Lab 1: Using the LegUp High-level Synthesis Framework

1 Introduction and Motivation

This lab will give you an overview of how to use the LegUp high-level synthesis framework. In LegUp, you can compile the entire `C` program to hardware, or you can also select one or more functions in the program to be compiled to hardware accelerators, with the remaining program segments running in software on the MIPS *soft* processor. Compiling the entire program to hardware can give you the most benefits in terms of performance and energy efficiency. However, there may be parts of the program which are not suited for hardware, such as linked list traversal, recursion, or dynamic memory operations. In this case, computationally intensive functions can be accelerated by hardware, with the remainder of the program running in software. This allows supporting a wider range of applications and enables a broad exploration of the hardware/software co-design space. With the MIPS soft processor, you can also execute the entire program in software.

In this lab, you are given an example `C` program, *matrix multiply*, and you will use the different flows in LegUp and compare their results in terms of execution time and circuit area. You will also see how you can improve the circuit throughput with *loop pipelining*. LegUp can currently target 2 Altera FPGAs: the Cyclone II (EP2C35F672C6) FPGA on the DE-2 board, and the Stratix IV (EP4SGX530KH40C2) FPGA on the DE-4 board. In this lab, all of the circuits are targeted for the Cyclone II FPGA.

2 Software Flow

In this section, you will learn how you can compile a `C` program to execute on the MIPS processor. This is called the pure software flow in LegUp. We currently use an open-source MIPS processor called Tiger, which was developed at Cambridge University [3]. Parts of the processor, such as the cache, have been modified to work in our framework. First let's go into the benchmark directory:

```
cd legup-3.0/examples/matrixmultiply
```

Open `matrixmultiply.c` with a text editor (gedit, gvim, emacs). You will see two input arrays, `A1` and `B1`, each of which is 20 by 20 elements in size. The output array, `resultAB1`, is also declared with the same size. In the `main` function, you will see a double nested loop, which calls the `multiply` function to calculate each element of the output matrix. The `count` variable keeps a running sum of each element of the output matrix, which is checked at the end of the program to verify that the program has produced the correct result.

Let's first compile the benchmark with `gcc` to verify the output. Run the following:

```
gcc matrixmultiply.c
./a.out
```

You will see the following output:

```
Result: 962122000
RESULT: PASS
```

The sum of all the elements in the output array is 962,122,000 and since this matches the expected output, `RESULT: PASS` is printed out. Now that we know what the expected output is, let's run this program on the MIPS processor. To do this, first run the following make target:

```
make tiger
```

This compiles the `C` file to MIPS assembly, which is then compiled to an ELF file. This ELF file is disassembled and made into a format compatible with the processor's test bench. The Tiger MIPS processor is copied over to the current directory and the simulation inputs are copied into the Tiger processor directory. To view the MIPS assembly, open `matrixmultiply.src`. You will see all of the MIPS instructions and data, as well as the addresses they are mapped to. To run this assembly code on the Tiger processor, run the following make target:

```
make sim_proc
```

This simulates the processor system with ModelSim using the compiled simulation inputs. Depending on your machine, this simulation may take several minutes. Note that both of the previous steps can be done with a single command, `make tigersim`. You will see the following output from the simulation:

```
# Result: 962122000
#
# RESULT: PASS
#
# counter =                217770
```

The output shows that we have produced the correct result. `Counter = 217770` indicates that the processor took 217,770 clock cycles to execute the program. Let's synthesize the processor with Altera Quartus II to obtain its *FMax* and area results. To do this, run the following make target.

```
make hybridquartus
```

This synthesizes the tiger processor. Now look in the `tiger_top.sta.rpt` file inside the *tiger* subdirectory for the *Slow Model Fmax Summary* section. You should see an *FMax* of 71.49MHz. So the circuit took a total of 3,046.16μs $(217,770 * (1/71.49MHz))$ to run – a quantity referred to as wall-clock time. To see the area results, look in the `tiger_top.fit.summary` file inside the tiger directory. You will see the following:

```
Total logic elements : 11,729 / 33,216 ( 35 % )
    Total combinational functions : 10,345 / 33,216 ( 31 % )
    Dedicated logic registers : 5,658 / 33,216 ( 17 % )
Total registers : 5726
Total pins : 65 / 475 ( 14 % )
Total virtual pins : 0
Total memory bits : 158,889 / 483,840 ( 33 % )
Embedded Multiplier 9-bit elements : 16 / 70 ( 23 % )
```

You may notice that the area is quite large for the Tiger processor. We are currently working on implementing our own processor to reduce the area and improve its maximum operating frequency.

