EEE3530-01 **Spring 2024**

Assignment 4: Branch Prediction

Due: Sunday, June 2, 2024, 11:59 PM

1 Introduction

- The objective of this assignment is to implement a branch predictor in a processor pipeline. This assignment requires working on the Kite C++ code.
- To begin the assignment, go to the `kite/` directory that you worked for Assignments 1 and 2.
- Download `branch.sh`, and execute it inside the `kite/` directory to update Kite.
- In this assignment, `Makefile` uses `-DBR_PRED` and `-DDATA_FWD` options to enable branch prediction and data forwarding in the processor pipeline.

```
$ cd kite/
$ wget https://casl.yonsei.ac.kr/teaching/eee3530/branch.sh
$ chmod +x branch.sh
$ ./branch.sh
$ make
g++ -g -Wall -O3 -DBR_PRED -DDATA_FWD  -o alu.o -c alu.cc
g++ -g -Wall -O3 -DBR_PRED -DDATA_FWD  -o br_predictor.o -c br_predictor.cc

...
```

- The following describes how Kite handles branch instructions along the pipeline stages in `proc.cc` - you do not have to modify this file.
- **Fetch stage:** Instruction fetch and branch prediction
  - Open the `proc.cc` file, and find the `void proc_t::fetch()` function near the end of the file.
  - In the fetch stage, `inst = inst_memory->read(pc)` fetches an instruction from the instruction memory based on the program counter (PC).
  - If the fetched instruction is SB type (i.e., `get_op_type(inst->op) == op_sb_type`), it makes branch prediction based on the instruction's PC; `inst->pred_taken = br_predictor->is_taken(inst)`. The instruction's PC can be retrieved by reading `inst->pc` in the function. `inst->pred_taken` stores if the branch is predicted to be taken (1) or not (0).
  - If the branch is predicted to be taken, the PC is set to the predicted branch target by consulting the branch target buffer (BTB); `pc = br_target_buffer->get_target(inst->pc)`. Otherwise, the PC remains PC+4. `inst->pred_target` stores the predicted branch target address.
  - If Kite does not use branch prediction (which is not the case in this assignment), PC is set to zero to halt instruction fetch.
- **Execute stage:** Resolving branch condition and branch target
  - In `void proc_t::execute()`, the ALU runs an instruction; `alu->run(inst)`.
  - Detailed ALU operations are in the `void alu_t::run(inst_t *m_inst)` function of `alu.cc`.
  - Near the bottom of the long switch-case statements in `alu.cc`, there are four cases of branch instructions; `op_beq`, `op_bge`, `op_blt`, and `op_bne`.
  - For instance, a `beq` instruction checks if its `rs1` and `rs2` values are equal; `m_inst->rs1_val == m_inst->rs2_val`. If so, the PC-relative address, `m_inst->pc + (m_inst->imm << 1)`, is the branch target. Otherwise, the branch target is `m_inst->pc + 4`.

  - `m_inst->branch_taken` and `m_inst->branch_target` are the actual branch state and target, which can be different from the predicted branch state and target (i.e., `m_inst->pred_taken` and `m_inst->pred_target`) obtained from branch prediction in the fetch stage if the prediction is incorrect.

- **Writeback stage:** Updating the branch predictor and branch target buffer
  - In `void proc_t::writeback()`, a branch instruction updates the branch predictor and BTB based on the actual branch information resolved in the execute stage.
  - It calls `br_predictor->update(inst)` to update the branch predictor. If the branch is taken (i.e., `inst->branch_taken == 1`), a two-bit saturating counter is incremented. Otherwise, the counter is decremented.
  - If the branch is taken, it also updates the BTB by calling `br_target_buffer->update(inst->pc, inst->branch_target)`. It records the branch target address corresponding to the branch instruction's PC in the BTB, which serves as a lookup table (or cache).
  - If the predicted branch target is different from the actual branch address (i.e., `inst->pred_target != inst->branch_target`), the processor pipeline is flushed to get rid of wrong-path instructions, and the PC is set to the correct branch target; `pc = inst->branch_target`.
  - Flushing the processor pipeline removes all instructions staged in pipeline registers (i.e., IF/ID, ID/EX, EX/MEM, and MEM/WB). It also flushes the ALU, and the dependency map of the register file is reset.

