

Lab 3: List Processor & ChipScope

Contents

| | |
|------------------------------------|----|
| 0 Introduction | 1 |
| 1 Prelab | 2 |
| 2 List Accumulator | 2 |
| 2.1 Skeleton Files | 3 |
| 2.2 Block RAM | 3 |
| 2.3 Datapath Design | 4 |
| 2.4 Controller | 6 |
| 2.5 Deploying to Hardware | 6 |
| 3 ChipScope | 6 |
| 3.1 CORE Generator (coregen) | 7 |
| 3.2 Integrating ChipScope | 12 |
| 3.3 ChipScope Analyzer | 12 |
| 4 Checkoff | 15 |

0 Introduction

Although the software simulation techniques from the previous lab should remain your first line of defense against bugs, at the end of the day, designs must work on the physical device. Verilog was created for hardware modeling, not synthesis, as a result not all Verilog constructs map to hardware. This leads to designs that run correctly in simulation but fail on hardware. In this lab, you will learn to use ChipScope to examine actual signals generated by a list processor running on the FPGA.

Additionally, this lab is intended to introduce design of complete digital systems (rather than individual components, as in previous labs). Datapath and control design will be a major focus of the upcoming term project - it is therefore in your interest to ensure you understand the reasoning behind the design decisions in this lab.

Finally, **this lab requires that you complete a prelab document before attending section!**

1 Prelab

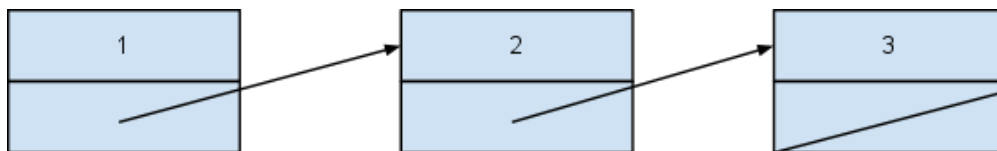
Complete this before attending lab section.

Read all of section 2 and complete the worksheet before attending lab. A TA will check your prelab at the beginning of the lab section.

2 List Accumulator

A common feature of early computer architectures is an “Accumulator” circuit. In general, this pattern uses a register to hold intermediate values of a series of operations rather than storing and fetching from memory for each one. In this lab, we will build an accumulator that processes linked lists using the ALU created in Lab 2.

Recall that a linked-list is a sequence of nodes, each of which contains a data element and a pointer to the next node. This example shows the list 1, 2, 3 with null-termination:



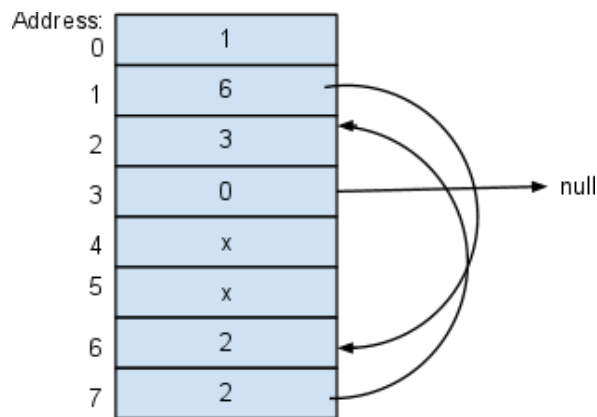
In C code, each node would likely be represented using a struct:

```
typedef struct list_element {
    int val;
    struct list_element *next;
} Node;
```

When instances of this data structure are stored in memory, the two fields will be in subsequent addresses. This lab simulates that behavior. The skeleton files contain a 1024 x 32-bit single-port read-only memory. The list will be stored in the following manner:

- The first node will always be at address 0.
- The data and pointer fields of a given node will be in sequential memory addresses, with the data before the pointer.
- The pointer field in the last node of the list will be 0.

The diagram below shows an example of a valid memory configuration representing the list {1, 2, 3}:



Use this list to complete prelab question 3.

2.1 Skeleton Files

In your `labs` directory, enter the following commands to acquire the skeleton files:

```
wget http://inst.eecs.berkeley.edu/~cs150/fall2/lab3/lab3.tar.gz
tar -xzf lab3.tar.gz
```

The skeleton files contain a top-level module, a datapath module, a controller module and a directory containing configuration files for the memory described in the previous section. For this lab, you will need to complete `Lab3Datapath` and `Lab3Controller`. You should not modify `m1505top` or any of the module interfaces.

You will need to copy your completed `ALU.v` and `ALUdec.v` from Lab 2 into the `src` directory.

2.2 Block RAM

As mentioned above, the skeleton files provide configuration files for a 1024x32-bit read-only memory (ROM). This memory utilizes the built-in Block RAMs on the FPGA and the memory contents can be initialized. Go to the `blk_ram` directory in the skeleton files:

```
cd ~/labs/lab3/src/blk_ram/
```

This directory contains 4 files:

- `blk_mem_gen_v4_3.xco`: This file is generated using coregen (more in the next section) and contains configuration information.
- `small_list.coe`: This file is used to initialize the memory. Each row corresponds to an address (starting with 0) and the contents are in hex.
- `build`: Run this script (`./build`) to generate the Block RAM
- `clean`: Run this script (`./clean`) to delete the generated files. **Important: do not run this from the GUI or you will lose work.**

Open the configuration file (blk_mem_gen_v4_3.xco) and find the following line:

```
CSET coe_file=./small_list.coe
```

The .coe file specified in this option is used to initialize the contents of the block RAM when it is generated. Every time this file (or the file referenced in the configuration) changes, the memory needs to be re-generated using the build script.

The skeleton files provide an example list to test your design with. Complete section 2 in the prelab to familiarize yourself with the list.

2.3 Datapath Design

There are two primary tasks that need to be accomplished in the list accumulator datapath: list traversal and data accumulation. These tasks can be considered separately to simplify the datapath design process.

