Prof. Kozyrakis

STANFORD UNIVERSITY

Lab 4: Let's build a hardware accelerator

Due: Tuesday, March 8th, 2022 at 11:59 PM

1 INTRODUCTION

In Lab 3, you implemented a pipelined MIPS processor that was capable of running general-purpose programs. One of these programs was the Sobel edge detection convolution, which was based on Lab 2. You saw that the flexibility of general-purpose hardware comes at a cost: The MIPS processor required millions of cycles to process a single frame of video. In this lab we will trade generality for efficiency and performance. We will implement the same Sobel algorithm directly in hardware by building a custom accelerator. When you are done, you will see dramatically improved performance compared to the previous labs.

2 REQUIREMENTS

We will provide you with a hardware accelerator that is mostly complete. You will need to implement the control logic for the design as well as finish the calculation of the convolution computation.

We provide tools that enable you to develop your design in a test-driven simulation environment. We also provide tests to get you started, but it is your responsibility to make sure that your design is functionally correct. When you are confident with your implementation, you will be able to synthesize and run your design on the lab board and use the video file as input.

In this lab, you will be required to:

1. Finish implementing the sobel convolution in the accelerator.
2. Implement the accelerator's state machine and all other control logic.
3. Synthesize and run your design on your lab board.

3 STARTER CODE

Obtain the starter code with from canvas.

The relevant components of the starter code are organized as follows:

- **README**: The project report that you will write.
- **make_submission.sh**: A script that packages your code, tests, and report for submission.
- **hw/**: Verilog source code and Zedboard files.
  - **Makeftle**: Makefile for building the Zedboard image.
  - **hdl/verilog/sobel/**: Incomplete Verilog source code for the Sobel accelerator.
    * **sobel_deftnes.v**: Helpers and configuration options for the Sobel accelerator, including the number of accelerator cores to be instantiate and macros specifying the commands that can be sent to the row data registers.
    * **sobel_top.v**: Top-level Sobel accelerator module, instantiates all other hardware modules and implements the interfaces between them.
    * **sobel_accelerator.v**: Source code for the Sobel accelerator cores, one of the hardware modules that must be completed as part of this lab's required tasks.
    * **sobel_control.v**: Source code for the Sobel accelerator control logic, one of the hardware modules that must be completed as part of this lab's required tasks.
    * **sobel_*.v**: Other hardware modules that form part of the Sobel accelerator, with implementation already complete (not part of the required tasks for this lab but might be interesting to read).
- **sim/**: Simulation and test infrastructure
  - **Makefile**: Test driver
  - **tests/**: Sobel accelerator unit tests

4 ACCELERATOR MODEL

Unlike Lab 3, in this lab there is no MIPS processor and, therefore, no C or assembly code. Instead, the Sobel algorithm is coded in Verilog and implemented directly in hardware. The basic flow of execution is as follows.

1. The host processor sends an input image to the hardware Sobel accelerator by transferring pixel data to its input memory buffer.
2. The hardware accelerator performs Sobel edge detection on the input image, placing output into its output memory buffer and, when finished, signals completion to the host processor.
3. The host processor reads the output from the accelerator's output memory buffer.

All images used in this lab are stored in row-major order. The driver running on the host processor will automatically pre-process the input image by converting it to grayscale. The driver will also pad the image with 0's where appropriate, so the accelerator hardware does not need to worry about detecting the edge of the image. Likewise, the driver will cut into pieces any image too large to fit in the hardware's input and output buffers; this is completely transparent to the hardware. The driver informs the hardware of the dimensions of the input image, and the hardware need simply perform Sobel convolution calculations on the contents of the input buffer, without worrying about edge-of-image or other special cases. As a result of the driver's automatic padding of the input buffer, for an input image of dimensions w by h, the corresponding output will have dimensions w-2 by $h-2$.

The subsections that follow provide greater details on the intended operation of the Sobel hardware accelerator that is to be constructed in this lab.

