Write and Synthesize a Two-Stage RISC-V-v2 Processor

CS250 Laboratory 2 (Version 082511)
Written by Yunsup Lee (2010)
Updated by Brian Zimmer (2011)

Overview

For the second lab assignment, you will write an RTL model of a two-stage pipelined RISC-V-v2 (a subset of the RISC-V instruction set) processor using Chisel and synthesize your RTL model. After producing a working RTL model, you will attempt to optimize your design to increase performance and/or decrease area. The instruction set allows for both a 32 bit or 64 bit datapath, but as a 64 bit version will take more time to run through the tools, we will be building a 32 bit version.

Deliverables

This lab is due **Monday, September 26th at 1pm**. Deliverables for this lab are:

- (a) your optimized and verified Chisel source and all of the scripts necessary to completely synthesize your RTL implementation checked into Git
- (b) one assembly test program and one C benchmark program to test your implementation
- (c) written answers to the questions given at the end of this document checked into git as `writeup/report.pdf` or `writeup/report.txt`

You are encouraged to discuss your design with others in the class, but you must turn in your own work. The two-stage pipeline should perform instruction fetch in the first stage, while the second pipeline stage should do everything else including data memory access.

If you need to refresh your memory about pipelining and the MIPS instruction set, we recommend *Computer Organization and Design: The Hardware/Software Interface*, Fourth Edition, by Patterson and Hennessey.

A mandatory prerequisite is to read about the RISC-V architecture, which can be found in the *RISC-V Instruction Set Manual* on the website. In this lab, you will be implementing a subset of this instruction set, but to understand what each instruction does, you will need to read the relevant sections of this manual.

For more information about using Synopsys VCS for Verilog simulation consult *Tutorial 4: Simulating Verilog RTL using Synopsys VCS*. To learn more about Synopsys Design Compiler for synthesis please refer to *Tutorial 5: RTL-to-Gates Synthesis using Synopsys Design Compiler*. Detailed information about building, running, and writing RISC-V assembly and C codes could be found in *Tutorial 3: Build, Run, and Write RISC-V Programs*.

Make sure to separate out datapath and memory components from control circuitry. The system diagram in Figure 3 can be used as an initial template for your two-stage RISC-V-v2 processor

implementation, but please treat it as a suggestion. Your objective in this lab is to implement the RISC-V-v2 ISA, not to implement the system diagram so feel free to add new control signals, merge modules, or make any other modifications to the system. You will need to turn in a diagram of your datapath anyway, so it is highly recommended that you draw your datapath from the beginning in a program such as Omnigraffle or Visio, and keep it updated as you design. This reference will be very useful to speed up debugging.

Processor Interface

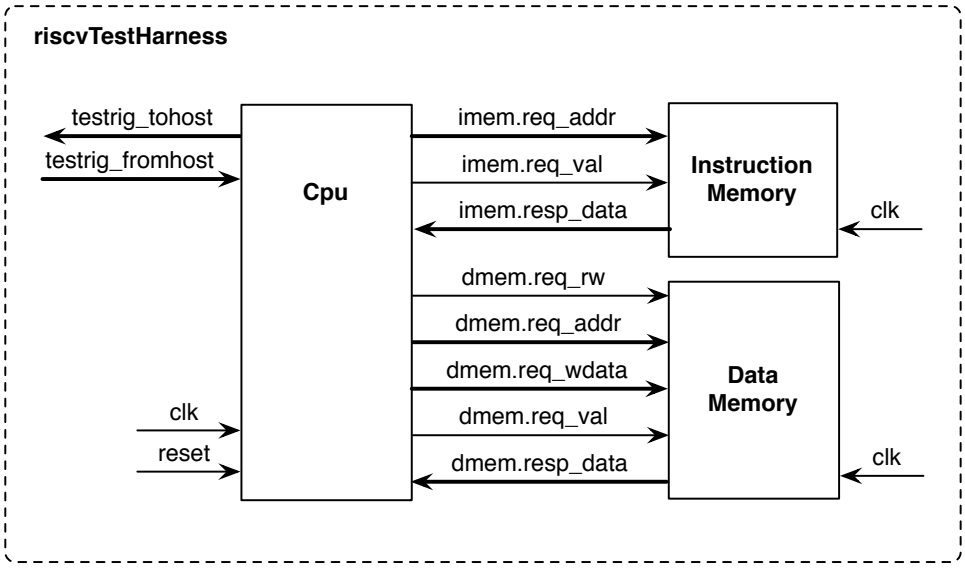

Figure 1: Block diagram for RISC-V-v2 Processor Test Harness

We have given you a 1-stage processor which already has a working interface to a test harness and memories. The test harness will drive the inputs and check the outputs of your design for a number of tests written in assembly language. These tests are targeted to verify correct functionality of every instruction. The test harness includes the data and instruction memories. We have provided separate instruction and data memory ports to simplify the construction of the pipeline, but both ports access the same memory space. The memory ports can only access 32-bit words, and so the lowest two bits of the addresses are ignored (i.e., only `imemreq_bits_addr[31:2]` and `dmemreq_bits_addr[31:2]` are significant). To make an instruction memory request, set `imemreq_bits_addr` to the fetch address and set `imemreq_val` to one. The data will be returned combinationally (i.e. there are no clock edges between when a request is made and when the response returns). To make a data memory request set `dmemreq_bits_rw` to zero for a load or one for a store, set `dmemreq_bits_addr` to the address, set `dmemreq_bits_data` to the store data if needed, and finally set `dmemreq_val` to one. The data will be returned combinationally for loads, while for stores the data will be written at the end of the current clock cycle. Notice that the data write bus is a separate unidirectional bus from the data read bus. Bidirectional tri-state buses are usually avoided on chip in ASIC designs.

