Lab 3: Changing How LegUp Implements Computations in Hardware

1 Introduction and Motivation

In this lab, you will gain exposure to the scheduling and Verilog generation steps of LegUp. Your goal is to change the way unsigned division and remainder are implemented in the hardware. By default, LegUp uses the Altera divider core to implement these operations. The Altera core is pipelined and can accept new inputs on *every* cycle – its *initiation interval* is 1 cycle. For this lab, you will change LegUp so that it implements unsigned division/remainder using a *serial divider* which computes 1 bit of the quotient per cycle. The serial divider can only accept new inputs every n cycles, where n is the bitwidth of the input operands. This way of doing division trades off area for time – the serial divider uses significantly less area than the Altera core, however, it is not pipelined. The skills developed in this lab are applicable if you wish to modify LegUp to support a new FPGA device architecture.

Two tasks are needed to change the divider implementation: 1) You will modify the scheduling algorithm in LegUp so that it does not initiate two unsigned division/remainder operations within n cycles of one another. That is, you must change the scheduler so that it honours the initiation interval of the serial divider. 2) You will change the Verilog generator in LegUp to instantiate the serial divider module, instead of using Altera's divider core.

Note that this lab is intended to be used with LegUp 4.0.

2 Serial Divider

The serial divider intended for use in this lab is included with the LegUp 4.0 distribution. Execute the following command:

```
cd ~/legup-4.0/serial_divider
```

In the directory above, take a look at the Verilog file `SerialDivider.v` using your favourite editor (gedit, emacs, gvim). While a detailed explanation of its functionality is outside the scope of this lab, the inputs and outputs of the module are described in the file as comments. The relevant inputs are the n-bit-wide dividend and divisor, along with a `go` input signal which must be asserted to start the division. After $n+1$ cycles, the n-bit-wide quotient and remainder are available as outputs and the divider asserts the `done` signal. You are referred to Appendix A if you wish to learn more about the serial divider.

3 Toy Example: Dividers

Before we make any code changes to LegUp, let's synthesize a program with divisions using the default LegUp divider implementation. Enter the following directory of the LegUp 4.0 distribution:

```
cd ~/legup-4.0/examples/dividers
```

Open the source file `dividers.c`. This is a toy program developed for this lab. Scroll down to the `main(...)` function to find a loop which executes 5 unsigned 32-bit division operations and sums the results. The loop trip count is 10, so the program does a total of 50 division operations. Notice that the program includes the typical built-in test vectors to verify that the hardware works as expected after LegUp synthesis.

Let's first compile and run `dividers.c` using gcc:

```
gcc dividers.c
./a.out
```

You should verify the program output matches:

```
Result 0: 72
Result 1: 23
Result 2: 46
Result 3: 63
Result 4: 56
Result 5: 39
Result 6: 54
Result 7: 84
Result 8: 59
Result 9: 81
Sum: 577
RESULT: PASS
```

The last line, `RESULT: PASS`, indicates that the built-in test vectors resulted in the expected output. Returning to the command line, type:

```
make
make v
```

The above commands synthesize the `C` file to Verilog and simulate it with ModelSim. You should see the following ModelSim output:

```
# Cycle:   39    Time:   830     Result 0:   72
# Cycle:   78    Time:   1610    Result 1:   23
# Cycle:   117   Time:   2390    Result 2:   46
# Cycle:   156   Time:   3170    Result 3:   63
# Cycle:   195   Time:   3950    Result 4:   56
# Cycle:   234   Time:   4730    Result 5:   39
# Cycle:   273   Time:   5510    Result 6:   54
# Cycle:   312   Time:   6290    Result 7:   84
# Cycle:   351   Time:   7070    Result 8:   59
# Cycle:   390   Time:   7850    Result 9:   81
# Cycle:   391   Time:   7870    Sum:   577
# Cycle:   392   Time:   7890    RESULT: PASS
# At t=   7910000 clk=1 finish=1 return_val=  577
# Cycles:  393
```


Notice that the output matches the gcc output, with an additional line indicating the simulation timestamp (in cycles) of each test vector result. The last line of the simulation, `Cycles:`, indicates that the circuit took 393 cycles to complete. Now let's run Quartus synthesis targeting the default LegUp FPGA family: Cyclone V. First setup the Quartus project for Cyclone V:

```
make p
```