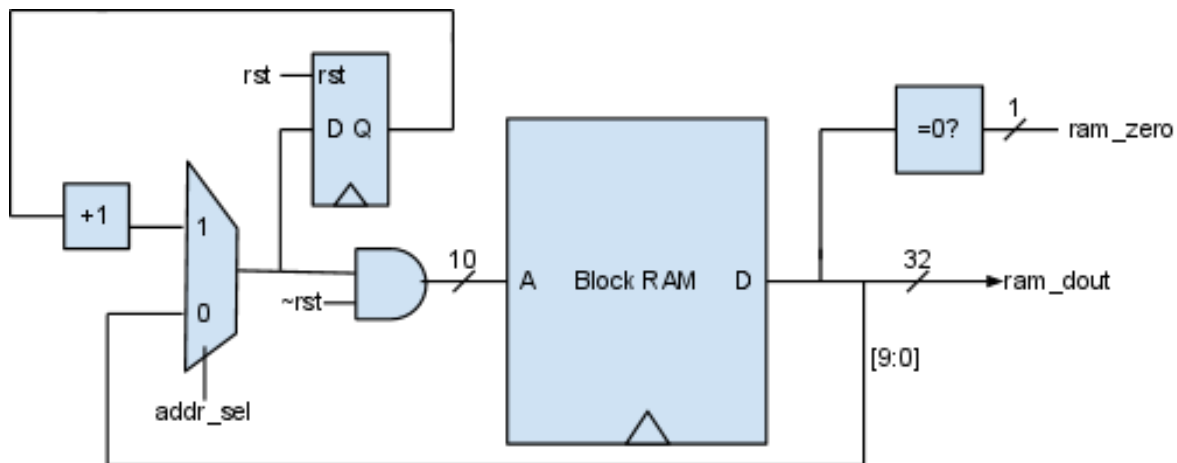
List Traversal

The memory provided in the skeleton files is single-ported, but for each node both the data and pointer fields need to be accessed. The list processor will therefore require two cycles for each element in the list. Each cycle has a different purpose:

Cycle 1: Fetch data: The address is 0 on reset, otherwise, the previous value loaded from the memory (which would have been a pointer) is the address.

Cycle 2: Fetch a pointer: Pointers are stored immediately after the data, so use the previous address incremented by 1.

This block diagram enables the functionality outlined above:

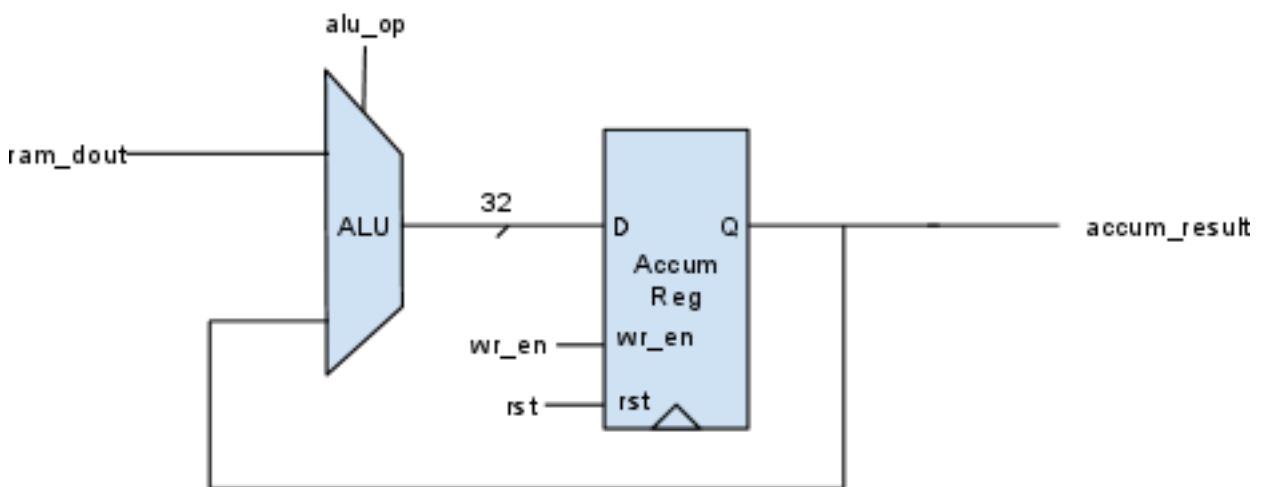


The list traversal component of this processor requires just one control signal, `addr_sel`, to determine the address for the block ram. Notice that on reset, the block ram will load address 0, which is the beginning of the list.

Accumulator Register

The accumulator component is straightforward: When data is loaded from the block ram, apply the operation on the data and the contents of the accumulator register. The use of a write enable signal allows the controller to prevent the accumulator from storing operations on the pointers.

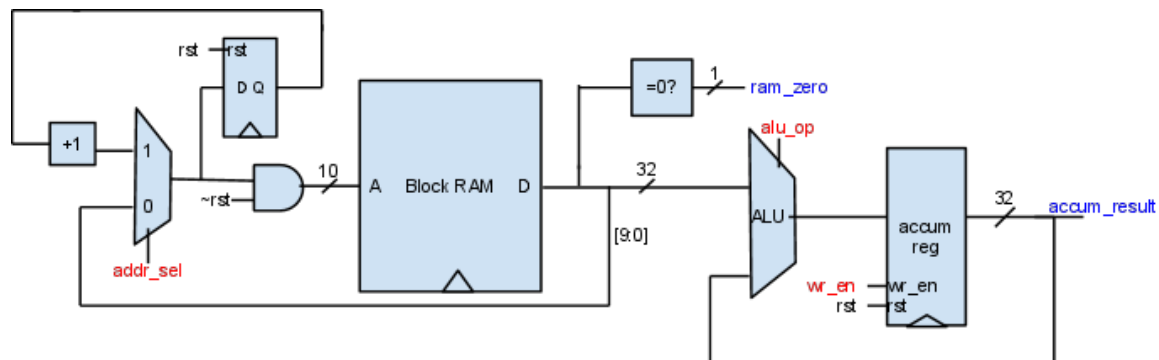
The accumulator portion of the datapath can be implemented in the following way:



This requires two control signals: the `alu_op` to set the accumulator function and a `wr_en` signal to prevent modifying the accumulator register contents when pointers are loaded or after the list ends.