Test Harness

There are two test harnesses: one for the Chisel-generated emulator, and another for the Chisel-generated Verilog. You should design your processor such that it passes the emulator testbench first. Then, when you are ready to synthesize your design, you should run the Verilog testbench to ensure that there are no bugs in the Chisel verilog generator.

We are providing a test harness to connect to your processor model. The test harness is identical to the one described in *Tutorial 4: Simulating Verilog RTL using Synopsys VCS* and *Tutorial 5: RTL-to-Gates Synthesis using Synopsys Design Compiler*. 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 processor. The test harness will clock the simulation until it sees a non-zero value coming back on the `testrig_tohost` register, signifying that your processor 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 2.

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

Figure 2: Simple test program

Implemented Instructions

RISC-V ISA is a research ISA we use in Berkeley. This ISA defines 32-bit, 64-bit operations, including single and double precision floating point operations, as well as supervisor operations. Consult the *RISC-V Processor Specification* for more details about the RISC-V architecture. You may also want to read *Tutorial 3: Build, Run, and Write RISC-V Programs*. For this lab assignment, you will only be implementing the RISC-V-v2 subset. Figure **??** shows the 33 instructions which make up the RISC-V-v2 subset.

You do not need to support any exceptions or interrupt handling (apart from reset). The only pieces of the privileged control registers you have to implement are the `tohost` and `fromhost` registers, and the MTPCR and MFPCR instructions that access these registers. These registers are used to communicate with the test harness. The test harness drives `testrig_fromhost`, while you should implement an 8-bit register which drives the `testrig_tohost` port of the `riscvProc` module interface.

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 and you have setup your account according to the setup instructions given on the class website. Also, this lab guide assumes that you have already completed lab 1. If you have not completed lab 1, run the following commands, assuming that your username is `yunsup`.

```
% cd /scratch/
% mkdir yunsup
% cd yunsup
% git init
% git remote add template git@github.com:ucberkeley-cs250/lab-templates.git
% git remote add origin git@github.com:ucberkeley-cs250/yunsup.git
```

You will be using Git to manage your CS250 laboratory assignments. Please see *Tutorial 1: Using Git to Manage Source RTL* for more information on how to use Git. Every student has their own private repository which is not accessible to other students. This lab assumes that you have finished Lab 1.

To begin the lab you will need to make use of the lab harness. The lab harness provides makefiles, scripts, and the emulator/Verilog test harness required to complete the lab. Assuming your username is `yunsup`, follow these steps to download the materials for this lab:

```
% cd /scratch/yunsup
% git pull template master
% git push origin master
% cd lab2
% LABROOT=$PWD
```

The resulting `lab2/` project directory contains the following primary subdirectories: `src/` contains your source Chisel; `vlsi/csrc/` contains the Direct C source files to simulate memory, parse and load ELF files; `vlsi/lib/` and /textttvlsi/include/ contain files needed to disassemble instruciton codes for debugging purposes; `vlsi/build/` contains automated makefiles and scripts for building your design; `vlsi/riscv-tests/` contains local test assembly programs; and `vlsi/riscv-bmarks/` contains local C benchmark programs. The `src/` directory contains various RISC-V instruction constants you may find helpful in `instructions.scala`.

The `src` directory contains the Chisel files that describe a simple 1 stage RISC-V processor that implements only four instructions.

- `consts.scala` - Bit enumerations for control signals
- `cpu.scala` - Wrapper that holds control and datapath
- `ctrl.scala` - Control part of the RISC-V processor
- `dpath.scala` - Datapath part of the RISC-V processor
- `instructions.scala` - RISC-V instruction definition
- `memories.scala` - "Magic" instruction and data memory
- `top.scala` - Used by Chisel to instantiate design

Chisel Implementation

When you design your processor, you will do so by writing Chisel code in the `src/` directory, compiling an emulator, then running assembly tests on the emulator. Only once everything works here will you try to push your design through the flow.

To compile the emulator, run the following instructions until you see a [passed] message:

```
% cd $LABROOT
% make emulator
```

Once this works, you will attempt to run RISC-V assembly tests on your processor to check for correct operation. To do this, run the following instructions:

```
% cd $LABROOT
% cd emulator
% make run-asm-tests
% make run-bmarks-test
```

To start, every `riscv-v1*` test should pass, but the rest will not because you have not implemented them yet. Please read the ISA spec and look at the provided system diagram in order to attempt to implement the instruction. If it still doesn't work, carefully read through your code and think of any possible corner cases you missed. As a last step, you will need to do real debugging. We have setup two mechanisms to allow efficient debugging. First, you can use a waveform viewer to see all signals in your file using the following command (change the test name `simple` to the instruciton of your choice):

```
% dve -vpd riscv-v2_simple.vpd &
```

More information about using `dve` can be found in the tutorials.