- As described, the branch predictor and BTB are accessed twice in the processor pipeline for every branch instruction, once in the fetch stage and another in the writeback stage.

- In the fetch stage, the processor calls `inst->pred_taken = br_predictor->is_taken(inst)` to predict if a branch is taken.

- If the branch is predicted to be taken, `inst->pred_target = br_target_buffer->get_target(inst->pc)` gets a predicted branch target.

- The writeback stage calls `br_predictor->update(inst)` to update the branch predictor based on the actual branch outcome.

- If the branch is taken, `br_target_buffer->update(inst->pc, inst->branch_target)` is called to update the branch target buffer.

- Your work in this assignment is to fill in the four functions above, i.e., `is_taken()` and `update()` functions of the branch predictor and the `get_target()` and `update()` functions of the BTB in `br_predictor.cc`.

2 Implementation

- The lecture slides show the simplest branch predictor called a *last-outcome predictor*. A PC value is mapped to one of the entries in the counter array of the branch predictor, each containing a two-bit saturating counter. Each counter traces the last (or two) behavior(s) of a branch instruction.

- Such a branch predictor can handle biased branch behaviors (e.g., loops) but does not work well with an alternating branch pattern, such as T, NT, T, NT, T, NT, $\cdots$.

- Suppose all two-bit saturating counters are initialized to ones (i.e., weakly not-taken). For the alternating T-NT pattern of T, NT, T, NT, T $\cdots$, the last-outcome predictor achieves 0% branch prediction accuracy since the counter toggles between 1 and 2 and misses every branch prediction.

- Thus, the question addressed in this assignment is *"Can we design a branch predictor that can handle such branch patterns?"* To achieve this, the branch predictor needs to observe longer branch history, not just the last one or two branch outcomes.

- Fig. 1 shows the structure of a two-level branch predictor comprised of a branch history register (BHR) and pattern history table (PHT).

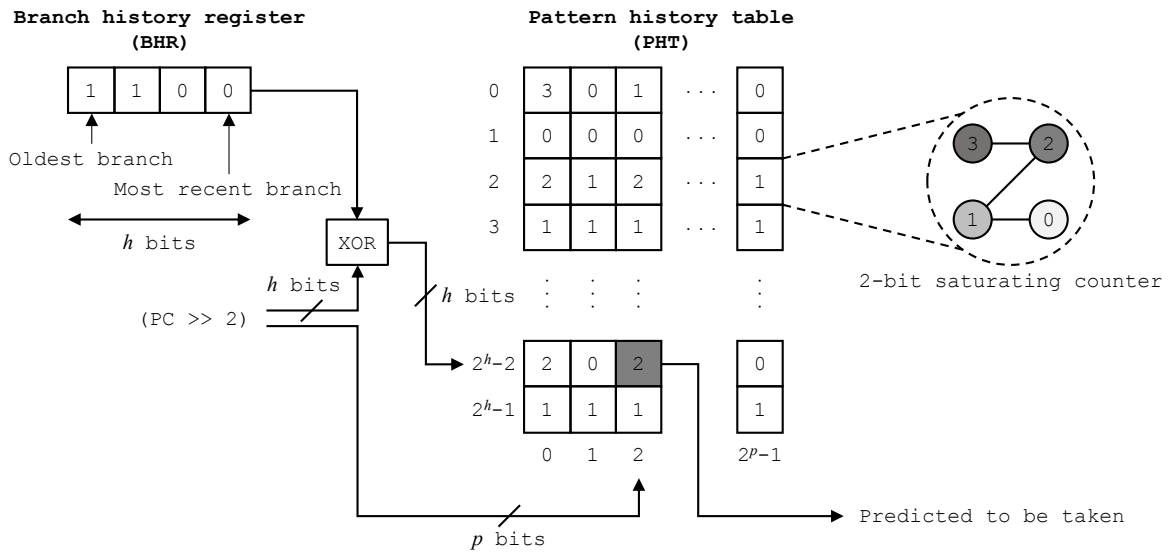

Fig. 1. A two-level branch predictor with a h-bit branch history register and $2^h \times 2^p$-entry pattern history table, where each entry contains a two-bit saturating counter.