The Complete Datapath

Combining these two components yields the complete datapath:



The interface to the datapath has been specified in `Lab3Datapath.v`. The

block diagram above shows the control signals in red and the outputs in blue.

Implement the datapath in `Lab3Datapath.v` after a TA has checked off question 3 of your prelab.

2.4 Controller

After a TA has checked off the prelab questions, implement the controller in `Lab3Control.v`. The controller should use the `ALUdec` module from Lab 2 to generate the `alu_op` output. Hard-code the `opcode` input to `6'b0` (R-type). The `funct` is set using the bottom six GPIO switches (this has already been wired into `m1505top.v`).

2.5 Deploying to Hardware

Once you have completed `Lab3Datapath` and `Lab3Control`, run `make` in the `lab3` directory. Next, run `make report` and verify that there are no unexpected synthesis warnings. (Though you should see warnings about “instantiating a black box module `<blk_mem_gen_v4_3`” and `accum_result[31:8]` unused).

After building the design successfully, deploy to hardware using `make impact`.

Verifying the Design

The skeleton files are configured such that the center compass switch resets the design, the center compass switch LED indicates that the list traversal is complete, and the GPIO LEDs show the bottom 8 bits of the accumulator register.

To check if your list accumulator is working, first configure the DIP switches to a known `funct`. For example, `ADD` is `6'b100001`. After setting the lower 6 DIP switches, press the center compass button to reset the CPU. The example list provided in `small_list.coe` runs nearly instantly, so the center switch LED should be on (if not, traversal isn't terminating). Finally, verify the sum displayed on the LEDs: the sum of the elements in `small_list` is 36, which corresponds to LEDs 2 and 5 lit.

Unfortunately, it's incredibly difficult to write correct Verilog on the first try. Furthermore, twiddling switches on hardware yields very little insight into design problems. The next step is to bring out the heavy artillery of FPGA debugging: ChipScope.

3 ChipScope

The ChipScope Analyzer is a tool that allows you to record internal signals of your design running on the FPGA when triggering conditions are met. In some circumstances this is preferable to software simulation because it removes

uncertainty created in the process of mapping Verilog to hardware.

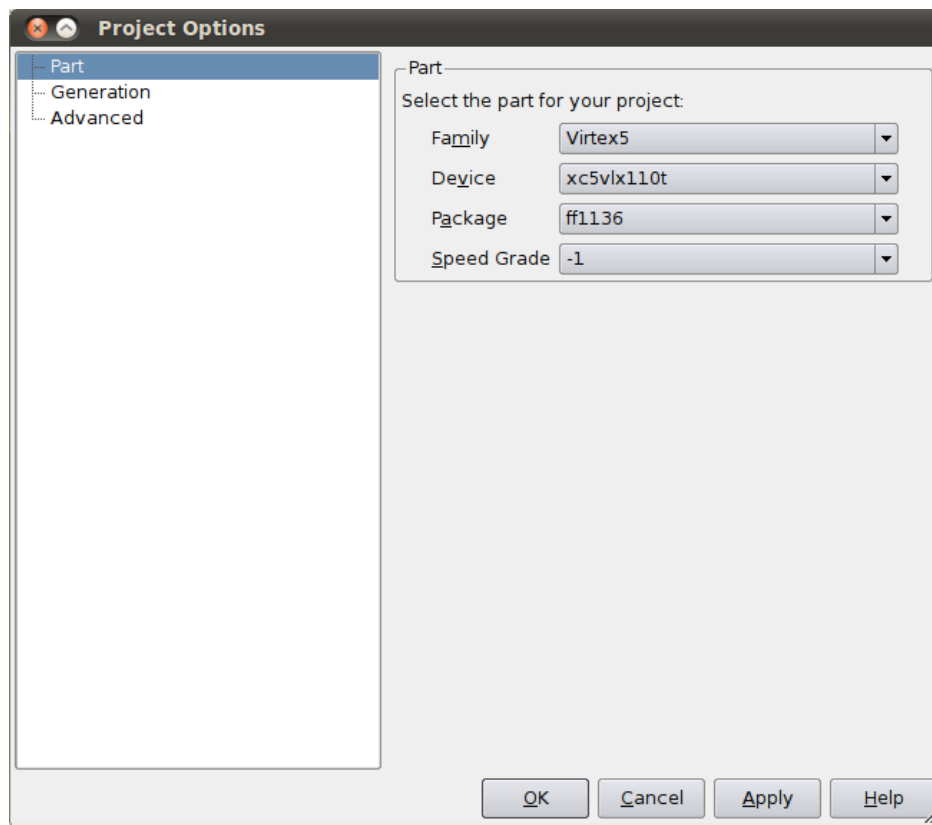
There are two modules that you will need to add to your design to enable ChipScope: The “Integrated Controller Core” (ICON) and the “Integrated Logic Analyzer” (ILA). You will generate these using Xilinx CORE Generator, a tool for customizing a variety of blocks from a library.

