INF5410 Spring 2011

Lab Assignment 4
Digital Arithmetic

Lab goals

- Extension of VHDL skills
- Implementing digital circuits learned in the course (MUL, DIV, SQRT)
- Embedding the circuits into the instruction set of our MIPS processor from lab1

Deliverables
For passing this lab, a zip file archive named in the format lab4_UseName_FirstName_FamilyName (e.g., `lab4_hsimpson_Homer_Simpson.zip`) has to be send to `koch@ifi.uio.no`.
The submission deadline is **May. 1st.**
The file has to contain:

- your commented source codes (only the files you touched)
- the place&route reports generated by the ISE tools (`top.par`)
- the corresponding simulation list-files generated by Modelsim (`File->Export->Tabular list`)
- a report as shown in the appendix

Task 1 (Integer Multiplication and Wallace Trees)

For implementing the MUL instruction in lab 1, we used the VHDL `'*'` operator. By default, this operator is mapped to three multiplier blocks in order to perform the 32-bit multiplication. However, multipliers can alternatively be implemented using logic only (look-up tables). This is relevant, if we run out of multiplier blocks on the FPGA while still having logic resources left over. In this task, we replace the multiplier by a radix-2 multiplier using a Wallace tree for adding the partial products. The entity of the multiplier shall look as follows:

```
entity CarrySaveMUL is
    Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);
           B : in  STD_LOGIC_VECTOR (31 downto 0);
           MUL_out : out  STD_LOGIC_VECTOR (31 downto 0);
           -- we write only the LOW word to the register file
           );
end CarrySaveMUL ;
```

Note that the output is only 32 bit wide and not 64 as usual for a multiplier with 32 bit wide input operands. The highest 32 bit can be truncated as already done in lab1. For a sake of simplicity, we will implement the multiplier combinatory (no flip-flops).
The implementation shall follow the following steps:

- Write a carry save adder (CSA) component with the inputs `A_in`, `B_in`, `C_in` and the outputs `S_out` and `C_out`. The size of the input operands should be defined by a generic parameter that denotes the number of bits.

- Write a testbench for a 8-bit CSA component and use the following test vector:
  `A_in=0010 1000` (40_{10}), `B_in=0010 1001` (41_{10}), `C_in=0010 1010` (42_{10})
  What is the expected output result for `S_out` and `C_out`?

- Implement a Wallace tree with the help of the CSA component.
  We will implement a 32-bit radix-2 multiplier which will add the partial products `0`, `A`, `2A` or `3A`, depending on a tuple of two bits of the input operand `B`.
  How many partial products do we have to add together?
  How many bits are required for the CSA component?
  Note that the synthesis tool can truncate unused logic. For example, if we do not use the outputs of some logic, the tool can detect this for removing all unnecessary logic. This works on a single bit granularity. We can exploit this in order to simplify the VHDL code by defining all CSA components of the same size.

- Write a testbench for the Wallace tree.
  Go on the following website and enter for the operands: $40_{10}, 41_{10}, \ldots$
  `www.ecs.umass.edu/ece/koren/arith/simulator/Wallace/main.html`
  Attach a screenshot of the Wallace tree and the simulator trace from the website to your report.

- Implement the multiplier. Write a process that computes `A`, `2A` and `3A`. The results will be connected to multiplexers (one for each input of the Wallace tree). The multiplexers are controlled by two bit pairs of operand `B`.

- Integrate the multiplier into your MIPS processor design from lab 1. Simulate the system using the code for Task 1 (which uses multiplication).

- Implement the whole system in Xilinx ISE. Run the synthesis process four times. The report shall include the achieved clock frequency and implementation cost in terms of LUTs and slices for:
  1) a CPU without any multiplier
  2) a CPU using DSP48 multiplier primitives (you can take the result from lab 1)
  3) a CPU using LUTs to implement the VHDL `'*'` operator
  This can be done by taking lab 1 and by changing the synthesis option `use_dsp48` from `"Auto"` to `"No"` (under section HDL Options).
  4) a CPU using our CarrySaveMUL multiplier

Task 2 (Division)

In this task, we will implement the DIV instruction for the MIPS CPU. The division produces two results: the quotient q and the remainder r. In the MIPS instruction set, two extra 32-bit registers are used to store the results: LO and HI. These registers haven't been used so far. Data between these registers and the register file is transferred with the help of dedicated move instructions. The DIV command computes for the two registers from the register file: $LO = GPR[rs] div GPR[rt]$ (quotient) and $HI = GPR[rs] mod GPR[rt]$ (remainder). The operands are signed (two's complement).
The divider shall be implemented using the non-restoring algorithm. The entity of the multiplier shall look as follows:

```
entity DIV is
    Port ( A : in  STD_LOGIC_VECTOR (31 downto 0);  -- dividend
           B : in  STD_LOGIC_VECTOR (31 downto 0);  -- divisor
           Q : out  STD_LOGIC_VECTOR (31 downto 0); -- quotient
           R : out  STD_LOGIC_VECTOR (31 downto 0); -- remainder
           stall : out STD_LOGIC;   -- drive '1' during computation
           clk : in  STD_LOGIC);
end DIV;
```

The divider shall perform one iteration of the non-restoring division per clock cycle. The output stall shall be active until the results are computed. This signal is needed to stop the PC from being updated during computation.
The implementation shall follow the following steps:

- Write an iterative non-restoring divider that follows the algorithm introduced in the lecture. We don't have to implement overflow or divide-by-zero indication,

because MIPS is not generating exceptions for this. These issues have to be resolved in software.
How wide do you have to chose the internal registers for 32 bit operands?
How many clock cycles does it need to compute the result?

- Write a testbench for the divider and use the following test vector:
  `A=0010 1010` (42_{10}) and `B=0000 1101` (13_{10})
  Repeat the test for all four sign combinations of A and B. Repeat this test with another large number example.
  What are the expected output results for the tests?
  For generating a reference trace, you can use the online simulator from:
  `http://www.ecs.umass.edu/ece/koren/arith/simulator/NRDiv/`

- Integrate the divider into the MIPS CPU. For a sake of simplicity, we can use directly the divider outputs *Q* and *R* without extra registers for *LO* and *HI*.
  How do you deal with internal or external stalls or wait requests?

- Integrate the divider into the MIPS CPU. For a sake of simplicity, we can use directly the divider outputs *Q* and *R* without extra registers for *LO* and *HI*.
  How do you deal with internal or external stalls or wait requests?

- Test the DIV command using the setup from lab 1, Task 1. Open the file `instr_rom_bram_com.vhd` and change line 63 from
  `X"00000000", -- ###bfc00034,00000000,nop` to
  `X"019A001A", -- ###bfc00034,019A001A,div t3,t4`
  Next, add the quotient and remainder output signals to the waveform. Run the simulation and observe the multiplier outputs. Here, we do not need the MOV instructions to access *LO* and *HI*.

- Implement the divider in Xilinx ISE and list the resource requirements in terms of LUTs/slices. Note when not connecting *LO* and *HI*, no logic will be generated for the divider. Therefore, you have to add the MFLO and MFHI to your simple MIPS.