High-Level Synthesis: Implementing and Optimizing the Mandelbrot Set Computations

Croucher Summer Course on Performance-Aware Programming with
Application Accelerators
The Unversity of Hong Kong
28 July – 1 August, 2014

1 Introduction and Motivation

This lab will give you an overview of how to use LegUp – an open-source high-level synthesis (HLS) tool under development at the University of Toronto. LegUp accepts a `C` program as input and produces a Verilog RTL hardware implementation of the program. The Verilog is intended for implementation on an Altera FPGA.

The objective of this lab is for you to gain familiarity with a state-of-the-art HLS tool, and how its input program and constraints can be changed to improve hardware performance. The skills acquired are generally applicable to *any* HLS tool. You will implement the Mandelbrot set calculations (see Figure 1) in hardware for a 64×64 pixel image.

The lab will work as follows: You will begin with a run-of-the-mill `C` implementation of the program, and use LegUp to synthesize this to hardware. You will then analyze the results: the number of clock cycles for execution (*cycle latency*), the hardware clock period ($FMax$) and the area consumed on the FPGA chip. Following this, you will optimize the hardware implementation by giving directives to the HLS tool, and also by changing the `C` code itself. Finally, at the end, you will parallelize the `C` code, directing LegUp to produce parallel hardware. For each of these optimizations, you will evaluate the impact on cycle latency, $FMax$ and area, and (hopefully!) achieve improved results with each optimization.

For this lab, all of the circuits are targeted to the Altera Cyclone II $90nm$ FPGA, however, LegUp also targets other Altera FPGAs, including Cyclone V and Stratix IV. The LegUp tool is open-source and downloadable at: `http://legup.eecg.toronto.edu`.

2 Hardware Flow

In this step, you will compile Mandelbrot directly to hardware without any modifications to the `C` code. Change into the example's directory:

```
cd ~/legup_repo/examples/mandelbrot/lab/part_a
```

Open `mandelbrot.c` with a text editor (gedit, gvim, emacs) and take a look at the code. It should match closely with what has been discussed in class. You will see the `mandelbrot` function that has an outer loop walking over the height of the image, and a level-2 loop that walks over the width of the image.
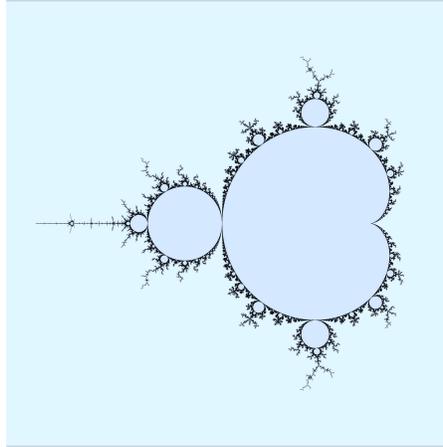
Figure 1: Mandelbrot Set illustration.

For each pixel, an inner-most loop (`for (iter ...)`) computes whether it lies in the Mandelbrot set. The function counts the total number of pixels that lie outside the Mandelbrot set, and returns this count. The `main` function in the program is straightforward: it simply calls the `mandelbrot` function. At the top of the file, you will see `#define` statements that implement the fixed-point calculations, as described in class.

Before even running HLS, verify that the software is correct by compiling and running it on your workstation. This is the typical HLS design methodology, where an engineer first runs and verifies their software, and only once they are confident the software is correct, they compile to hardware. Type:

```
gcc mandelbrot.c -o mandelbrot
./mandelbrot
```

You should see:

```
Count: 2989
PASS
```

In other words, the program incorporates correctness checking within itself, reporting PASS or FAIL.

After you are familiar with the software code and its behaviour, you can compile it to hardware by typing:

```
make
```

HLS executes and then produces a Verilog file called `mandelbrot.v`. Open `mandelbrot.v` in a text editor and scroll down the machine-generated Verilog. Indeed, it looks very difficult to follow! Debugging Verilog code produced by HLS is a very challenging problem!

At the bottom of `mandelbrot.v`, you will find a Verilog module called `main_tb`. This module is called a testbench, and it is used for simulation of the main Verilog module (called `top`) that implements Mandelbrot. You will see the testbench is quite simple: it simulates the main Verilog module with a clock (`clk`), as well as asserts `start` and `reset` signals, and waits for a `finish` signal to become high when Mandelbrot has finished its work computing 64×64 pixels.

