ASIC Implementation of a RISC-V v2 Core with On-Chip Caches

CS250 Laboratory 3 (Version 092110a)
September 21, 2010
Yunsup Lee

In the second lab assignment, you wrote an RTL model of a two-stage and a three-stage pipelined RISC-V v2 processor using Verilog and synthesized your RTL model. In the third lab assignment, you will substitute magic memory with on-chip caches and will use all the tools you have learned so far to simulate, synthesize, place and route your design and finally analyze energy consumption. After producing a preliminary ASIC implementation, you will attempt to optimize your design to make it more energy efficient while meeting your time constraint. The objective of this lab is to introduce memory blocks that you might use in your final project, as well as to give you some intuition into how high-level hardware descriptions are transformed into layout.

We have provided a 16 KB instruction cache and a 16 KB data cache to use. Both caches have a 32-bit wide CPU access port and a 128-bit wide memory refill port. These caches block on a cache miss. Cache lines are 64 bytes, and both caches are initially set to be a 8-way set associative cache. If you do the math, you should be able to figure out that there are 32 sets. These numbers can be changed, since the cache is parameterizable. You will integrate these cache blocks into your three-stage RISC-V v2 core (the one with a branch target buffer), which you wrote for the second lab. After producing a working RTL model for the core, you will attempt to optimize your design to increase energy efficiency. The deliverables for this lab are (a) your optimized Verilog source and all of the scripts necessary to completely generate your ASIC implementation and analyze energy consumption checked into SVN, (b) an analytic energy model for your core implementation, and (c) written answers to the critical questions given at the end of this document. The lab assignment is due at the start of class on Monday, October 4. You must submit your written answers electronically by adding a directory titled writeup to your lab project directory (lab3/writeup). Electronic submissions must be in plain text or PDF format.

Before starting this lab, it is recommended that you revisit the Verilog model you wrote in the second lab. Take some time to clean up your code, add comments, and enforce a consistent naming scheme. You will find as you work through this lab assignment that having a more extensive module hierarchy can be very advantageous; initially we will be preserving module boundaries throughout the toolflow which means that you will be able to obtain performance and power results for each module. It will be much more difficult to gain any intuition about the performance or power of a specific assign statement or always block within a module. Thus you might want to consider breaking your design into smaller pieces. For example, if your entire ALU datapath is in one module, you might want to create separate submodules for the adder/subtractor unit, shifter unit, and the logic unit. Unfortunately, preserving the module hierarchy throughout the toolflow means that the VLSI tools will not be able to optimize across module boundaries. If you are concerned about this you can explicitly instruct the VLSI tools to flatten a portion of the module hierarchy during the synthesis process. Flattening during synthesis is a much better approach than lumping large amounts of Verilog into a single module yourself.
Block Diagram and Module Interfaces

Figure 2 shows the overall diagram. The core module contains the processor, instruction cache, data cache, and an arbiter. The processor is connected to the instruction cache and the data cache with a similar interface used in the second lab. These interfaces are 32-bits wide. Notice that there is a ready signal on the request side, and a valid signal on the response side. This is a more realistic memory interface, where you might not be able to issue a memory request every cycle because the cache is blocked on a miss (when the ready signal on the request side is deasserted), or not be able to read from memory every cycle because you missed in a cache (when the valid signal on the response side is deasserted). Also notice that the instruction memory refill port and the data memory refill port needs to arbitrate on the memory refill port going out the core. There are several arbitration policies you can implement, but in this lab you will use a fixed priority arbitration scheme, where the instruction memory refill port always gets priority over the data memory refill port. The refill port is 128-bits wide. There will be four transactions on the memory refill interface when a cache line transfer occurs, since the cache line size is 64 bytes.

Your core should be in a module named riscvCore and must have the interface shown in Figure 1. We have provided you with a test harness that will drive the inputs and check the outputs of your design. The signal log_control is used to control the logging. The value on this signal should change when you write something to the $cr10 privileged control register with the mtpcr instruction. The riscvCore module contains riscvProc, and ICache_32x4096_BC, DCache_32x4096_BC, and riscvArbiter. Interfaces for these modules follow.

```verilog
module riscvCore(
    input clk,
    input reset_ext,

    output log_control, // Log control

    input [7:0] testrig_fromhost, // Testrig fromhost port
    output [7:0] testrig_tohost, // Testrig tohost port

    output mem_req_val, // is mem request valid?
    input mem_req_rdy, // is mem ready?
    output mem_req_rw, // read or write (r=0/w=1)
    output [‘MEM_ADDR_BITS-1:0] mem_req_addr, // read/write address
    output [‘MEM_TAG_BITS-1:0] mem_req_tag, // request tag
    output [‘MEM_DATA_BITS-1:0] mem_req_data, // write data

    input mem_resp_val, // is mem response valid?
    input [‘MEM_TAG_BITS-1:0] mem_resp_tag, // response tag
    input [‘MEM_DATA_BITS-1:0] mem_resp_data // returned read data
);
```