4.1 CONTROL FLOW

The Sobel accelerator processes input pixels in column order, meaning it computes outputs for all rows in a set of columns before moving to the next set of columns. By default, the hardware produces outputs for two consecutive columns at a time. After producing the Sobel output for pixels (i, j) and $(i, j+1)$, it proceeds with pixels $(i+1, j)$ and $(i+1, j+1)$. To produce two simultaneous columns of output pixels, the hardware needs to read a strip of 4 columns of input, namely columns $j-1$, j, $j+1$, and $j+2$. By extension, to produce n columns of output pixels, the hardware needs to read a strip of $n+2$ columns of input; as the number of columns
processed simultaneously is increased, the strips get wider, but otherwise there is no key operational change. A strip of columns, or "column strip," is identified by the number of the first column in it ($j-1$ in the preceding example).

Once the accelerator finishes processing all rows in a given column strip, it must move to the next column strip to continue processing input. It detects this condition by comparing the current row number i to the number of rows in the input image, as indicated by the host processor. If this condition arises while processing the final column strip in the input image, the hardware accelerator knows it has processed the entire image and signals completion to the host processor. The accelerator can detect it is processing its final column strip by combining its knowledge of the number of columns in the input image, the current column strip it is processing, and the number of columns of output it produces simultaneously.

Figure 4.1 demonstrates the hardware's control flow for a 6-by-6 input image in which 2 columns are processed simultaneously. The result is a 4-by-4 output image computed over 8 steps. The hardware detects the bottom of the image in step 4, thus causing a switch to the next column strip, and detects the completion of its processing in step 8, after which it signals completion to the host processor. Notice that, when moving from one column strip to the next, there is some overlap in terms of the columns consumed as input (refer to the blue rectangles). In step 1, the accelerator is consuming columns 0, 1, 2, and 3, while in step 5, the accelerator is consuming columns 2, 3, 4, and 5.