Another useful way to debug is to compare the log of the operations of your processor with those of the "golden reference" (the ISA simulator).

```
% cd $LABROOT
% cd emulator
% make run-asm-tests
% make run-asm-correct
% vimdiff riscv-v2_simple.out riscv-v2_simple.correct.out
```

`run-asm-tests` will generate a riscv-v2_inst.out file that provides the debugging information defined in `testbench/emulator.cpp`. Feel free to add other signals to this file by defining additional printf commands. All signals that are available for you are listed in `generated-src/Top.h` and are named similar to (but not exactly the same) as the signal names in your Chisel source code. By default, very important lines are included (what is being written to and read from the register file every cycle). Other signals, such as the address and data of the data memory, might be very useful to view when you try to implement more complicated instructions.

`run-asm-correct` will generate a riscv-v2_inst.out.correct file that provides the result of this assembly test being run on the guaranteed-correct ISA simulator. For a single-stage implementation,

the output of both of these files should be exactly the same. For multiple stages, the output will be different (even for correct operation) as branch mispredictions will cause cycles to be killed, but the overall flow of the program will remain the same. When debugging a multi-stage processor, you can also print out the signal that determines whether or not an instruction will be killed, then ignore these lines while running a `diff` or `vimdiff` on these files.

Verilog Implementation

In order to synthesize your design, it must described in Verilog. Chisel will generate Verilog for you, but a separate testbench is needed to ensure that the Chisel generator worked correctly. The generated Verilog is output to `vlsi/generated-src/`.

The `vlsi/build` directory contains the following subdirectories which you will use when synthesizing your design.

- `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

The `vlsi/testbench/` contains a verilog testbench that instantiates magic memories, then loads and runs programs on the processor.

To make sure that the Chisel-generated Verilog still works, rerun all of the assembly tests using the following commands:

```
% cd $LABROOT/vlsi/build
% cd vcs-sim-rtl
% make run
```

Each subdirectory includes its own makefile and additional script files.

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, and do post synthesis gate-level netlist simulation.

```
% cd $LABROOT2/build
% make vcs-sim-gl-syn
```

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 in verification mode, and run benchmarks in performance mode.

