FINAL PROJECT REPORT

Performance of Branch Prediction on SweRV EH1 core

ECEN 5593

Submitted by

[Pranav Bharadwaj](mailto:prbh1731@colorado.edu)

[Malcolm Mckellips](mailto:mamc9392@colorado.edu)

Contents

1. Abstract
2. Introduction and Objectives
3. Motivation
4. Related Work
5. Methodology/Evolution of Design
6. Implementation
7. Evaluation
8. Results
9. Conclusions
10. Appendix

**ABSTRACT**

Branch prediction is a major component in modern day CPU architectures and plays a considerable role in improving performance (measured in CPI). Without it, a processor would have to wait until the conditional jump passes through the execute (EX) stage in the pipeline, where the decision to a conditional jump or not is calculated. There are several implementations of branch predictors used in the industry. GSHARE is one such method, where a shared history of all conditional jumps is maintained. This is advantageous especially for history where there is correlation between different conditional jumps. We aim to explore a similar implementation of GSHARE on a SweRVolf EH1 core, which is a 32-bit superscalar processor based on the RISC-V ISA. As an open-source design, RTL implementation of the core is available for synthesis and debug operations. Using the Digilent Nexys A7-100T board, which is based on a Xilinx Artix-7 FPGA, we synthesize the core on it, and use the onboard UART-JTAG port for debugging. PlatformIO is our IDE of choice, which is an extension for the popular code editor Visual Studio Code. Using a custom assembly subroutine, we aim to test the efficiency and performance of the branch prediction.

**OBJECTIVES**

The goals set out for this project are divided into two main sections: hardware and software. The hardware portion is primarily concerned with the following:

1. Synthesizing RTL for the EH1 core using Xilinx’s Vivado IDE. The hardware description of the core itself is readily available from Western Digital. However, additional modules such as I/O, path trace for JTAG need to be added as “wrappers” to the main core code and synthesized together.
2. Programming the Nexys A7 board with the bit file from the previous step. The programmer needs to communicate with the board, which requires using the appropriate FTDI USB driver for the board family.

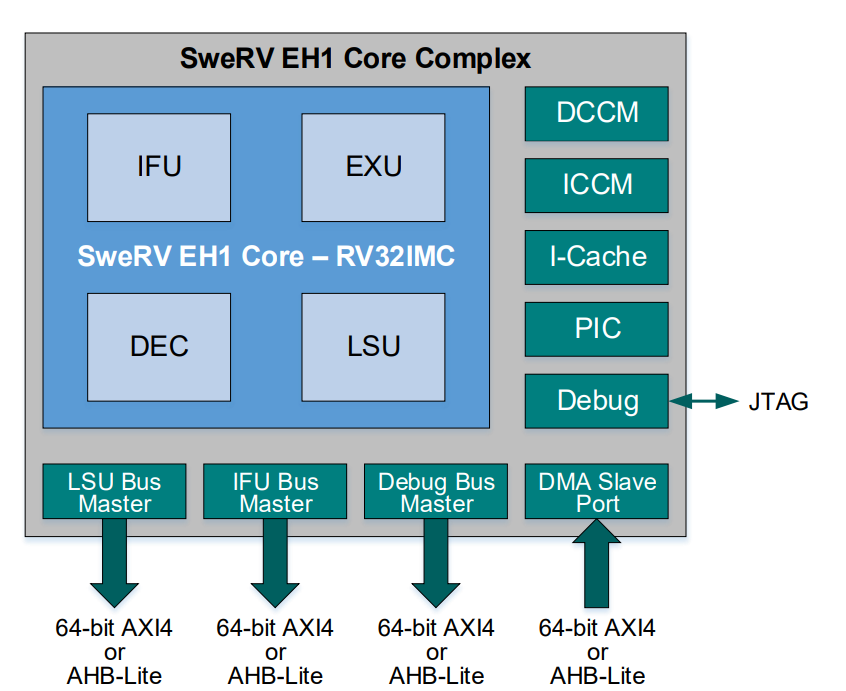
The software section is primarily concerned with the following:

1. Connecting to the board using PlatformIO and OpenOCD as the debugger of choice to dump a test assembly subroutine onto it. This is used to test the operation of the various registers in the RISC-V ISA.
2. Establish a metric to measure the performance of the branch predictor. This can be achieved by using cycle count for the execution of an assembly subroutine. Based on the performance of the branch predictor, the values will differ.
3. Disable the default branch prediction on the core using the onboard ‘mfdc’ register, which is part of the control status register (CSR). This allows us to test and establish a baseline for core performance without branch prediction.
4. Calculate the number of cycles required for the execution of a custom assembly subroutine with branch prediction enabled and disabled. Repeat this process for different run lengths of the subroutine and tabulate the result.