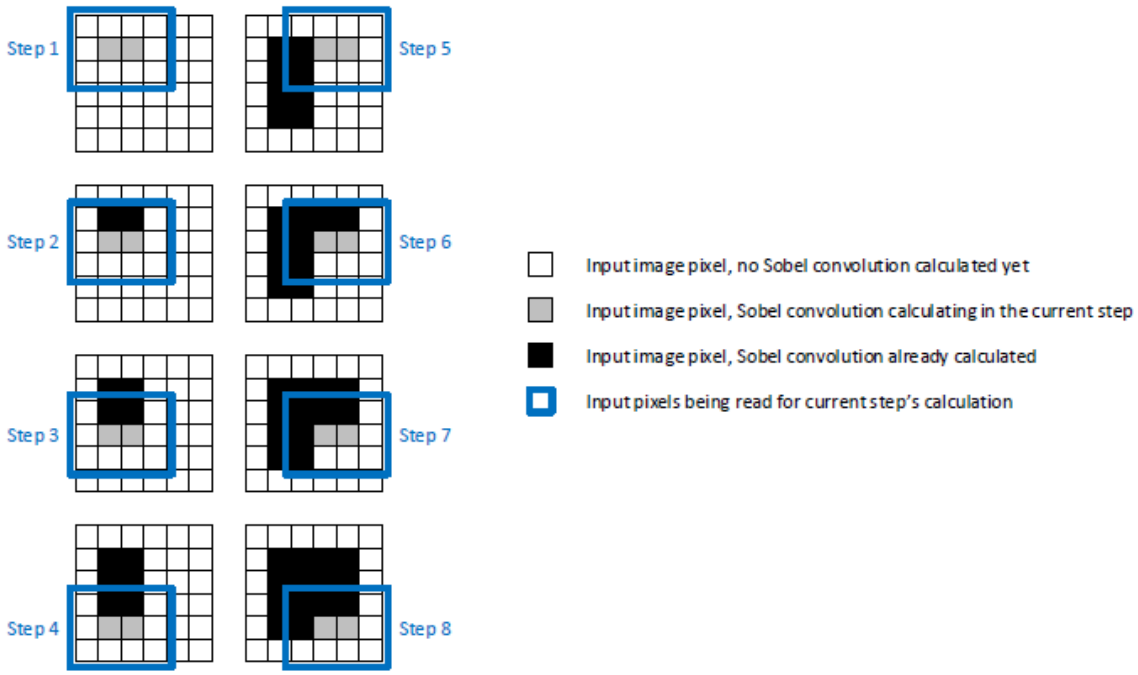

Figure 4.1: Illustration of high-level Sobel accelerator control flow.

4.2 Pixel Data Access

As with Lab 3, input and output pixel data are stored in separate byte-addressable memory buffers, one byte per pixel. Storage is row-major, so consecutive byte addresses represent different columns within the same row before moving onto column 0 in the next row. Pixel data are also ordered in memory as would be displayed in an image, with the top-left corner (0,0) stored at byte address 0 and increasing from there. In general, the byte address of the pixel at row i, column j is given by $i \times r + j$, where r is the image's row width.

All input and output memory addresses are *byte addresses* and must be 2-byte aligned (i.e. the last bit must be zero). There should never be a need to access memory other than with 2-byte-aligned accesses; at a minimum, the hardware will process 2 columns simultaneously.

4.3 Hardware Structure

Figure 4.2 contains a block diagram showing all the hardware modules in the Sobel accelerator and the interfaces between them. Red paths show the flow of control, and blue paths show the flow of data. Shaded boxes represent the modules that must be implemented for this lab's required tasks; anything unshaded is provided in its completed form and need not be modified.

Each hardware module is defined in an identically-named Verilog source file which can be found in the starter code. The subsections that follow detail the intended functionality of each of the hardware modules.

4.3.1 Memory Abstraction Layer

These modules, sobel_read_transform and sobel_write_transform, implement an abstraction layer required to make memory accesses work as described previously. For the purposes of this lab it is not necessary to understand how they function, but those curious are welcome to inquire during office hours.

4.3.2 Row Shift Register

As shown in Figure 4.1 and noted previously, the progression from one step to the next involves the re-use of two of the already-processed rows of inputs and only requires a single row of new input data. The sobel_image_rowregs module captures the required behavior in the form of what is essentially a two-dimensional shift register; the blue rectangles in Figure 4.1 show the pixels contained in the shift register at each step in the hardware's operation.

The row shift register accepts new data from the input buffer and supplies input data directly to the accelerator cores. The hardware module supports a very specific command set for determining what to do:

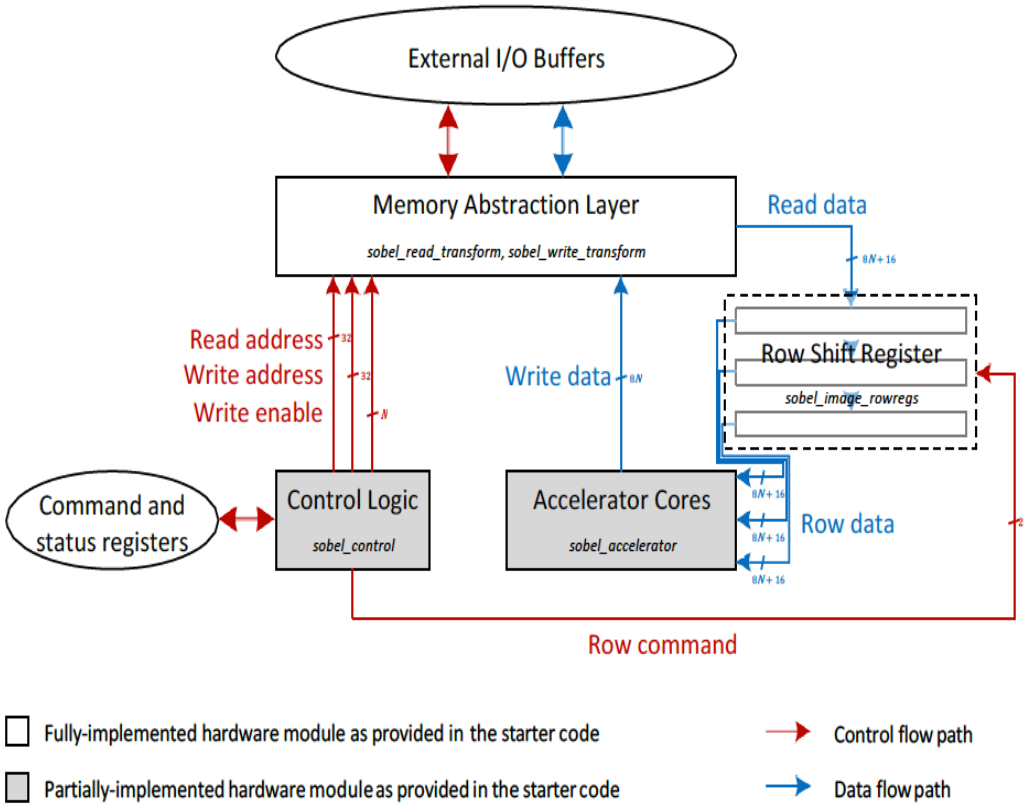

Figure 4.2: Top-level hardware block diagram of the Sobel accelerator; N is the number of columns produced simultaneously.

- **Hold**: do nothing, maintain existing contents.
- **Shift Rows**: place the incoming data in row 3, move old row 3 data to row 2, move old row 2 data to row 1, and discard old row 1 data. This operation is what is reflected in the blue rectangle as it progresses from one step to the next in Figure 4.1.

4.3.3 Accelerator Cores

Each instance of a Sobel accelerator core calculates a single Sobel convolution, consuming 3 input columns and producing a single pixel of output. In performing these calculations, notice that any multiplication operations are by a constant of 2. No multipliers are necessary; a simple left-shift by 1 bit position is sufficient. The circuitry in an accelerator core is entirely combinational.

The previous examples describing multiple consecutive columns of output being produced simultaneously are implemented by simply instantiating multiple Sobel accelerator cores. The number of accelerator cores is controlled by a pre-processor constant contained in the file "sobel_defines.v" with a default of 2; the hardware infrastructure supports values of 2, 4, 6, and 8. You are required to run a minimum of 2 accelerator cores.

All of the Sobel accelerator cores are captured in a single hardware module, sobel_accelerator. This module uses a Verilog generate block to programmatically create the correct number of accelerator cores. The body of the generate block implements each individual convolution.

This module is provided in an incomplete state in the starter code; required tasks for this lab include completing it.

4.3.4 Control Logic

The sobel_control module encapsulates all of the control flow for the entire hardware accelerator. At its core is a state machine, which keeps track of the hardware's overall progress through the image. At the start of the operation of the accelerator, it receives metadata (dimensions of the input image) from the host and, once instructed to do so, it begins processing the image. It derives other control information, as reflected in the red paths in Figure 4.2, from state it holds.

This module is provided in an incomplete state in the starter code; required tasks for this lab include completing it. The skeleton code defines all of the states it requires, but it omits all of the state transitions and all of the logic for deriving other outbound control signals. The accelerator states are described in Table 4.1 and defined in the "sobel_control.v" source file.

Table 4.1: List and descriptions of all Sobel accelerator control states.

