CS 505 Computer Structures, Fall 2017
Lab 2: Pipelined PARCv2 Processor
(adapted from ECE 4750 at Cornell University)
(adapted from ECE 475 at Princeton University)

For this lab, we will work with various implementations of the **PARC ISA**, which is based on the MIPS32 ISA. A full description of the ISA is provided in the lab tarball in *parc-isa.txt*. **PARCv1** implements the simplest subset of instructions that can run a set of assembly tests. The **PARCv2** ISA includes the full suite of immediate, arithmetic, subword memory, and branch instructions that are required to run a C program without any system calls. Both versions are described in full detail in the ISA description document provided in the lab tarball.

In this lab, your task is to take a pipelined, 5-stage PARCv1 processor with stalling, and expand it into a pipelined, 5-stage PARCv2 processor with bypassing. The reference processor provided will have a completed PARCv2 *datapath*, but the *control unit* will only support PARCv1 instructions. You will need to upgrade the control unit to support the PARCv2 instructions using pipelined controls, as well as add bypassing capabilities to the processor

Pipelining allow us to achieve higher performance than the multicycle *mcparc* implementation (which executes one pipeline stage per cycle without pipelining) by lowering the CPI without significantly affecting the cycle time. This is achieved by allowing multiple instructions to occupy the processor simultaneously, instead of only allowing one at a time. The tradeoff is that now we must be careful about the data and control hazards we discussed in class, which complicates the control logic. The reference processor uses stalling and squashing to avoid these hazards, but you will be adding an additional feature, bypassing, to improve performance even further. Of course, there are tradeoffs with this technique as well, which you will discuss in the report.

1. Getting Started

First, download the lab tarball from **sakai**. Then execute the following commands (assuming that you working directory for the class is *cs505*):

```
cd cs505
tar xzf cs505-lab2.tar.gz
cd cs505-lab2
export LAB2_ROOT=$PWD
```

If you are working on **ilab**, also make sure to source the correct environment for the lab:

```
source /ilab/users/class/cs505/toolchain/env.sh
```

There will be eight directories in the lab root. The purpose of each directory is outlined below:

- *build*: Build directory for Verilog source code
- *mcparc*: Multicycle **PARCv1** processor source code
- *pv2stall*: Pipelined **PARCv1** processor with stalling source code
- *pv2byp*: Pipelined **PARCv2** processor with bypassing source code
- *pv2long*: Pipelined **PARCv2** processor with bypassing and pipelined *muldiv unit* source code
- *tests*: Assembly test build system
  - *parcv1*: **PARCv1** assembly tests
  - *parcv2*: **PARCv2** assembly tests
  - *scripts*: miscellaneous scripts for build system
- *ubmark*: Benchmarks for evaluation
- *vc*: Additional Verilog components

The *tests* directory contains its own build system to compile and convert assembly tests into a Verilog memory hex (*.vmh*) file, which can be used to initialize the processor's memory. The *parcv1* and *parcv2* subdirectories contain the assembly tests that test each instruction in the corresponding ISA. **It is very important that we test each new instruction we implement for**

the lab with its own assembly test to make sure that it is correct before moving onto the next instruction. Please see the Testing Methodology section for more information. In particular, before running assembly tests make sure to compile them as described in Compiling New Assembly Tests section.

pv2stall, *pv2byp*, and *pv2long* contain the source code for the pipelined PARC processor we will be building in this lab. *pv2stall* will be used for the processor with just stalling, whereas *pv2byp* will be used for the processor with additional bypassing. *pv2long* adds a multicycle, pipelined *muldiv* unit.

We will begin by compiling the multicycle reference processor and running the PARCv1 and PARCv2 assembly tests:

```
cd $LAB2_ROOT/build
make
make check
make check-asm-pv2stall
make check-asm-rand-pv2stall
make check-asm-pv2byp
make check-asm-rand-pv2byp
make check-asm-pv2long
make check-asm-rand-pv2long
```

You need to compile asm tests (see Testing Methodology) running the check-asm- commands. Running *make* will compile our processor simulators. The simulators **without *-rand-*** use a synchronous memory with a fixed 1-cycle response time, whereas the ones **with *-rand-*** use a synchronous memory with random delays. Take a look at the *Makefile* by opening it in your text editor of choice. You will see a section titled *List of Assembly Tests*, and under it will be the *tests* variable defined with a list of assembly tests that we want to run. You will need to add any other assembly tests you want to run to the end of this list. Running `make check-asm-pv2stall` will use the simulator without any random delays. You can also run the simulator with random delays by running `make check-asm-rand-pv2stall`. Notice that this command uses the *pv2stall-randdelay-sim* instead of the *pv2stall-sim simulator*.