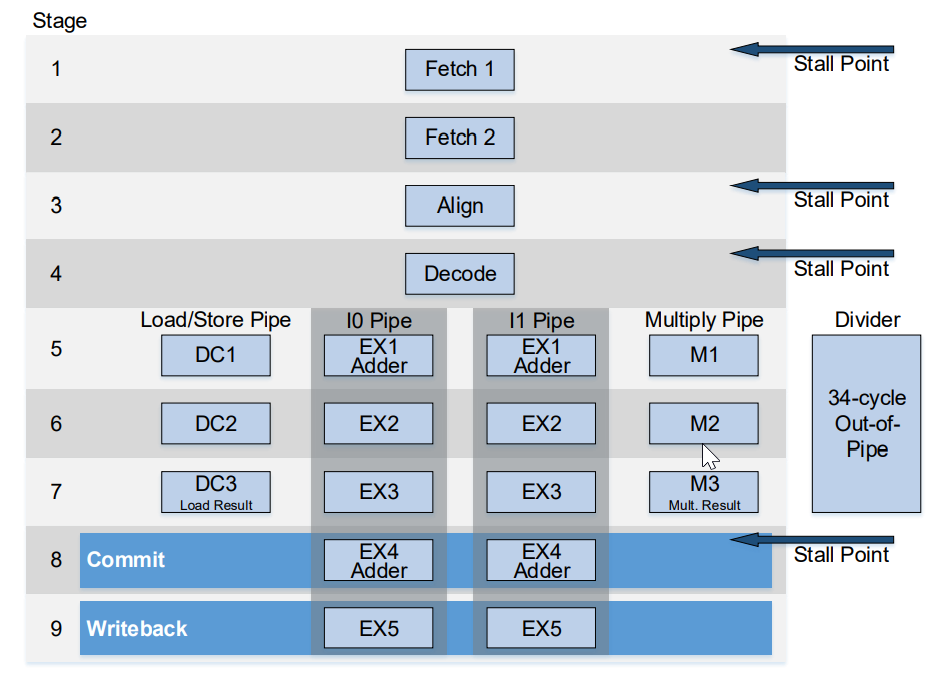
**SweRV CORE**

The SweRV cores is a family of RISC-V based processors developed by Western Digital. As one of the first implementations of the new RISC-V ISA, Western Digital have developed three versions of it, namely the EH1, EH2 and EL2 cores. These are based on the RV32IMC ISA extension, meaning that they support integer, multiplication and division (dedicated hardware) and compressed instructions. Our primary concern in this project is the EH1 core, which is a 32-bit, 2-way superscalar processor with a 9-stage pipeline. The core is designed for machine-mode implementation, where all functions are accessible except virtual memory management and a dedicated MMU. The core also has openly accessible CSR registers and commands to extract values from them. Some of the additional features offered are as follows:

* RV32IMC compliant with branch prediction (GSHARE implementation)
* Optional 4-way associative instruction cache (L1)
* Optional interrupt controller supporting up to 255 external interrupts
* RISC-V specified core debug unit, supporting standard JTAG
* Targeted for 28 nm process and 1 GHz target frequency



The pipeline employed has 9-stages, with two dedicated integer ALU units and separate multiplier (which is further pipelined) and an out-of-pipe 34-cycle divider. Load store operations are handled by a separate unit which increases throughput. The pipeline is configured with 4 stall points as shown in the diagram below. The design also employs prefetching of instructions before the execute section. There are 32 architectural registers numbered x0 to x31. X0 is the default ‘zero’ register, which is used in several instructions. All these register specifications are derived from the base RV32IMC ISA extension.