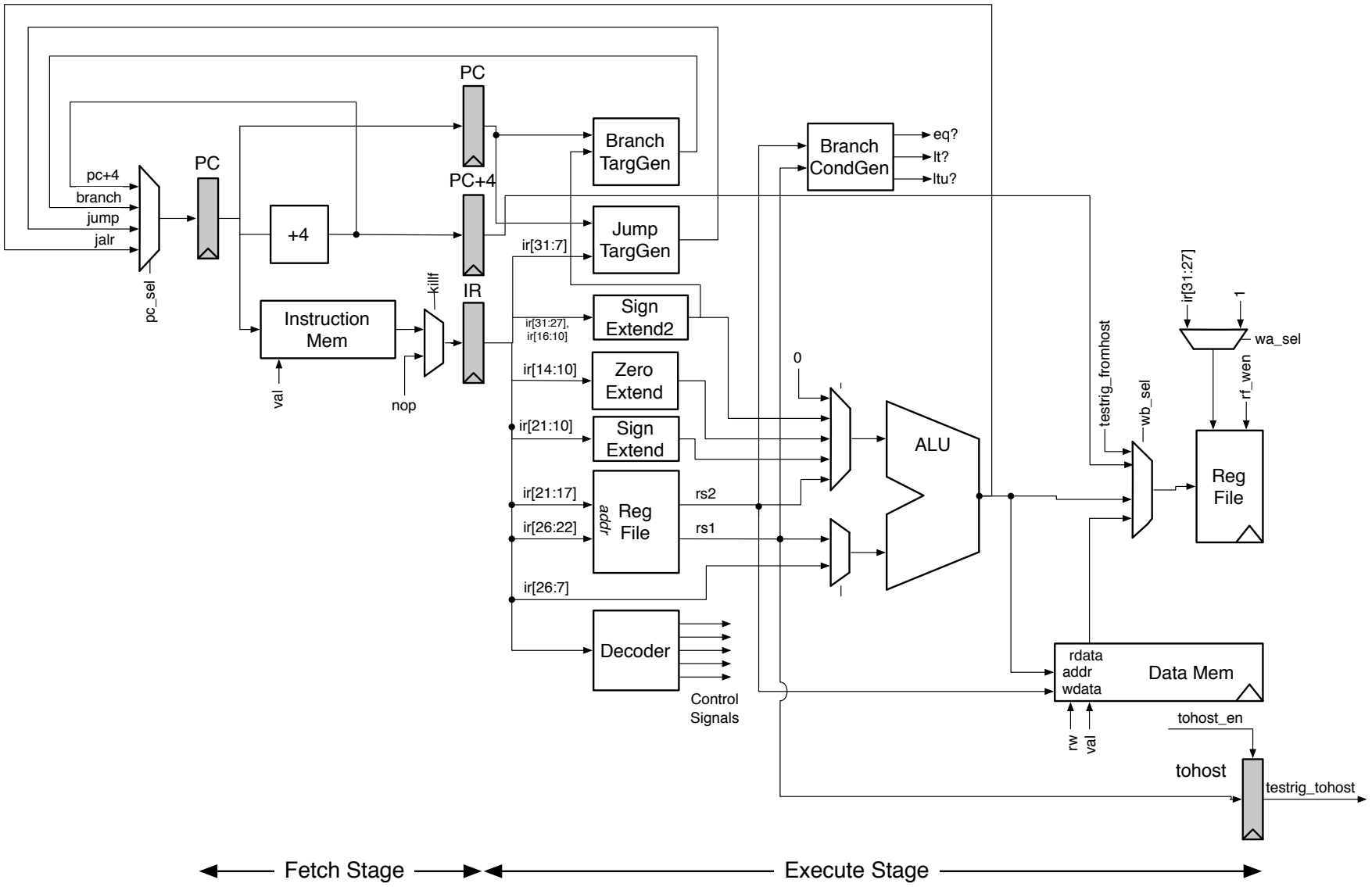

Figure 3: Two-Stage Pipeline for RISC-V-v2 Processor. Shaded state elements need to be correctly loaded on reset.

31 27	26 22	21 17	16 15	14 12	11 10	9 8	7 6 0	
jump target							opcode	J-type
rd	LUI-immediate						opcode	LUI-type
rd	rs1	imm[11:7]	imm[6:0]			funct3	opcode	I-type
imm[11:7]	rs1	rs2	imm[6:0]			funct3	opcode	B-type
rd	rs1	rs2	funct10				opcode	R-type
rd	rs1	rs2	rs3		funct5		opcode	R4-type

Control Transfer Instructions

imm25					1100111	J imm25
imm25					1101111	JAL imm25
imm12hi	rs1	rs2	imm12lo	000	1100011	BEQ rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	001	1100011	BNE rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	100	1100011	BLT rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	101	1100011	BGE rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	110	1100011	BLTU rs1,rs2,imm12
imm12hi	rs1	rs2	imm12lo	111	1100011	BGEU rs1,rs2,imm12
rd	rs1	imm12		000	1101011	JALR.C rd,rs1,imm12
rd	rs1	imm12		001	1101011	JALR.R rd,rs1,imm12
rd	rs1	imm12		010	1101011	JALR.J rd,rs1,imm12

Memory Instructions

rd	rs1	imm12		010	0000011	LW rd,rs1,imm12
imm12hi	rs1	rs2	imm12lo	010	0100011	SW rs1,rs2,imm12

Integer Compute Instructions

rd	rs1	imm12		000	0010011	ADDI rd,rs1,imm12
rd	rs1	000000	shamt	001	0010011	SLLI rd,rs1,shamt
rd	rs1	imm12		010	0010011	SLTI rd,rs1,imm12
rd	rs1	imm12		011	0010011	SLTIU rd,rs1,imm12
rd	rs1	imm12		100	0010011	XORI rd,rs1,imm12
rd	rs1	000000	shamt	101	0010011	SRLI rd,rs1,shamt
rd	rs1	imm12		110	0010011	ORI rd,rs1,imm12
rd	rs1	imm12		111	0010011	ANDI rd,rs1,imm12
rd	rs1	rs2	0000000	000	0110011	ADD rd,rs1,rs2
rd	rs1	rs2	1000000	000	0110011	SUB rd,rs1,rs2
rd	rs1	rs2	0000000	001	0110011	SLL rd,rs1,rs2
rd	rs1	rs2	0000000	010	0110011	SLT rd,rs1,rs2
rd	rs1	rs2	0000000	011	0110011	SLTU rd,rs1,rs2
rd	rs1	rs2	0000000	100	0110011	XOR rd,rs1,rs2
rd	rs1	rs2	0000000	101	0110011	SRL rd,rs1,rs2
rd	rs1	rs2	0000000	110	0110011	OR rd,rs1,rs2
rd	rs1	rs2	0000000	111	0110011	AND rd,rs1,rs2
rd	imm20				0110111	LUI rd,imm20

Table 1: Instruction listing for RISC-V

Add a Simple Branch Predictor

Congratulations! Now you have a working two-stage pipeline. Have you looked at your IPC (Instructions Per Cycle) numbers? The current two-stage RISC-V-v2 design incurs a one-cycle penalty for every taken branch. Obviously, the IPC of your design cannot exceed one. It can, however, be significantly less than one due to the single cycle taken branch delay penalty. Use the following commands to measure the IPC of your processor. You can find IPC as well as other statistics in the corresponding `*.pref.out` file. Note, this will not work until `make run-bmarks-test` passes. Also, you will need to modify the `emulator/testbench/emulator.cpp` file to only increment `inst_count` when the instruction is not killed.

```
% cd $LABROOT/emulator
% make run-bmarks-perf
```

Your next task is to add a simple branch predictor to your original RISC-V-v2 processor. For more information on branch predictors please consult *Computer Organization and Design: The Hardware/Software Interface*, by Patterson and Hennessey or the CS252 course lecture slides. The TA would also be happy to chat with you about branch predictors.