You can also run a single assembly separately by invoking the simulator directly. For example, if we want to run the assembly test for *parcv1-addiu* on its own:

```
cd $LAB2_ROOT/build
./pv2stall-sim +exe=../tests/build/vmh/parcv1-addiu.vmh +stats=1
```

You can choose to display statistical information about the simulation by using the `+stats=1` option. By default, this option is turned off.

Running a simulation with the `+vcd=1` command line argument will always produce a *.vcd* file with waveform data for the simulation that you can view with ***gtkwave***. Be aware that running simulations in succession **will overwrite the *.vcd* files**, so if you want to hold onto a VCD dump, you should rename the file to something more unique.

The *Makefile* uses the *pv2stall-sim* and *pv2stall-randdelay-sim* binaries to run the assembly tests for `make check-asm-pv2stall` and `make check-asm-rand-pv2stall`, respectively. The *pv2stall* simulators use the stall-based processor. Similarly, the *pv2byp-* counterparts use the bypass-based processor. The *pv2byp* simulators use the same stall-based processor as the *pv2stall* simulators out of the box, since the source code is the same. After you implement bypassing in the lab, you will be able to run both the stall and bypass versions of the processor. Notice that the multicycle reference processor simulators also compile when we run `make`, but these will not be used for automatic testing.

Running `make check-asm-*`, `make check-asm-rand-*`, or `make run-bmark-*` will generate a *vcd* dump for each assembly test that runs with the corresponding test name. For instance, the *vcd* dump for the *parcv1-addiu.vmh* test will be named *parcv1-addiu.vcd*. The *vcd* dumps for the random delay memory simulations will have the *-rand* suffix. So for *parcv1-addiu.vmh* running on *parc-randdelay-sim*, the *vcd* dump will be named *parcv1-addiu-rand.vcd*. **All vcd dumps will be placed in the same directory as the tests – this can be either $LAB2_ROOT/tests/build/vmh, or $LAB2_ROOT/ubmark/build/vmh.** Running the automatic tests for the *pv2stall*, *pv2byp*, and *pv2long* counterparts back-to-back will overwrite the vcd dumps.

2. Pipelined 5-Stage PARCv2 Processor with Bypassing

For this lab, you will be provided a complete **PARCv2 *datapath* with stalling only** and a **PARCv1 control unit**. The *datapath* uses the reference solution iterative *muldiv unit*, feel free to replace the imuldiv source files with your own solution.

There are 3 primary objectives for this lab:

- Add PARCv2 instructions to control unit
- Add bypassing to *datapath* and control uniti
- Integrate a pipelined *muldiv unit*

You should complete **objective 1** in the **pv2stall** directory. Once you are done, copy over your source files to the *pv2byp* directory and continue with **objective 2** in the *pv2byp* directory. Be sure to rename all your files to have the *pv2byp-* prefix as well as change any *`include* file names to have the same prefix. Then, complete **objective 3** in the *pv2long* directory, again changing filenames and paths as necessary. It is important that you have access to both the completed stalling processor and bypassing processor for evaluation.

2.A Objective 1: Upgrading the Control Unit to PARCv2

For the first objective, **you will not need to change anything else besides the control unit**, *parc-CoreCtrl.v*. All the signals you will need from the *datapath* are already available as inputs to the control unit, and all the control signals you need to set are already declared as outputs. **You will need to add entries to the control output table for each PARCv2 instruction**. All the columns representing the control signals you will need are already defined in the table – you

will not need to add any more. You are free to add any additional helper signals to implement the instructions, but you will likely only need to for branch instructions. **Be sure to study the localparams for the control signals carefully, as you will need to use them to fill in columns for new instruction entries in the table.**

The pipelined 5-stage PARC processor is composed of the 5 stages we learned about in class: (**F**)etch, (**D**)ecode, E(**X**)ecute, (**M**)emory, and (**W**)riteback. You will notice in the reference code that nearly all of the signals in the *datapath* and control are suffixed with **_Fhl**, **_Dhl**, **_Xhl**, **_Mhl**, or **_Whl**. As you might guess, the first letter represents the stage in which the signal is used – this helps us categorize signals into appropriate stages as well as allow us to quickly catch bugs, such as in cases when a **_Fhl** control is being used to drive hardware in the **X** stage. The second and third letters represent for how long after the clock edge the signal will be valid. The *'h'* stands for a high clock level, and the *'l'* stands for a low clock level. So *'hl'* would mean the signal is valid for exactly one cycle after the clock edge. We will only be using flip-flop-based design in our labs so you will only ever use the *'hl'* suffix.