In this section, you have learned how to execute a `C` program on the Tiger MIPS processor using the pure software flow. In the next section, you will learn how to compile a `C` function to hardware and execute it using the processor/accelerator hybrid system.

3 Processor/Accelerator Hybrid Flow

In this section, you will learn how to execute the *matrix multiply* benchmark using our processor/accelerator hybrid system. The hybrid flow allows the user to select one or more `C` functions, which are compiled into hardware accelerators, with the remainder of the program running on the Tiger MIPS processor. To do this, the user first designates the function to be accelerated. Then, the function calls to the designated functions are replaced with calls to wrapper functions. These wrapper functions are generated by LegUp and allow the software process to communicate with the hardware process. The wrapper functions perform memory-mapped reads/writes to transfer function arguments, start the accelerators, and retrieve any return values. The communication fabric between the processor and accelerators, called the Avalon Interface, is generated by Altera's SOPC (System-On-a-Programmable-Chip) Builder. The Avalon Interface provides the hardware interconnection network, which allows the processor to communicate with the hardware accelerators, and also allows both the processor and accelerators to access the shared memory space. This entire hybrid flow is automated so that the user simply has to specify the name of the function to be accelerated.

Now, let's try the hybrid flow on the *matrix multiply* benchmark. Open `matrixmultiply.c` again. You can see that the *multiply* function is a good candidate for hardware acceleration. The *main* function cannot be accelerated since that would compile the entire program to hardware, in which case the pure hardware flow should be used instead (described in the next section). To mark the *multiply* function for acceleration, open the *config.tcl* file. Here you will see two lines as shown below.

```
#set_accelerator_function "multiply"
#loop_pipeline "loop"
```

Uncomment the first line with *set_accelerator_function* and save the file. This parameter marks the function for hardware acceleration. To accelerate multiple functions, you need to use this parameter for each new function. We will discuss the second parameter, *loop_pipeline*, in Section 5. After the function has been designated for acceleration, run the following make target:

```
make hybrid
```

This make target runs a sequence of LLVM compiler passes in LegUp. The compiler passes first separate the program into two parts, the hardware part and the software part. The hardware part contains the program segments for the designated function and all of its descendant functions. This hardware part is taken through LegUp's high-level synthesis algorithms to be compiled to Verilog. The software part contains the remaining program segments without the designated function (and all of its descendants). This software part, after some transformations, is compiled to execute on the MIPS processor.

In the software part, calls to the hardware designated function, *multiply*, are replaced with calls to the wrapper function, *legup_seq_multiply*. This indicates that the *multiply* function will be executed in hardware instead of in software. LegUp generates a wrapper function for each of the hardware accelerated function. Let's look at the wrapper function inside *legup_wrappers.c*.

```
#define multiply_DATA (volatile int * ) 0xf0000000
#define multiply_STATUS (volatile int * ) 0xf0000008
#define multiply_ARG1 (volatile int * ) 0xf000000c
#define multiply_ARG2 (volatile int * ) 0xf0000010

int legup_seq_multiply(int i, int j)
{
    *multiply_ARG1 = (volatile int) i;
    *multiply_ARG2 = (volatile int) j;
    *multiply_STATUS = 1;
    return *multiply_DATA;
}
```