- The BHR consists of h bits, and the PHT has $2^h \times 2^p$ entries of two-bit saturating counters. The example is shown for $h = 4$ bits and $p = 4$ bits.
- The least significant bit (LSB) of the BHR is the most recently committed branch outcome, and the most significant bit is the oldest branch result.
- The BHR is XOR'd with `pc >> 2` for indexing PHT rows. For instance, the BHR value of [`1100`] XOR'd with the PC of [$\cdots$ `0010 00`] results in the 4-bit index of [`1110`], which points to row #14 of the PHT.
- PHT columns are indexed by the lowest p bits of (`pc >> 2`). For the PC value of [$\cdots$ `0010 00`], it points to column #2 of the PHT.
- The corresponding PHT entry in the example has a counter value of two, so the branch is predicted to be taken.
- When the branch instruction reaches the writeback stage, it updates the branch predictor based on the actual branch result. The BHR is shifted to the left, and the branch state (i.e., 1 for taken or 0 for not-taken) is written to the LSB.
- For instance, if the branch is really taken, the referenced two-bit counter is incremented from 2 to 3, and the BHR becomes [`1001`]. Then, the next branch instruction will reference row-index #9 of the PHT.
- Notably, all branch instructions globally share the same BHR. Such a two-level branch predictor is called a *global predictor*.
- Globally sharing the BHR can cause a branch instruction to accidentally update a different PHT counter than what it referenced in the fetch stage if the PHT is indexed again using the BHR in the writeback stage.
- After a branch instruction reads the BHR for indexing the PHT in the fetch stage, the BHR can be altered by other preceding in-flight branch instructions. Thus, when this instruction reaches the writeback stage, it may see a different BHR value, pointing to a different PHT entry.
- To update the same PHT counter that the branch instruction referenced in the fetch stage, the instruction needs to capture the BHR or PHT information at the moment of branch prediction and carry it along the processor pipeline so that the instruction can access the same PHT counter in the writeback stage.

- You should add a variable to `class inst_t` in `inst.h` to store the necessary information during branch prediction. When the instruction updates the branch predictor, it uses this carry-on information to identify the right PHT counter instead of re-indexing the PHT with the altered BHR.
- `program_code` contains a RISC-V code for testing. The code is given, and you will not work on assembly programming in this assignment.
- The following shows a part of `program_code` with five branch instructions in doubly nested loops.

```
# Outer loop
            addi    x24,    x10,    0
L1:         bge     x0,     x24,    L7      # Branch
            andi    x25,    x24,    1
            addi    x24,    x24,    -1
            bne     x25,    x0,     L2      # Branch
            jal     x0,     L1
# Inner loop
L2:         addi    x26,    x0,     1
            addi    x27,    x11,    0
L3:         blt     x27,    x26,    L6      # Branch
            andi    x25,    x26,    3
            bne     x25,    x0,     L4      # Branch
            addi    x26,    x26,    1
            jal     x0,     L3
L4:         addi    x25,    x25,    -1
            addi    x26,    x26,    1
            bne     x25,    x0,     L5      # Branch
            addi    x26,    x26,    4
L5:         jal     x0,     L3
# End of the inner loop
L6:         jal     x0,     L1
# End of the outer loop
```

- The first branch instruction (i.e., `bge x0, x24, L7`) is mostly not taken. The second branch (i.e., `bne x25, x0, L2`) is taken with 50% frequency, and the third branch (i.e., `blt x27, x26, L6`) is mostly not taken.
- The fourth branch (i.e., `bne x25, x0, L4`) is taken with 75% frequency, and the fifth branch (i.e., `bne x25, x0, L5`) is taken with 67% frequency.
- The default branch predictor in the skeleton code is an always-not-taken predictor, which returns 0 for all `br_predictor->is_taken(inst)` calls. This predictor shows 54.6% branch prediction accuracy as follows.

```
$ ./kite program_code
...

Start running ...
Done.

======== [Kite Pipeline Stats] =========
Total number of clock cycles = 759835
Total number of stalled cycles = 0
Total number of executed instructions = 488628
Cycles per instruction = 1.555
Number of pipeline flushes = 90401
Number of branch mispredictions = 90401
Number of branch target mispredictions = 0
Branch prediction accuracy = 0.546 (108600/199001)
...
```

