Design and verification of a branch predictor

# Introduction

In this project we have designed a branch predictor which includes a Branch Transfer Buffer (BTB), Branch History Table (BHT), and a Global History Register (GHR). Branch prediction is used to overcome the fetch limitation imposed by control hazards in order to expose instruction-level parallelism (ILP), the key ingredient to pipelined and superscalar architectures that mask instruction execution latencies by exploiting (ILP). Branch prediction can be thought of as a sophisticated form of prefetch or a limited form of data prediction that attempts to predict the result of branch instructions so that a processor can speculatively fetch across basic-block boundaries. Once a prediction is made, the processor can speculatively execute instructions depending on the predicted outcome of the branch. If branch prediction rates are high enough to offset misprediction penalties, the processor will likely have a better overall performance. Without branch prediction, such a processor must stall whenever there are unresolved branch instructions. Since on average 20 percent of instructions are branches, this imposes a substantial penalty on the performance of such processors. Many branch prediction schemes have been proposed. This work uses selected benchmark programs from the SPEC95 to compare several well known branch prediction schemes.

# Design

### BTB Implementation

It is implemented 4-way associativily. In set associative the branch address is directly mapped to sets of multiple table entries, and within those sets the entries are searched fully associative. This is a good compromise of the positive aspects of the different mappings.and an LRU replacement policy is used. Tag is hashed and LRU is implemented with 3 bit counters.

### BHT Implementation

it is implemented as an array of 2 bit entries. The tag used is hashed and the table gets updated after every branch instruction

### GHR Implementation

Global history register is a shift register which records the behavior of previous branches.

# Infrastructure

Varilater is used to run the core. Instruction set simulator is SPIKE. Lint is used for debugging,

• RISCV gnu tool-chain installed works for 64-bit architecture by default to make it work on 32-bit architecture toolchain is need to be configured with –with-arch=rv32im option.

• Used spike as a golden reference functional RISCV ISA software simulator to verify core.

• Spike works on 64-bit architecture and elf produced by spike was different from core’s elf. So core’s elf is used on spike to create log. That log is then compared with core’s trace log.

• Spike starts pc addresses from 0x80000000 to make elf compatible, changed core’s data and instruction memory addressing from 0x00000000 to 0x800000000.

• Compared core’s trace log with spike’s log to confirm functionality.

• Generated tests using force-riscv.

• Problem while running branch instruction:

The default code model, medlow, can only access addresses below 0x80000000 on a 64-bit part, or the entire memory on a 32-bit part due to wraparound. If you want to access memory above 2GB on a 64-bit part, you have to use the medany code model, via the -mcmodel=medany compiler option. If you are linking with libraries, like the C library, then those libraries must also be compiled with medany. It you are building your own toolchain, this works best if you use – with-cmodel=medany when configuring. The toolchains on the SiFive web site are configured this way.

• To configure code-model medany :

./configure --prefix=/opt/riscv --with-arch=rv32imc --with-cmodel=medany –enable-multilib

### CODE MODEL

• The code model determines which software addressing mode is used, and, therefore, what constraints are enforced on the linked program. Software addressing modes determine how the programmer sees addresses, as opposed to hardware addressing modes which determine how address bits in instructions are handled.

Code models are necessary due to the split between the compiler and the linker: when generating an unlinked object, the compiler doesn't know the absolute address of any symbol but it still must know what addressing mode to use as some addressing modes may require scratch registers to operate. As the compiler cannot generate actual addressing code, it generates addressing templates (known as relocations) that the linker can then fix up once it knows the actual addresses of each symbol. The code model determines what these addressing templates look like, and thus which relocations are emitted.

• Specifically, the medany code model generates auipc/ld pairs to refer to global symbols, which allows the code to be linked at any address; while medlow generates lui/ld pairs to refer to global symbols, which restricts the code to be linked around address zero. They both generate 32- bit signed offsets for referring to symbols, so they both restrict the generated code to being linked within a 2GiB window.

• “bne x2,x3,8” is asking for a branch to an absolute address 8, but bne is pc-relative, and can’t accept an absolute address as a target, so the assembler changes this to an inverted branch around a jump instruction. If you want to branch to pc+8, then use “bne x2 , x3, .+8”, but in the presence of linker relaxation that might not go where you want if the instruction after it is converted to a compressed instruction, so it is always best to use a label as a target of a branch instruction.

• Your testcase has “.8” which isn’t correct. It should be “.+8”. With “.8”, the assembler assumes this is a label, and since this is an undefined label with an unknown address which can’t be assumed to be in range of a branch instruction, it gets converted to an inverted branch followed by a jump, as a jump has a longer range than branch. The inverted branch is why you get beq instead of bne.

# Verification

A verification plan is formulated by brainstorming each possible case to be faced by branches predictor. Testbench is optimized for new tests. The tests for each case are generated using force-RISCV. Python scripts are written in a directed-randomized fashion.