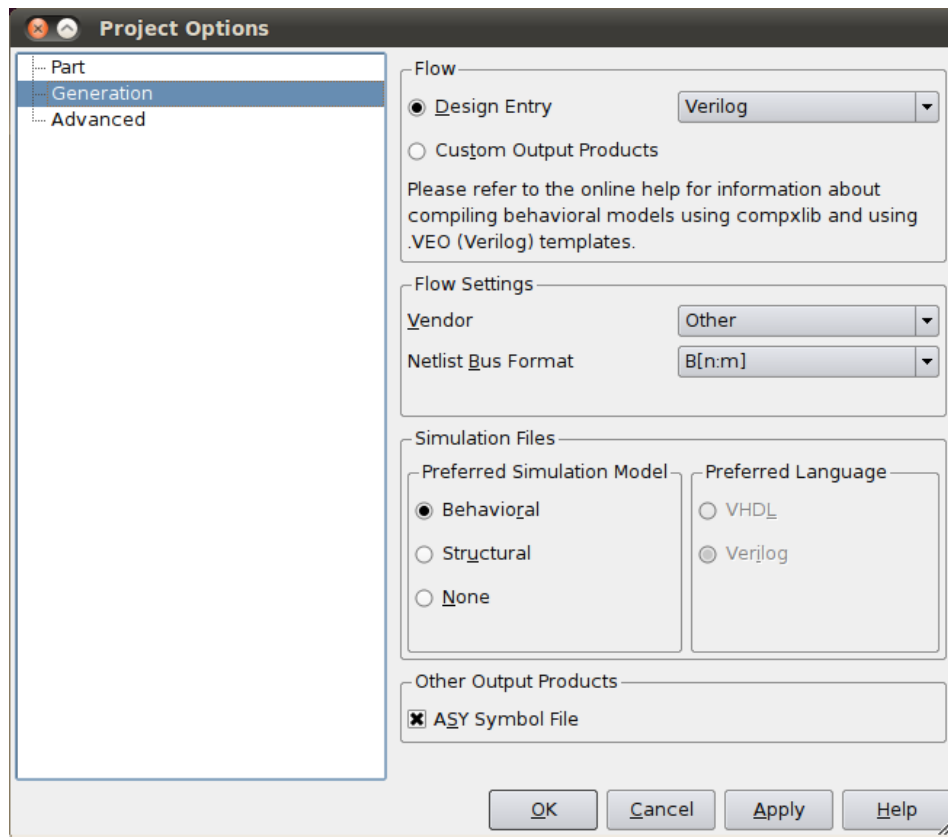
3.1 CORE Generator (coregen)

The first step is to make a new directory (within the `src` directory) for the generation output, then run `coregen`:

```
cd ~/labs/lab3/src
mkdir chipscope
cd chipscope
coregen &
```

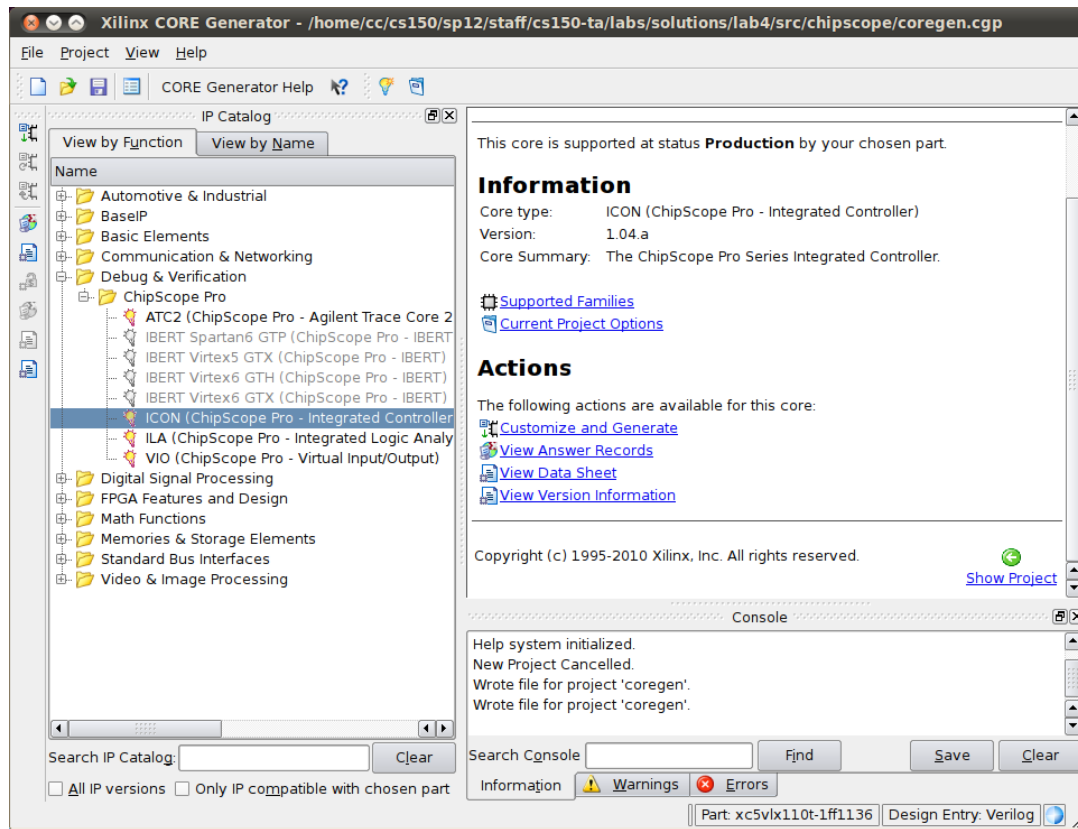
The `coregen` home screen will appear. Go to File -> New Project and press “Save” to store the project file in the `chipscope` directory. You will then need to customize the “Part” and “Generation” panels of the next dialog to match the screenshots:



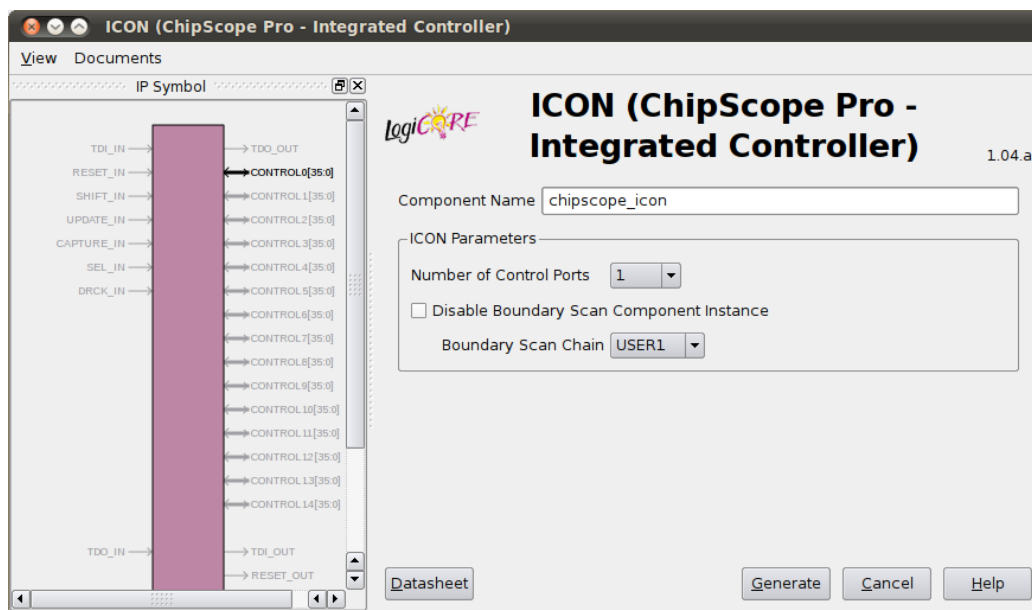


Leave the defaults in the “Advanced” options panel. Click “OK” once you have entered the project settings.

You should now be at the coregen home screen again. In the IP Catalog file tree on the left, select “Debug & Verification” -> “ChipScope Pro” -> “ICON (ChipScope Pro - Integrated Controller)”. After selecting this, the ICON project should appear in the main pane. Click “Customize and Generate” under “Actions” (you may need to scroll down):

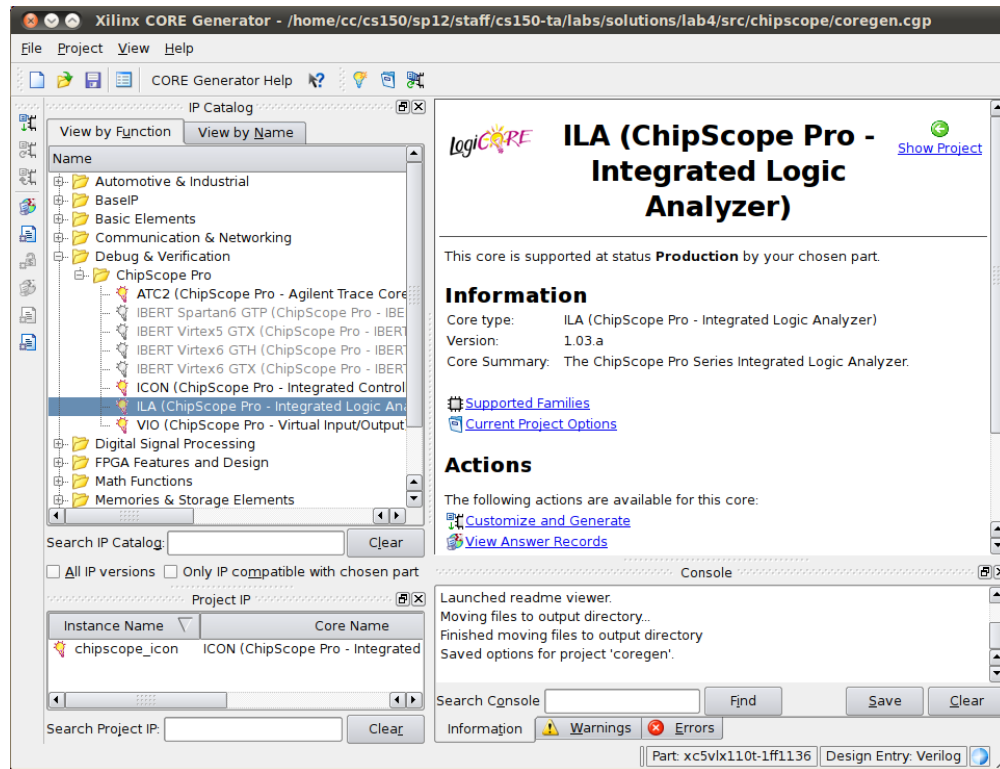


Use the default generation options (shown below) for the ICON. Click “Generate” to generate the module:



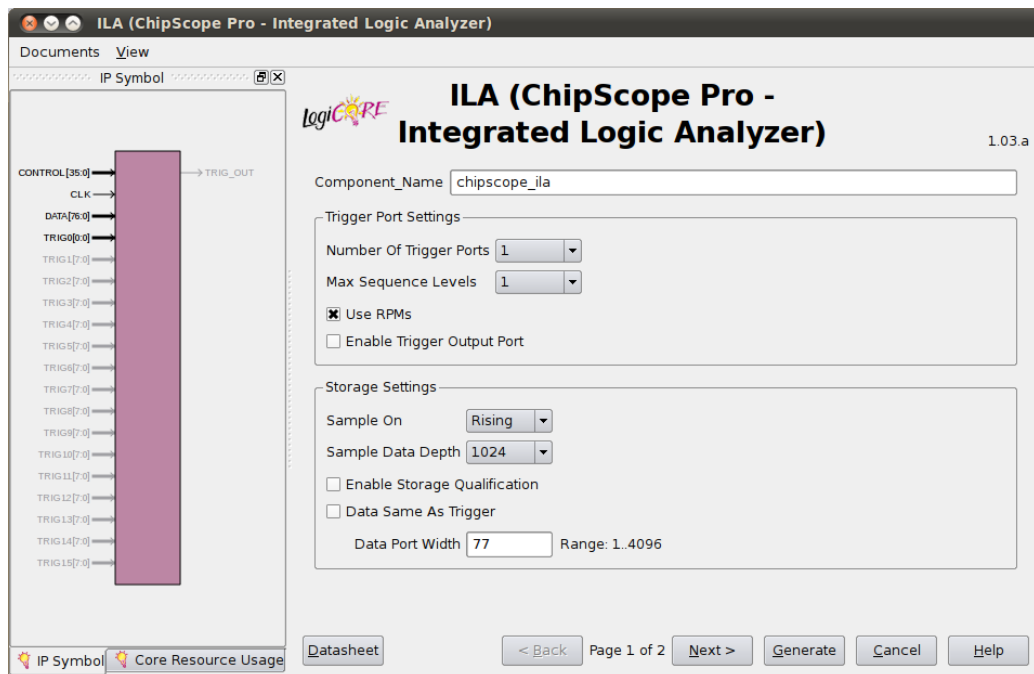
The generation process will take a few minutes. Once complete, you should again be at the main coregen window.