Figure 1: Interface for RISC-V v2 Core

Interfaces for the instruction cache and data cache are omitted. Both caches have a request/response port to/from the processor and a memory refill port. Notice that the instruction cache interface is simpler compared to the data cache interface since it is a read-only memory. You can take a look at ICache_32x4096_BC.v and DCache_32x4096_BC.v for more details.
Figure 2: Block diagram for RISC-V v2 Core Test Harness
The arbiter sees both memory refill requests coming from the instruction cache and the data cache and puts its decision on the request ready signals. Notice that the mem_resp_data port is directly connected to both caches. The arbiter will only control the response valid signal. There’s no mux for the mem_req_data port, since the instruction cache is a read-only memory.

module riscvProc
(
    input clk,
    input reset,
    output log_control, // Log control
    input [7:0] testrig_fromhost, // Testrig fromhost port
    output [7:0] testrig_tohost, // Testrig tohost port (must reset to zero)
    output imemreq_val, // Inst mem port: is icache request valid?
    input imemreq_rdy, // Inst mem port: is icache ready?
    output [31:0] imemreq_bits_addr, // Inst mem port: addr to fetch
    input imemresp_val, // Inst mem port: is icache response valid?
    input [31:0] imemresp_bits_data, // Inst mem port: returned instruction
    output dmemreq_val, // Data mem port: is dcache request valid?
    input dmemreq_rdy, // Data mem port: is dcache request valid?
    output dmemreq_bits_rw, // Data mem port: read or write (r=0/w=1)
    output [31:0] dmemreq_bits_addr, // Data mem port: read/write address
    output [31:0] dmemreq_bits_data, // Data mem port: write data
    output dmemresp_val, // Data mem port: is dcache response valid?
    input [31:0] dmemresp_bits_data // Data mem port: returned read data
);

module riscvArbiter
(
    input ic_mem_req_val, // I$ refill: is request valid?
    output ic_mem_req_rdy, // I$ refill: is mem ready?
    input ["MEM_ADDR_BITS-1:0"] ic_mem_req_addr, // I$ refill: address
    output ic_mem_resp_val, // I$ refill: is response valid?
    input dc_mem_req_val, // D$ refill: is request valid?
    output dc_mem_req_rdy, // D$ refill: is mem ready?
    input ["MEM_ADDR_BITS-1:0"] dc_mem_req_rw, // D$ refill: read or write (r=0/w=1)
    input ["MEM_ADDR_BITS-1:0"] dc_mem_req_addr, // D$ refill: address
    output dc_mem_resp_val, // D$ refill: is response valid?
    output mem_req_val, // mem: is request valid?
    input mem_req_rdy, // mem: is mem ready?
    output mem_req_rw, // mem: read or write (r=0/w=1)
    output ["MEM_ADDR_BITS-1:0"] mem_req_addr, // mem: address
    output ["MEM_TAG_BITS-1:0"] mem_req_tag, // mem: request tag
    input mem_resp_val, // mem: is response valid?
    input ["MEM_TAG_BITS-1:0"] mem_resp_tag // mem: response tag
);

Figure 3: Interface for RISC-V v2 Processor and the refill port arbiter
Test Harness

We are providing a test harness to connect to your core model. The test harness loads a RISC-V binary into the memory. The provided makefile can load both assembly tests as well as C benchmarks to run on your core. The test harness will clock the simulation until it sees a non-zero value coming back on the testrig_tohost register, signifying that your core has completed a test program. The testrig_tohost port should be set to zero on reset. A very simple test program is shown in Figure 4.

```
# 0x00000000: Reset vector.
addiw $x1, $x0, 1  # Load constant 1 into register x1
mtpr $x1, $cr16    # Write x1 to tohost register
loop: beq $x0, $x0, loop # Loop forever
```

Figure 4: Simple test program

Getting Started

You can follow along through the lab yourself by typing in the commands marked with a ‘%’ symbol at the shell prompt. To cut and paste commands from this lab into your bash shell (and make sure bash ignores the ‘%’ character) just use an alias to “undefine” the ‘%’ character like this:

```
% alias %=""
```