This command generates a `top.qsf` file for Quartus. Now run a full Quartus synthesis:

```
make f
```

Take a look inside `top.fit.summary`, you should see:

```
Family : Cyclone V
Device : 5CSEMA5F31C6
Timing Models : Final
Logic utilization (in ALMs) : 1,037 / 32,070 ( 3 % )
```

Note that the numbers you observe may differ slightly due to Quartus place and route noise. Open `top.sta.rpt` in an editor and search for the *Slow 1100mV 85C Model Fmax Summary* section. You should see an $Fmax$ of approximately 117 MHz (+/- 5 MHz). We can use the cycle latency and the $Fmax$ to calculate the circuit wall-clock time with the equation:

$$wall\_clock\_time = cycle\_latency \times (1/Fmax) \quad (1)$$

In this case, the circuit had a wall-clock time of: $393 \times 1/(117 \times 10^6) = 3.3\mu s$.

4 Scheduling Changes

Now let's move on to the scheduling code changes. First we're going to add a new parameter to the LegUp TCL configuration file. Open `config.tcl` in your editor and uncomment:

```
set_parameter SERIAL_DIVIDER 1
```

As we will see below, this user-specified parameter can be accessed in the LegUp codebase using the global variable `LEGUP_CONFIG`:

```
LEGUP_CONFIG->getParameterInt("SERIAL_DIVIDER")
```

Now change into the following directory, which contains most of the LegUp codebase:

```
cd ~/legup-4.0/llvm/lib/Target/Verilog
```

Before we begin making code changes, let's backup the original versions of the source files we will be modifying:

```
mkdir backup_lab3
cp * backup_lab3
```

First let's make a helper function to tell us when an LLVM instruction is a candidate to use a serial divider functional unit. We can add this function to the `utils.h|.cpp` file, which holds common helper functions. Open `utils.h` in your editor and add the following function prototype right below the `isUnsignedDivOrRem` prototype:


```
bool isSerialDivider(Instruction *instr);
```

Next open `utils.cpp` and add the following function right below the `isUnsignedDivOrRem` function:

```
bool isSerialDivider(Instruction *instr) {
    return (LEGUP_CONFIG->getParameterInt("SERIAL_DIVIDER") &&
            isUnsignedDivOrRem(instr));
}
```

This function takes a LLVM instruction as an input parameter and returns `true` if the operation can be computed by a serial divider functional unit. This is the case if the operation is an unsigned divide or modulus, and the user has set the parameter `SERIAL_DIVIDER` to non-zero in the TCL configuration file. The function `isUnsignedDivOrRem` is defined in `utils.cpp`.

Now we can modify the scheduler. In this same directory, open the source file `Scheduler.cpp` in a text editor. This file contains helper functionality that is used by the LegUp scheduler (the scheduler implementation is in `SDCScheduler.h|.cpp`).

In `Scheduler.cpp`, navigate to the method:

```
unsigned Scheduler::getNumInstructionCycles(Instruction *instr)
```

The `getNumInstructionCycles` method returns the number of cycles of latency required to compute the hardware operation corresponding to the LLVM instruction: `instr`. Find the section of this method that returns the latency for a divider:

```
switch(instr->getOpcode()) {
  case (Instruction::UDiv):
  case (Instruction::URem):
  case (Instruction::SDiv):
  case (Instruction::SRem): {
        unsigned pipelineDepth;
```

LegUp instantiates the default divider functional unit, Altera's `lpm_divide` megafunction, with a pipeline depth equal to the operand bitwidth. This pipeline depth was selected to ensure that the $Fmax$ of the `lpm_divide` is about 100MHz on a Cyclone V FPGA.

Recall that the serial divider has a latency of n + 1 cycles, where n is the bitwidth of the input operands. Therefore, we must modify this section of code for the serial divider. Add the following code before the return statement of that case block:

```
if (isSerialDivider(instr)) pipelineDepth++;
```

In the same file, `Scheduler.cpp`, search for the following method:

```
unsigned Scheduler::getInitiationInterval(Instruction *instr)
```

The `getInitiationInterval` method has a parameter `instr`, which is a pointer to an LLVM `Instruction` object. This method returns the initiation interval for the instruction's hardware implementation. The function's body contains a switch statement, which at present, returns 1 for all types of instructions. The reason for this is that by default, all hardware functional units used by LegUp can accept new inputs *every* cycle.

