Lab 2: Modifying LegUp to Limit the Number of Hardware Functional Units

1 Introduction and Motivation

In this lab, you will gain exposure to the scheduling and binding steps of LegUp. You will modify the tool to limit the number of *adder* functional units in the generated hardware. By default, LegUp generates hardware with no restrictions on the number of adders when targeting the Cyclone II FPGA. Your goal for this lab is to modify LegUp high-level synthesis such that no more than *two* adders are used in the generated circuit. Naturally, multiplexers will be needed on the inputs to the adders for any programs that need to perform more than two addition operations. While we focus on adder functional units in this lab, the same procedure can be applied to impose resource constraints on any type of functional unit (e.g. floating point division, or shift). The purpose of these resource constraints is to reduce the logic elements and registers required on the FPGA.

There are two key steps to realizing the desired functionality: 1) The scheduler must be changed such that no more than two addition operations are scheduled in a given cycle. Without this change, it will be impossible to get by with just two hardware adders, as there could be a scheduling step where more than two additions are computed in parallel. 2) The binding step of LegUp must be changed to map addition operations in the program to one of the two adder functional units. In this case, LegUp will balance the number of addition operations that are assigned to each adder functional unit, thereby limiting the sizes of the multiplexers on each unit's inputs.

After implementing the code changes, you will use the LegUp test infrastructure to evaluate the impact of the changes in the circuit latency, FMax and area. In particular, you will be able to see that the scheduling changes lead to longer hardware latencies and that the binding changes introduce additional multiplexers, increasing circuit area and lowering FMax.

The lab is intended to be used with the LegUp 3.0 release.

2 Modifying Scheduling

Our goal here is to incorporate resource constraints on adders by changing the scheduler to schedule at most two addition operations per cycle. LegUp already imposes resource constraints on other types of units, namely, it ensures that only a single division/remainder operation is scheduled per cycle, and also that no more than two memory accesses (loads/stores) occur in a cycle (as the underlying FPGA device has dual-port RAMs).

Before we begin, we will evaluate the latency of a benchmark circuit scheduled with no restriction on the number of adder functional units; that is, scheduled such that *any* number of add operations may occur in any cycle. Open a terminal (click the icon on the left sidebar). Change into the following directory which

contains the `adpcm` benchmark from the CHStone high-level synthesis benchmark suite [3], included with the LegUp distribution:

```
cd ~/legup-3.0/examples/chstone/adpcm
```

The CHStone benchmarks contain input vectors within the programs themselves, as well as golden output vectors (expected outputs). This allows us to simulate the circuits without injecting vectors from a separate HDL testbench. Type the following command which will compile the `adpcm` benchmark using LegUp (from C to Verilog RTL).

```
make
```

Make sure there are no errors, then simulate the resulting Verilog with ModelSim:

```
make v
```

At the end of the ModelSim simulation, you will see the number of cycles the benchmark took to execute – its *latency* (look for a message that begins `Cycles`):

```
# Result:        150
# RESULT: PASS
# At t=           478110000 clk=1 finish=1 return_val=       150
# Cycles:               23903
```

Make a note of the number of cycles reported for later comparison. Also, later on in this lab, you will synthesize the Verilog produced by LegUp to an Altera FPGA.

We now turn to the scheduling algorithm modifications. The LegUp scheduling algorithm is based on the system of difference constraints (SDC) formulation proposed by Cong and Zhang [1]. In SDC scheduling, the scheduling problem is represented mathematically as a linear program (LP) that can be solved by a standard linear programming solver. Change into the following directory of the distribution in which the majority of the LegUp code can be found:

```
cd ~/legup-3.0/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_lab2
cp * backup_lab2
```

The LegUp scheduler implementation is in the following files: `SDCScheduler.h` and `SDCScheduler.cpp`. Open `SDCScheduler.cpp` in an editor of your choosing (gedit, emacs, gvim).