Next, to generate the ILA, select “Debug & Verification” -> “ChipScope Pro” -> “ILA (ChipScope Pro - Integrated Logic Analyzer)” from the catalog tree on the left. As before, click “Customize and Generate” under “Actions”:

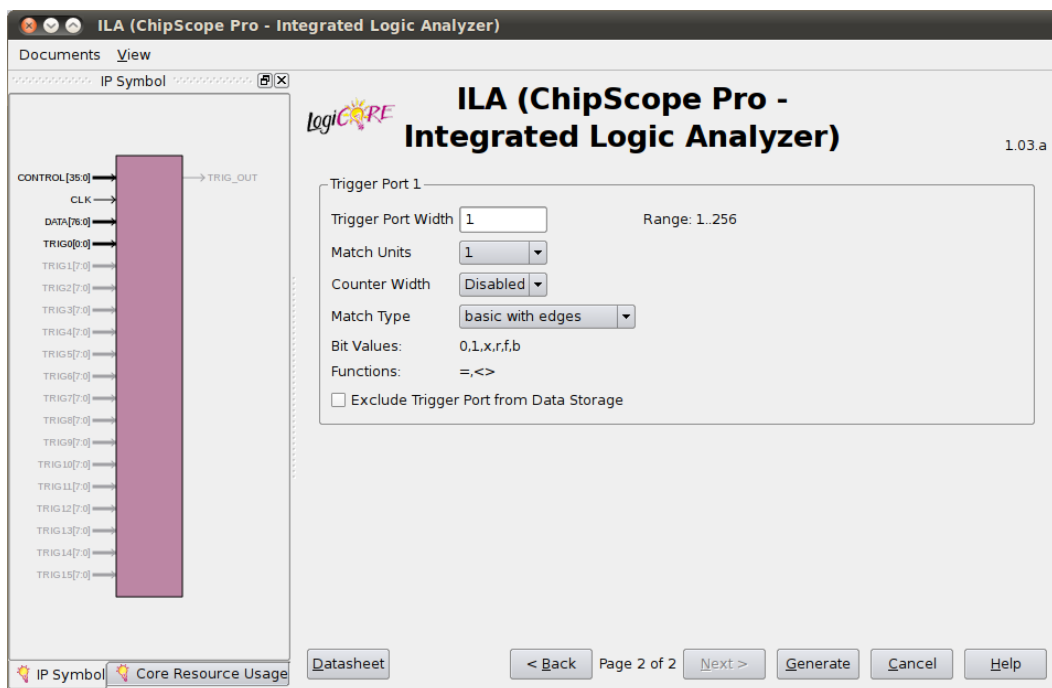


You will need to modify the options for the ILA. On the first configuration page, select 1 trigger port, 1 sequence level, uncheck all checkboxes except “Use RPMs”, sample on rising edge and set data depth to 1024.

You will need to determine the data port width yourself. This is the number of bits you capture for storage on each cycle. You will likely want to capture the control signals and some of the data signals.



After completing the first options page, click “Next”. On the second page, set the trigger port width to 1, match units to 1, counter width disabled, match type basic with edges and uncheck exclude trigger port from data storage:



Finally, click “Generate” and wait. You should now see a litany of chipscope ICON and ILA files in the chipscope directory.

3.2 Integrating ChipScope

Now that you have generated the ChipScope modules, you need to integrate them into your design. You should instantiate these in your datapath so that the data and control signals are accessible. Use the following Verilog code to instantiate the modules:

```
// ChipScope components:
wire [35:0] chipscope_control;

chipscope_icon icon(
    .CONTROL0(chipscope_control)
) /* synthesis syn_noprune=1 */;

chipscope_ila ila(
    .CONTROL(chipscope_control),
    .CLK(clk),
    .DATA({rst, addr_sel, wr_en, mem_addr, mem_dout, accum_reg}),
    .TRIG0(rst)
) /* synthesis syn_noprune=1 */;
```

The comments following the module instantiation (but before the semi-colon) are synthesis directives to indicate that the modules should not be pruned (which could happen, because they produce no externally visible outputs).

You will likely have different variable names in your datapath (and you may want to record a different subset of signals, depending on the data depth you selected during generation), so the DATA input is provided as an example only.

The trigger is the reset signal - this means that 1024 cycles of the DATA input will be stored when you press the center compass button.