The wrapper function has the same function prototype as the original function, except that the function name has been prepended with *legup_seq_*. Its function body is removed and replaced with memory-mapped operations. This causes the processor to communicate with the hardware accelerator over the Avalon Interconnect. The *#define* statements at the top of the file define the memory-mapped addresses assigned to the accelerator. Hence, performing a write to the memory-mapped address will send data to the accelerator, and performing a read will retrieve data from the accelerator. The *multiply_DATA* pointer is used to retrieve the return data from the accelerator, the *multiply_STATUS* pointer is used to give the start signal to the accelerator, and the *multiply_ARG1* and the *multiply_ARG2* pointers are used to send the function arguments to the accelerator. The wrapper function first sends all of the arguments to the accelerator. These arguments are stored in registers in the hardware accelerator. Then the wrapper function asserts the start signal by writing a `1` to the accelerator via the `multiply_STATUS` address. When the start signal is received, the accelerator responds by asserting a stall signal back to the processor. This stalls the processor until the accelerator has finished execution. When the accelerator is done, a *done* signal is asserted, which allows the processor to resume its execution by reading from the *multiply_DATA* pointer, which retrieves the return value from the accelerator. This data is then returned to the caller function.

LegUp also generates the script to control the SOPC Builder. The script contains the tcl commands to add the accelerator to the processor system, make the necessary connections between the processor and the accelerator, and generate the complete system. The script can be found in *legup_sopc.tcl*. This script is read in by the SOPC builder to generate the system, which allows the entire flow to work without user intervention.

Once the system has been successfully generated, let's simulate the system. This can be done by running the following make target.

```
make sim_proc
```

Note that both the generation as well as the simulation of the system can be done with a single command, `make hybridsim`. Now let's look at the simulation outputs. A number of outputs, as shown below, are printed out first.

```
# At t=           105408000 clk=1 finish=1 return_val=    125400
# At t=           106563000 clk=1 finish=1 return_val=    125590
# At t=           109398000 clk=1 finish=1 return_val=    125780
# At t=           110193000 clk=1 finish=1 return_val=    125970
# At t=           110988000 clk=1 finish=1 return_val=    126160
# At t=           111783000 clk=1 finish=1 return_val=    126350
# At t=           114618000 clk=1 finish=1 return_val=    126540
```

Each line is printed out whenever the accelerator finishes and returns to the processor. Hence if there are multiple calls to the accelerator, as in this example, multiple lines of outputs are displayed. Each output shows the time at which it returned to the processor as well as the return value. The first time the accelerator was called, it returned a value of 125,400, and the second time, it returned 125,590. Since the accelerator was called 400 times in this program (iterating over a 20x20 matrix), the accelerator displayed 400 lines of return values. At the end, the processor prints out the final result.

```
# Result: 962122000
#
# RESULT: PASS
#
# counter =                73320
```

You can see that the program produced the correct result and the benchmark passed. For the processor/accelerator hybrid system, it took 73,320 cycles to complete its execution. Let's synthesize the system to get its *FMax* and area results. Once again, run the following target to synthesize the tiger directory.

```
make hybridquartus
```

Now look in the `tiger_top.sta.rpt` inside the tiger directory for the *Slow Model Fmax Summary* section. You should see an *FMax* of 65.02MHz. So the circuit took a total of 1,127.65μs $(73,320 * (1/65.02MHz))$ to run. Even though the *FMax* of the hybrid system is slightly lower than the pure software system, its run-time is much faster due to reduced clock cycles. To see area results, look in the `tiger_top.fit.summary` file inside the tiger directory. You should see the following.

```
Total logic elements : 13,674 / 33,216 ( 41 % )
    Total combinational functions : 11,970 / 33,216 ( 36 % )
    Dedicated logic registers : 6,657 / 33,216 ( 20 % )
Total registers : 6725
Total pins : 65 / 475 ( 14 % )
Total virtual pins : 0
Total memory bits : 158,889 / 483,840 ( 33 % )
Embedded Multiplier 9-bit elements : 22 / 70 ( 31 % )
```