In the source file, search for the following method (line 679):

```
SchedulerMapping* SDCScheduler::createMapping(...)
```

createMapping() is the main governing function for the scheduler – it sets up the LP problem, adds constraints and calls the solver. Scroll down the function until you come to the point where resource constraints are being added to the LP formulation. In particular, look for the following snippet of code (line 772) which creates the resource constraints on the number of unsigned and signed dividers:


```
if (LEGUP_CONFIG->getParameterInt("NUM_DIVIDERS")) {
  unsigned numDividers = LEGUP_CONFIG->getParameterInt("NUM_DIVIDERS");
  addResourceConstraint(F, isUnsignedDivOrRem, numDividers);
  addResourceConstraint(F, isSignedDivOrRem, numDividers);
}
```

Immediately after the above lines, add a new line to the source to constrain the number of addition operations per cycle to 2 (line 777):

```
addResourceConstraint(F, isAdd, 2);
```

Perhaps surprisingly, that is all that is needed for this part of lab 2! Let's disect the meaning of the newly added line. The first argument, `F`, is a pointer to a type called `Function`, which is an LLVM class representing a function in the `C` program being synthesized by LegUp. This is the function currently being scheduled by the scheduler. LegUp schedules functions one at a time. The third argument is the maximum number of addition operations we wish to allow in any given cycle of the schedule – 2 in our case. The second argument, `isAdd`, is a pointer to a function with the following signature (declared on line 295 `utils.cpp` in the same directory as the scheduler):

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

The `isAdd` function takes a pointer to an LLVM `Instruction` as an argument and returns `true` if that instruction is an add instruction; otherwise, it returns false. Take a moment to look at the implementation of `isAdd` in `utils.cpp`.

The `addResourceConstraint` function (line 275) walks through all instructions in the function `F` and for each one, it calls the `isAdd` method to determine if the instruction is an add operation. The resource constraints are then added to all such add instructions. While it's beyond the scope of this lab to show all details of the scheduler, let's take a cursory look at the `addResourceConstraint` function, which is in the same source file: `SDCScheduler.cpp`. A few lines into this function, you will see a `for` loop iterating over all of the basic blocks of the function `F` being scheduled:

```
for (Function::iterator b = F->begin(), be = F->end(); b != be; b++)
```

Within this outer loop, you will see an inner loop iterating over the instructions in each basic block of the outer loop:

```
for (BasicBlock::iterator i = b->begin(), ie = b->end(); i != ie; i++)
```

For each instruction visited by this inner loop, the `isAdd` function is being called (the parameter name is `instrTypePredicate`). Instructions that are addition operations are added to an STL vector container called `constrainedInstructions`.

Following the nested loop, the `constrainedInstructions` container is sorted. A second loop in the function then begins as follows:

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

This loop iterates through the sorted list of instructions and adds constraints to the LP formulation through calls to `add_constraintex`, which is a function defined by the `lpsolve` linear program solver used by LegUp. The constraints ensure that no more than two of the constrained instructions (additions) will be scheduled in the same cycle.

Congratulations on your first modifications of LegUp! Now let's test them out. Change directories, going up three levels back to the `llvm` directory:


```
cd ~/legup-3.0/llvm
```

Type the following command to re-compile LegUp with your code modifications:

```
make
```

After the compilation is complete, return to the CHStone `adpcm` benchmark directory as above:

```
cd ~/legup-3.0/examples/chstone/adpcm
```

Resynthesize the circuit by running `make` and make sure there are no errors. Then simulate the circuit by executing `make v` in the `adpcm` directory. You should see the output:

```
# Result:        150
# RESULT: PASS
# At t=           493110000 clk=1 finish=1 return_val=       150
# Cycles:               24653
```

Note the number of cycles needed to implement the benchmark and compare it with the figure you recorded earlier. The latency of the circuit should have increased by several hundred cycles. With a restriction on the number of additions per cycle, the schedule has been made longer (stretched out), thereby increasing the latency of the circuit. Indeed, you have just exercised a key trade-off in scheduling: constraining resources reduces circuit area but generally leads to slower circuit performance. However, since LegUp does not share adders by default, we will not see any area benefit from our new scheduling constraint. In the next section we will fix this by turning on adder resource sharing.