Name	Description
STATE_WAIT	The wait state, during which the accelerator is waiting for a signal from the host indicating that the input buffer has been filled with valid data and that processing should commence.
STATE_LOADING_1	In this state, the accelerator loads the first row from the newly-filled input buffer into the row shift register. There are not yet sufficient data in the row shift register to calculate any Sobel convolutions.
STATE_LOADING_2	Same as STATE_LOADING_1, except that the accelerator is loading the second row.
STATE_LOADING_3	Same as STATE_LOADING_2, except that the accelerator is loading the third row. Once this row has loaded, the row shift register contains sufficient data to calculate a Sobel convolution.
STATE_PROCESSING_CALC	In this state, the Sobel accelerator is computing a Sobel convolution and producing output.
STATE_PROCESSING_LOADSS	In this state, the Sobel accelerator is shifting in a new input row and discarding the oldest row in preparation for the next round of Sobel convolution calculations.
STATE_PROCESSING_CALC_LAST	Same as STATE_PROCESSING_CALC, except that the accelerator has reached the bottom of the image; this is the final convolution that is to be calculated for the current column strip.
STATE_PROCESSING_LOADSS_LAST	Same as STATE_PROCESSING_LOADSS, except that the accelerator has reached the bottom of the image; this is the final row that is to be loaded into the row shift register for the current column strip.
STATE_PROCESSING_DONE	The completion state; the accelerator has finished processing the entire input buffer and is waiting for the host to provide it with a new one.
STATE_ERROR	The error state, indicating that an error has occurred. The starter code by default causes all transitions to go to this state which, once reached, will never be left.

4.3.5 Helpers and Configurable Options

The file "sobel_defines.v" is where Sobel accelerator configuration options and helpers are defined. For the purposes of completing this lab, this file should be used:

- To specify how many Sobel accelerator cores to instantiate (either 2, 4, 6, or 8, with a default of 2 in the starter code)
- As a reference point for the commands supported by the row shift register, including symbolic names that should be used in place of numeric values

4.4 Accelerator Core Scaling

The design supports increasing the number of Sobel accelerator cores from 2 to 4, 6, or 8. For this lab, we require only that you implement the base case of 2 cores, however you may optionally choose to support more.

Changing the number of accelerator cores is as simple as modifying the appropriate line in "sobel_defines.v," but successfully running with a higher number of accelerator cores requires careful consideration while completing this lab's required tasks; be sure to avoid making any assumptions that depend on a particular number of accelerator cores. For example, if the number of columns in the input image is not evenly divisible by the number of accelerator cores, there will be situations in which some of the accelerator cores need to be disabled, situations which must be detected and handled properly by the control logic. Disabling an accelerator simply means setting its pixel_write_en bit to 0 (see "sobel_control.v").

5 Setup Instructions

We use the same simulation infrastructure as lab 3. If you didn't work locally for lab3, it is recommended you work on the `rice` machines (`rice.stanford.edu`). All software needed for the simulation part of this assignment is installed on these machines. See section 7 for information regarding bitstream generation. You should unzip the starter files work inside the **afs-home** directory. This will be required later on for synthesis in section 7. You can get the starter files by scp-ing the files and then unzipping:

```
cd afs-home
unzip lab4_starter.zip
```

Once you unzip the starter code, it should be ready to run and no further setup is required.

5.1 Running from home

Alternatively, you may be able to do your design testing and simulation on your own computer instead of using the lab machines (you will still need the lab machines to synthesize the hardware design for the demo). For this you will need a unix-like environment, and you will first need to install **iverilog,** which simulates Verilog designs, and **gtkwave,** which is a waveform viewer.

6 Simulation infrastructure

The simulation tools require iverilog and gtkwave. If you are running on a lab machine then all of these packages are installed. If you are running on your own machine, you need to make sure these are available.

The starter code comes with a small selection of tests in the form of an input image and golden output image against which your accelerator's output will be compared. Supplied tests include:

- **basic**: sanity check. Input is all 0s, output should be 0s. Intended to verify the correct number of output bytes are produced in the correct locations in the output buffer and no spurious memory writes are performed.
- **image***XX*: small image tests which exercise various parts and corner cases of the Sobel algorithm. It is recommended that you use these tests when performing waveform debugging, as the images are small and manageable.
- **imageMIPS:** a real full-sized image test, taken from the same cat input video used in Labs 2 and 3.