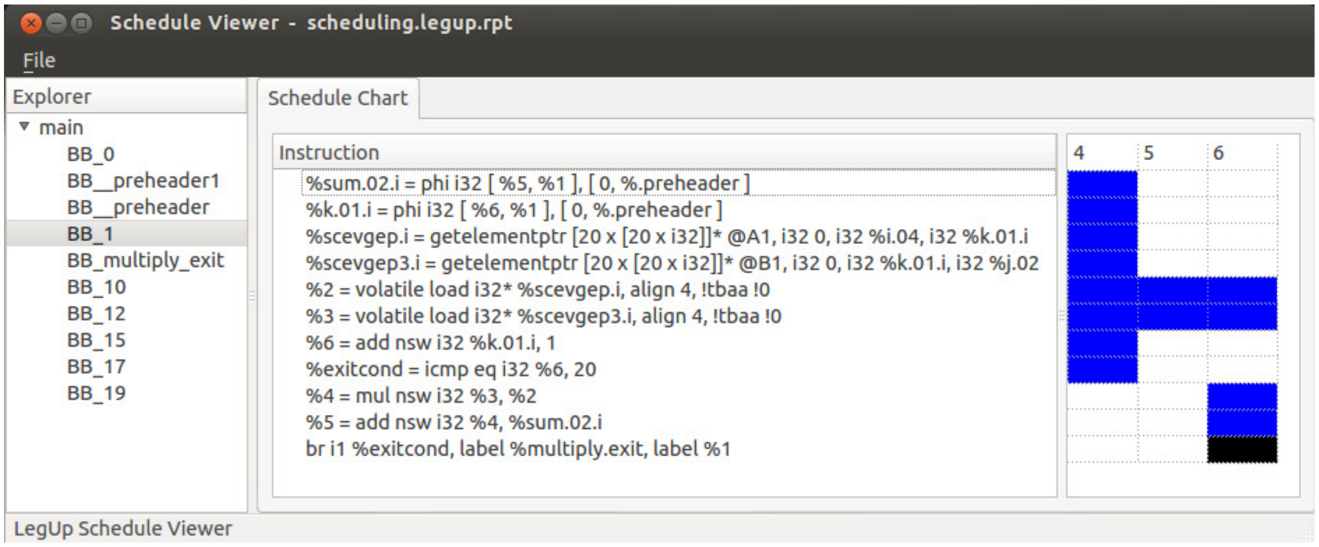
The area has increased from the pure software flow since the hardware accelerator has been added to the system.

In this section, you have learned how to use the hybrid flow in LegUp to accelerate a `C` function by hardware. In the next section, you will learn how to use LegUp to compile the entire program to hardware.

4 Hardware Flow

LegUp can compile the entire program to hardware, which can give the most benefits in performance and energy efficiency. LegUp supports a large subset of ANSI `C`, such as arrays, structs, pointer arithmetic and floating point operations, but it does not support recursion and dynamic memory. Thus, if the program does not contain any of the unsupported operations, the entire program can be compiled to hardware.

Now, let's try to compile the *matrix multiply* benchmark to hardware. Since the entire program will be compiled to hardware, first open *config.tcl* and comment out the first line with `set_accelerator_function "multiply"`. Then run the following:



```
%sum.02.i = phi i32 [ %5, %1 ], [ 0, %.preheader ]
%k.01.i = phi i32 [ %6, %1 ], [ 0, %.preheader ]
%scevgep.i = getelementptr [20 x [20 x i32]]* @A1, i32 0, i32 %i.04, i32 %k.01.i
%scevgep3.i = getelementptr [20 x [20 x i32]]* @B1, i32 0, i32 %k.01.i, i32 %j.02
%2 = volatile load i32* %scevgep.i, align 4, !tbaa !0
%3 = volatile load i32* %scevgep3.i, align 4, !tbaa !0
%6 = add nsw i32 %k.01.i, 1
%exitcond = icmp eq i32 %6, 20
%4 = mul nsw i32 %3, %2
%5 = add nsw i32 %4, %sum.02.i
br i1 %exitcond, label %multiply.exit, label %1
```

Figure 1: LegUp ScheduleViewer displaying the instructions and the schedule for a basic block.

```
make
```