3 Modifying Binding

LegUp performs a stage of high-level synthesis called *binding* after all operations have been scheduled. Binding solves the problem of assigning each scheduled operation to one of the available function units. For instance, if there are two loads from memory that occur in the same state of the finite state machine, each load must be assigned to different memory port of our memory controller.

In this lab, we have already scheduled a maximum of two additions per cycle. Therefore, we can get away with having only two adder functional units in the entire circuit. These two adders can be shared by all the addition operations, using multiplexing on the adder inputs. We need to make sure that if two additions occur in parallel, that each addition is bound to a different adder functional unit. If only one addition occurs in a given cycle, then we can bind it to either of the two available adders.

LegUp has two different binding algorithms: pattern sharing [2] and bipartite weighted matching [4]. To limit the complexity of this lab we will focus only on the bipartite weighted matching algorithm, which is a common binding method used by many high-level synthesis tools.

In bipartite weighted matching, we construct a *bipartite graph*: two independent sets of vertices that only have edges from one set to the other set, there are no edges connecting vertices within the same set. The operations for the given scheduling step make up the vertices of one set and the functional units make up the vertices of the other. We assign a weight for each edge to indicate the cost of connecting an operation to that specific functional unit. The goal of bipartite weighted matching is to match each operation to a functional unit while minimizing the sum of the edge weights. This problem can be solved with the Hungarian method in $O(n^3)$ time [5]. We construct this bipartite graph for each scheduling step, and bind one state at a time in order. An example is shown in Figure 1, in this case we would match `addition1` to `adderFuncUnit2`

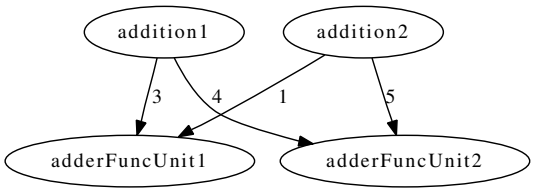

Figure 1: Bipartite Graph

and `addition2` to `adderFuncUnit1` for a minimum edge weight of 5. The exact edge weight function is beyond the scope of this lab, but the factors affecting the weight are: 1) The number of operations already shared to a particular functional unit; we prefer less operations, to balance the size of multiplexers in front of each unit, 2) sharing the same input/output as another operation already assigned to the functional unit is preferred.

For the purposes of this lab, we do not have to modify the bipartite weighted matching algorithm. We simply need to modify the LegUp binding code to turn on sharing for adders.

The LegUp binding code is located in the same directory as the scheduling code. Change into the directory below:

```
cd ~/legup-3.0/llvm/lib/Target/Verilog
```

The LegUp binding implementation is in the following files: `Binding.cpp`, `BipartiteWeightedMatchingBinding.cpp`, and `PatternBinding.cpp`.

Open `BipartiteWeightedMatchingBinding.cpp` in an editor of your choosing. In the source file, search for the following method (line 141):

```
bool BipartiteWeightedMatchingBinding::isInstructionSharable(...)
```

The isInstructionSharable() function returns true if sharing (binding) is enabled for the specified LLVM instruction. Look at the body of the function:

```
return (share_div(I) ||
        share_rem(I) ||
        share_mem(I) ||
        is_floating_point(I) ||
        share_dsp(I, alloc));
```

Currently, the function returns true if the instruction is a divide, modulus, load/store, floating point operation, or if we have run out of DSP blocks on the FPGA. We need to modify this statement and add an additional condition to allow sharing for adders. We can use the `isAdd` function we have already seen in scheduling to allow sharing for add instructions. Modify the function body to read:

```
return (share_div(I) ||
        share_rem(I) ||
        isAdd(I) ||
        share_mem(I) ||
        is_floating_point(I) ||
        share_dsp(I, alloc));
```


Now save the file. This is the only change you will need to make to the LegUp binding algorithm to share adders.