A list of **PARCv1** and **PARCv2** instructions are replicated below for your convenience, please refer to the PARC ISA document *parc-isa.txt* for details.

PARCv1: simplest subset in order to run assembly tests

- Register-Immediate Arithmetic Instruction
  - *addiu, ori, lui*
- Register-Register Arithmetic Instructions
  - *addu*
- Memory Instructions
  - *lw, sw*
- Jump Instructions
  - *jal, jr*
- Branch Instructions
  - *bne*
- MulDiv Instructions
  - *mtc0*

PARCv2: simplest subset in order to run raw C code (no syscalls)

- Register-Immediate Arithmetic Instructions
  - *andi, xori, sll, srl, sra, slti, sltiu*
- Register-Register Arithmetic Instructions
  - *subu, slt, sltu, sllv, srlv, srav, and, or, xor, nor*
- Memory Instructions
  - *lb, lbu, lh, lhu, sb, sh*
- Jump Instructions
  - *j, jalr*
- Branch Instructions
  - *beq, blez, bgtz, bltz, bgez*
- MulDiv Instructions

- *mul*, *div*, *divu*, *rem*, *remu*

The control unit we are using for this lab differs significantly from the multicycle reference processor. Instead of using an FSM, we pipeline control signals and output them to the *datapath* at the stages they will be used. This allows us to have a more compact control output table with each row representing the set of control signals that need to be pipelined for a given instruction, rather than a state. A brief summary of each control signal is shown below in Table 1. These signals correspond to the controls driving the *datapath* in Figure 1. **Be sure you understand the interaction between the controls and the *datapath* before adding new instructions**.

INST_VAL	signifies a valid instruction, used for assertions
J_EN	is there a jump that can be determined in decode? (i.e. *jal, jr*)
BR_SEL	type of branch instruction, used to determine which branch conditions need to be checked to see if branch is taken
PC_SEL	PC mux select
OP0_SEL	operand 0 mux select
RS_EN	is operand 0 being read from the regfile? Used to determine when to stall or bypass
OP1_SEL	operand 1 mux select
RT_EN	is operand 1 being read from the regfile? Used to determine when to stall or bypass
ALU_FN	ALU function
MULDIV_FN	muldiv unit function
MULDIV_EN	muldiv request valid
MULDIV_SEL	muldiv response mux select, used to choose lower or upper 32-bits of result
EX_SEL	execute stage output mux select, used to choose between ALU output or muldiv output
MEM_REQ	type of memory request
MEM_LEN	length of memory request in bytes – use 0 for word, 1 for byte, and 2 for halfword
MEM_SEL	memory response mux select, used to choose unsigned or signed subword memory responses, if necessary
WB_SEL	writeback mux select, used to choose execute stage output or memory response
RF_WEN	regfile write enable
RF_WADDR	regfile write address
CP0_WEN	cp0 write enable

Table 1

All of these signals are defined in the **D** stage, but are pipelined until they get to the stage where they are used. For example, the *alu_fn_Dhl* signal is set in Decode, but is pipelined until the Execute stage, as *alu_fn_Xhl*, which is what drives the ALU in the datapath.

Instruction encodings, instruction fields, and control signal fields are all defined in a new instruction message type header, *parc-InstMsg.v*. These parameters are declared as `*defines*, which are global, so we use a unique prefix to avoid namespace conflicts. Such macros all have the *PARC_INST_MSG_* prefix.

2.B Example: Adding the LH Instruction

This section will walk you through adding an instruction to the reference processor. In this example, we will implement the *LH* instruction of the **PARCv2** ISA.

Since the *datapath* already implements all of the **PARCv2** instructions, we only need to modify the *control unit* to add a new instruction. As with any new instruction we are going to add, we want to examine the *datapath* and see how the instruction will flow down the pipeline. For a subword memop, like *LB*, we can take the *LW* instruction as an example. A *LW* instruction uses the regfile read data for operand 0 and a sign-extended immediate for operand 1, then uses the ALU to add these operands to calculate the memory address. After making a memory request in **X** and receiving a valid response in **M**, it proceeds to write the response to the regfile write address in the **W** stage. The *LH* instruction is nearly identical to *LW*, except that we only want to read a halfword and sign extend the response. Taking a look at the control output table entry for *`PARC_INST_MSG_LW*, we see that we only need to modify the memory request length column and the memory response mux select column. Specifically, we want to set the length to be 2, which is defined in a **localparam** as *ml_h*, and set the mux select to choose the sign extended halfword, which is defined as *dmm_h*. We can add the new row for *`PARC_INST_MSG_LH* as follows:

```
`PARC_INST_MSG_LH : cs={ y, n, br_none, pm_p, am_rdat, y, bm_si, n, alu_add,
                         md_x, n, mdm_x, em_x, ld, ml_h, dmm_h, wm_mem, y, rt, n };
```