This compiles the `C` program to Verilog. Before simulating the program to check its results, let's first look at its *schedule*. The schedule shows when each operation is executed in a program. In other words, it shows the FSM state assignment for each operation in the program. We have a prototype GUI, called the *ScheduleViewer*, which graphically displays the schedule by parsing a text file, *scheduling.legup.rpt*, which contains the scheduling data. Let's use the GUI to see the schedule.

```
scheduleviewer scheduling.legup.rpt
```

Figure 1 shows a screenshot of the ScheduleViewer. On the left *Explorer* panel, you can see all of the functions in the program as well as their associated basic blocks. In this case, the *multiply* function is a small function which only has one call site in the *main* function. Hence the *multiply* function has been *inlined* into the *main* function. In the GUI, click on *BB_1*, which is the basic block containing the instructions for the loop body of the multiply function. On the right *Schedule Chart*, it shows all of the instructions in the basic block, as well as their assigned states. If you move your mouse over to one of the highlighted boxes, it also shows the data dependencies. A red rectangle represents where an input to the current instruction is coming from; an orange rectangle represents where the output of the current instruction is being used. You can see that this basic block is divided into three states: 4, 5, and 6. A complete discussion of all instructions in this basic block are outside the scope of this lab (see the Appendix of this lab for more information), however, it's worth drawing attention to a few aspects. First, notice that there are two load instructions scheduled in state 4. LegUp uses dual-ported on-chip memories, thus there can be up to 2 memory accesses in a clock cycle. Each memory access has a latency of 2 cycles (both inputs and outputs are registered in on-chip RAM) and the memory accesses are pipelined so that a new memory access can start a cycle after the current memory access. However, in this example, there are no other memory accesses, and the remaining instructions need the data from memory (`%2, %3`) as their inputs. Therefore, the consumer of the load instructions, the multiply (`mul`) instruction, needs to wait until the data is returned from memory, which leaves state 5 empty. In state 6, `%2` and `%3` are multiplied together, added to the running sum of `%sum.02.i`, and assigned to `%5`. Hold your mouse over the blue bar corresponding to the `mul` instruction to see its input and output dependencies.

Now that you have learned how to view the schedule produced by LegUp, let's simulate the program using:

```
make v
```

This simulates the Verilog code with ModelSim. You will see an output as below.

```
# Result:          962122000
# RESULT: PASS
# At t=          496930000 clk=1 finish=1 return_val= 962122000
# Cycles:                24844
```

The circuit produced the correct result. It took 24,844 cycles to complete its execution. Let's synthesize this circuit with Quartus II to obtain the *FMax* and area results. First you need to make a Quartus project with the generated Verilog file, *matrixmultiply.v*. To do this, run the following:

```
make p
```

This makes the Quartus project for the target FPGA, the Cyclone II. To synthesize, run the following:

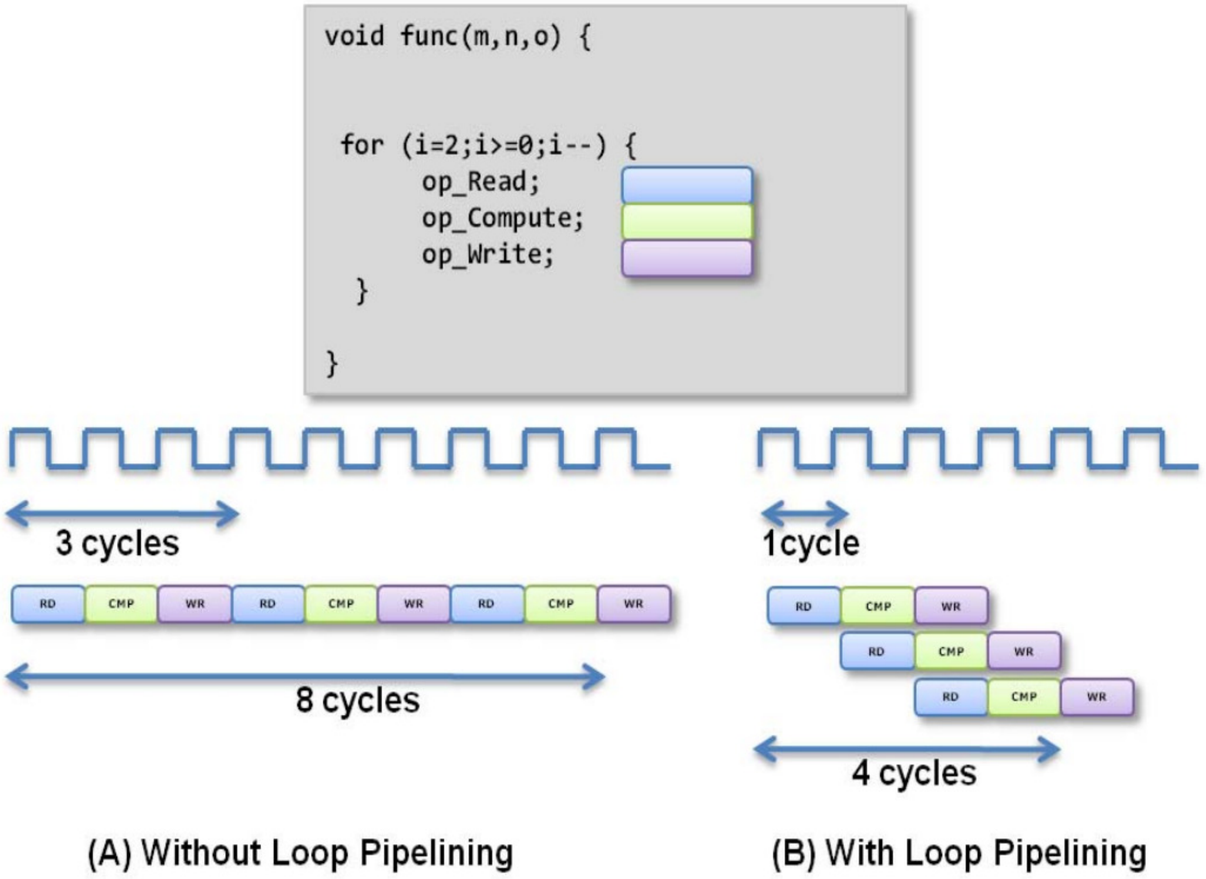
```
make f
```

Now look in the `top.sta.rpt` file in the current directory for the *Slow Model Fmax Summary* section. You should see an *FMax* of 101.04MHz. So the circuit took a total of 245.88μs ($24,844 * (1/101.04MHz)$) to run. To see the area results, look in the `top.fit.summary` file in the current directory. You should see the following.

```
Total logic elements : 1,240 / 4,608 ( 27 % )
    Total combinational functions : 1,131 / 4,608 ( 25 % )
    Dedicated logic registers : 679 / 4,608 ( 15 % )
Total registers : 679
Total pins : 36 / 89 ( 40 % )
Total virtual pins : 0
Total memory bits : 38,400 / 119,808 ( 32 % )
Embedded Multiplier 9-bit elements : 6 / 26 ( 23 % )
```

The area has decreased significantly from the previous two flows, as the system no longer contains the MIPS processor.

In this section, you have learned how to use the pure hardware flow in LegUp to compile an entire `C` program to hardware. You have also learned how to view the schedule produced by LegUp. In the next section, you will learn how you can improve the performance of the hardware circuit by using *loop pipelining*.


```
void func(m,n,o) {


 for (i=2;i>=0;i--) {
      op_Read;
      op_Compute;
      op_Write;
  }

}
```

Figure 2: Loop Pipelining Example [4].

5 Hardware Flow with Loop Pipelining

In this section, you will use *loop pipelining* to improve the throughput of the hardware generated by LegUp. Loop pipelining allows a new iteration of the loop to be started before the current iteration has finished [1]. By allowing the execution of the loop iterations to be overlapped, higher throughput can be achieved. The amount of overlap is controlled by the *initiation interval*. The initiation interval (II) indicates how many cycles are taken before starting the next loop iteration [1]. Thus an II of `1` means a new loop iteration can be started every clock cycle, which is the best case. The II needs to be larger than `1` in other cases, such as when there is a resource contention (multiple loop iterations need the same resource in the same clock cycle) or when there are loop carried dependencies (the output of a previous iteration is needed as an input to the subsequent iteration).