To give you a brief overview of the `BipartiteWeightedMatchingBinding.cpp` file, search for the method (line 49):

```
void BipartiteWeightedMatchingBinding::operatorAssignment()
```

This is the main top-level function that runs the bipartite weighted matching algorithm. The first line of this function is:

```
    std::map <std::string, int> &numFuncUnitsMap =
        this->alloc->getNumFuncUnits(this->Fp);
```

This line creates a map from the name of a functional unit type, for instance `signed_add_32`, to the number of those functional units available. The `alloc` variable points to the LegUp Allocation pass, which has already calculated the minimum number of functional units available based on the schedule and the user constraints.

The next set of nested loops calculates the set of all operations (LLVM instructions) that should be shared, stored in the variable:

```
    std::set<Instruction*> Instructions;
```

The following loop iterates over all states of the finite state machine:

```
for (FiniteStateMachine::iterator state = this->fsm->begin(), se =
        this->fsm->end(); state != se; ++state) {
```

Then we iterate over all functional unit types:

```
for (std::map<std::string, int>::iterator f = numFuncUnitsMap.begin(),
        fe = numFuncUnitsMap.end(); f != fe; ++f) {
```

Finally, we iterate over all instructions within each state of the FSM:

```
for (State::iterator instr = state->begin(), ie = state->end();
        instr != ie; ++instr) {
```

For each instruction we call the function:

```
    shareInstructionWithFU(*instr, funcUnitType)
```

This function determines whether we can share the operation `instr` on the given functional unit type. Search for `shareInstructionWithFU` in the file (line 155), and you'll see that the first line calls `isInstructionSharable()`, the function we modified above.

Go back to the `operatorAssignment` function (line 49), and scroll to the end of the function. There are two nested loops that call the function `bindFunctUnitInState` for every state of the finite state machine and every functional unit type. Search for that function in the file (line 425). The `bindFunctUnitInState` function constructs the weights for the bipartite weighted matching problem, by calling the function
`constructWeights` for every shared operation (instruction) in the given FSM state. Then we pass these weights to the function `solveBipartiteWeightedMatching` to solve the bipartite weighted matching problem using the Hungarian method. Finally, using the results from the matching, we can call

the function `UpdateAssignments` to update the member variable `BindingInstrFU` that maps every operation (instruction) to the name of its assigned functional unit.

Now let's try out our new binding code. Change your directory into the `llvm` directory of the LegUp 3.0 distribution:

```
cd ~/legup-3.0/llvm
```

Now run `make`. After the compilation is complete, return to the CHStone `adpcm` benchmark directory as above:

```
cd ~/legup-3.0/examples/chstone/adpcm
```

Resynthesize the circuit by running `make`. If there are no errors, simulate the circuit by executing `make v` in the `adpcm` directory. You should see the same simulation output as in the previous section:

```
# Result:        150
# RESULT: PASS
# At t=           493110000 clk=1 finish=1 return_val=       150
# Cycles:               24653
```

You have now successfully enabled binding for addition operations in LegUp!

Open the `binding.legup.rpt` file in your editor and scroll to the very bottom. You should see the following log messages (compacted for brevity) from the bipartite weighted matching algorithm:

```
State: LEGUP_F_main_BB__preheader_211
Binding functional unit type: signed_add_32
Weight matrix for operation/function unit matching:
  ...
                              main_signed_add_32_0  main_signed_add_32_1
%.main_result.2 = add i32 ...  84                    83
Solving Bipartite Weighted Matching (minimize weights)...
Assignment matrix after operation/function unit matching:
                              main_signed_add_32_0  main_signed_add_32_1
%.main_result.2 = add i32 ...  0                     1
Checking that every operator was assigned to a functional unit...yes
Binding operator -> functional unit assignments:
  %.main_result.2 = add i32 ... -> main_signed_add_32_1 (mux inputs: 70)
```