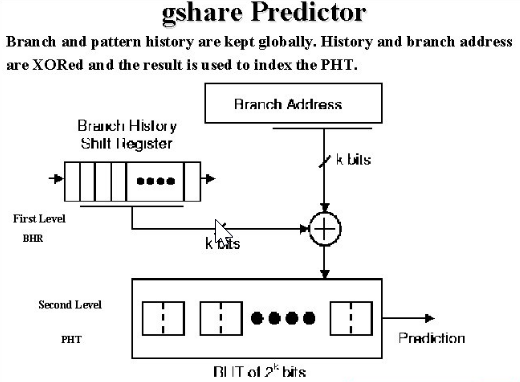


An important thing to note here is that conditional branches are executed at the 7th stage in the pipeline, when the value of whether a conditional jump passes the condition or not is calculated. Branch prediction units come into picture much earlier, performing a prediction for the jump destination and fetching newer instructions before the jump condition itself is satisfied.

**BRANCH PREDICTION**

One of the major components in modern day processors is the branch prediction unit, which contributes significantly to the increases in performance we observe in every new generation. Since most modern processor pipelines have over 10 stages (beyond 20 for CISC machines), branch prediction becomes crucial as the penalty from waiting for the calculation of conditional branch address to finish becomes considerable. Following instructions would have to wait until then to enter the pipeline itself. The argument for branch prediction is that the worst case scenario would still have an accuracy of 50%, while the advantages from it would result in a significant rise in IPC (Instructions per count).

The simplest form of branch prediction is a static predictor, which is independent of the dynamic history of the code being executed. Instead, each branch statement would have a fixed outcome decided beforehand. Early implementations of MIPS and SPARC employed this form of prediction. An improvement on this is a dynamic branch predictor, which uses information about taken or not-taken branches at run-time as history to predict the outcome of the current branch. There are various flavors of dynamic predictors, such as saturating counter, two-level predictor, and global branch prediction. These maintain a branch history table (not to be confused with Branch Target Buffer, aka BTB), which is used to predict the current outcome. Global branch predictors such as GSHARE maintain a combined history for all conditional jumps. This results in better prediction for correlated branch instructions and jumps. It is highly dependent on instructions being correlated, else the branch history stored would be diluted and result in poor prediction accuracy. With a large enough buffer for storing branch history, GSHARE can result in considerably better prediction accuracy.



MOTIVATION

**RISC-V**

RISC-V is an open ISA that allows for free, modular development and has recently gained a spike in popularity. After its original conception at UC Berkeley in 2010, RISC-V has grown from a simple ISA exploration project into a full-fledged global organization. RISC-V International is a nonprofit organization with over two thousand industry members which now maintains and extends RISC-V. RISC-V is built on the principles of open-source architecture, a trend which has gained huge popularity in recent years across all fields of tech. Besides the exciting and accessible nature of RISC-V there are several other reasons that it was chosen as the base architecture for this project. For one, the project design team has more experience with the RISC-V ISA than any other architecture. Also, recent news has pointed to the current importance of RISC-V in the general ecosystem of computer architecture. Intel has made recent attempts to acquire SiFive, one of the world’s top designers of RISC-V processors. Due to Intel’s massive power over the computer architecture industry as a whole, their move to acquire SiFive lends support to the fact that RISC-V is an important facet of modern processor design and points towards it only increasing in importance in coming years.

**BRANCH PREDICTION**

When assessing concrete metrics for the evaluation of physical cores, branch prediction jumped out as a simple architectural change that would produce very apparent and measurable performance changes. In addition, early in the course when this project was conceived, branch prediction was one of the architectural elements that had already been discussed in class. Branch prediction itself is also an interesting and ever-evolving field that has profound implications for processor performance. Branch prediction is not only relevant as one of the largest single architectural elements that contributes to processor performance, it has had very interesting implications in the security field in recent years. Spectre attacks leverage knowledge that can be gained from caches modified by speculative execution to glean sensitive user data and were not discovered until 2018! These attacks, while niche, led to a complete reformulation of the paradigms surrounding speculative execution and fueled the fire behind branch prediction research. Also, both of the design team members of the project had previous experience with branch prediction, so it seemed a feasible architectural change to evaluate a physical core by extracting meaningful comparison metrics.