Add the following code at the beginning of the `getInitiationInterval` method:


```
if (isSerialDivider(instr)) {
    unsigned bitwidth = getBitWidth(instr->getType());
    return bitwidth + 2;
}
```

This new code checks if the `instr` parameter is going to be computed by a serial divider using our helper function, and if so, it returns an initiation interval equal to $n + 2$, where n is the bitwidth of the instruction. The initiation interval is derived by observing that we must wait for a total latency of $n + 1$ for the serial divider to compute the quotient and remainder. After these output signals are valid, the serial divider done signal is asserted for a single cycle. After waiting for the done signal to be deasserted, we can start the next divide operation. Therefore, the initiation interval is equal to the latency of the serial divider ($n + 1$) plus an additional cycle. For example, if the instruction were a 32-bit division, then the initiation interval would be 34 – division computations cannot commence within 34 cycles of one another. The `getBitWidth` function called above can be found in `utils.cpp` in the same directory. Take a moment to look at `getBitWidth`, which in turn makes an LLVM call (`getPrimitiveSizeInBits`) that returns the number of bits used by the instruction.

By default, LegUp instantiates a single divider unit and applies resource constraints in scheduling to meet that constraint; i.e. it ensures that at most one division/remainder operation is scheduled in any cycle. Your code changes above, combined with the existing resource constraints, ensure that no two unsigned division/remainder operations will be scheduled within $n + 2$-cycles of one another, where n is the bitwidth of the instruction.

The LegUp scheduler uses the system-of-difference constraints (SDC) formulation proposed in [2], where scheduling is formulated as a linear program (LP) and fed into a standard LP solver. Open `SDCScheduler.cpp` and go to the following method:

```
void SDCScheduler::addResourceConstraint(...)
```

Scrolling down the function, you will pass by a nested loop, and then you will see a second loop that begins as follows:

```
for (i = constraint; i < constrainedInstructions.size(); i++) {
```

This loop selects pairs of constrained instructions, unsigned division/remainder instructions in our case, and for each pair, introduces a constraint into the scheduler's LP formulation. The call to `add_constraintex` adds a constraint to the LP formulation. You can see the call to the `getInitiationInterval` function you modified above. The scheduling formulation is constrained to create a "gap" in the schedule between the unsigned division/remainder operations according to the initiation interval you specified. You can refer to [2] for all the gritty details on how the resource constraints are managed in the LP.

Change directories, going up three levels back to the `llvm` directory:

```
cd ~/legup-4.0/llvm
```

Now re-compile LegUp with your Scheduling code modifications by running:

```
make
```

Return now to the `examples/dividers` directory:

```
cd ~/legup-4.0/examples/dividers
```


Resynthesize the design using LegUp and simulate with ModelSim by typing `make` and then `make v` as you did above. You should see the following output:

```
# Cycle:   1721   Time:   34470    Sum: 577
# Cycle:   1722   Time:   34490    RESULT: PASS
# At t=    34510000  clk=1 finish=1 return_val= 577
# Cycles:  1723
```

When the simulation completes, you will observe a drastic increase in the number of simulation cycles, owing to your changing the divider's initiation interval! But we are still using the same `lpm_divide` divider functional unit, in the next section we will change the default divider functional unit to be a `SerialDivider`.

5 Verilog Generator Changes

Now we will move on to the Verilog generation changes needed to instantiate the serial divider functional unit in the LegUp synthesized circuit. Go back to the LegUp source directory:

```
cd ~/legup-4.0/llvm/lib/Target/Verilog
```

Let's start by opening the `GenerateRTL.cpp` file in your editor. The `GenerateRTL.cpp` source file takes the scheduling and binding information, and constructs the final circuit to be synthesized.

Search for the following method:

```
RTLSignal *GenerateRTL::createFU(Instruction *instr, RTLSignal *op0,
        RTLSignal *op1);
```

The method `createFU`, creates a functional unit (FU) Verilog module instantiation for the given instruction and input operand signals. LegUp represents a circuit in an internal data structure consisting of: Verilog modules (`RTLModule` objects), wires and registers (`RTLSignal` objects), and Verilog operations like addition (`RTLOp` objects).