The row is broken up into multiple lines for the handout, but make sure that this entry is all on one line in the code. That's all we need to add for this instruction, as simple as that.

Let's add the assembly test for *LH* by navigating to the lab build directory and adding the test to the list in the `Makefile`. Under *"List of Assembly Tests"*, add the name of the test for *LH*:

```
tests = \
parcv1-addiu.vmh \
Parcv1-ori.vmh \
parcv1-lui.vmh \
parcv1-addu.vmh \
parcv1-lw.vmh \
parcv1-sw.vmh \
parcv1-bne.vmh \
parcv1-jal.vmh \
parcv1-jr.vmh \
parcv2-lh.vmh \ <-- This is our new test
```

Now we can run `make check-asm-pv2stall` and it should run and pass the test for *LH*. Most instructions you will be implementing will only require another entry in the control output table like we did here. Just examine the *datapath* and figure out how each control signal needs to be set for the given instruction.

2.C Objective 2: Adding Bypassing

After you have added the **PARCv2** instructions to the control unit in the *pv2stall* directory, copy over your source files to the *pv2byp* directory. You will continue with adding bypassing in the *pv2byp* directory. You can copy over your source files and rename them easily by executing the following:

```
cd $LAB2_ROOT/pv2byp
cp ../pv2stall/* .
for filename in *; do mv $filename ${filename/pv2stall/pv2byp}; done
```

You will still need to change the prefixes of any files that are referenced as *`includes* inside the source files to *pv2byp*. It is especially important that you search and replace all instances of *pv2stall* to *pv2byp* in *pv2byp.mk*.

To add bypassing to the processor, you will need to add bypass muxes to the *datapath*. Examine Figure 1 and determine where the muxes would have to be placed, as well as where the values would need to be bypassed from. **Draw an updated datapath diagram, you will need to submit this with your report.**

You will need to add additional controls for the bypass mux select in the control unit. Bypassing is another way of avoiding data hazards by forwarding values that need to be read from the regfile in the **D**ecode stage before they are actually written back. **The processor should be fully-bypassed, meaning it should be able to forward values from the X, M, and W stages**. It will probably be helpful to have separate helper signals that represent whether a value can and should be bypassed from each of these stages, for both operands. For example, a signal called *rs_X_byp_Dhl* could be used to represent if a value for register `rs` needs to be forwarded from the **X** stage. The helper signals should be used to determine the bypass mux selects.

Implementing bypassing does not mean we can completely get rid of stalling. We still need to stall for load-use hazards because the data memory response needs to come back before we can bypass it. You will need to modify the stall signal to stall just for load-use hazards, instead of for any data hazard.

For the load-use stall signal, you will need a new pipelined signal that signifies whether or not an instruction was a load. Call this signal, *is_load*, with the appropriate suffixes for each stage.

2.D Other Architectural Differences

The following information is ancillary to the objectives of the lab, but some may find it interesting.

The bubble bits for each stage in the pipeline are used to mark when an instruction in that stage is invalid. The bubble bit for the next stage is set when the current stage is stalling or squashed and the next stage is not stalling. This allows us to avoid sending *nops* down the pipeline whenever we need to insert a bubble due to stalls or squashes, which expends more energy since we usually need to change the values of many pipeline registers. Instead, a single bit is sent down the line to effectively invalidate instructions while the pipeline registers retain their value.

You might have noticed additional bypass registers attached to the response side of both the instruction and data memories. These are essentially 1-deep queues that are used to decouple the processor from the memory, which is a technique known as **skid-buffering**. Because we are using more realistic synchronous test memories with the *val/rdy* interface, there will be cases in which we send out a valid memory request before we know the processor is going to stall. When a valid memory response comes back, but the processor is stalled, we will lose the response forever unless we store it somewhere. This is because we always set the memory response ready to high, to prevent the processor from stalling the memory network, causing the response to be valid for only one cycle. After that, there is no guarantee that the value of the response is what we are expecting. The queue allows us to store that response until the processor stops stalling so we do not miss the response.