METHODOLOGY/EVOLUTION OF DESIGN

**HARDWARE AND BRANCH PREDICTION**

At the initial inception of the project, the initial goal was to take a variation of a RISC-V core that contained no branch prediction and implement branch prediction by modifying the source HDL files that made up the core. The key objective throughout the process was to find some way to test the effects of branch prediction on physical hardware. However, after exploring the available cores it became quickly apparent that there were far more flavors of RISC-V cores than we initially thought and the vast majority contain branch prediction already. As it was already known that the project was to target the Digilent Nexys A7 board, the project or core to be used was narrowed down to one that had already been verified for this platform. This eventually led to the SweRV EH1 being selected and it already includes the Gshare branch prediction scheme described above. This led to a shift in project approach. The new scope of the hardware design for the project involved modifying the HDL source code of the EH1 core to remove the existing, complex branch prediction module rather than adding simple custom branch prediction as initially intended. However, after inspecting the HDL files for the core and reviewing the branch prediction documentation in the programming reference manual, it was found that the EH1 core actually allows the branch prediction module to be disabled from within software (as described below in the Branch Prediction Register section.

This led to our final design implementation. It allowed differences in performance to be evaluated on a core with branch prediction enabled vs. a core with branch prediction disabled. This was a positive shift in the project as it moved away from the simple 2-stage branch predictor that was originally going to be implemented in hardware and instead offered learning about the complex branch prediction scheme used in the EH1 core. It also allowed the core to be modified by simply changing a line of assembly software to disable branch prediction when testing rather than going through the lengthy FPGA compile and load process for two versions of hardware.

**SOFTWARE AND EVALUATION**

While using VSCode and PlatformIO as the IDE through which software would be run stayed constant throughout the lifetime of the project, almost every other aspect of the initial evaluation framework changed several times. At the very start of the project, naive evaluation metrics were contrived including reviewing timing information from the FPGA design environment Vivado itself to extract performance metrics to compare branch prediction. After learning a bit more about benchmarking however, a shift was made towards incorporating software to run established benchmarks and extract performance metrics from third party tools to evaluate the core. This general evaluation framework then underwent several more transformations before arriving at its final iteration.

For one, the operating system itself that all of the evaluation software would be run from changed about midway through the project. Initially, Ubuntu 18.04.6 LTS Bionic Beaver running on an Oracle VirtualBox VM was utilized as the OS from which to run the software. This OS was chosen because of its compatibility with Whisper and Verilator, which were evaluation tools originally hoped to be used at early stages of the project. Whisper is a RISCV instruction set simulator specifically designed by Western Digital to verify SweRV cores. This would allow for simulation of the core to gain insight into its functionality and could even be used to extract simulation data if success was not achieved on the hardware itself. Verilator is a powerful tool that converts Verilog to cycle-accurate behavioral models. Even better, Verilator interfaces with PlatformIO, which was the desired software development platform. When used in conjunction with GTK wave (a hardware level system timing viewer), it was hoped to be able to detect and count branch mispredictions when running a benchmark program. However, as can be imagined, running these large software programs from within a VM led to a host of issues that hindered development. Drivers were a prevailing problem throughout the project. From within the VM managing the drivers required to connect to the board for each of the many tools used was even more difficult and often the board could not be connected to at all. Also, due to the limited resources allocated to the VM, hardware compilation times for the core were massive within Vivado. This led to a switch away from the Ubuntu VM and gave motivation towards being able to develop a software design and evaluation framework entirely supported by the Windows 10 operating system.

Ultimately the switch of operating systems led to several evaluation design changes. For one, Verilator and Whisper are built to run on Unix systems (primarily Linux distributions) and thus could no longer be utilized. The next iteration of the evaluation framework thus took shape as an attempt to use the debugging techniques present within VSCode PlatformIO itself to extract branch prediction metrics. Initially, this was intended to be done by viewing the registers and program counter when running an assembly program with loops, thus identifying mispredictions in real time with software debugging. This technique was extended in an attempt to automate the process by using the GDB debugger’s printing and scripting capabilities to automatically run the program until reaching breakpoints at which point register contents could be dumped to a file. This file was to later be parsed by a python script in order to identify misprediction events. However, it became readily apparent that branch mispredictions are not so easy to identify in software. As any lasting effects of mispredictions are supposed to be cleared when the pipeline is flushed before the writeback stage, the register contents viewable through GDB ended up being inadequate indicators for branch misprediction. This led to the final evaluation framework being developed. This framework, described in detail below in the Evaluation section, consists of running a series of software loops over the two cores being compared and using the register watch window in VSCode updated over JTAG to extract cycle count information from internal privileged core registers to evaluate performance.

