Lab 2: Control Flow and Branch Prediction

Assigned: Sat., 07.10; Due: **Sat., 27.10** (Midnight)

Instructor: Onur Mutlu
TAs: Mohammed Alser, Can Firtina, Hasan Hassan, Jeremie Kim, Juan Gomez Luna, Geraldo Francisco de Oliveira, Minesh Patel, Ivan Puddu, Giray Yaglikci

1. Introduction

In this lab, you will extend the pipelined MIPS machine that we provide to support the control flow instructions (conditional and unconditional branches). You will also implement two branch predictors: (1) A simple global history-based branch predictor, and (2) a tournament-style branch predictor similar to the Alpha 21264 that we learned about in class. Along the way, you will learn about the squashing and correctness issues that arise due to control flow in a real pipeline, and when you are finished, you will have a complete pipeline that supports the core essentials of a real ISA.

This lab is to be done individually. You are allowed to consult with the TAs, but the lab should be completely your own individual work.

2. Your Task: Additions to the MIPS Machine

2.1. Architecture

- **Instruction Set.** Using the provided five-stage pipelined MIPS core **with forwarding** as a starting point, you will add support for control flow instructions (jumps and branches) *without* branch delay slots. After implementing the control flow instructions, the machine should support the 51 instructions in the following table.

ADD	ADDU	ADDI	ADDIU	AND	ANDI	BEQ
BGEZ	BGEZAL	BGTZ	BLEZ	BLTZ	BLTZAL	BNE
~~DIV~~	~~DIVU~~	J	JAL	JALR	JR	LB
LBU	LH	LHU	LUI	LW	MFHI	MFLO
MTHI	MTLO	MULT	MULTU	NOR	OR	ORI
SB	SH	SLL	SLLV	SLT	SLTI	SLTIU
SLTU	SRA	SRAV	SRL	SRLV	SUB	SUBU
SW	SYSCALL	XOR	XORI			

- **System Call Instruction.** Terminates the program (same as Lab 1). In addition, for debugging and grading purposes, you must ensure that the contents of the register file are dumped out in the **same format** as Lab 1.
- **Exceptions.** No support (same as Lab 1).
- **Branch Delay Slot.** The machine does *not* support a branch delay slot after a jump or branch instruction. The semantics of branch and jump instructions are listed for your reference in the table below.

JAL	R[31] $\leftarrow$ PC + $\underline{4}$ PC $\leftarrow$ PC$_{31...28}$ + (Immediate $\ll$ 2)
JALR	R[rd] $\leftarrow$ PC + $\underline{4}$ PC $\leftarrow$ R[rs]
B<cond>AL	R[31] $\leftarrow$ PC + $\underline{4}$ if (cond) PC $\leftarrow$ PC + 4 + (Sign_Extended_Immediate $\ll$ 2)

2.2. Microarchitecture

Pipeline Modifications.

In this lab, you will modify the five-stage pipeline we provide you for this lab to support branch and jump instructions by modifying the execute stage. Specifically, in addition to performing ALU and load/store operations, the execute stage will handle instructions that modify *control flow* as well. To do so, you will modify the execute stage to calculate the branch condition and the branch target address. Once the target address is computed, it is sent to the fetch stage to change the instruction fetch address when a branch is taken.

Pipeline Flushing.

In between when a branch is fetched and when the first instruction from the resolved branch target is fetched, other instructions will have entered your pipeline. It is your job to figure out how to flush the pipeline so that, from the programmer's point of view, only the correct instructions are executed.

Branch Prediction.

Now that you have control flow instructions working, you can start improving the performance of your machine using branch prediction. You will implement two types of branch predictors that we learned about in class, and submit each of these implementations *separately*. For each branch predictor, start from scratch with the Vivado project we provide, and submit your work as two separate projects in a single tarball.

1. **Global Branch Predictor.** The first branch predictor you will implement will be a simple global predictor, similar to the one we talked about in class. The global predictor consists of the following components:
   - A global history register that contains the outcome of the past 12 branches.
   - A pattern history table that contains 4096 entries.
   - A 2-bit saturating counter for each entry in the pattern history table.

   You are free to determine the initial values for these components. An important aspect of your design will be *when* you decide to update the global history register and pattern history table entries. One option is to update them speculatively based on the *predicted* outcome of the branch, another is to wait until the branch direction has been determined and update them based on the *actual* outcome of the branch. Experiment with both techniques and find a configuration that works well.[1]
2. **Tournament Branch Predictor.** The second branch predictor you will implement is a hybrid branch predictor that includes a local branch predictor and a global branch predictor. A separate choice predictor chooses which of the two branch predictors to use for each branch.

   For the global branch predictor, please reuse the design that you implemented earlier. The local predictor consists of the following components:
   - 1024 local history registers.
   - A 10-bit branch history for each local history register based on the PC of the branch.
   - A pattern history table containing 1024 entries.
   - A 2-bit saturating counter for each entry in the pattern history table.

   The choice predictor uses the 12 bits from the global history register to index into a 4096-entry choice history table. Each entry in the choice history table contains a 2-bit saturating counter specifying which predictor (local or global) to use, and is updated during branch retirement if the local and global predictors disagreed in the branch direction.