2.E Objective 3: Integrating a Pipelined *Muldiv* Unit

The reference implementation of the processor uses the iterative *muldiv* unit from lab 1 and stalls in **D** and **X** to wait for the result. Clearly this is not the most efficient method of integrating the *muldiv* unit. One alternative is to integrate a pipelined *muldiv* unit that is decoupled from the processor, allowing it to run in parallel, only stalling when there is a dependency.

Provided for you in the *pv2long* directory is a functional model of a 4-stage, pipelined *muldiv* unit called *pv2long-CoreDpathPipeMulDiv.v*. This is not an actual pipelined *muldiv* unit, but rather is a simplified representation of one. The actual computation is performed using functional operators (i.e. *, /, %) in the first stage, and 3 dummy stages are used to pipeline the result. Assume that you cannot bypass the result from inside the *muldiv* unit. The goal is to integrate this pipelined *muldiv* unit into the processor and add the necessary logic to allow the processor to run in parallel with it while still handling data hazards correctly.

When you are finished with the *pv2byp* processor, copy your source code from *pv2byp* into *pv2long* and rename where appropriate, just as was done when starting the *pv2byp* processor. Replace the *imuldiv_IntMulDivIterative muldiv* implementation with the provided pipelined *muldiv* unit, and then implement and test any new stall and bypass signals as needed.

As a hint, the simplest solution would be to simply extend the entire pipeline by two stages. The first two stages of the pipelined *muldiv* unit can overlap with the existing X and M stages, while the last two stages need to be inserted before the **W** stage. **Pay attention to the interface of the pipeline and the multiplier: the multiplier should receive values from the pipeline right after the (D)ecode stage instead of the e(X)ecute stage. Also the stall signals from pipeline need to be hooked up correctly to the multiplier**. Once the stages are built, add all of the resulting new forwarding, stalling, and bypassing logic needed. Certainly, there are more efficient ways to implement a multicycle *muldiv* unit without extending the entire **X** stage to multiple cycles. However, this simpler solution is acceptable for this lab.

3. Testing Methodology

For this lab, all **PARCv1** and **PARCv2** assembly tests are provided. You will need to add at least one custom assembly test that focus tests any bugs you encountered during the lab, or specific bypass paths in your completed processor. You may use the macros in *parc-macros.h*, but it is also fine to write raw assembly. **In the report, discuss why you designed your test the way you did and how it proves that your processor correctly implements bypassing or that the bug has been fixed.**

All assembly tests can be found in the `$LAB2_ROOT/tests` directory under the appropriate directory for each PARC ISA version. For instance, all **PARCv1** assembly tests are in the `$LAB2_ROOT/tests/parcv1` directory. Notice that all assembly tests in each directory are prefaced with the same name as the directory they reside in. For example, all assembly tests in `$LAB2_ROOT/tests/parcv1` start with *parcv1-[instruction name].s*. Any tests that you add to the existing suite should also follow the same conventions for it to compile correctly. You will also need to add any new tests to the appropriate *.mk* file, in addition to the *tests* variable in the `Makefile` in `$LAB2_ROOT/build`.

Try opening up one of the assembly tests. You will notice that most of the tests are written using macros defined in the *parc-macros.h* header file. This file contains useful macros that will make writing clean assembly tests much easiser than having to write out the MIPS assembly code by hand every time. Let's walk through the assembly test for *addiu* as an example. Navigate to `$LAB2_ROOT/tests/parcv1` and open up *parcv1-addiu.s* in a text editor.

The very first piece of code is an include statement for *parc-macros.h*. Right after this is a macro called **`TEST_PARC_BEGIN** which configures the **.text** section of the assembly code. Every assembly test needs these two statements at the beginning of the file. Every assembly test also needs a **`TEST_PARC_END** macro at the very end of the file, which contains the code for pass and fail routines.

The body of the code is all written using macros with the **`TEST_IMM** prefix. This is the set of macros that are designed to test immediate instructions such as *addiu*, *ori*, *lui*, etc. Let's look at **`TEST_IMM_OP** macro. This is the most basic test macro for this macro set. The syntax is:

```
`TEST_IMM_OP(instruction, source value, imm. value, expected result)
```

We can see the first test checks to see if `0+0=0`, the next checks if `1+1=2`, and so on. You can take a look inside the *parc-macros.h* file to see exactly how these macros are implemented in assembly. In general, though, we use the *mtc0* instruction to write a special coprocessor **0** status register whenever we want to send information about a test to the simulator.