To run all of the supplied tests, go into the sim directory and type

```
make test
```

This command will search the tests directory for any existing subdirectories. Each subdirectory is the name of a test and contains image buffers and simulation configuration files for that particular test.

When make test finishes running each of the tests in the tests directory, it will aggregate each test's result and display a report which lists each test and displays whether the test passed or failed. When you change the hardware design (Verilog) files, running make test again will automatically rebuild any outdated components and run those tests again.

To run a single test, prefix the test name with "test_". For example, to run the 'basic' test you would run

```
make test_basic
```

To debug a test with the waveform viewer, prefix the test name with "wave_". For example, to view the waveform of the 'basic' test, run

```
make wave_basic
```

Note that if you are working remotely over SSH, you will need to enable X forwarding (ssh -Y) to view the waveform.

To add a new test, create a folder in the sim/tests directory with the name of the new test (it is probably easiest to copy an existing test and modify it for your needs). The test infrastructure will automatically include the new test the next time you run it.

The test infrastructure determines if a test passes or fails by comparing the output produced by your design with a golden output produced by a reference implementation. The harness is a Verilog module that instantiates your design, loads the input image buffer, and waits for your design to enter STATE_DONE, at which point the simulator exits and the infrastructure determines if the test passes or fails.

It is important to note that the supplied tests are only a starting point for your design. They do not necessarily cover all of the functionality that you will need to implement and are not guaranteed to even work in all situations.

When the tests pass you are ready to synthesize your design and try it out with baxter.avi.

7 Running your Design on the Board

In order to see your Sobel accelerator in action, you should generate a bitstream from your code and use the bitstream to program the FPGA. You will follow a procedure similar to the one used in lab 2. The following sections should serve as a refresher on this procedure.

NOTE: You should attempt to test your design on the board **only after all tests pass in simulation**. Logic synthesis is a time-consuming process. Also, you have almost zero visibility into the individual design signals when running on the board. Synthesizing your design without passing simulation tests is a bad idea, and you will waste a considerable amount of time on it unnecessarily.

7.1 Logic Synthesis

Logic synthesis is the process by which a hardware design description – like your accelerator – is turned into a design implementation in terms of logic gates. This process involves multiple synthesis tools, which convert your design HDL into a binary file called *bitstream*. To the end user like you, the flow resembles a (fairly slow) compilation-like process.

You have been provided with a Makefile interface to generate the bitstream.

- Navigate to the hw directory containing the Makefile and type make. This process will take a very long time and will create fpga.img.
- To clean up files from an old build, type make clean.

You must be logged into the caddy.best.stanford.edu to build the bitstream (just as in lab 3). Running logic synthesis requires proprietary Xilinx tools and licenses, which can be accessed through the lab workstations.

7.2 Programming the FPGA

Once the bitstream is generated, programming the FPGA is a simple two-step process:

1. Copy fpga.img to your Zedboard using scp. For example:
   scp fpga.img user@ee180-40z:
2. From your Zedboard, run the prog_fpga command:
   prog_fpga fpga.img

Programming the FPGA can take a few seconds to complete. During the programming process, you can observe that the blue LED on the Zedboard turns off. Once the programming has completed successfully, the blue LED turns back on again.

7.3 Running your Sobel Hardware

We have installed a binary on your board named lab4 which can be found in /usr/bin. You should be able to run this binary from any folder.

The generated executable lab4 accepts numerous options. We have highlighted a few use cases below:

1. Run the accelerator once with a specific input buffer and image size, optionally dump output buffer to a file:
   lab4 -i inbuf.hex -r numRows -c numCols
   lab4 -i inbuf.hex -r numRows -c numCols -o outbuf.hex # Dump output buffer
2. Run accelerator in a loop with input video from a video file. Note that the frame dimensions are fixed; using -r or -c has no effect in this case.

   lab4 -f <path_to_file>

Run the executable without any options to see a helpful message describing each of the options in detail. We will only be using the second use case with baxter.avi to test your implementation. Note that it may take until around frame 11 (~1 minute after running) before the video appears over X11.