Make your changes in such a way that you can build your processor both with and without the branch predictor. We have provided the necessary scripts to enable this functionality. It is important to learn how to use Chisel as a "generator" which generates different code based on different input options as we will be using Chisel to explore design spaces in the projects. To understand how we can generate different code, we will step through the entire generation process. First, open the file that controls this.

```
% cd $LABROOT
% vim gen-design-space.py
```

Now that we want to generate multiple design outputs, we can no longer send them to this same base directory. This script will create a "symbolic tree" directory for each design space that perfectly mirrors the top level directory structure, but instead of copying files, it makes symbolic links. This way you can generate multiple design points, and if you make a fix to the source file, all of these directories are kept up to date. The design spaces are set up by adding entries to the `design_spaces` variable, and the name of each entry determines the folder name. There is also code that will delete these design point directories for you. Note: Unless you add or remove files from the original build directory, you do not need to run this script to generate these directories again. For example, if you modify the source code or the testbench, no additional effort is needed. You can run this script with the following command:

```
% make ds-setup
```

Try browsing into these directories. The only difference is the options.mk file. The options set in the `design_spaces` variable get sent here, then are forwarded to chisel when any `make` command is run.

In `src/top.scala` you can see a hook that sets the global variable `isBranchPrediction` to true if the `-bp` option is sent through. You can set additional options by adding additional lines here. This flag variable is set in `src/consts.scala` which needs to be imported to every file that the variable is used in.

Now inside your chisel code, you can use `if/else` structures to generate different code for your version with and without branch prediction. The following code snippets should be useful as you parameterize your code:

Include an additional component:

```
var btb:DpathBTB = null;
if(isBranchPrediction){
btb = new DpathBTB;
}
```

Now you can access these signals elsewhere in the file:

```
if(isBranchPrediction){
  btb.io.current_pc4 := id_reg_pc;
  if_predicted_target := btp.io.predicted_target;
  // Note: CANNOT use val inside an if structure.
  // eg. val if_predicted_target =
  // won't work
} else {
  // Normal case
}
```

You will probably also need to conditionally communicate between components. Define new bundles that are specific to your additional logic, then include them and conditionally connect them.

```
// In dpath.scala
val ctrl_bp = new IoCtrlToDpathBP().flip();
...
...
// In cpu.scala
if(isBranchPrediction){
  c.io.ctrl_bp  <> d.io.ctrl_bp;
  c.io.dpath_bp  <> d.io.dpath_bp;
}
```

You are now going to sketch a simple predictor. We recommend that you start by implementing this simple predictor. If you have time and are interested, you are welcome to improve the predictor.

You will be implementing a simple direct-mapped branch-target-buffer (BTB). You will *not* be predicting JALR instructions. These instructions should be handled exactly as they are in the baseline design: squash the fetch stage when taken. Figure 4 shows the modified system diagram. The predictor should contain a small (four entry) table. Each entry in the table is a <pc+4,target> pair. You will also need some form of valid bits so that you can correctly handle uninitialized table entries. You will create the table using the `Mem` construct in the same way that the register file is created in the sample 1-stage processor The operation of the predictor is described below.

In the fetch stage, the predictor uses the low order bits of PC+4 to index into the table. You use PC+4 instead of the PC because it makes the pipelining simpler. The predictor should read out the corresponding <pc+4,target> pair from the table, and if the PC's match then this is a hit. You then choose the proper `pc_sel` signal so that the predicted target gets clocked into the PC on

the next cycle. If the PC's do not match, then this is a miss and you use PC+4 as the next PC. Our simple predictor uses the following invariant: if a PC is in the table then it is always predicted taken, but if a pc is not in the table then we always predict not-taken. Entries are never removed from the table they are only overwritten. Since you are not predicting JALR, we know that the target address is *always* correct even if your taken/non-taken prediction is incorrect. You do not need to verify the target address, only the taken/not-taken prediction.

You pipeline the predictor hit/miss signal to the execute stage. Because of the invariant mentioned above, this hit/miss bit also tells you if the branch was predicted taken or not-taken. In the execute stage, the predictor should compare the predicted taken/non-taken bit to the calculated taken/not-taken bit. This is how the predictor can determine if there is a misprediction. There are four possible scenarios shown in the following table.