Now, let's gain some insight into what LegUp actually did in synthesizing the C to hardware. Start up the LegUp hardware visualizer GUI:

```
scheduleviewer scheduling.legup.rpt
```
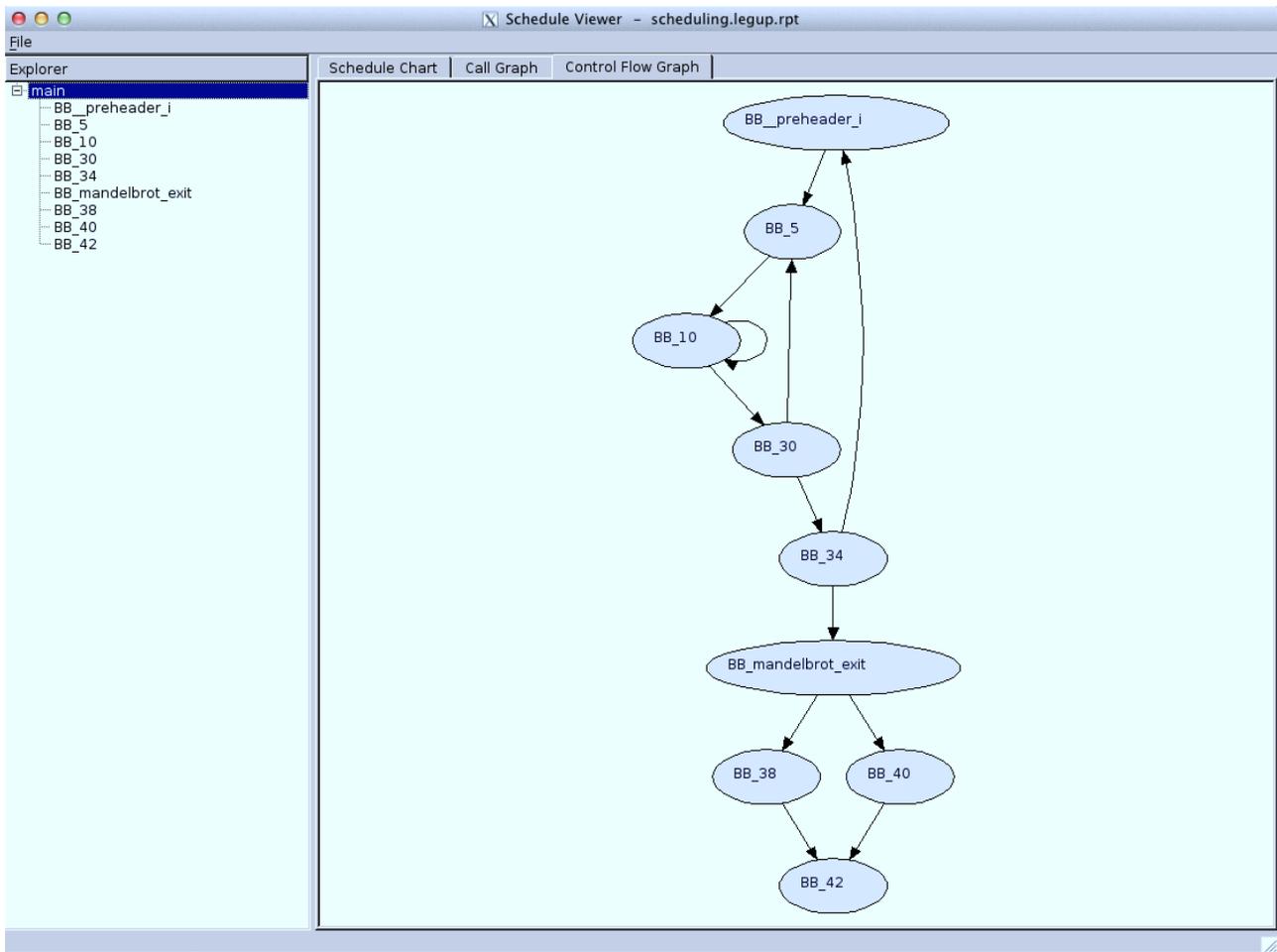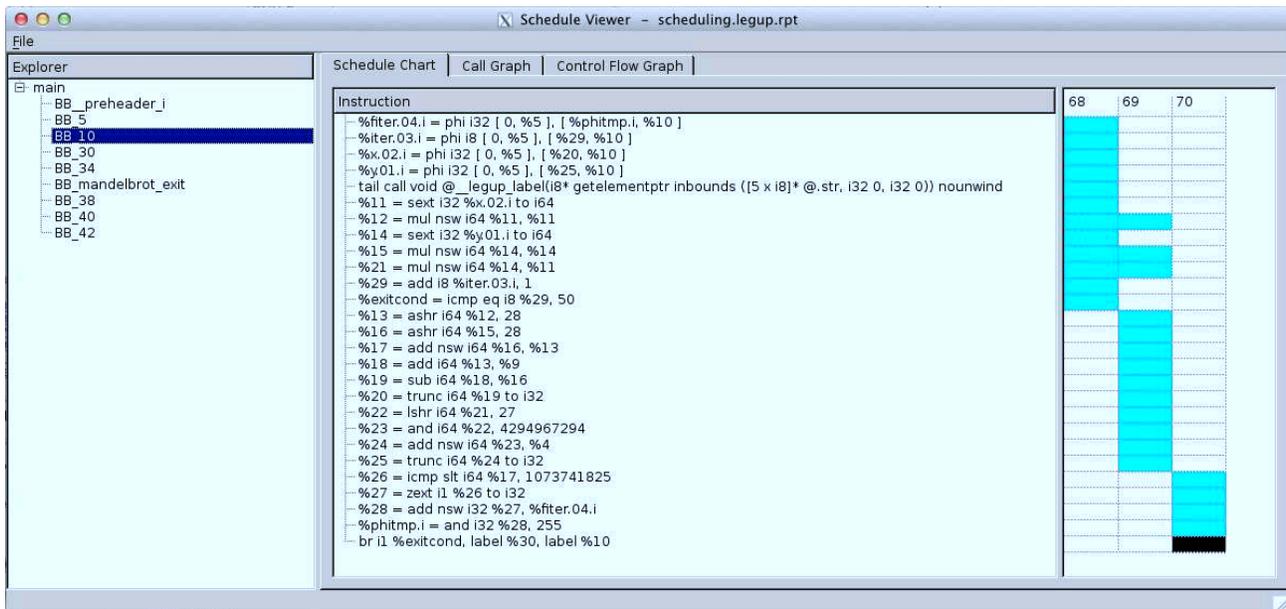
Figure 2: Control-flow graph (CFG).