In this snippet, we are binding the scheduling step corresponding to the state `LEGUP_F_main_BB_preheader_211` of the finite state machine. This state contains a 32-bit addition operation named `%.main_result.2 = add i32 ...`, and we have two choices for adder functional units (as expected). We construct a matrix that contains the edge weights for assigning this addition to either adder. In this case the lowest edge weight is 83 for the 32-bit signed adder functional unit `main_signed_add_32_1` so we assign the addition operation to that functional unit:

```
  %.main_result.2 = add i32 ... -> main_signed_add_32_1 (mux inputs: 70)
```

Take note that the multiplexer on the input of this adder is huge. According to the line above, there are 70 other 32-bit addition operations assigned to this adder functional unit. We expect that the size of this multiplexer to significantly decrease the FMax of the resulting circuit.

4 Evaluating the Results

To evaluate the change in circuit area and clock frequency caused by sharing adders on Cyclone II, we must synthesize `adpcm` with Quartus. Note that gathering these metrics can take some time, synthesizing `adpcm` using Quartus took about 10 minutes on an Intel Quad Core2 2.4Ghz.

First change into the `adpcm` directory:

```
cd ~/legup-3.0/examples/chstone/adpcm
```

To setup the Quartus project for Cyclone II (the default LegUp family) run:

```
make p
```

To ensure we are targeting the exact same Cyclone II device during Quartus synthesis, open up the generated `top.qsf` file and look for the following line:

```
set_global_assignment -name DEVICE AUTO
```

Change this line to specify the Cyclone II device:

```
set_global_assignment -name DEVICE EP2C35F484C6
```

Now run a full Quartus synthesis including static timing analysis:

```
make f
```

After Quartus finishes, you can find the circuit area metrics in the `top.fit.rpt` file and the clock frequency (FMax) can be found in the `top.sta.rpt` for the *Slow Model FMax Summary* section.

Here are the expected results after compiling adpcm using Quartus targeting a Cyclone II FPGA. Your cycle latency should match exactly, while the area and FMax metrics might be slightly different due to Quartus place and route noise.

Originally:

```
Latency: 23903 cycles
FMax: 67 MHz
Total logic elements : 17,451
    Total combinational functions : 15,559
    Dedicated logic registers : 8,617
```

After scheduling modification:

```
FMax: 70 Mhz (+4%)
Latency: 24653 cycles (+3% vs original)
Total logic elements : 17,886 (+2% vs original)
    Total combinational functions : 15,739 (+1% vs original)
    Dedicated logic registers : 9,260 (+7% vs original)
```

After binding modification:

```
FMax: 29 MHz (-57% vs original)
Latency: 24653 cycles (+3% vs original)
Total logic elements : 21,894 (+25% vs original)
    Total combinational functions : 19,598 (+26% vs original)
    Dedicated logic registers : 9,262 (+7% vs original)
```


From these results, we can draw the conclusion that turning on adders sharing for the adpcm benchmark on Cyclone II increases the total logic elements required by 25% (by increasing multiplexing) and causes a 57% drop in clock frequency. This matches our intuition – multiplexers are at least as big as adders on the Cyclone II FPGA architecture, so sharing adders is not advisable.

Before starting the next lab, let's restore all of the LegUp code from our backup folder:

```
cd ~/legup-3.0/llvm/lib/Target/Verilog
cp backup_lab2/* .
```

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

```
cd ~/legup-3.0/llvm
make
```

References

[1] 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.

[2] S. Hadjis, A. Canis, J.H. Anderson, J. Choi, K. Nam, S. Brown, and T. Czajkowski. Impact of FPGA architecture on resource sharing in high-level synthesis. In *ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, pages 111–114, 2012.

[3] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[4] C.Y. Huang, Y.S. Che, Y.L. Lin, and Y.C. Hsu. Data path allocation based on bipartite weighted matching. In *IEEE/ACM Design Automation Conference*, volume 27, pages 499–504, 1990.

[5] H. Kuhn. The Hungarian method for the assignment problem. In *50 Years of Integer Programming 1958-2008*, pages 29–47. Springer, 2010.