- If the branch predictor is implemented to work as a last-outcome predictor with 256 entries in the PHT indexed by instruction PCs (i.e., `pc >> 2`) such as the lecture slides, it achieves 81.7% branch prediction accuracy. You do not have to implement the last-outcome predictor in this assignment, but it will be an interesting test.
- In contrast, the global predictor with h = 4 bits and p = 4 bits achieves 99.9% branch prediction accuracy. Your assignment is to implement a branch predictor with 99.9% prediction accuracy.
- The following shows a simple BTB designed as a direct-mapped cache.

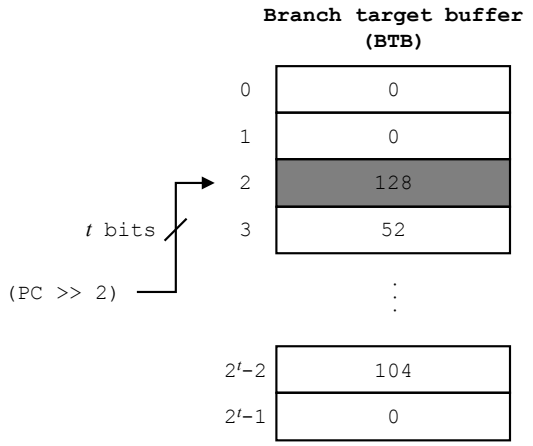

Fig. 2. A 2^t-entry branch target buffer, where each entry contains the branch target address of a taken branch.

- It is indexed by the lowest t bits of `pc >> 2`. The BTB has 16 entries for t = 4 bits in this assignment. Each entry of the BTB stores the target address of a taken branch.
- For instance, if a branch instruction has the PC of [$\cdots$ `0010 00`], the lowest four bits of `pc >> 2` points to index #2 of the BTB. The predicted branch target address is 128 in the example.
- The following shows the definition of `class br_predictor_t` in `br_predictor.h`, representing a branch predictor.

```
// Branch predictor
class br_predictor_t {
public:
    br_predictor_t(unsigned m_hist_len, unsigned m_pht_bits);
    ~br_predictor_t();

    bool is_taken(inst_t *m_inst);      // Is a branch predicted to be taken?
    void update(inst_t *m_inst);        // Update a prediction counter.

private:
    uint8_t  bhr;                       // Branch history register (BHR)
    uint8_t *pht;                       // Pattern history table (PHT)
    unsigned h;                         // h-bit branch history
    unsigned p;                         // p bits for PHT indexing
};
```

- `br_predictor_t(unsigned m_hist_len)` and `~br_predictor_t()` are the constructor and destructor functions of the C++ class. These functions are automatically called when the class is created or deleted.
- `is_taken()` is called in the fetch stage to make branch prediction based on an instruction's PC (i.e., `m_inst->pc`).
- The writeback stage calls `update()` to update the branch predictor based on the actual branch state (i.e., `m_inst->branch_taken`).

- The BHR is defined as a `uint8_t` variable, but you only need the lowest four bits of the variable for $h = 4$ in this assignment.
- The PHT is defined as a `uint8_t` array, where each entry represents a branch prediction counter. Since a two-bit saturating counter has a value between 0 and 3, it only uses the lowest two bits of `uint8_t`.
- The following shows a branch predictor implementation in `br_predictor.cc`. Your assignment is to fill in the empty functions of this file.

```
// Branch predictor
br_predictor_t::br_predictor_t(unsigned m_hist_len, unsigned m_pht_bits) :
    bhr(0),
    pht(0),
    h(m_hist_len),
    p(m_pht_bits) {
    // Create a pattern history table (PHT).
    pht = new uint8_t[(1 << h) * (1 << p)];
}

br_predictor_t::~br_predictor_t() {
    // Deallocate the PHT.
    delete [] pht;
}

// Is a branch predicted to be taken?
bool br_predictor_t::is_taken(inst_t *m_inst) {
    // Predict always not taken.
    return false;
}

// Update a prediction counter.
void br_predictor_t::update(inst_t *m_inst) {
    bhr = 0;
}
```