Look at the very first condition in the function:

```
    if (isDiv(instr) || isRem(instr)) {
        return createDivFU(instr, op0, op1);
```

This condition creates the default `lpm_divide` functional unit. We need to override this default, so you should add the following code immediately before the condition above:

```
if (isSerialDivider(instr)) {
    return createSerialDivFU(instr, op0, op1);
}
```

The function `createSerialDivFU` is a new function that we will now create to instantiate a serial divider module. In Verilog, this instantiation would look something like:


```
    SerialDivider div_inst (
        .clk       (clk),
        .clken     (clken),
        .resetn    (~ reset),
        .go        (~ reset & run),
        .dividend  (numerator),
        .divisor   (denominator),
        .remainder (my_result[63:32]),
        .quotient  (my_result[31: 0]),
        .done      (my_done)
    );
    defparam
        div_inst.n = 32,
        div_inst.log2n = 5;
```

We need to re-create this module instantiation using the LegUp internal RTL data structures. Copy the code in Figure 1 into `GenerateRTL.cpp` right before the `createFU` function. For those of you using the LegUp 4.0 Virtual Machine on VirtualBox, you can find this code snippet in the file:

```
~/tutorials/lab3_createSerialDivFU.cpp
```

You can also find this file on the tutorials page of www.legup.org.

You'll notice that the member variable `rtl` is used extensively in this new method. Each function in the `C` code gets mapped to a Verilog module, this module is represented by a `RTLModule` object pointed to by the member variable `rtl`. Every time we wish to add a new wire or register to the current Verilog module, we must call the `addWire` or `addReg` methods of the `rtl RTLModule` object. The `RTLModule` objects keep track of all the signals and connections, and then eventually these data structures are passed to the source file `VerilogWriter.cpp`, which prints out the final Verilog RTL corresponding to the circuit data structures. The Verilog printing step is beyond the scope of this lab.

Now return to the `GenerateRTL.cpp` source file and search for the method:

```
void GenerateRTL::visitBinaryOperator(Instruction &I);
```

The `visitBinaryOperator` method is called on every two-operand operation in the `C` program we wish to convert to hardware. We must ensure that for every divide operation, we assert the appropriate `go` signal of a serial divider functional unit. Add the following code at the very end of the `visitBinaryOperator` method:

```
if (isSerialDivider(instr)) {
    RTLSignal *go;
    if(this->binding->existsBindingInstrFU(instr)) {
        std::string fuId = this->binding->getBindingInstrFU(instr);
        go = rtl->find(fuId + "_go");
    } else {
        go = rtl->find(verilogName(instr) + "_go");
    }

    connectSignalToDriverInState(go, ONE, this->state, instr);
}
```



```
RTLSignal *GenerateRTL::createSerialDivFU(Instruction *instr,
         RTLSignal *numerator, RTLSignal *denominator) {
  std::string serial_div_name = "serial_divider_" + verilogName(instr);
  RTLModule *serial_div = rtl->addModule("SerialDivider", serial_div_name);

  serial_div->addIn("clk")->connect(rtl->find("clk"));
  std::string div_enable = "lpm_divide_" + verilogName(instr) + "_en";
  RTLSignal *en = rtl->addWire(div_enable);
  serial_div->addIn("clken")->connect(en);
  RTLSignal *reset = rtl->find("reset");
  RTLSignal *not_reset = rtl->addOp(RTLOp::Not)->setOperands(reset);
  serial_div->addIn("resetn")->connect(not_reset);

  std::string go_signal_name = verilogName(instr) + "_go";
  if(this->binding->existsBindingInstrFU(instr)) {
      std::string fuId = this->binding->getBindingInstrFU(instr);
      go_signal_name = fuId + "_go";
  }
  RTLSignal *go = rtl->addWire(go_signal_name);
  go->setDefaultDriver(ZERO);
  serial_div->addIn("go")->connect(go);

  serial_div->addIn("dividend")->connect(numerator);
  serial_div->addIn("divisor")->connect(denominator);

  RTLWidth width = RTLWidth(instr->getType());
  RTLSignal *FU = rtl->addWire(serial_div_name + "_temp_out", width);
  RTLSignal *unused = rtl->addWire(verilogName(instr) + "_unused", width);

  if (isDiv(instr)) {
      serial_div->addOut("quotient")->connect(FU);
      serial_div->addOut("remainder")->connect(unused);
  } else {
      assert(isRem(instr));
      serial_div->addOut("quotient")->connect(unused);
      serial_div->addOut("remainder")->connect(FU);
  }

  RTLSignal *done = rtl->addWire(serial_div_name + "_done");
  serial_div->addOut("done")->connect(done);

  unsigned bitwidth = instr->getType()->getPrimitiveSizeInBits();
  serial_div->addParam("n", utostr(bitwidth));
  serial_div->addParam("log2n", utostr(log2(bitwidth)));

  return FU;
}
```