More specifically, we write a **1** when a test passes or write the line number at which the test fails otherwise. The simulator polls the status register until it detects a non-zero value and displays information about the test based on whether the test passed or failed.

Some other common macro patterns are the **SRC0_EQ_X** macros, which test matching source and/or destination registers, and the **BYP** macros, which test for correct bypassing logic. The latter doesn't really test anything in this lab since we don't have any bypassing logic, but they will be very important for the next lab.

You can look at the *parc-macros.h* file for detailed information on other macros, in addition to their syntax.

3.A Disassembling Instructions for Debugging

The simulators for this lab are equipped with disassembly features that may prove useful during debugging. **There are 3 levels of disassembly, which is specified as a command line option to the simulator, `+disasm=#`.** For instance, you can run the *pv2stall* simulator with a disassembly level of **2** like this:

```
cd $LAB2_ROOT/build
./pv2stall-sim +disasm=2 +exe=../tests/build/vmh/parcv1-addiu.vmh
```

Of course, this option can be used in conjunction with other options like, **+vcd** or **+stats**.

Level 1 of disassembly will show you the PC and instruction for every instruction in the program you just ran. The instructions are in program order and do not capture instructions that have been squashed or stalled.

Level 2 of disassembly will show you the same as **level 1**, with the addition of a regfile dump for each instruction that shows the value of each register after that instruction has executed. This level still does not capture squashing or stalling, just high-order program execution flow.

Level 3 of disassembly will show you a basic line trace of instructions flowing down the pipeline. Each stage is delimited by the '|' separator. The first stage, **F**, shows the PC of the instruction that was fetched, and all other stages show the name of the instruction in that stage at that cycle. Each new row represents a cycle, with time progressing downwards. '#' symbols preceding an instruction represent a stall, and instructions surrounded by '-' represent a squashed instruction. The '**(-_-)**' symbols represent bubbles in the pipeline (i.e. an invalid instruction). Here is an example snippet from the trace of the *parcv1-jr* assembly test:

```
{-00080014-|# jr    | addiu | (-_-) | (-_-) }
{-00080014-|# jr    | (-_-) | addiu | (-_-) }
{-00080014-|# jr    | (-_-) | (-_-) | addiu }
{-00080014-| jr     | (-_-) | (-_-) | (-_-) }
{ 00080020 | (-_-) | jr    | (-_-) | (-_-) }
```

We can see above that the processor is stalling at `PC=00080014` due to some data hazard between the *jr* and *addiu* instructions in **D** and **X**. Over the next few cycles, the processor waits until the *addiu* flows down the pipeline and writes back its result. Once the value is in the regfile, the processor comes out of stall, squashes the instruction in **F**, since it knows there will be a jump, and jumps to `PC=00080020`, after which the *jr* continues to flow down the pipeline.

3.B Compiling New Assembly Tests

It will be necessary to compile assembly tests.

Navigate to the *tests* directory and create a separate build directory. Then we can run a configure script to generate a `Makefile` based on the platform we are using. The commands below demonstrate this process:

For assembly tests:

```
cd $LAB2_ROOT/tests
mkdir build
cd build
../configure --host=maven
make
../convert
```

And for benchmarks:

```
cd $LAB2_ROOT/ubmark
mkdir build
cd build
../configure --host=maven
```

The `--host=maven` option to the configure script tells the configuration to use our course-specific cross compiler to compile our assembly tests instead of the standard *gcc*. Next, let's compile the assembly tests in the *parcv1* and *parcv2* directories and convert them into *.vmh* files. In the same directory:

```
make
../convert
```

Running `make` will compile the assembly test sources into binaries. Unfortunately, our Verilog processor cannot understand this format (yet!) so we will have to convert them into something it can understand. The convert script is a shell script that automatically generates the object dumps of binaries and in turn converts the object dumps into *.vmh* files. It will place all the binary files into a directory called *bin*, all the object dumps into a directory called *dump*, and all the *.vmh* files into a directory called *vmh*. As long as the *.vmh* files for the assembly tests we want to run are in the *vmh* directory, the simulator will be able to find and run them without any further input from the user.

When you add your own assembly tests or modify existing ones and need to recompile, you do not need to do all the steps above again, unless you delete your *build* directory. It is sufficient to run `make` and `../convert` in the *build* directory to recompile all the assembly tests.

The process that we just went through to compile the unit tests is essentially the standard process to compile projects on Linux from source. In general, we create a *build* directory in the project root, change into that directory and run `configure` and `make`. If you want to install the project, you would additionally run `make install`.

