions that we have on our processor which depend on the regwrite by looking at it's assignment in the ControlUnit.v file. There we have the following line:

```
assign RegWrite = (Op == OP_RTYPE) | (Op == OP_LW) | ( Op == OP_ADDI);
```

This means that when we have an R-Type instruction, a load word instruction or an add immediate instruction. We will activate the register write to store the result of the instructions (e.g. a computation or a memory fetch) in a register. Without the RegWrite we could not store the results. On our processor we do not have that many R-Type instructions: In our design we have the following R-Type instructions (the ones we have implemented in our ALU):

- add
- sub
- and
- or
- xor
- nor
- slt

However, if we talk about general MIPS processors there are many more instructions. First of all, there are other R-Type instructions for example jalr or jr and other shifts etc. which we did not implement but need the RegWrite enabled.

Many of the I-Type instructions (mostly the ones analogous ones to the ALU R-Type instructions also need the RegWrite signal)

Furthermore there are a lot of other load instructions in a full-fledged MIPS processor which would all also require the RegWrite signal to be 1.

Last but not least, we would like to add that there are many MIPS instructions that do not need the RegWrite signal enabled such as branche, some jumps, stores, etc.

Exercise 2

Explain why a 6-bit address is enough for the instruction and data memory. *(Hint: size of the memory.)*

Instruction Memory:
We note that the Instruction Memory is defined as follows (in InstructionMemory.v):

```
reg [31:0] InsArr[63:0];
```

This means that we have 64 words, each 32 bits in length. Since MIPS has byte addressable memory we must have four addresses per word (since 4*8=32). Hence we need to be able to address 64*4 = 256 = 2^8 different bytes. This means that we would theoretically need an 8 bit wide bus.

However, we use a small trick. Since we always increase the program counter by 4 and only work with whole words we do not need the byte addressability, hence the last two bits of the address bus will always be zero which allows us to eliminate the last two bits and address our memory using a 6 bit wide bus.

We can see this trick being used for example in the MIPS.v file where we pass `PC[7:2]` as an argument to the instruction memory. Since the last two bits are zero anyway we do not bother passing them.

We note that exactly the same applies to the data memory which is defined as follows:

```
reg [31:0] DataArr[63:0];
```

The addressing is completely analogous. We also only use the first 6 most significant bits. The only difference we have here is that the address doesn't come from the program counter but from the ALU:

```
ALUResult[7:2]
```

Exercise 3

As you might have noticed, there are three different counters used in this lab. One is present in the *snake_patterns.asm* file, the second is in the clock_div module and the third is the `DispCount` signal for the 7-segment display. Explain the functions of each of these three counters/dividers in a sentence or two each.

***snake_patterns.asm* counter:**
This counter makes the pattern visible. Without this counter the computer would be way too fast. Using the counter we can update the display, wait for a long time and only after the relatively long waiting period update the display again. This means the processor spends pretty much all of its time waiting and only very rarely updates the screen.
This makes the whole pattern visible since when the pattern is not changing the processor is incrementing the counter.

In addition to the loop/wait counter there is also a counter in $t4 which cycles through the snake movement pattern by storing the address of the current pattern state

clock_div counter:

From page 2 of the lab manual of lab 8 we know that the system clock of the FPGA runs at 50 Mhz as well as that the critical path of our processor is around 25ns.

This technically means we can never be faster than 1/(25ns) which is 40MHz. In order to be save we use the two bit register in the clock divider which has four different states and hence makes a new clock that is four times slower than the system clock. Therefore our clock divider creates a new clock that runs at 12.5MHz.

Since 12.5Mhz is significantly slower than the maximal 40Mhz, this is a safe operating frequency.

DispCount counter:

The main problem we have is that we cannot set all four 7 segment displays at once but must select which one we would like to turn off/on. As a result we cycle trough the displays and only light up one display at a time.

Furthermore, if the *DispCount* register is only 4 bits wide we cycle through the displays way too fast which results in something comparable to the screen tearing effect in monitors. This is due to the fact that the display does not have adequate time to fully light up and that there is a lag between setting the segment and having the segment light up. This results in having multiple segments on different displays light up in different intensities at the same time. Having a large enough counter and letting the digits light up for 1.6ms *(as seen in the provided top.v file)* solves this problem.

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.

- Question 1 is very broad. Are we talking about our implementation of a MIPS processor or a general full fledged MIPS processor.