[1] A useful paper that analyzes and discusses the trade-offs of these two update strategies is on the course web page at: `https://safari.ethz.ch/architecture/fall2017/lib/exe/fetch.php?media=p228-hao.pdf`.

You are free to determine when to update the local history register and also the starting values for the components. Once again, experiment with different designs to try and find a configuration that achieves good performance.

3. Submission

Use the corresponding assignment in Moodle (`https://moodle-app2.let.ethz.ch/`). You should submit all the files needed to compile and simulate your code in a single tarball (with the name lab1 YourSurname YourName.tar.gz). Please include comments to explain what you have done, in the Verilog files.

3.1. Source Code

Make sure that your source code is readable and documented.

The source code of the pipelined MIPS machine is provided in Moodle. `inputs` contains the `primes` program for evaluating the performance of your design. `assembler` contains SPIM and the script (`asm2hex`) that you could use to generate binary for your own assembly program. In `lab2_vivado_project` you will find the Vivado project for the pipelined MIPS machine. The files in the project are similar to what you had in Lab 1. Please treat all source files confidentially and do not share with anyone.

3.2. Software Tools

For this course, we use the software Vivado for programming and simulation. The computers in rooms HG E26.1 and 26.3 are already installed with the necessary software. If you wish to use your own computer, you can refer to the following instructions: `https://goo.gl/VUJ34J`

3.3. README

In addition, please submit two `README.txt` files, one for each branch predictor.

The `README.txt` file must contain the following three pieces of information.

1. A high-level description of your design (including what are the initial values of your components).
2. The results of your experimentation with updating the various history registers and history table entries. Note that you do not need to report every permutation of local versus global update for each component, but at least show results for (1) updating all components with the predicted outcome, (2) updating all components with the actual outcome, and (3) discuss which design that you experimented with performs the best (which may be a hybrid of the two update strategies) and why.
3. The percentage speedup of each of your branch predictor designs over your machine without branch prediction for the `primes` input program (provided in the tarball).

It may also contain information about any additional aspect of your lab. You are not required to hand in your block diagrams.

4. Getting Started & Tips

4.1. Getting Started

1. **Please do not distribute the provided MIPS pipeline and program files. These are for exclusive individual use of each student of the Computer Architecture course. Distribution and sharing violates the copyright of the software provided to you.**
2. Open the project we provide in Vivado, modify the existing files and create new ones when needed.

4.2. Tips

- **Read this handout in detail.**
- **If needed, please ask questions to the TAs using the online Q&A forum in Moodle.**

- When you encounter a technical problem, please first read the logs/reports generated by the software tools. A search on the web can usually solve many of tool related debugging issues, and error messages.
- Your Verilog code will require many wires. Please adopt a consistent scheme for naming them. **You will review your code with us. So please make sure your code is readable by a third person.**
- The system call instruction should terminate the program only after all other preceding instructions have completed execution.
- Make sure your Verilog implementation is **synthesizable**.

5. Extra Credit

We will offer up to 100% additional credit for this lab for exploring different design aspects of the branch predictor. Please make sure you attempt the extra credit work only after you get the branch predictors (in Section 2) functionally correct and test them well.

For extra credit on this lab, we will hold two performance competitions.

1. Among all *global branch predictor* and *tournament branch predictor* implementations that are correct, the "top"[2] students that have the shortest critical path in their microarchitecture will receive up to 25% additional credit *for this lab*. Please make sure that you should use the constraint file (.xdc) that we provide when synthesizing your design. You can easily get the critical path delay by running the following *tcl* command in Vivado after opening the synthesized design:
   report_design_analysis -timing
2. We will also hold a competition on branch prediction accuracy. Among all submissions, the top students that have the best branch prediction accuracy and the best branch predictor designs will receive up to 75% additional credit *for this lab* as well as "prizes" (at the discretion of the instructor). To improve the branch prediction accuracy of your processor, you may choose to devise any or all of the following:
   (a) *Perceptron* branch predictor [1]
   (b) *TAGE* branch predictor [2]
   (c) Any other branch prediction mechanism, including optimization of the *global branch predictor* or the *tournament branch predictor* you have implemented. You can find more branch prediction proposals in `https://www.jilp.org/cbp2016/program.html`.

   Regardless, you should document how you optimized your design, to get extra credit.

Please write a clear and detailed-enough report (`report.pdf`) that describes 1) how you optimize the critical path, 2) how you improve the branch prediction accuracy, 3) what branch predictors you implemented and how you implemented them, i.e., the challenges you faced and the design decisions you made to resolve the challenges. Your report does not need to be more than four pages. Please also submit the version of your simulator that implements the extra credit parts you have completed.

You should submit your extra credit work separately using the designated Moodle submission link.

References

[1] Daniel A Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*, pages 197–206. IEEE, 2001.

[2] The instructor reserves all rights for the precise definition of the word "top".

[2] André Seznec. A case for (partially) tagged geometric history length branch prediction. *The Journal of Instruction Level Parallelism*, 8:1–23, 2006.