Figure 3: Schedule for the inner-most loop body.

In the left panel of the GUI, you will see the names of the functions and basic blocks in the program. Observe that the `mandelbrot` function doesn't appear in the list! LLVM has inlined the function into the `main` function. Click on the `main` function name, and you will see the control-flow graph (CFG) for the program, similar to Figure 2. The names of the basic blocks in the program are prefixed with `BB`. It's difficult to determine how the names of the basic blocks relate to the original `C` code; however, can you see the loops in the CFG? Once you determine which basic block corresponds to the inner-most loop of Mandelbrot, double-click it or click on its name on the left-hand panel.

Figure 3 shows the schedule for the body of the inner-most loop. The middle panel shows the names of the LLVM instructions (see the Appendix for more information). The right-most panel shows how the instructions are scheduled into states. Observe that the loop body has been scheduled by LegUp into three clock cycles. Hold your mouse over top of some of the blue boxes in the schedule: you will see the inputs and outputs of each instruction become coloured. Look closely at the names of the LLVM instructions and try to connect the computations with those in the original `C` program.

Let's do a back-of-the-envelope performance analysis: since we observe that the inner-most loop has been scheduled with 3 clock cycles, and since that loop body executes $64 \times 64 \times 50 = 204800$ times, we expect the total number of cycles spent executing the hardware to be *roughly* $204800 \times 3 = 614400$ cycles. Of course, this doesn't count some of the overhead operations outside of the inner-most loop, yet it gives us a rough idea of how many cycles are needed for the hardware.

Okay, we're now ready to simulate the hardware with ModelSim to find out the actual number of cycles needed – the cycle latency. To do this, type:

```
make v
```

You will see ModelSim printing a lot of text on the screen, which is related to loading the simulation models it needs to know about in order to properly simulate Altera hardware. Focus just on the last messages output by the simulator, which should appear something like this:

```
# Count:        2989
# PASS
```

```
# At t=            15198830000 clk=1 finish=1 return_val=         2989
# Cycles:               759939
# ** Note: $finish    : mandelbrot.v(2663)
#    Time: 15198830 ns  Iteration: 2  Instance: /main_tb
```

We see that the simulation took 759K cycles, which is fairly close to our hand estimate above. Observe that the simulation passed and the correct value of `count` was computed. This means that the LegUp-generated hardware produced the same results as the original software.

The simulation above is called an RTL (register-transfer level) simulation, because we are simulating the design before actually mapping it to an FPGA device. This is also called a *functional* simulation. Now, it's time to map the Verilog into the Altera Cyclone II FPGA. To do this, type:

```
make p
make f
```

The first command sets up an Altera project file (`top.qsf`) that contains the name of the design, the target device (Cyclone II), and other constraints. The second command invokes Altera's synthesis, placement, routing and timing analysis tools (called Quartus II). The second command may take a few minutes to complete. This design is quite small and actually, the run-time of vendor tools is a significant problem, as it can be hours or days for state-of-the-art large designs. Once the commands are complete, we can see how much area of the FPGA the design consumed. Open the file `top.fit.rpt` which is the report produced by Altera's fitter tool (packing, placement and routing). Scroll down to see text like that shown below. This design used 2.5K logic elements and 24 multipliers (the Cyclone II device has hard ASIC-like multipliers embedded within the FPGA fabric).

```
; Total logic elements                    ; 2,548 / 33,216 ( 8 % )
;     Total combinational functions       ; 2,196 / 33,216 ( 7 % )
;     Dedicated logic registers           ; 1,557 / 33,216 ( 5 % )
; Total registers                         ; 1557
; Total pins                              ; 89 / 475 ( 19 % )
; Total virtual pins                      ; 0
; Total memory bits                       ; 0 / 483,840 ( 0 % )
; Embedded Multiplier 9-bit elements ; 24 / 70 ( 34 % )
```

You can also view the speed performance of the design, after its complete implementation on the Cyclone II. Open `top.sta.rpt`, which is the report produced by Altera's timing analysis tool. Page down a few times, and you will see a table like that below. This circuit can operate at 108MHz on the Cyclone II.

```
+-------------------------------------------------+
; Slow Model Fmax Summary                         ;
+----------+---------------+-----------+------+
; Fmax       ; Restricted Fmax ; Clock Name ; Note ;
+----------+---------------+-----------+------+
; 108.7 MHz ; 108.7 MHz       ; CLOCK_50  ;      ;
+----------+---------------+-----------+------+
```

Wall-clock time is the key performance metric for HLS, computed as the product of the cycle latency and the clock period. In this case, our cycle latency was 759937 and the clock period was 9.2ns. The wall-clock time of our implementation is therefore $759937 \times 9.2 = 6991ms$.