IMPLEMENTATION

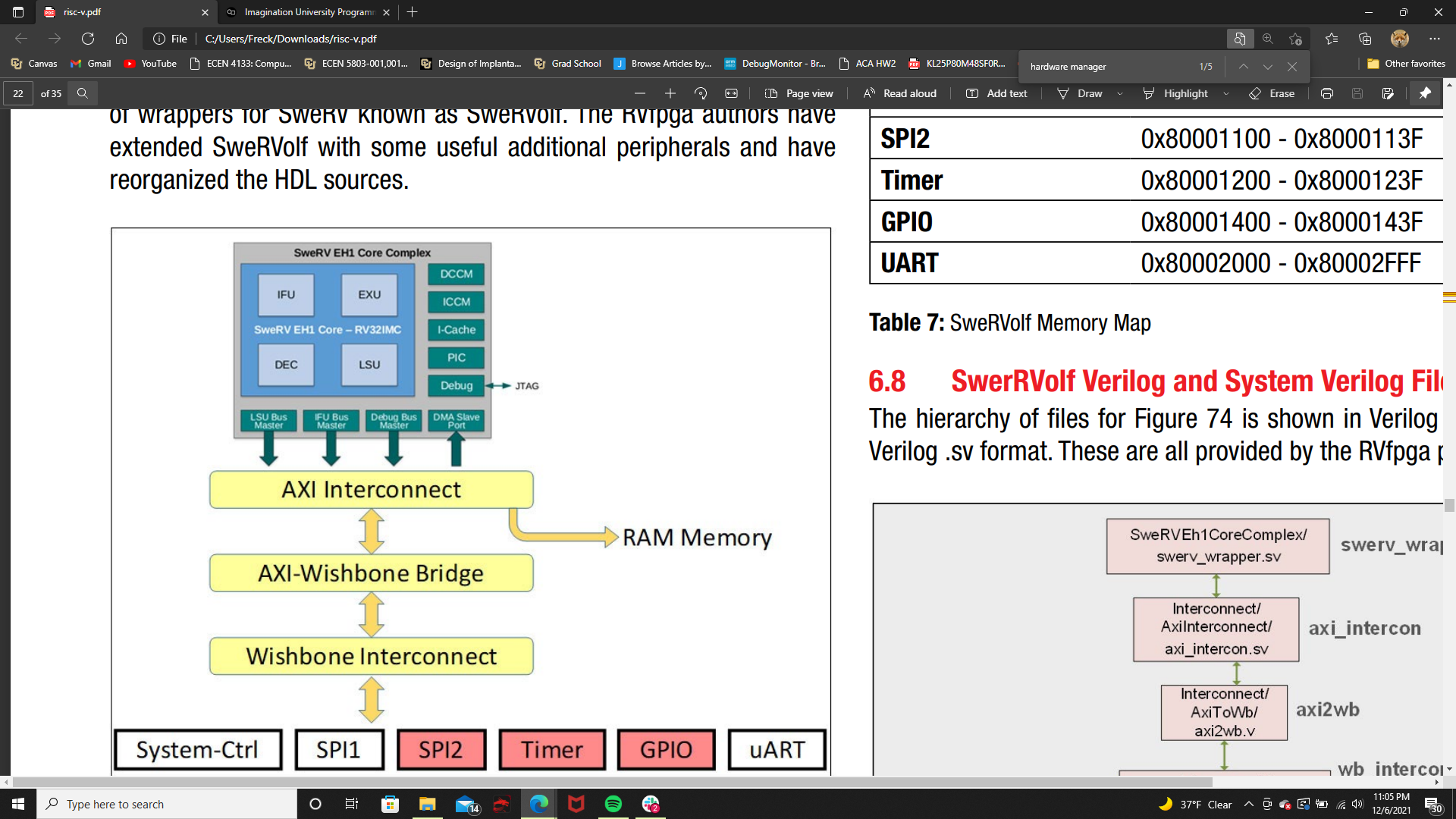
**HARDWARE**

DIGILENT NEXYS A7 BOARD

The project is targeted at the Digilent Nexys A7 digital circuit development platform. The Nexys A7 board features the Xilinx Artix-7 FPGA (XC7A100T-1CSG324C) with 15,850 logic slices and an internal clock of 450 MHz+. This board was selected in part because there is already a host of support surrounding implementing RISC-V based IP within the Nexys A7.