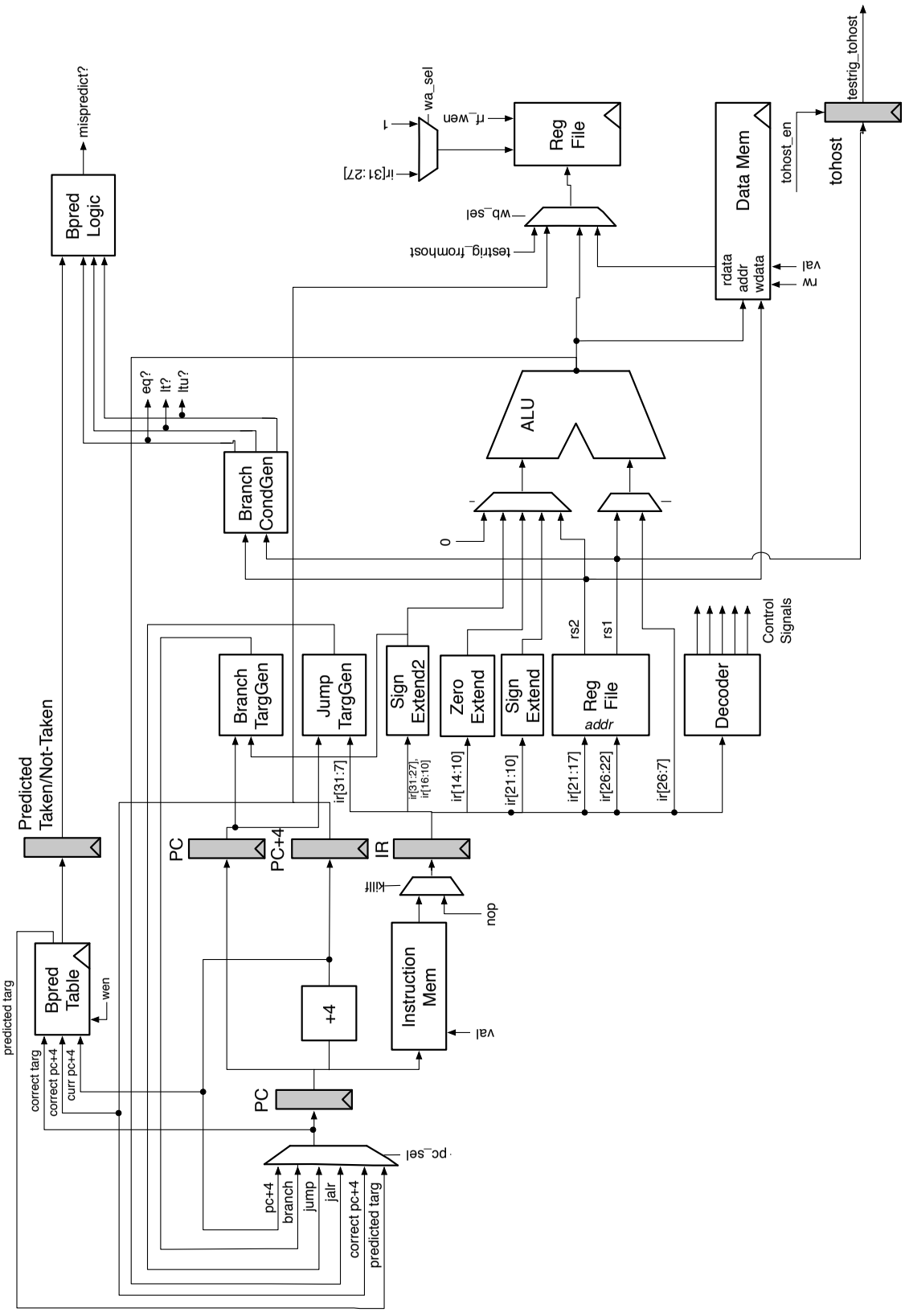

Figure 4: Two-Stage Pipeline for RISC-V-v2 Processor with Branch Predictor.

Predicted	Actual	Mispredict?	Action to take
taken	taken	no	no action required
not-taken	taken	yes	kill instr in fetch, update table, pc := branch or jump targ
taken	non-taken	yes	kill instr in fetch, do not update table, pc := correct pc+4
not-taken	not-taken	no	no action required

If the branch was predicted not-taken, and it was actually taken (i.e. we enter a loop), then you update the table by adding the appropriate PC+4 and branch or jump target. This corresponds to writing the `correct targ` and `correct pc+4` signals shown in Figure 4. If the branch was predicted taken and it was actually not-taken (i.e. we fall out of a loop), then you do *not* update the table. You could invalidate the appropriate entry in the table, but to make things simpler you just leave the table unchanged.

There are several subtle issues to be aware of when implementing the predictor. The most important is to carefully think about the situation when there are back-to-back branches. For example, what happens if the execute stage identifies a misprediction, but at the same time there is a branch in the fetch stage which is being predicted taken?

If you are feeling particularly ambitious there are several ways to improve on this simple design including: adding support for JALR prediction, using a set-associative table instead of a direct mapped table, increasing the size of the table, or adding some hysteresis to the table (i.e. it takes more than one taken branch before you predict taken).