8 Extra Credit: Feeling Adventurous?

A close examination of the finite state machines and the overall architecture of the accelerator will reveal that the current design does not utilize memory bandwidth from the input buffer effectively. Data read from the input buffer and the Sobel accelerator run in sequence, one after another. In principle, this flow can be pipelined; data elements needed for the next accelerator run can be read in parallel with the current run of the accelerator.

For extra credit, implement a pipelined dataflow path as described above. Describe all the changes you make to the datapath and the FSM in your README. Report the speedup in the *accelerator's* run time compared to a baseline (but functional) accelerator that does not have such a pipelined datapath.

Note that your extra credit submission must run correctly and pass all simulation tests for it to be considered. You should be able to achieve a speedup of at least 30% in order to receive extra credit.

If you have other ideas on optimizing the accelerator, we would love to hear them. Talk to the TAs about what you have in mind!

9 Suggested workflow

We recommend that you establish your workflow based on the following guidelines:

- Version and backup your code regularly: Use your favorite version control tool.
- Start early: This lab has fewer days than lab 3. It is all too easy to underestimate the amount of work.
- Get comfortable with waveforms: You will be staring at waveforms for quite a bit in this lab using GTKWave during debugging.
- Code – Test in Simulation – Code: While your final design must run on the board, simulation offers substantially faster turnaround times during coding or debugging. Logic synthesis is a slow process and each build takes a few minutes to run, compared to a few seconds for simulation. A wise idea is try synthesis only if all tests are passing in simulation.
- Draw out the state transition diagram: Make sure you understand the role of each state in your state machine and the conditions required to transition from one state to another. Drawing out a state transition diagram will not only help you in this process, but will also give you a detailed picture of how the control signals are assigned in each state.

- Think of the block diagram: Before starting out to write Verilog, make sure you clearly understand what you are implementing. If you cannot explain something on a drawing board, chances are you cannot implement it in Verilog because you don't yet fully understand all parts.
- Write new test cases: Think of ways in which you can stress test your accelerator. Every good test written will save you time in the future.
- Test often: Implementing one piece of functionality could break another. Running your tests often can help catch such issues quickly.
- Do not wait until the last day to run logic synthesis. You have to make sure that your design meets all the timing constraints. Waiting until the last day means that you have less time to debug any last minute timing/board related issues.

10 Resources

- Verilog reference guides and tutorials:
  - Stuart Sutherland's Verilog 2001 reference guide
    http://www.sutherland-hdl.com/online_verilog_ref_guide/verilog_2001_ref_guide.pdf
  - Verilog tutorials, reference information
    http://asic-world.com/verilog/index.html
- Reference material on iverilog: http://iverilog.wikia.com/wiki/User_Guide
- Reference material on GTKWave: http://gtkwave.sourceforge.net/gtkwave.pdf

11 Grading

1. Base requirements(100%)
   a) Simulation tests (70%)
   b) README (15%) containing the following:
      - Partner and group names
      - Detailed description of changes made to implement the accelerator core and state machines.
      - (Extra credit only) Description of changes made to implement a pipelined datapath.

c) Accelerator on the board (15%)

2. Extra credit - Pipelined datapath (10%)

Note: **Extra credit will be awarded only if the base requirements are also met**. Work for extra credit will NOT be regarded if your design fails to meet all base requirements listed above.

12 SUBMISSION

In order to create a submission tarball, run "make_submission.sh" in your top-level directory. The submission tarball will be named lab4_submission.tar.gz. This tarball contains all your Verilog source files, all test files, and your README.

Please submit through Gradescope as a group. All submissions must be made electronically.