All of the CS250 laboratory assignments should be completed on an EECS Instructional machine. Please see the course website for more information on the computing resources available for CS250 students. Once you have logged into an EECS Instructional you will need to setup the CS250 toolflow with the following commands.

```
% source ~/cs250/tools/cs250.bashrc
```

You will be using SVN to manage your CS250 laboratory assignments. Please see Tutorial 1: Using SVN to Manage Source RTL for more information on how to use SVN. Every student has their own directory in the repository which is not accessible to other students. Assuming your username is yunsup, you can checkout your personal SVN directory using the following command.

```
% svn checkout $SVNREPO/yunsup vc
```

To begin the lab you will need to make use of the lab harness located in ~cs250/lab3. The lab harness provides makefiles, scripts, and the Verilog test harness required to complete the lab. The following commands copy the lab harness into your SVN directory and adds the new project to SVN.

```
% cd vc
% mkdir lab3
% svn add lab3
% cd lab3
% LABROOT=$PWD
% mkdir v-riscv-v2-4stage
```
% cd v-riscv-v2-4stage
% mkdir trunk branches tags
% cd trunk
% LABROOT4=$PWD
% cp -R ~cs250/lab3/v-riscv-v2-4stage/* $LABROOT4
% cd $LABROOT
% svn add *
% svn commit -m "Initial checkin"
% svn update

The resulting lab3/v-riscv-v2-4stage/trunk project directory looks similar to lab 2’s working directory structure: src contains your source Verilog; build contains automated makefiles and scripts for building your design; riscv-tests contains local test assembly programs; and riscv-bmarks contains local C benchmark programs.

The src directory contains the Verilog test harness and other Verilog modules you will need in this lab assignment. The files marked with (empty) are the files you need to fill in. Copy over your RISC-V v2 implementation from lab 2.

- riscvCore.v (empty) - RISC-V core
- riscvProc.v (empty) - RISC-V processor
- riscvProcCtrl.v (empty) - Control part of the RISC-V processor
- riscvProcDpath.v (empty) - Datapath part of the RISC-V processor
- riscvArbiter.v (empty) - Arbiter
- ICache_32x4096_BC.v - Instruction cache
- DCache_32x4096_BC.v - Data cache
- defCommon.vh - Common definitions used for RTL debugging
- riscvInst.vh - RISC-V instruction definition
- riscvConst.vh - Important constant definition
- riscvTestHarness.v - Test harness

The build directory contains the following subdirectories which you will use when building your chip.

- vcs-sim-rtl - RTL simulation using Synopsys VCS
- dc-syn - Synthesis using Synopsys Design Compiler
- vcs-sim-gl-syn - Post synthesis gate-level simulation using Synopsys VCS
- icc-par - Automatic placement and routing using Synopsys IC Compiler
- vcs-sim-gl-par - Post place and route gate-level simulation using Synopsys VCS
- pt-pwr - Power analysis using Synopsys PrimeTime PX

Each subdirectory includes its own makefile and additional script files. If you end up adding more files, you will have to make modification to these script files as you push your design through the toolflow. Once you have all the tools working you can use the toplevel makefile in the build directory to run multiple tools at once. For example, once all the scripts are properly setup you should be able to use the following command to synthesize, place and route, do post place and route gate-level netlist simulation, and power analysis.
\% cd $LABROOT4/build
\% make pt-pwr

Now go ahead and copy your lab 2’s RISC-V processor implementation to the src directory. Then change your core design to cope with a more realistic memory interface. Remember you also need to implement an arbiter. Then you can use the following commands to build local assembly tests and C benchmarks, run them on the RISC-V ISA simulator, build your simulator, run assembly level tests, run benchmarks. You will need to modify the makefile so that it has a listing of all your Verilog source files.

\% cd $LABROOT4/riscv-tests
\% make
\% make run
\% cd $LABROOT4/riscv-bmarks
\% make
\% make run-host
\% make run-riscv
\% cd $LABROOT4/build/vcs-sim-rtl
\% make
\% make run-asm-tests
\% make run-bmarks-test

Guidelines for Lab Grading

Your final lab submission should pass all of the assembly tests and also be able to successfully run the globally installed benchmarks on both RTL simulation, post synthesis and post place and route gate-level netlist simulation. The most important metric for lab grading will be energy efficiency (energy/instruction). You will also develop your own analytic energy model by measuring energy consumption of assembly tests and benchmarks, and investigating instruction mixes of assembly tests and benchmarks (see end of document for lab questions).

Lab Hints and Tips

This section contains several hints and tips which should help you in completing the lab assignment.