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 and gate-level netlist simulation. When you first start working on your processor it will not pass the multiply benchmark. This is because the multiply benchmark is incomplete. You will finish writing the benchmark in question 4 of this lab assignment (see end of the document for the lab questions). You need to submit one assembly test program and one C benchmark program (not counting the multiply benchmark), and we will run your programs on others processor. Metrics for lab grading will be post synthesis area, post synthesis critical path clock period, and IPC measurements on benchmarks. Notice these metrics are not independent. If you make one metric better, it is likely that a different metric will get worse. Try to find the best design point.

Lab Hints and Tips

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

Tip 1: Use Incremental Development

We suggest taking an incremental approach when implementing the two-stage RISC-V-v2 processor. Start by implementing just a few instructions and verify that they are working before slowly adding more instructions. First create a two-stage RISC-V v1 processor and verifying that it can pass the five RISC-V v1 tests. The simplest approach for handling branches is for the fetch stage to always "predict" not-taken. Branches and jumps resolve in the execute stage; thus if the branch or jump

was actually taken then you simply set the PC multiplexer appropriately and kill the instruction currently in the fetch stage. You can kill an instruction by inserting a NOP into the instruction register. Once you have a two-stage RISC-V v1 processor working, begin to refine the system into the two-stage RISC-V v2 system show in Figure 3 (for example you might want to create a separate branch address generator).

Now you can gradually add additional instructions and attempt to incrementally pass more of the assembly test suite. Your system should be able to complete all `riscv-v2_` tests. We strongly discourage implementing the entire system and all instructions before trying to pass any tests. A more incremental approach will greatly reduce your verification time.

Tip 2: Handle Reset Carefully

It is very important to plan how you will handle reset. Figure 3 highlights those state elements which you should probably reset. The *RISC-V Processor Specification* states that the `tohost` register must be reset to zero. Ask yourself what should the PC and the IR be reset to?

Tip 3: Use Text Tracing Wisely

We recommend adding some text tracing to your `riscvTestHarness` such that the value of various nets in your processor are displayed each cycle (one cycle per line). See the `v-riscv-v1-1stage` code for an example of text tracing. Make sure that you annotate on the text trace what cycle each signal belongs to. For example, in the 1-stage example the disassembler prints the instruction (eg. `addi $x1, $x0, 2`) based on the instruction bits coming from the instruction memory. But in a multi-stage processor, this should come from the instruction stored in a later pipeline register.

There is a signal called `instruction` under the `Testbench` module that you can add to your signals to show the disassembled instruction. You will need to change the radix to ASCII.

A common debugging technique is to first try running a test program on your processor. From examining the text trace you should be able to get a good feel for how your processor is executing. If you need to do more detailed debugging, then start Discovery Visualization Environment (DVE) and use the waveform viewer to trace more signals. You can start this with the command:

```
% cd $LABROOT/emulator
% make run-asm-tests
% dve -vpd testname.vpd &
```

See *Tutorial 4* for more on using DVE.

Tip 4: Run Tests Individually

Although the makefile contains convenient targets for running all of the assembly tests at once, when you are initially debugging your processor you will want to get a simple test working before trying all of the tests. The following commands will build the simulator, and then run the test on the simulator. After running the test you can open the `riscv-v1_simple.out` file to examine the text trace output. Each assembly test is a self-checking test. If the test passes, it will write one to the `tohost` register and the test harness will stop and print `*** PASSED ***`. If the test fails, it will write a value greater than one to the `tohost` register and the test harness will stop and print

`*** FAILED ***`. For RISC-V v2 tests, the value in the `tohost` register corresponds to which test case in the assembly test failed. Consult the appropriate RISC-V v2 assembly test file to learn more about the specific test case.

```
% cd $LABROOT2/build/vcs-sim-rtl
% make riscv-v1_simple.out
```

Tip 5: Use the RISC-V Assembly and Objdump Files for Debugging

When debugging your processor, you can view the .corrected.out files to see correct operation. If you would like to learn more, you should consult both the RISC-V assembly file as well as the objdump file. The assembly test files are installed globally at `~/cs250/install/riscv-tests`. Parallel to the `riscv-v1_simple` binary you will also see a `riscv-v1_simiple.dump` file. The assembly code that generates these binaries are in `~/cs250/fa11/riscv-tests`. The objdump file shows the exact instructions (and their addresses) which makes up the corresponding assembly test. By examining which PC your processor is executing and correlating this to the objdump and assembly file you should be able to figure out what your processor is doing.

Tip 6: Use the RISC-V v2 ISA Simulator as a Reference

When you make a new assembly test program or a new C benchmark program, run it on the ISA simulator first to confirm that it is functionally correct. For more information consult *Tutorial 3*.

Critical Thinking Questions

The primary deliverable for this lab assignment is your optimized Verilog RTL for the two-stage/three-stage RISC-V v2 processor with/without the branch target buffer. In addition, you should prepare written answers to the following questions and turn them in electronically.

Question 1: Design Partitioning

Tell us how your two-stage `riscvProc` works in detail. Clearly identify on the system diagram shown in Figure 3 which components you placed in your datapath and which components you placed in your control logic. Also clearly highlight all of your control signals including any additional signals which you may have added.