SWERVOLF

The specific EH1 SweRV core utilized is a wrapped version written by Olof Kindgren called SweRVolf. The core is a reorganized version of that distributed by Western Digital and includes additional useful peripheral support. The components of the core are memory mapped, which allows for easy updating and control of peripherals using simple memory writes in assembly. The distribution utilized is downloadable from the Imagination University Program. This distribution relies heavily on the use of block diagram/schematic elements within a Vivado project to combine the HDL components that make up the core.



*SweRVolf Core. Image Courtesy of Imagination University Program*

BRANCH PREDICTION REGISTER

The final implementation of the design utilizes the branch prediction disable or bpd field of the feature disable control register (mfdc) of the EH1 core. This allows for toggling on and off of branch prediction through assembly code by simply writing to a bit in the register. This bit essentially works as a select bit in the RISC-V control path such that the program counter value is not updated with a predicted next address but rather always assumes that branches are always not taken and loads the PC with PC+4.

LOADING THE CORE ONTO THE NEXYS A7

The Core can be built within Xilinx Vivado using the standard FPGA design flow. When the core has been built from the source files a .bit file will be generated. Assuming that the necessary board support packages for the Nexys A7 board have been downloaded in Vivado, the bit file can then be loaded directly onto the Artix-7 FPGA using the Vivado hardware manager. NOTE: Vivado requires specific serial drivers different from those used by PlatformIO for writing software.

**SOFTWARE**

PLATFORM IO

PlatformIO is an embedded development platform designed from the ground up to be cross-platform and cross-vendor (hence the name). It is a VSCode extension that allows a powerful IDE and debugging interface to be accessible from within VSCode. PlatformIO has support specifically for the Digilent Nexys A7 board including debugging capabilities through OpenOCD and GDB, and compilation for SweRVolf cores. In order to interact with the SweRVolf core, the Western Digital “CHIPS Alliance” platform must be installed within PlatformIO. This package contains support tailored to make a seamless interface to the Digilent Nexys A7. To format a project within PlatformIO, first an app must be set up using a .json file which contains necessary project information. In this case, this consisted of including the necessary assembly files, specifying the compiler settings (to compile for RISC-V RV32I), board information for the Nexys A7, and architecture information specifying the SweRVolf EH1 as the target. These .json PlatformIO configuration files are generally automatically generated, and for this project the configuration file could be copied from an example application designed for the SweRVolf core on the Nexys A7. After formatting the VSCode PlatformIO app, source files can be written to run upon the EH1 core.

ASSEMBLY

For this project all of the source files used for testing the core were written in RISC-V assembly code as per the RV32I specification. The assembly files can be edited from within VSCode and can also be built from the same interface with the power of PlatformIO. This process is one of the simplest embedded development experiences available and it is on the modern VSCode interface rather than one of the larger and clunkier platforms offered by board vendors.

To test the core, first, assembly programs that exercised the peripherals of the board were utilized in order to elicit a physical response from the board. For example, some of the first programs run were those that wrote to the memory mapped GPIO locations to blink the LEDs or control the LEDs with the switches. These programs gave an initial sign of life that verified the core was successfully loaded onto the board and running custom software. Finally, custom software was written to exercise branch prediction by writing to the mfdc register and using the software design described below in the “Evaluation” section.

EVALUATION

**REGISTER WATCH AND “MCYCLE” REGISTER**

VSCode offers the ability to view local variables directly from within the IDE’s debugging view. With the PlatformIO extension, architectural registers of the specific MCU or core being debugged can also be viewed. The JTAG connection to the board allows OpenOCD to extract a constant stream of data about the device state to PlatformIO. This data can be viewed at the left of a VSCode code in the “registers'' section. When the debugger is paused (or a breakpoint is reached), the registers will update with their correct current values. This was a useful technique used to ensure that the core was working properly after its first instantiation. However, the registers that can be viewed are not limited to the basic RV32I registers (x0-x31), but depending on the architecture of the core implemented can also display architectural or control registers. In the case of the SweRV core, this included control registers such as the featured disable control (mfdc) register. This meant that we could ensure that assembly code was actually modifying the register and toggling branch prediction on and off. Another useful pair of architectural registers that could be viewed from the register watch were the mcycle and mcycleh registers. According to the RISC-V privileged specification, these registers are designated for holding the clock cycle count for the current RISC-V hart running on the core. Mcycle holds the lower 32 bits of the clock cycle count while mcycleh holds the upper 32 bits.