Tip 1: Always Test Your Processor After Making Modifications

When pushing your processor through the physical toolflow, it is common to make some changes to your RTL and then evaluate their impact on area, power, and performance. Always retest your processor after making changes and before starting the physical toolflow. You can use the run-asm-tests make target to quickly verify that your processor is still functionally correct. Keep in mind, that a fast or small processor which is functionally incorrect is worse than a slow or large processor which works!
Tip 2: Deal With Synchronous Memory Reads

You have used combinational memories in the previous lab, where reads are asynchronous and writes are synchronous. That’s why you were able to connect the output of the PC register to the instruction memory, and the ALU output to the data memory. However, the memory interface used in this lab implements synchronous reads. The signals to the memory should be ready before the rising clock edge. You need to connect the output of the PC mux to the instruction cache. This is also one of the reasons to separate out a memory stage so you can get the memory address and the store data calculated before the rising edge.

Tip 3: Think Carefully About Microarchitectural Trade-offs

There are a couple ways you could incorporate caches into your processor. Figure 5 shows two possible designs. You can simply add another stage (i.e., memory stage) for memory instructions resulting in a traditional five-stage pipeline (the upper design shown in the figure). However, you will stall the pipeline waiting for a load value to come back from the cache, even though the following instructions actually are not depending on the load value. You can fix this problem by having two writeback stages, one for execute and one for memory (the design on the bottom). However, there is a price you need to pay, since you need one more write port to the register file, which costs more area. This allows you to keep executing non-memory instructions until you hit an instruction which depends on the load value. The baseline RISC-V processor, which we will provide for your projects, implements two write ports. It schedules writebacks from the long-latency functional unit and memory to the second write port, while one-cycle ALU instruction gets its own write port to the register file.

Tip 4: Register Ready (Grant) Signals on the Negative Edge

You should register ready signals \( \text{ic_mem_req_rdy, dc_mem_req_rdy} \) coming out from the arbiter on the negative edge. If you don’t register the ready signals, there is a high probability to make a combinational loop in your circuit. RTL simulation might fail in a weird way.
Tip 5: Reporting Clock Period

As discussed in the tutorials, you need to specify a clock period constraint during both synthesis and place+route. The tools will try and meet this constraint the best they can. If your constraint is too aggressive, the tools will take a very long time to finish. They may not even be able to correctly place+route your design. If your constraint is too conservative, the resulting implementation will be suboptimal.

Even if your design does not meet the clock period constraint it is still a valid piece of hardware which will operate correctly at some clock period (it is just slower than the desired clock period). If your design does not meet timing the tools will report a negative slack. Similarly, a design which makes the timing constraint but does so with a positive slack can run faster than the constrained clock period. For this lab we are more concerned about the effective clock period of your design as opposed to the clock constraint you set before synthesis. The effective clock period is simply the clock period constraint minus the worst slack ($T_{clk} - T_{slack}$). The synthesis and place+route timing reports are all sorted by slack so that the path with the worst slack is listed first. To determine the effective clock period for your design simply choose the smaller of the rising and falling edge slacks. Figure 6 illustrates two examples: one with positive slack and one with negative slack. In this example, our clock period constraint is 10 ns. In Figure 6(b), the post-place+route reports indicate a positive slack value of 1.5 ns and thus the effective clock period is 8.5 ns. In Figure 6(c), the post-place+route reports indicate a negative slack value of 4.75 ns and thus the effective clock period is 14.75 ns. Notice that the effective clock period in Figure 6(c) is not equal to the worst case combinational critical path (i.e. 13.5 ns). This is because we must also factor in setup time and clock skew.

Determining the effective clock period when your design does not meet timing can be more tricky when we include latches (as opposed to flip-flops) in our design. Latches impose additional half-cycle constraints. For this lab it is adequate to ignore any half-cycle constraints and focus instead on the full cycle constraints.

Figure 6: Determining your hardware’s effective clock period
Critical Thinking Questions

The primary deliverable for this lab assignment is your optimized Verilog source and all of the scripts necessary to completely generate your ASIC implementation checked into SVN. In addition, you should prepare written answers to the following questions and turn them in electronically.

Question 1: Draw a Block Diagram

Tell us how your riscvCore works in detail. How many pipeline stages did you use? Draw a block diagram and identify on the system diagram which components you placed in your datapath and which components you placed in your control logic.

Question 2: Take a Screenshot

Take a screenshot of your core and include it in your writeup. You should highlight the register file and the ALU with different colors. Point the caches from your design. Do you think pre-placement of the cache blocks would help minimize core area?

Question 3: Evaluate Your Baseline Core

Push your baseline processor through the physical toolflow and report the following numbers. (Summarize! don’t just copy and paste.)