At the end of this handout, you will find a blank table. Fill in the results for this part of the lab under the column labelled "Pure hardware".
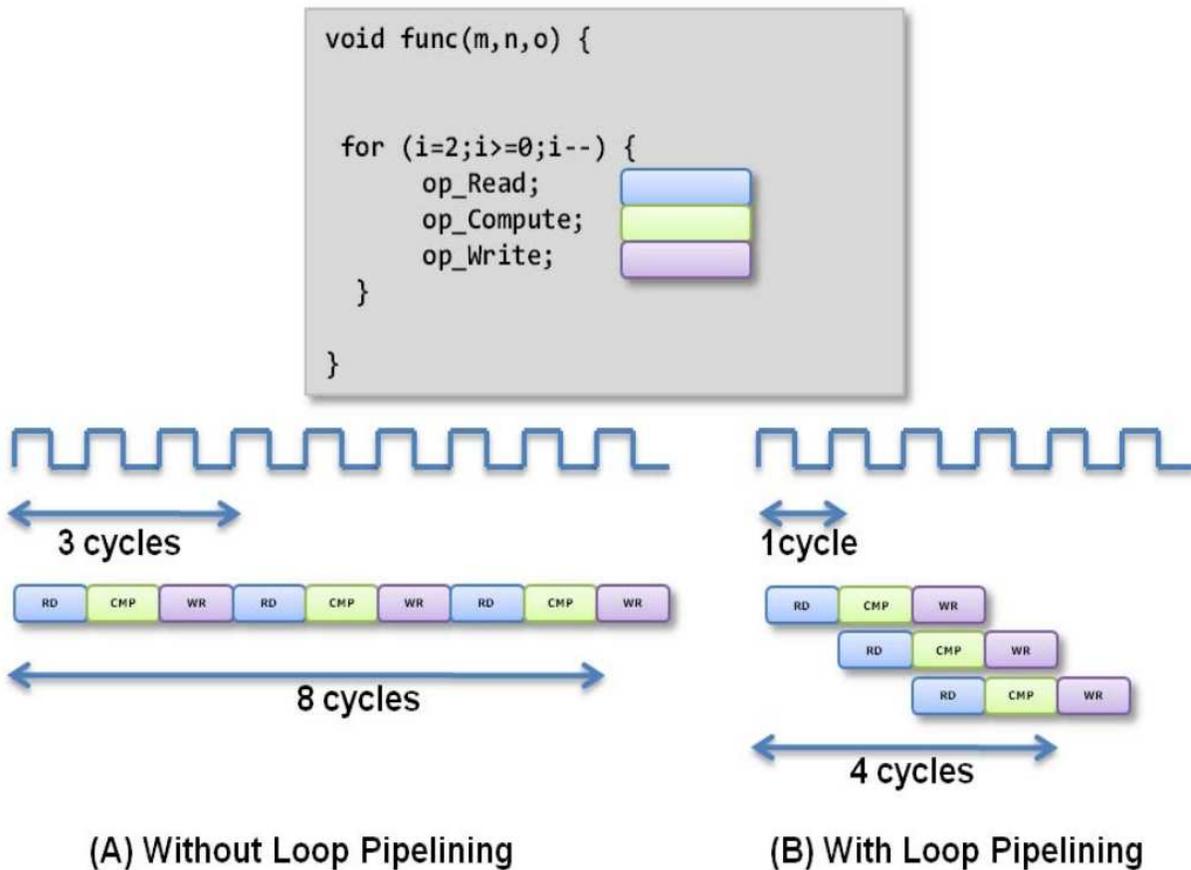
5

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


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


}
```

(A) Without Loop Pipelining

3 cycles

8 cycles

(B) With Loop Pipelining

1cycle

4 cycles

Figure 4: Loop Pipelining Example [3].

3   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 4 shows an example of loop pipelining [3]. Figure 4(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 4(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.

To invoke loop pipelining, you must tell the HLS tool which loop you wish to pipeline. In a loop nest, only the inner-most loop can be pipelined. There are two steps to do loop pipelining. First, open

mandelbrot.c and add a label "loop:" before the for loop to pipeline. After your modifications, the relevant line of C should appear like this:

```
loop: for (iter = 0; iter < MAX_ITER; iter++) {
```

Next, open the configuration file called config.tcl and add the following line below source ../config.tcl, which tells LegUp to pipelining the loop:

```
loop_pipeline "loop"
```

Having made those modifications, you can now synthesize the design by typing:

```
make
```

To see the results produced by loop pipelining, check the log file produced by LegUp called: pipelining.rtl.legup.rpt to find the following:

```
Found 1 loops to pipeline
Generating Loop Pipeline for label: "loop_1"
BB: %10
II: 2
Time: 3
maxStage: 1
Induction var:   %iter.03.i = phi i8 [ 0, %5 ], [ %29, %10 ], !legup.canonical_inducti
Label: loop_1
Constant tripCount: 50
Generating datapath for loop pipeline state: LEGUP_loop_pipeline_wait_loop_1_68
```

Observe that an II of 2 was achieved for the loop, and that the pipeline length is 3.

A better way to understand the effect of loop pipelining is visually. Open up the GUI again, this time with an option that allows you to view the pipelined schedule:

```
scheduleviewer scheduling.legup.rpt -p pipelining.legup.rpt
```

In the CFG, you will see a basic block called wait_loop_1. Double-click it to reveal the loop pipeline schedule, similar to that shown in Figure 5. Here, you can see that the II of the loop is 2, and that the length of the pipeline is 3 cycles. The dark black rectangle illustrates what the pipeline looks like in the steady state. In the steady state, two iterations of the loop are "in flight" at once.

Since we observe that the loop II is 2, and the loop executes $64 \times 64 \times 50 = 204800$ times, the total time spend in the inner-most loop is roughly $204800 \times 2 = 409600$ cycles. This is an approximation, as it doesn't include the initial time to fill the pipeline, nor does it include the time to flush the pipeline for each pixel. To check the esimate, simulate the design with ModelSim:

```
make v
```

You should see results similar to this:

```
# Count:         2989
# PASS
# At t=          11184750000 clk=1 finish=1 return_val=      2989
# Cycles:              559235
# ** Note: $finish    : mandelbrot.v(3026)
#    Time: 11184750 ns  Iteration: 2  Instance: /main_tb
```
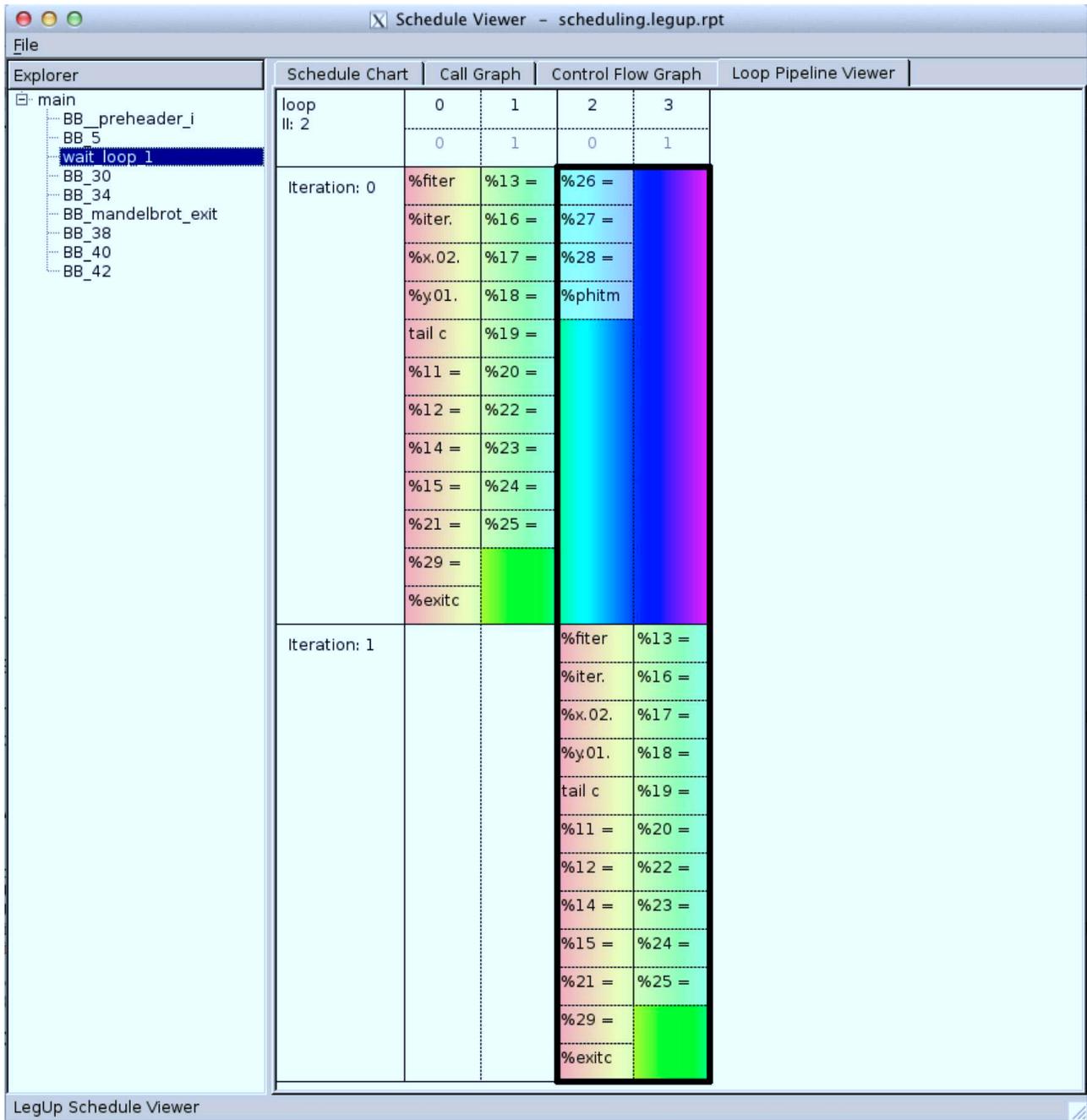
Figure 5: Schedule for pipelined loop.

Observe that loop pipelining has dramatically improved the cycle latency for the design, reducing it from 759K cycles to 559K cycles in total.

Finally, use Altera's Quartus II tool to map the design onto the Cyclone II FPGA:

```
make p
make f
```

Extract the FPGA speed ($FMax$) and area data from the files `top.sta.rpt` and `top.fit.rpt` (as you did in the previous part of this lab) and complete the "Pipelining" column in the table at the end of this handout. Compare the data for the implementations without and with loop pipelining.

4   Changing the Multiplier Latency

As mentioned in the lecture portion of the course, certain computations in hardware may take multiple clock cycles to complete. For example, a load from memory takes two cycles, and a store to memory takes one cycle. Other operations, however, are completely combinational – they take *zero* cycles to complete and may be chained together with other zero-latency operations in a single clock cycle. In LegUp, for example, add, subtract, and all logical operations are zero-latency operations.

By default in LegUp, multiplication operations take a single cycle to complete. The rationale for this default setting is to improve the $FMax$ of the circuits produced. We have observed with zero-latency multiplies, that multiplies are often on the critical path of the resultant circuits. So, setting the latency of multiplies to 1 is generally good for $FMax$. On the other hand, non-zero-latency operations are generally worse for cycle latency, as they cannot be chained with other operations in the same clock cycle.

In the Mandelbrot example, the inner loop is heavy on multiplies, thereby lengthening the number of clock cycles needed for each inner loop iteration. LegUp provides an easy mechanism to change the cycle latency of an operation. In this case, you will set the multiply latency to 0 cycles and synthesize the design again (with loop pipelining) to see if a lower initiation interval can be achieved.

To do this, open the `config.tcl` file in the directory, and add the following line:

```
set_operation_latency multiply 0
```

Leave loop pipelining on, as in the previous step – that is, we are going to resynthesize the design with the multiplier latency set to 0, and with loop pipelining. To do this type:

```
make
```

Examine the log file `pipelining.rtl.legup.rpt`. You should see that an initiation interval of 1 has been achieved for the inner loop (this is good!). At this point, you should open the GUI again to examine the loop pipelining schedule graphically, as you did in the previous step. To do this:

```
scheduleviewer scheduling.legup.rpt -p pipelining.legup.rpt
```

In the GUI, when you look at the loop pipeling schedule, you should observe an iteration of the loop starting each cycle.

Now let's check the impact of the lower II on cycle latency for the entire hardware execution. Type:

```
make v
```

From the ModelSim output, record the cycle latency in the table at the end of this document under the column "Multiplier lat=0". You should observe that the lower II has drastically reduced cycle latency in comparison to the prior steps. This is good! Now, run the Altera tools to find the $FMax$ and area:

```
make p
make f
```

Look at `top.fit.rpt` and record the area in the table. Look at `top.sta.rpt` and record the $FMax$. Here, you will see what is perhaps a surprising result! In comparison with the prior synthesis runs, the $FMax$ has dropped to about 55MHz.

```
+----------------------------------------------+
; Slow Model Fmax Summary                      ;
+-----------+----------------+-----------+------+
; Fmax      ; Restricted Fmax ; Clock Name ; Note ;
+-----------+----------------+-----------+------+
; 57.22 MHz ; 57.22 MHz       ; CLOCK_50   ;      ;
+-----------+----------------+-----------+------+
```

By reducing the multiplier latency to 0, you have reduced its cycle latency; however, the circuit's critical path is considerably worse. This is a key trade-off in high-level synthesis: cycle latency can typically be traded-off with $FMax$ – by lengthening the clock period (worse $FMax$), we can normally reduce cycle latency by chaining operations together in a clock cycle.

At this point, three columns of the table at the end of this lab should be completely filled in. Good work!

5   Loop Transformations

The code for the Mandelbrot example contains a triply-nested loop. The outer loop walks over the rows; the first inner loop walks over the columns; the inner-most loop iterates through the iterations for a particular pixel.

If you look carefully at the inner-most loop, you can see it contains a *loop-carried* dependency: the ith iteration of the loop depends on the $i - 1^{th}$ iteration. Because of this dependency, loop pipelining could not achieve an II of 1 without reducing the multiplier latency to 0, as you did in the prior step of the lab. For this step, you will modify the C code to remove the dependency entirely, by using an approach called *loop interchange*.

The basic idea is to interchange the first inner loop with the inner-most loop, thereby producing an inner-most loop without *any* loop-carried dependency. Conceptually, in the original version of the code, each pixel was considered in turn. That is, the code computed whether a given pixel was in the Mandelbrot set, and then moved onto determine whether the next pixel was in the Mandelbrot set. With loop interchange, the order in which things are computed will be changed fundamentally: the inner-most loop will compute z_i for an entire row of pixels. After this is completed, the next run of the inner-most loop will compute z_{i+1} for the same row of pixels. In essence, the Mandelbrot set computations for an entire row of pixels will be "in flight". To make this work, we will need to store the z_i values for all such pixels that are in flight: an entire row of pixels.

In the code below, the first few lines of the function declare arrays that allow us to store intermediate data for an entire row of the image. Following these declarations, observe that the outer loop is over the rows of the image (as before). Within the outer loop, the original functionality has been split into three sections: In the first section, labelled `lp1`, there is an inner loop that initializes all data for a row of the image. The second section, has a doubly nested loop, where the loop interchange has been implemented. The top-level loop of this nest corresponds to the inner-most loop of the original code; the inner-most loop here, labelled

lp2, iterates over all the columns of a row. The third and final section of the code below, labelled lp3, performs the accumulation to count for all pixels in a row.

The code below implements loop interchange, and also *loop fission* (also sometimes called *loop distribution*), where the triply nested loop in the original code has been split into multiple separate loops[1].