Figure 1: Function to instantiate the SerialDivider module.

This code finds the `go` signal of the serial divider functional unit assigned to this divide operation. Next, we call the method `connectSignalToDriverInState`, which has the following parameters: 1) a destination signal to drive, in this case `go`, 2) a value to drive to the destination signal, in this case a high bit, 3) the state at which to drive the destination signal, in this case the first state (`this->state`) of the divide operation, and finally 4) an optional LLVM instruction for Verilog comments.

You've now finished making all necessary changes to the LegUp code. Change directories, going up three levels back to the `llvm` directory:

```
cd ~/legup-4.0/llvm
```

Now re-compile LegUp with your code modifications by running:

```
make
```

6 Evaluating the Results

We will now evaluate the results of our new serial divider functional unit changes. First we need to make some changes to the LegUp Makefiles. Change into the LegUp 4.0 distribution `examples` directory:

```
cd ~/legup-4.0/examples
```

and open the file `Makefile.config`. Search for the Makefile variable: `VERILOG_LIBS`. Directly below the following line

```
$(ALTERA_SIM_LIBS_DIR)/altfp_fptosi64.v \
```

Add:

```
$(LEVEL)/../serial_divider/SerialDivider.v \
```

This change tells ModelSim where to find the `SerialDivider` module.

Now change into the dividers example directory we looked at previously:

```
cd ~/legup-4.0/examples/dividers
```

Synthesize the C file with LegUp using the new serial divider by running: `make`. Remember that we have already modified the `config.tcl` file to set the `SERIAL_DIVIDER` parameter.

Now simulate the resulting Verilog with ModelSim by running: `make v`. Verify that the built-in test vectors match their expected values by looking for the output: `RESULT: PASS`. You should see the same ModelSim output as in the previous section:

```
# Cycle:   1721   Time:   34470    Sum: 577
# Cycle:   1722   Time:   34490    RESULT: PASS
# At t=    34510000 clk=1 finish=1 return_val= 577
# Cycles:  1723
```

Take a look at the total cycle latency of the dividers example now (look for `Cycles:`). The latency has increased from 414 cycles (using the default `lpm_divide`) to 1723 cycles with the serial divider. This is due to the large initiation interval of the serial divider – we must wait until a divide is totally finished before starting a new one, unlike the pipelined `lpm_divide` Altera megafunction.

Let's see how the area and $FMax$ changed by running a Quartus synthesis. Run `make p` to generate a new Quartus project for Cyclone V. Open the *top.qsf* file and add this line so Quartus knows where to find the SerialDivider module:


```
set_global_assignment -name SOURCE_FILE ../../serial_divider/SerialDivider.v
```

Now run a full Quartus synthesis using the command: `make f`. After Quartus finishes, look inside `top.fit.summary` for area metrics and `top.sta.rpt` for the FMax (in the *Slow 1100mV 85C Model Fmax Summary* section). You should get results similar to the following:

```
Family : Cyclone V
Device : 5CSEMA5F31C6
Timing Models : Final
Logic utilization (in ALMs) : 424 / 32,070 (-59% vs original)
Latency: 1723 cycles (+316% vs original)
Fmax: 115 MHz (-2% vs original)
Wall Clock Time: 14.9us (+352% vs original)
```

Compare this to the default LegUp `lpm_divide` version from the beginning of the lab:

```
Family : Cyclone V
Device : 5CSEMA5F31C6
Timing Models : Final
Logic utilization (in ALMs) : 1,037 / 32,070 ( 3 % )
Latency: 414
Fmax: 117 MHz
Wall Clock Time: 3.3us
```

We find that for the evaluated `dividers` program, the cycle latency of the serial divider version is over three times longer than the default. However, we save a significant amount of logic modules (59%) due to the smaller size of the serial divider. As future work, we could investigate instantiating many serial dividers and running them in parallel to hide the high initiation interval.