Figure 2 shows an example of loop pipelining [4]. Figure 2(A) shows the sequential loop, where the II=3, and it takes 8 clock cycles for the 3 loop iterations before the final write is performed. Figure 2(B) shows the pipelined loop. In this case, there are no resource contentions or data dependencies. Hence the II=1, and it takes 4 clock cycles before the final write is performed. You can see that loop pipelining can significantly improve the performance of your circuit, especially when there are no data dependencies or resource contentions.

Now let's try loop pipelining with LegUp. First you need to choose the loop to pipeline.
Open `matrixmultiply.c` with a text editor. Let's pipeline the for loop inside the *multiply* function. To

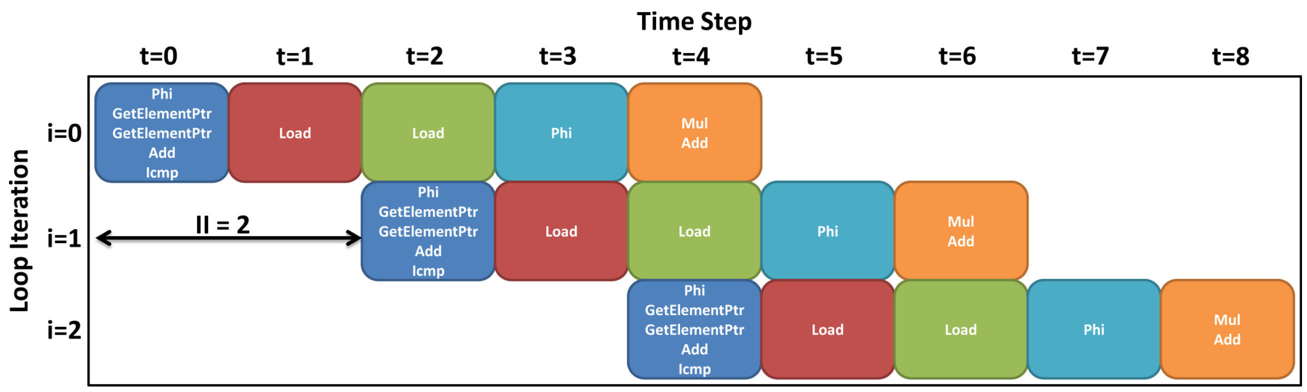


Figure 3: Schedule for Pipelined Loop.

do this, you need to add a label to the loop. Add a label called `loop` before the for loop as shown below.

```
loop: for(k = 0; k < SIZE; k++)
```

Once the loop has been labeled, open *config.tcl*. Uncomment the line:

```
loop_pipeline "loop"
```

This tells LegUp to pipeline the loop labeled as *loop*. Compile the program by running,

```
make
```

Let's look at the schedule for the pipelined loop. Our ScheduleViewer GUI does not yet support viewing the schedule for pipelined loops. Instead, you can look at *pipelining.legup.rpt*. At the bottom of the file, it shows the schedule for the loop.

Figure 3 shows the schedule for three iterations of the loop. Currently, with loop pipelining, there can only be one memory access per clock cycle. This makes the II=2 to make sure that the loads from the current iteration do not overlap with the loads from other iterations. You can see in the figure that there is only one load per time step for all loop iterations. We are currently working on supporting dual-ported memories with loop pipelining.

Now let's simulate the circuit. Run the following:

```
make v
```

It will produce the following result.

```
# Result:              962122000
# RESULT: PASS
# At t=            368930000 clk=1 finish=1 return_val= 962122000
# Cycles:                  18444
```

The circuit produced the correct result and has completed its execution in 18,444 clock cycles. Let's synthesize the circuit with Quartus II to obtain the *FMax* and area results. Run the following make targets.

```
make p; make f
```

Table 1: Summary of Results for different LegUp Flows.