```
int mandelbrot() {
    int i, j;
    int count = 0;

    int x_0 [WIDTH] = {0};
    int y_0 [WIDTH] = {0};
    int x [WIDTH] = {0};
    int y [WIDTH] = {0};
    unsigned char fiter[WIDTH] = {0};

    for (j = 0; j < HEIGHT; j++) {

lp1:    for (i = 0; i < WIDTH; i++) {

            // find x_0 and y_0 in fixed point by interpolation
            // Mandelbrot x-range [-2:1] -- "width" is 3
            // y-range [-1:1] -- "width" is 2
            x_0[i] = int2fixed(-2) + (((((3 << 20) * i)/WIDTH) ) << 8);
            y_0[i] = int2fixed(-1) + (((((2 << 20) * j)/HEIGHT) ) << 8);
            x[i]=0;
            y[i]=0;
            fiter[i]=0;
        }

        int xtmp;
        unsigned char iter;
        for (iter = 0; iter < MAX_ITER; iter++) {
lp2:        for (i=0; i < WIDTH; i++) {
                long long abs_squared = fixedmul(x[i],x[i]) + fixedmul(y[i],y[i]);
                xtmp = fixedmul(x[i],x[i]) - fixedmul(y[i],y[i]) + x_0[i];
                y[i] = fixedmul(int2fixed(2), fixedmul(x[i],y[i])) + y_0[i];
                x[i] = xtmp;

                fiter[i]  +=  abs_squared <= int2fixed(4);
            }
        }

lp3:    for (i=0; i < WIDTH; i++) {
            //get black or white
            unsigned char colour = (fiter[i] >= MAX_ITER) ? 0 : 1;
            //accumulate colour
            count += colour; // we count the # of pixels that "escaped" (NOT in Mandelbrot set
        }
    }

    return count;
}
```

To find the above code, change directories:

---

[1]In this example, the loop nest cannot be interchanged without first performing a loop distribution.

```
cd ~/legup_repo/examples/mandelbrot/lab/part_b
```

Apply loop pipelining to the loop where interchange has been applied: that labelled `lp2`. And, also set the multiplier latency back to its original value of 1. Open up `config.tcl` and incorporate the following:

```
loop_pipeline "lp2"
set_operation_latency multiply 1
```

Now, synthesize the design and run ModelSim:

```
make
make v
```

You should see cycle latency results roughly as:

```
# Cycles:                  378895
```

The cycle latency results are similar to that observed in the previous step, when the multiplier latency was set to 0.

However, observe, in the code above, that is also possible to pipeline the loops with the other labels, `lp1` and `lp3`. Change the configuration file to turn on loop pipelining for these loops as well. Add the following lines to the `config.tcl` file:

```
loop_pipeline "lp1"
loop_pipeline "lp3"
```

Now, synthesize the design (where three loops are pipelined) and run ModelSim:

```
make
make v
```

```
# Cycles:                  237775
```

A significant reduction in cycle latency has been achieved. You may wish to look at the design in the GUI:

```
scheduleviewer scheduling.legup.rpt -p pipelining.legup.rpt
```

In the left panel of the GUI, you will see three basic blocks with the names: `wait_lp1_1`, `wait_lp2_1`, and `wait_lp3_1`, corresponding to the three pipelined loops. Take a look at the schedules for these three loops. The schedule for `lp1` is particularly interesting: the II of the loop is 1, however, the length of the pipeline is over 30 cycles long, as illustrated in Figure 6. The reason for this is that `lp1` contains a division operation, which itself is heavily pipelined by LegUp to improve the $FMax$ of the design.

Implement the design on the Cyclone II FPGA:

```
make p
make f
```

Complete the table at the end of the document under the column "Loop Transform".
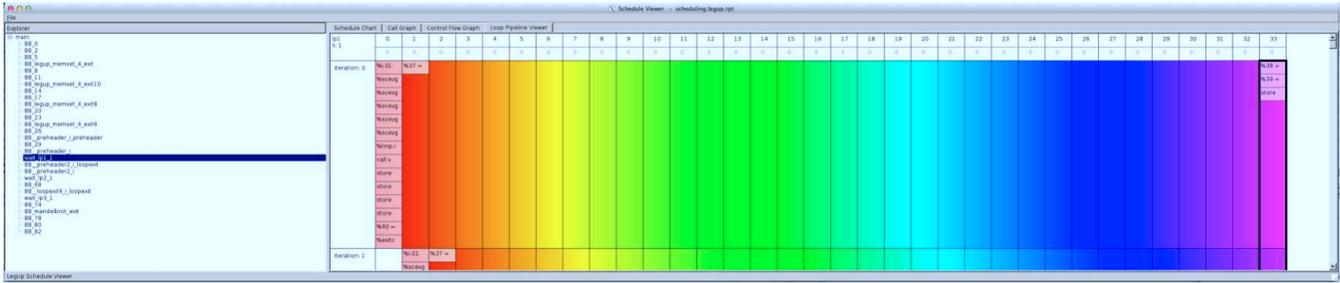
Figure 6: Schedule for loop `lp1` in the transformed Mandelbrot code.

6 Parallelizing Mandelbrot

Congratulations! If you are reading this, you have survived to make it to the last step of this exercise! In this step, we will exploit a key advantage of moving computations into hardware: the ability to exploit spatial parallelism. We will parallelize the software code using Pthreads, and then synthesize the parallel code into parallel hardware. LegUp is currently the only HLS tool that has such capability.

At a high-level, the idea is to launch four threads, each of which will be responsible for computing the pixel values for 1/4 of the rows of the image. The HLS tool will then synthesize four Mandelbrot "cores" in the hardware, that operate concurrently with one another. Since each core is responsible for only 1/4 of the original work, the total cycle latency should be reduced to roughly 25% of the original latency. Naturally, the resultant circuit is expected to take up about $4\times$ the area on the FPGA. In essence, we are trading off area for time – an optimization approach that is very typical in computer hardware design.

To find the parallel code, change directories:

```
cd ~/legup_repo/examples/mandelbrot/lab/part_c
```

Open `mandelbrot.c` in the editor and you will see the snippet of code below. A C structure is defined at the top of the file with two fields: `startidx` and `maxidx`. For each thread, these integers hold the starting row and ending row of the image that each thread is responsible for. The `mandelbrot` function is nearly identical to that used in the first case of this lab, however, in this case, the function parameter is a pointer to the structure (cast to `void *`). The first thing the function does is to extract the starting and ending rows from the structure. These are then the bounds of the outer loop (`for j`).

```
struct thread_data{
    int  startidx;
    int  maxidx;
};

volatile unsigned char img[WIDTH][HEIGHT];

void *mandelbrot(void *threadarg) {
    int i, j, tid;
    int count = 0;
    struct thread_data* arg = (struct thread_data*) threadarg;
    int startidx = arg->startidx;
    int maxidx = arg->maxidx;

    for (j = startidx; j < maxidx; j++) {
        for (i = 0; i < WIDTH; i++) {
```

Looking down in the file, you will see within the `main` function, the code snippet below, which sets up the work for each thread (the set of rows it is responsible for). Here, NUM_ACCEL is the number of accelerators that should be synthesized (4 in this case).

```
//initialize structs to pass into accels
for (i=0; i<NUM_ACCEL; i++) {
    data[i].startidx = i*OPS_PER_ACCEL;
    if (i == NUM_ACCEL-1) {
        data[i].maxidx = HEIGHT;
    } else {
        data[i].maxidx = (i+1)*OPS_PER_ACCEL;
    }
}
```

Further down in the file, you will see calls to `pthread_create` and `pthread_join`, which create the threads and wait for them to completely, respectively.

After you are comfortable with the code, execute HLS and simulation:

```
make parallel
make v
```

In this case, you will not be able to use the GUI visualizer as you did in the previous steps – it cannot properly display the hardware for software with Pthreads.

Make a note of the cycle latency and then synthesize the circuit to the Altera Cyclone II FPGA:

```
make p
make f
```

Complete the table at the end of the document by filling in the "Parallel" column.

7   Overall Results

Table 1: Results worksheet.

	Pure hardware	Pipelining	Multiplier lat=0	Loop transform	Parallel
Cycles					
FMax					
Clock period					
Wall-clock time					
LEs					
Multipliers					

One of the advantages of using HLS over writing custom hardware is that the space of solutions can be explored fairly easily – solely by changing software or the constraints on the tool. Imagine how difficult it would have been to try out all the different hardware configurations you did above if you had to manually modify Verilog code.

Having completed the results table, compare the different implementations to one another. Try to answer the following questions, which will be discussed in class:

- Which implementation has the lowest cycle latency?

- Which implementation has the best wall-clock time?

- Does loop pipelining increase area? If so, why?

- Which implementation uses the most area (#LEs) on the FPGA? Why?

- Why is the $FMax$ worse in the implementation where the multiplier latency was 0?

- Which implementation offers the best area-delay product and why? (#LEs × wall-clock time)

8 Appendix: More on the LLVM Intermediate Representation (IR)

Returning to the schedule viewer GUI in Figure 3, the first several 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.

The rest of the instructions have names that more closely resemble assembly code. For example, `mul` is a multiply operation, `add` is an addition, `lshr` is a logical-shift right, `zext` is a zero-extend operation, and so on. The variable names in the LLVM IR are preceded by a % sign. Let's dissect an example:

```
%20 = lshr i64 %21, 27
```

In this case, the variable %21 is shifted right by 27 bits and the result is stored in the variable %20. The `i64` indicates the datatype involved: a 64-bit integer. At the end of the instructions listed in Figure 3, you will see a branch instruction, `br`:

```
br i1 %exitcond, label %30, label %10
```

This is a condition branch that depends on `%exitcond`. Looking at the earlier instructions, you will see that `%exitcond` is the result of an integer compare operation (`icmp`). If `%exitcond` is true, control flows to the basic block with the label %30; otherwise, control flows to the basic block with the label %10.

References

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

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

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