4. Evaluation

The evaluation for this lab will be based on several C benchmarks which are provided in the `$LAB2_ROOT/ubmark` directory. There are 4 benchmarks in the suite:

- *ubmark-vvadd.c*: vector-vector add
- *ubmark-cmplx-mult.c*: complex multiply
- *ubmark-masked-filt.c*: masked filter
- *ubmark-bin-search.c*: binary search

Remember that you can run all the benchmarks in the build directory and the statistics will be saved in the corresponding *.out* file. The *pv2stall* simulations will use the *-stall.out* suffix, and the *pv2byp* simulations will use the *-byp.out* suffix. For example, the results for *ubmark-vvadd.c* will be saved in the *ubmark-vvadd-stall.out* file if using the *pv2stall* simulator. The following commands run the benchmarks on *pv2stall*. Replace *pv2stall* with *pv2byp* to run the benchmarks on *pv2byp*.

```
cd $LAB2_ROOT/build
make run-bmark-pv2stall
```

For this lab, you will be comparing the performance of the *pv2stall* processor to the *pv2byp* and *pv2long* processors. Report the cycle count, as well as the IPC of each benchmark for each of the *pv2stall, pv2byp,* and *pv2long* processors. Analyze the performance differences between the implementations in the report and discuss in which situations bypassing is helpful and in which situations it will have limited effect. Also, discuss the tradeoffs of bypassing compared to stalling. What parts of the Iron Law of Processor Performance are affected?

5. Extensions

5.A Writing a Custom Benchmark

Using the existing benchmarks for reference and the instructions in this lab handout, code your own benchmark that demonstrates an interesting performance comparison between the *pv2stall*, *pv2byp*, and *pv2long* processors. Be sure to analyze the performance in your report and justify your approach in designing your benchmark.

5.B Other

As always, students are encouraged to think of other creative ideas for extensions. Just be sure to document your process in the lab report.

6. Lab Report

In addition to the source code for the lab, you will need to submit a lab report, which includes the following sections:

- **Abstract**: introductory paragraph summarizing the lab
- **Design**: describe your implementation, justifications for design decisions (if any), deviations from the prescribed datapath, discussion of any extensions
- **Testing Methodology**: describe how you tested the modules and your overall testing strategy–justify the effectiveness of your assembly test(s) in testing the functionality of the processor
- **Evaluation**: report your simulation results and cycle counts
- **Discussion**: comparison and analysis of benchmark results, discussion of tradeoffs of using bypassing over stalling
- **Figures**: updated **PARCv2** datapaths with bypassing and with bypassing and pipelined muldiv unit

The report can be a maximum of 4 pages, you will be penalized if you go over. Figures do not count against this limit.

7. Deliverables

For submission, please turn in a **.tar.gz** file of your working directory **without changing the original directory structure**. All source files should be located in `$LAB2_ROOT/pv2stall`, `$LAB2_ROOT/pv2byp`, and `$LAB2_ROOT/pv2long`. Please be sure to delete any generated waveforms or compiled code by running `make clean` in each of the build directories. Also, delete the build directories of the *tests* and *ubmark* directories.

```
cd $LAB2_ROOT/build
make clean
cd $LAB2_ROOT/tests
rm -rf build
cd $LAB2_ROOT/ubmark
rm -rf build
```

You can execute the following commands to make a tarball of your completed lab, assuming that you did not change the name of the lab root directory.

```
cd $LAB2_ROOT
cd ..
tar -cvzf {netid}-lab2.tar.gz cs505-lab2
```

Below is a list of files we will need to submit to meet the expectations of this lab:

- *pv2stall* source code
- *pv2byp* source code
- *pv2long* source code
- Custom assembly test
- *Datapath* diagram of pipelined 5-stage **PARCv2** processor with bypassing
- *Datapath* diagram of pipelined 7-stage **PARCv2** processor with bypassing and pipelined *muldiv* unit
- Lab report

8. Tips

- Use incremental development–never code everything at once and hope it works!
- Use the unit testing framework!
- Always draw the hardware before you start coding!
- Clearly define the interaction between the control logic and the *datapath*!
- If you can't get everything working, be sure to thoroughly explain how far you did get in the lab report in addition to the debugging strategy you used!