Category	**Pure SW**	**Hybrid**	**Pure HW**	**Pure HW with Pipelining**
Clock Cycles	217770	73320	24844	18444
FMax	71.49	65.02	101.04	105.69
Execution Time (μs)	3,046.16	1,127.65	245.88	174.51
Speedup over Pure SW	1	2.70	12.39	17.46

Now look in the `top.sta.rpt` file in the current directory for the *Slow Model Fmax Summary* section. You should see an *FMax* of 105.69MHz. So the circuit took a total of 174.51μs $(18,444 * (1/105.69MHz)$ to run. To see the area results, look in the `top.fit.summary` file in the current directory. You will see the following:

```
Total logic elements : 1,015 / 4,608 ( 22 % )
    Total combinational functions : 941 / 4,608 ( 20 % )
    Dedicated logic registers : 640 / 4,608 ( 14 % )
Total registers : 640
Total pins : 36 / 89 ( 40 % )
Total virtual pins : 0
Total memory bits : 38,440 / 119,808 ( 32 % )
Embedded Multiplier 9-bit elements : 6 / 26 ( 23 % )
```

You may notice that the area has decreased slightly from the non-pipelined version. This is due to using single-ported memories in loop pipelining, which removes the multiplexing logic at the second memory port, as well as differences in scheduling.

In this section, you have learned how to use loop pipelining in LegUp to improve the performance of your circuit.

6 Evaluating the Results

Let's compare the results you have obtained from the previous experiments. Table 1 summarizes the results in terms of execution cycles, *FMax*, and total execution time (wall-clock time). Observe that wall-clock time is improved considerably as computations are moved from software to hardware.

Area-delay product is another important metric used for measuring the efficiency of hardware performance. Table 2 summarizes the results from the previous experiments. We have used the number of logic elements as the metric for area and the total execution time as the metric for delay.

7 Appendix: More on the LLVM Intermediate Representation

Returning to the schedule viewer GUI example, the first two instructions in the basic block are `phi` instructions. These instructions take a list of pairs as arguments, with one pair for each candidate predecessor basic block of the current block [2]. One of the pairs is chosen depending on which predecessor basic block was executed before the current basic block. Hence, in this example, `%sum.02.i`, which is created from

Table 2: Area-delay products for different LegUp Flows.

Category	**Pure SW**	**Hybrid**	**Pure HW**	**Pure HW with Pipelining**
Area (LEs)	11729	13674	1240	1015
Execution Time (μs)	3,046.16	1,127.65	245.88	174.51
Area-delay Product	35,728,414.18	15,419,527.53	304,894.70	177,128.02
Percentage vs Pure SW (%)	100.00	43.16	0.85	0.50

the `sum` variable in `C`, is assigned to `0`, if the program entered the loop body for the first time (coming from the `%.preheader` basic block), or else is assigned to `%5` (looping back from the current basic block). `%5` is assigned in state 6 which keeps a running sum of the multiplied matrix elements. The second *phi* instruction assigns a value to `%k.01.1`. `%k.01.1` is the loop induction variable `k`, which is assigned `0` if entering the loop for the first time (coming from `%.preheader` basic block) or is assigned `%6` if coming from the same basic block. `%6` is assigned to `%k.01.i + 1` in the same state (5 lines below).

The third and fourth instructions use `getelementPtr`. This LLVM instruction returns a pointer to a location in an array, based on the array's base address and an offset. A load is performed from each one of the calculated addresses, which are assigned to `%2` and `%3`. On the next two lines, the add instruction increments `%6`, which is the induction variable, and the `icmp` instruction checks whether the exit condition for the loop has been met. This completes state 4. The branch (`br`) instruction at the end of the basic block checks whether the previously evaluated `%exitcond` is true, in which case it exits the loop by branching to the `%matrix.exit` basic block, or else it loops back to the start of the current basic block.

References

[1] Michael Fingeroff. *High-Level Synthesis Blue Book*. Xlibris Corporation, 2010.

[2] LLVM. *LLVM Language Reference Manual*.

[3] University of Cambridge. *The Tiger MIPS processor (http://www.cl.cam.ac.uk/teaching/0910/ECAD+Arch/mips.html).*, 2010.

[4] Xilinx, Inc. *Vivado Design Suite User Guide*.