- Post-synthesis area of the register file, processor’s datapath (excluding register file), processor’s control unit, and cache blocks in $um^2$ from *.mapped.area.rpt
- Post-synthesis total area of core in $um^2$ from *.mapped.area.rpt
- Post-synthesis critical path and corresponding effective clock period in nanoseconds from *.mapped.qor.rpt
- Post-place+route area of the register file, processor’s datapath (excluding register file), processor’s control unit, and cache blocks in $um^2$ from IC Compiler *.output.area.rpt
- Post-place+route total area of the core in $um^2$ from IC Compiler *.output.area.rpt
- Post-place+route critical path and corresponding effective clock period in nanoseconds from IC Compiler *.output.qor.rpt

Your post-place+route numbers will probably be worse than your post-synthesis numbers. Explain why the place+route tool is reporting more area and a longer clock period than the synthesis tool.

Question 4: Analytic Energy Model for your RISC-V v2 Core

For this question you are to calculate energy/instruction for assembly tests and benchmarks. Fill in the following table by counting individual instructions (instruction mix) and measuring power and performance. You should write a Python script which collects data from your build directories.

If you define the instruction mix as a matrix $A$, the energy/instruction numbers as $x$, and the consumed energy column as $b$ then $Ax = b$ holds. Since you know $A$ and $b$ by filling up the
<table>
<thead>
<tr>
<th>Program</th>
<th>Average Power</th>
<th># of addiw</th>
<th># of addw</th>
<th>...</th>
<th># of xor</th>
<th>Target Time</th>
<th>Consumed Energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>riscv-v1_simple.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v1_addiw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v1_bne.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v1_lw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v1_sw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_addiw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_addw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_andi.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_and.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_beq.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_ble.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_bleu.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_blt.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_bltu.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_bne.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_j.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_jal.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_jalr.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_jalr_r.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_jalr_j.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_lui.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_lw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_nor.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_ori.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_or.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_simple.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_slti.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_sltiu.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_sl.t.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_sl.tu.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_sllw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_sllw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_sraiw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_sraw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_srlw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_srlw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_subw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_sw.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_xori.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>riscv-v2_xor.S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>median</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>qsort</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>towers</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vvadd</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>multiply</td>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
table, you can guess what $x$ should look like. Use MATLAB to run a regression to calculate $x$ (energy/instruction). Ideally after this calculation you have your own power/energy model that you can plug into your simulator. Now you don’t need to go through all the VLSI tools to get energy numbers out!

Provide answers to the following questions.

- List your energy/instruction numbers.

- Is there a natural grouping of energy/instruction numbers? Explain the grouping discipline. What instruction/group has the lowest energy/instruction? The highest energy/instruction? Why?

- How accurate is your model? Calculate the predicted energy consumption of benchmark programs. Compare this number to the measured energy consumption. What’s the error?

- Identify the instruction mix of the benchmark program you wrote for lab 2. Run it through your energy model and compare the predicted energy consumption to the actual measured energy consumption. What’s the error?

- Is the model accurate? How could you improve your energy model?

**Question 4: Provide us Feedback**

Please provide us feedback about the labs, tutorials, and experience using the VLSI tools.

**Read me before you commit!**

- For this lab, you don’t need to commit any build results. We will build the design from your Verilog source files.
- If you have added Verilog sources codes or changed the name of the Verilog source codes, however, you **must** commit the makefiles you’ve changed.
- We will checkout the stuff you committed to the trunk and use that for lab grading. Feel free to take advantage of branches and tags. We will also checkout the version which was committed just before 10:30am on Monday, October 4. Use your late days wisely! If you have used your late days, please email TA.
- To summarize, your SVN tree for lab3 should look like the following:

  /yunsup
  /lab3: COMMIT PYTHON SCRIPT
  /writeup: COMMIT REPORT
  /v-riscv-v2-4stage
  /trunk
   /src: COMMIT CODE
   /build
    /dc-syn: commit if changed
    /icc-par: commit if changed
    /pt-pwr: commit if changed
    /vcs-sim-gl-par: commit if changed
Acknowledgements

Many people have contributed to versions of this lab over the years. The lab was originally developed for 6.375 Complex Digital Systems course at Massachusetts Institute of Technology by Christopher Batten. Contributors include: Krste Asanović, John Lazzaro, Yunsup Lee, and John Wawrzynek. Versions of this lab have been used in the following courses:

- CS250 VLSI Systems Design (2009-2010) - University of California at Berkeley
- CSE291 Manycore System Design (2009) - University of California at San Diego