Question 2: Optimize Your ALU

Use the post-synthesis `*.mapped.area.rpt` and `*.mapped.resources.rpt` reports to identify how your ALU is being synthesized. How many Synopsys DesignWare components are being inferred? Ideally we should be able to implement the RISC-V v2 ALU with just an adder, a left shifter, and a logic unit. The logic unit would contain bit-wise and, or, and xor. Optimize your ALU for area and push it through the physical toolflow. Report the change in post-synthesis area and performance. Don't forget to retest your design with your optimized ALU!

Question 3: Analyzing a Simple RISC-V Benchmark

For this question you will first write a small RISC-V assembly routine, and then evaluate ISA changes which would affect the performance of that routine. The RISC-V v2 ISA does not include a multiply instruction, yet you can emulate multiplication using shifts and adds. The following pseudo code illustrates a straightforward algorithm for software multiplication.

```
function multiply ( op1, op2 ) {
  r1 := op1
  r2 := op2
  r3 := 0

  for ( i = 0; i < 32; i++ ) {
    if ( ( r1 & 0x1 ) == 1 )
      r3 := r3 + r2
    r1 := r1 >> 1
    r2 := r2 << 1
  }

  return r3
}
```

Your first task is to write a software multiplication routine in RISC-V assembly. We have provided you with a C test program and RISC-V assembly template. You can find a C version of the multiply routine in `riscv-bmarks/multiply/multiply.c`. Modify `riscv-bmarks/multiply/multiply_asm.S` by adding your RISC-V code where indicated. A useful reference is `~/cs250/fa11/riscv-bmarks/median/` as this benchmark uses a similar assembly file embedded in a C program. You can then build and test your multiply benchmark using the following commands.

```
% cd $LABROOT2/riscv-bmarks
% make
% make run-riscv
% cd $LABROOT2/build/vcs-sim-rtl
% make
% make multiply.riscv.out
```

Once your multiply routine is working you can evaluate the performance (measured in IPC) with the following commands.

```
% make multiply.riscv.perf.out
```

The resulting output will show some statistics including the IPC. In your answer to this question, report the IPC of this benchmark. You will need to modify the `emulator/testbench/emulator.cpp` file to only increment `inst_count` when the instruction is not killed. The multiply routine has two branches: the outer for loop branch and the inner if statement branch. For which branch will your predictor generate good predictions? For which branch will the predictor generate poor predictions?

A RISC-V assembly programmer approaches you and suggests that you add the following conditional add instruction to the RISC-V ISA. The syntax and semantics for this new instruction are shown below.

`addw.c x3, x2, x1` if ((x1 & 0x1) == 1) x3 := x3 + x2

Would this new instruction help? Why? What changes would you need to make to your RISC-V v2 datapath in order to support this new instruction? Do you think these changes would impact the cycle time or area of your design?

Question 4: New Assembly Test Program and C Benchmark Program

Tell us about the new assembly test and new C benchmark program you committed. What are you trying to test? How do they work? Which corner cases are covered by the test?

Question 5: Evaluate your Processor

Push your processors through synthesis (with the best clock period you can) and report the following numbers by filling up the table. Also measure IPCs of all benchmarks on all processors, and see how it changes with the trade-offs you have made for every processor. What trade-offs have you learned? You should be able to run a Python script which collects the following data. Commit the script to the repository.

- Post-synthesis area of the register file, datapath (excluding register file), and control unit in um^2 from `*.mapped.area.rpt`
- Post-synthesis total area of processor in um^2 from `*.mapped.area.rpt`
- Post-synthesis critical path and corresponding effective clock period in nanoseconds from `*.mapped.timing.rpt`
- Post-synthesis power estimates from `*.mapped.power.rpt`
- Post-synthesis cell count from `*.mapped.reference.rpt`
- Post-synthesis Design Ware block usage from `*.mapped.resources.rpt`

Post Synthesis	Unit	2-stage w/o BTB	2-stage w/ BTB
Area (Regfile)	um^2		
Area (Dpath)	um^2		
Area (Control)	um^2		
Area (Total)	um^2		
Critical Path Clock Period	ns		
Power Estimates	mW		
Cell Count			
DesignWare Block Usage			

IPC of	2-stage w/o BTB	2-stage w/ BTB
median		
qsort		
towers		
vvadd		
multiply		
your benchmark		

Read me before you commit!

- For this lab, you don't need to commit any build results for VCS or Design Compiler. We will build the design from your Chisel source files.
- However, if you changed any of the Makefiles or build scripts, you must commit these changes.

Acknowledgements

Many people have contributed to versions of this lab over the years. The lab was originally developed for CS250 VLSI Systems Design course at University of California at Berkeley by Yunsup Lee. Contributors include: Krste Asanović, Christopher Batten, John Lazzaro, Yunsup Lee, Brian Zimmer, Chris Terman, 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
- 6.371 Intrdouction to VLSI Systems (2002) - Massachusetts Institute of Technology
- 6.375 Complex Digital Systems (2005-2009) - Massachusetts Institute of Technology
- CSE291 Manycore System Design (2009) - University of California at San Diego