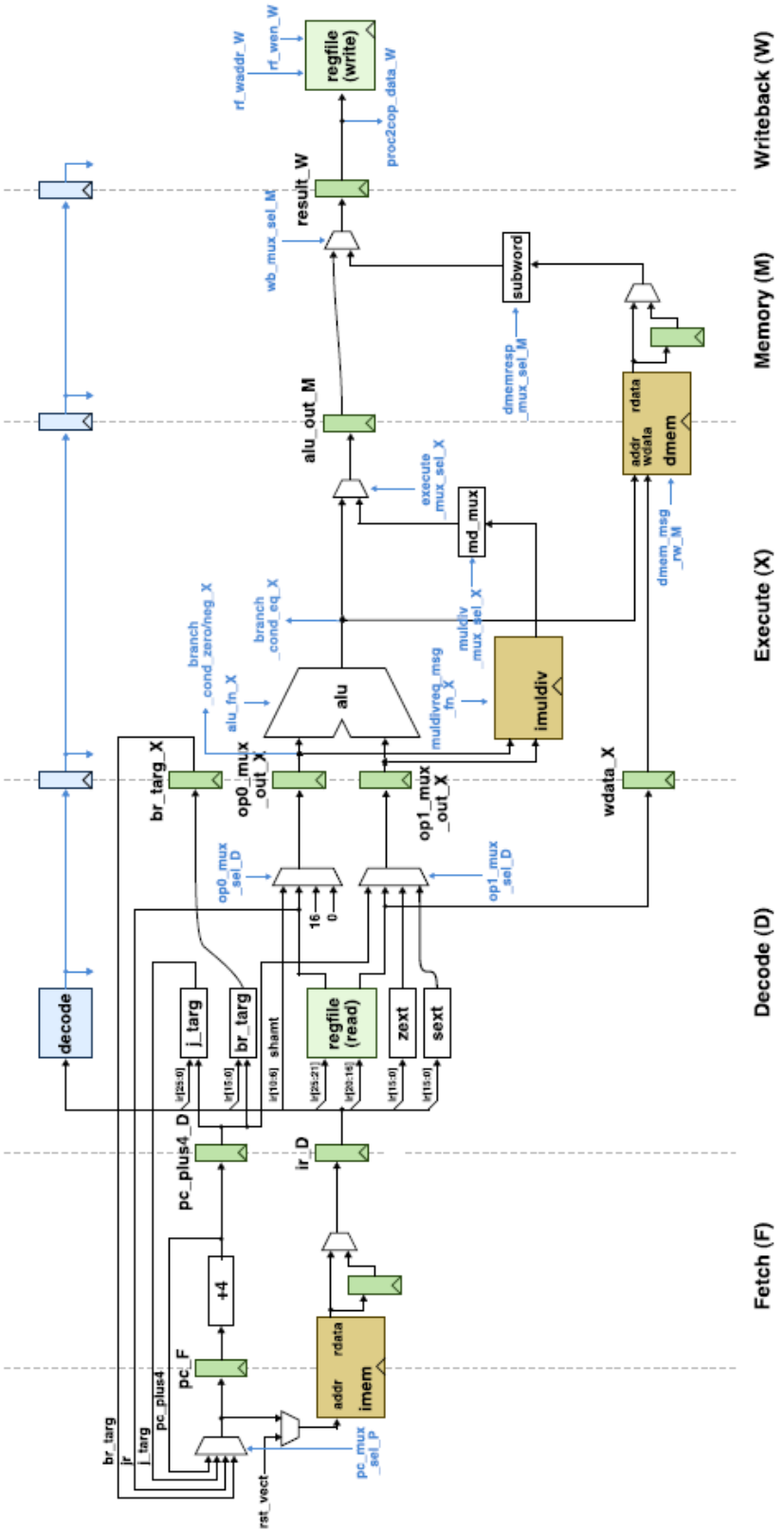
Writeback (W)
Memory (M)
Execute (X)
Decode (D)
Fetch (F)
regfile (write)
rf_waddr_W
rf_wen_W
proc2cop_data_W
result_W
wb_mux_sel_M
subword
dmemresp_mux_sel_M
alu_out_M
execute_mux_sel_X
md_mux
addr wdata rdata dmem
dmem_msg_rw_M
branch_cond_zero/neg_X
branch_cond_eq_X
muldiv_mux_sel_X
alu_fn_X
alu
muldivreq_msg_fn_X
imuldiv
br_targ_X
op0_mux_out_X
op1_mux_out_X
wdata_X
op0_mux_sel_D
op1_mux_sel_D
16
0
decode
j_targ
br_targ
shamt
regfile (read)
zext
sext
pc_plus4_D
ir_D
+4
pc_F
addr rdata imem
br_targ
jr
j_targ
pc_plus4
pc_mux_sel_P
rst_vect

Figure 1