Finally, let's restore all of the LegUp code from our backup folder:

```
cd ~/legup-4.0/llvm/lib/Target/Verilog
cp backup_lab3/* .
```

Now recompile LegUp by running `make` from the `llvm` folder:

```
cd ~/legup-4.0/llvm
make
```


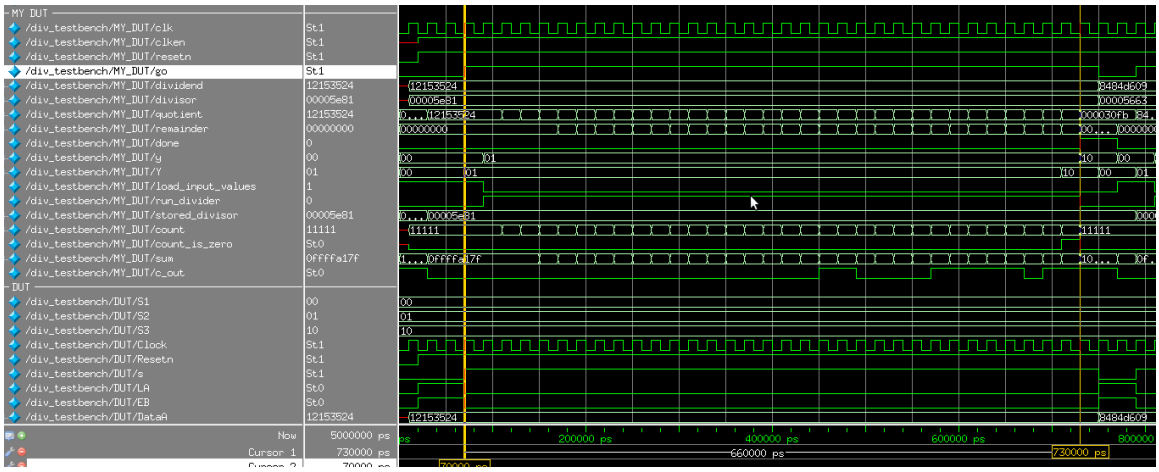

Figure 2: ModelSim simulation for the serial divider.

Appendices

A Serial Divider Algorithm

The serial divider is based on the implementation described by Brown and Vranesic [1]. Figure 3 shows an example of long-hand division for both binary and decimal. The pseudo code of the algorithm is given in Figure 4. The trip count of the loop is equal to the bitwidth of the unsigned operands and each iteration calculates a single bit of the quotient and remainder. To simulate the SerialDivider.v file, change into the directory:

```
legup-4.0/serial_divider/testbench
```

Now run `make`. ModelSim should open, and you should see a simulation similar to Figure 2. The serial divider instance is named `MY_DUT`. Notice that the latency between the go signal being asserted and the done signal being asserted is 660ns. The clock period is 20ns, so the divider latency is 33 cycles.


```
numerator: 10001100 (140)
denominator: 1001   (9)

       00001111 <- quotient (15)              15
      ---------                              ----
1001 ) 10001100                          9 ) 140
        1001                                  9
       -----                                 --
        10001                                 50
         1001                                 45
        ------                                --
         10000                                 5
          1001
         -----
           1110
           1001
           ----
            101 <- remainder (5)
```

Figure 3: Binary serial division example from [1].

```
quotient = 0
remainder = 0
for i: 1 to bit width
    left shift the remainder
    set the LSB of the remainder equal to the MSB of the numerator
    left shift the numerator and remove the MSB

    left shift quotient by 1
    if (remainder >= denominator):
        set LSB of quotient to 1
        remainder = remainder – denominator
    end if
end for
```

Figure 4: Pseudo-code for serial divide algorithm

References

[1] S. Brown and Z. Vranesic. *Fundamentals of Digital Logic with Verilog Design 2nd Edition*. McGraw-Hill Higher Education, 2008.

[2] J. Cong and Z. Zhang. An efficient and versatile scheduling algorithm based on SDC formulation. In *IEEE/ACM Design Automation Conference*, pages 433–438, 2006.