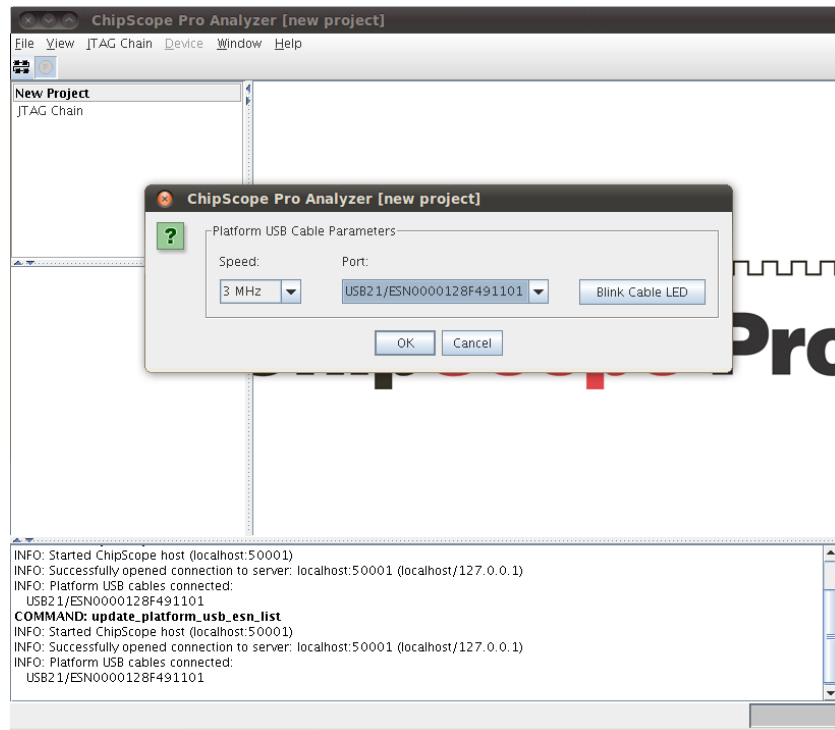
After you instantiate the ChipScope modules, you will need to run `make clean` followed by `make` from the `lab3` directory to build the design. The build time will increase to 4-5 minutes. Finally, run `make impact` to program the FPGA.

3.3 ChipScope Analyzer

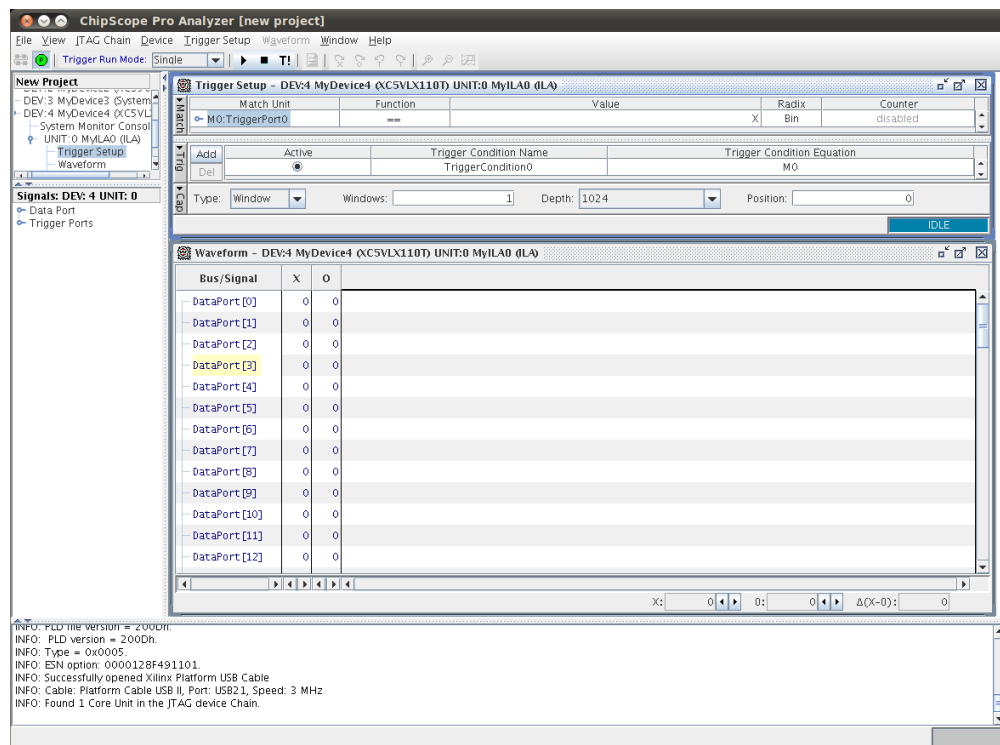
With ChipScope integrated you now have the ability to probe the internals of your design. To capture and view data, open the ChipScope Analyzer:

```
analyzer &
```

A mostly blank window with a large ChipScope Pro logo should appear. In the menu bar, select "JTAG Chain" -> "Xilinx Platform USB Cable". Leave the defaults in the two pop-up dialogs; just click "Okay" twice:

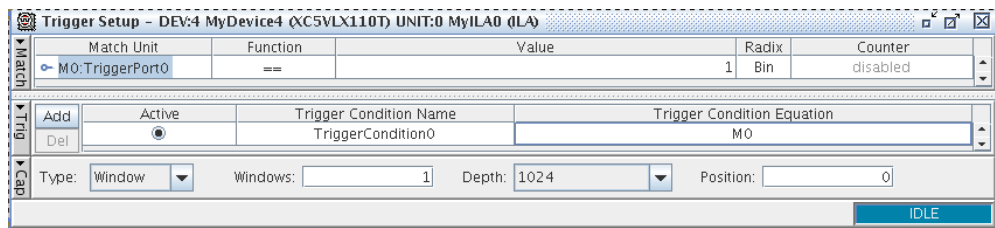


You should now see a screen similar to this:



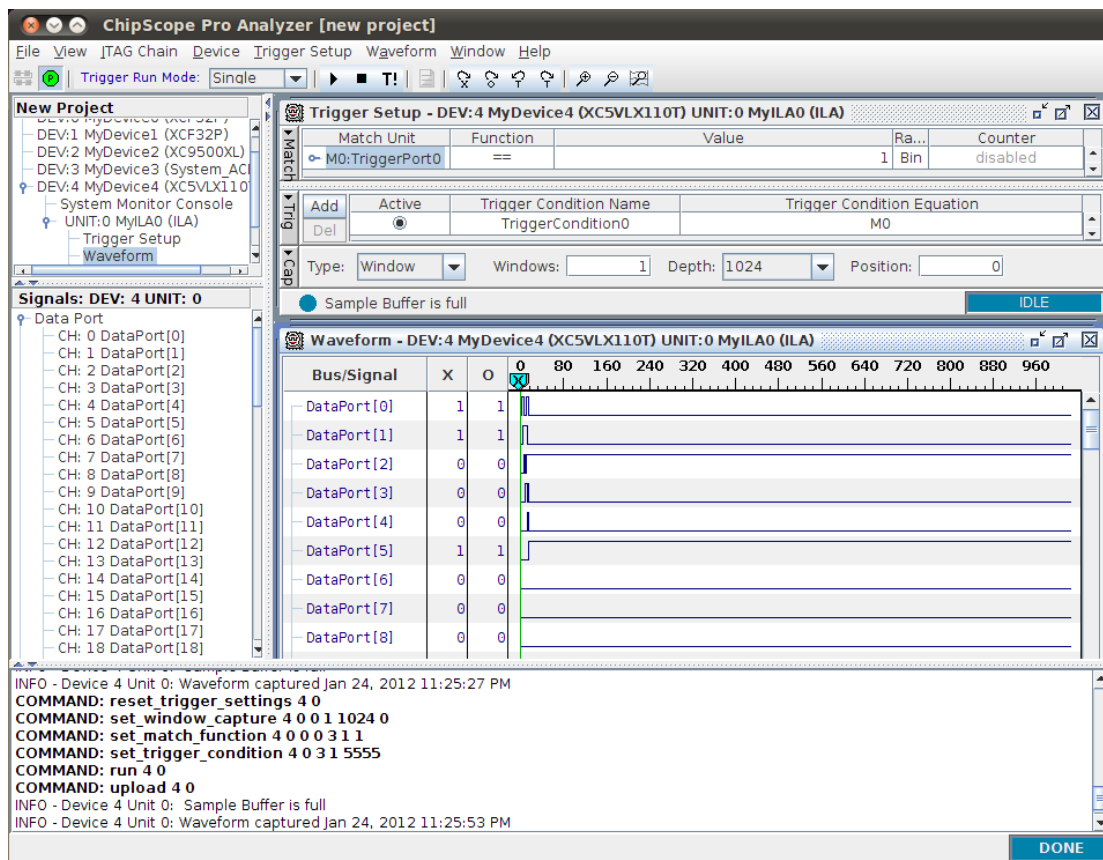
The first step is to configure the trigger. When the conditions specified in the

Trigger Setup panel are met, the ChipScope modules begin recording data. To debug your list processor, you want to record data when reset is pressed. Configure your trigger to match the following screenshot:



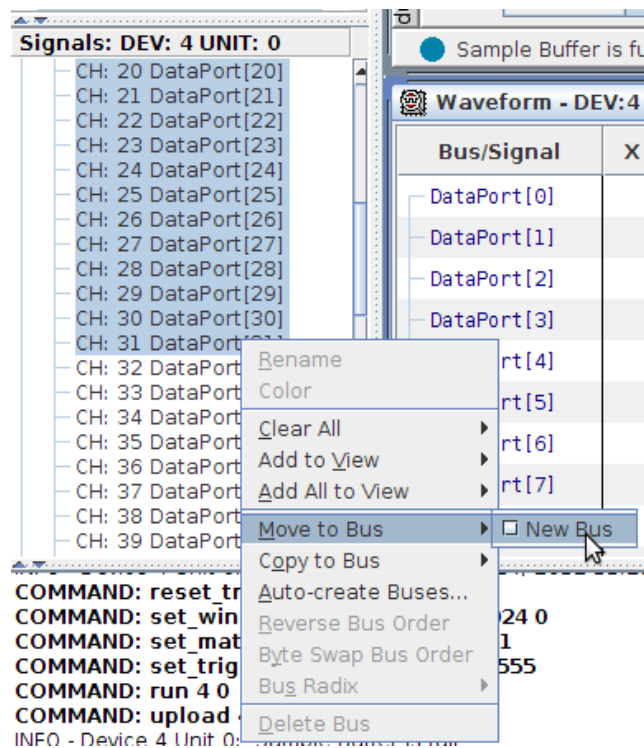
M0:TriggerPort0 corresponds to the TRIG0 port of the ILA. As such, this configures ChipScope to begin recording data when rst == 1 (i.e. you press the center compass switch).

To capture data, press F5 or click the run button on the toolbar. This arms the trigger: press the center compass switch on the board and watch as the waveform view is populated with data. After data capture, you should now see:





Unfortunately, this view is not particularly useful: there are no signal names and the data isn't grouped into buses. You will need to do this manually: select the

DataPort bits that correspond to a bus in your design and right-click them. Select “new bus”, and then rename the bus to match the signal. For example, if the first 32 bits are the block ram output, they can be grouped into a new bus:



Right click on the bus to rename it and optionally select a radix that is easier to read (e.g. hex, decimal). Organize each signal you have wired into the DATA input of the ILA so that it appears in the following manner:

| Bus/Signal | X | O | 0 | 5 | 10 | 15 | | | | | |
|--|----|----|--|---|----|----|---|----|----|----|----|
|  mem_dout | 35 | 35 |  35 | 0 | 1 | 3 | 6 | 10 | 15 | 21 | 28 |

You should now have a comprehensive view of the signals in your design. Use this to troubleshoot and debug. You should not need to close and reconfigure ChipScope when you make bug fixes; simply run `make impact` as you ordinarily would and then re-capture the data.

4 Checkoff

When you have the design working on the board, show the TA:

1. The sum of `small_list` on the GPIO LEDs
2. Named, organized waveforms in ChipScope of your list processor running `small_list`
3. Your control and datapath module source files.

Additionally, be prepared to answer the following questions:

1. What bugs did you find in your list processor and how did you use Chipscope to help you find those bugs?
2. Compared to the testing you performed with Modelsim last week, which tool took more time/effort to use?
3. When is it more appropriate to use Chipscope over testbenches/simulation?