The RISC-V privileged spec specifies the privilege levels and operating modes of RISC-V cores. When an operating system is used, these modes are very important (for security purposes, you do not want users to have direct access to architectural components of the core). The mcycle registers are present in the “m” or machine level privilege mode which is the lowest level and protected mode. In our simple application with no operating system, these registers were readily available and thus gave us a concrete performance metric (clock cycles). As per the RISC-V spec, the mcycle registers can start with any value, and thus only relative metrics are useful for measuring the amount of clock cycles executed. For example, to get the approximate clock cycles of a single instruction, first pause the debugger before the instruction and take note of the mcycle and mcycleh register values in the debugger. Then run past the single instruction and pause at a breakpoint directly after the instruction. Then, take note of the mcycle registers a second time. The difference between the values specified by these registers will give the increase in the cycle count for the single instruction. An interesting observation that was made when testing this process was that the mcycle count increased dramatically even for a single instruction. However, when executing several of the same instruction (i.e.in a loop), the mcycle count, while very large, increased proportional to the number of times the instruction was executed. This pointed towards the fact that for the specific EH1 core used, the cycle count is likely proportional, but not indicative of the literal clock cycle count. In order to more constructively analyze this cycle information to evaluate branch prediction, the cycle count metric was created by taking an initial and final mcycle count, through the process described above, and subtracting them. This difference in mcycle count for a given set of assembly instructions was then normalized to 100, such that the metric itself was easier to work with and display. The evaluation framework itself as well as the assembly code used to compare the performance of the core with and without branch prediction can be seen in the next section.

**ASSEMBLY TEST CODE**

The main objective of this test code is to provide a benchmarking solution exposing the cycle count consumed by the processor while performing branch prediction. Since BTB values are not architecturally exposed, counting the entries in it is not an option. Instead, we come up with an assembly subroutine that employs a calculated number of conditional branch executions also while executing arithmetic operations that follow the branch instructions. Once a branch instruction is pushed into the pipeline following a branch prediction, arithmetic instructions follow it in. If the prediction is accurate, there would be no increase in cycle count apart from the usual number of cycles stipulated in the ISA. In case of a misprediction, the arithmetic instructions in the pipeline are flushed and the processor resumes execution after the branch instruction. This would result in an increase in the cycle count, albeit significantly low.

To expose this change in cycle count, we increase the number of branches performed by creating a loop construct that loops a predefined number of times. As Gshare works better with correlated branching data, a loop construct would result in improved prediction accuracy and lower cycle count with branch prediction enabled, while also exposing the change in cycle count with branch prediction disabled. A sample of the test code is shown below:

li t2, 0xff

j next

branchx:

addi t2, t2, -1

bne t1, t2, next

li t1, 0x09

ret

next:

li a1, GPIO\_SWs # Read the Switches

lw t0, 0(a1)

li a0, GPIO\_LEDs

srl t0, t0, 16

sw t0, 0(a0) # Write the LEDs

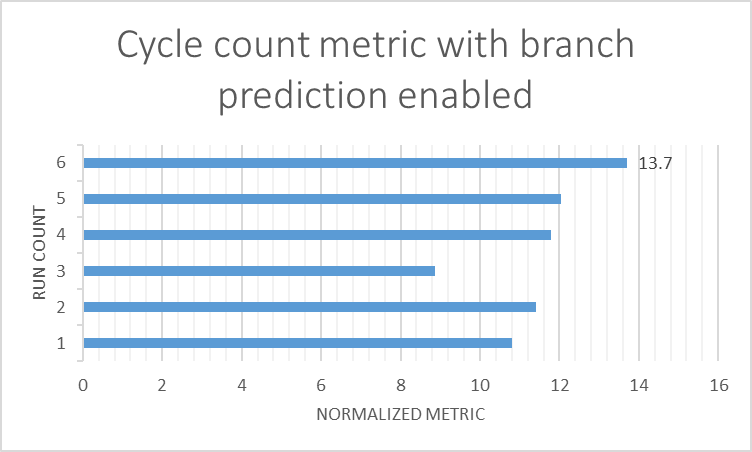
jal branchx

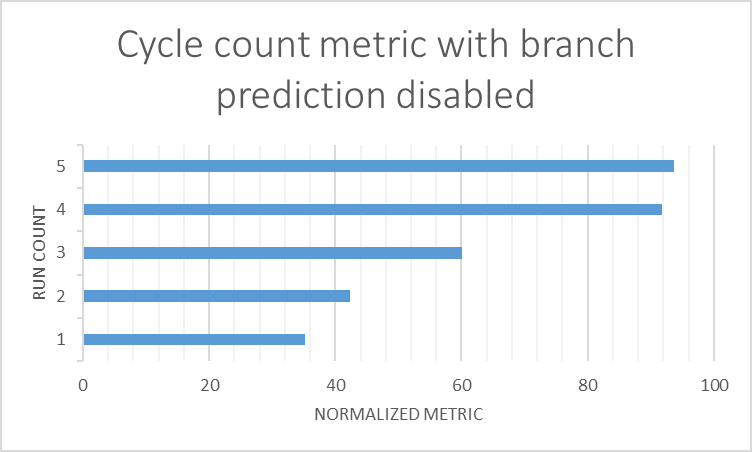
.end

The register labelled ‘t2’ here holds the predefined number of loops to be performed. The load instruction following the conditional branch is used as a breakpoint location. The code would be run in debug until it hits the breakpoint, which would occur after the loop register value is exhausted.

**RESULTS**

The normalized cycle count values were calculated for several runs with different loop counts. This process was repeated with branch prediction disabled as well. The comparisons are shown below in the form of graphs.





Here, we observe two important things: the significant difference in cycle counts between branch prediction enabled and disabled, and the relatively limited change in cycle count with branch prediction enabled when the loop count value is increased. These behaviors can be attributed to the following reasons:

1. Without branch prediction, there is always a 50% chance for a conditional branch to be taken or not. For a loop count of 100, this could result in 50 instances of mispredictions. Each misprediction would result in 7 instructions dropped from the pipeline, as the value of a conditional branch is calculated in the 7th stage. Thus, cycle count values increase significantly.
2. With branch prediction enabled, cycle count values naturally drop. However, the change between different loop runs is considerably less, especially when the loop count value increases. This is attributed to the Gshare implementation in the SweRV EH1 core. Gshare works best with correlated conditional branches, and loops represent the same. Hence, increasing loop count values are highly correlated with previous branch predictions of the same instruction. Thus, prediction accuracy increases and results in lower cycle count values.

**CONCLUSIONS**

The main goal of this project was to observe and tabulate the performance of the branch predictor onboard the SweRV EH1 core and devise a metric to quantify the values observed for better comparison. This problem was compounded by the fact that the BTB is architecturally invisible, preventing us from accessing a direct value for comparison. Instead, we devised a custom method to extract the performance of branch prediction by observing at cycle count values, which is stored in the CSR registers of the core.

The Gshare implementation onboard the EH1 core behaves as expected, with improved prediction accuracy when handling correlated conditional jumps and large variations when it is disabled. This also gives us an insight at how branch predictors are implemented in modern day processors, as their implementation and structure is not generally publicly available. It also introduced us to how an industry-level implementation of the RISC-V ISA would function, and all the additional features they support. This allows us to bridge the gap between computer architecture theory and real-world implementations and their operation.

Another facet of this project that deserves equal attention is the process of synthesizing the RTL for the core onto a FPGA. It sheds light on the multi-billion-dollar worth semiconductor design licensing market, showing how RTL for a processor core is altered for specific applications and how it is tested using FPGAs. Most modern mobile processors are variations of ARM designs, and this project gave a glimpse into the process of developing such custom chips.

Appendix

References

* Matt Godbolt’s blog and associated github repository on BTB in contemporary Intel chips: <https://xania.org/201602/bpu-part-three>
* Western Digital’s docs on SweRVolf family of cores: <https://www.westerndigital.com/solutions/business/risc-v>
* RISC-V org’s technical specifications about ISA: <https://riscv.org/technical/specifications/>