- The first function is the C++ constructor of the `br_predictor_t` class. This function is called when a branch predictor is created as `br_predictor = new br_predictor_t(4, 4)` in `proc.cc`.
- `pht = new uint8_t[(1 << h) * (1 << p)]` creates a $2^h \times 2^p$-entry array, which is the same as `pht = (uint8_t*)malloc((1 << h) * (1 << p)` in the C language. The only thing you have to add to this function is to initialize the `pht` array.
- The C++ destructor of the `br_predictor_t` class is called when the branch predictor is destroyed at the end of the Kite simulation. `delete [] pht` is the same as `free(pht)` in the C language. This function is complete, and you do not have to work on it.
- The skeleton code of `is_taken()` simply returns `false`, which makes it work as the always-not-taken predictor. You have to update the code to implement the global predictor with $h = 4$ bits and $p = 4$ bits.
- The `is_taken()` function is supposed to make a branch prediction based on an instruction's PC and the BHR. The lowest four bits of `bhr` is XOR'd with that of `m_inst->pc >> 2`, which is used for indexing the rows of the `pht` array. The columns of the `pht` array are indexed by the lowest four bits of `m_inst->pc >> 2`.
- You need to store either the `bhr` value or the `pht` index in `m_inst` to update the right counter later in the `update()` function. Since `pht` is created as a 1-D array, it should be an array index into the 2-D table.
- If a PHT entry contains a counter value of two or greater, the branch is predicted to be taken, and the function returns 1. Otherwise, the branch is predicted to be not taken, and the function returns 0.
- The `update()` function updates `bhr` and `pht`. The BHR is shifted left by one bit, and the branch state of the instruction (i.e., `m_inst->branch_taken`) is recorded in the lowest bit of the BHR.

- The `update()` function also updates a PHT counter. However, the PHT cannot be indexed in the same way as `is_taken()` because the BHR value could have been changed since the branch instruction referenced it. This is where you need the `bhr` value or the `pht` index stored in `m_inst` during the `is_taken()` call.
- The skeleton code meaninglessly writes 0 to `bhr` to avoid a compiler warning, which complains that the variable is never used. You should update the code to implement a properly working global predictor.
- The following shows the definition of `class br_target_buffer_t` in `br_predictor.h`, representing a branch target buffer.

```
// Branch target buffer
class br_target_buffer_t {
public:
    br_target_buffer_t(uint64_t m_size);
    ~br_target_buffer_t();

    uint64_t get_target(uint64_t m_pc);                  // Get a branch target address.
    void update(uint64_t m_pc, uint64_t m_target_addr); // Update the BTB.

private:
    uint64_t num_entries;                                // Number of BTB entries
    uint64_t *buffer;                                    // Target address array
};
```

- `br_target_buffer_t(uint64_t m_size)` and `~br_target_buffer_t()` are the C++ constructor and destructor of the class.
- The `get_target()` function is called in the fetch stage to make a branch target prediction based on an instruction's PC in the case the branch is predicted to be taken.
- The writeback stage calls the `update()` function to update the BTB with the actual branch target address.
- `num_entries` is the number of entries in the BTB, which is 16 in this assignment. The BTB is defined as a `uint64_t` array, where each entry stores the 64-bit address of a branch target.
- The following shows the BTB implementation in `br_predictor.cc`.

```
// Branch target buffer
br_target_buffer_t::br_target_buffer_t(uint64_t m_size) :
    num_entries(m_size),
    buffer(0) {
    // Create a direct-mapped branch target buffer (BTB).
    buffer = new uint64_t[num_entries];
}

br_target_buffer_t::~br_target_buffer_t() {
    // Deallocate the target address array.
    delete [] buffer;
}

// Get a branch target address.
uint64_t br_target_buffer_t::get_target(uint64_t m_pc) {
    // Always return PC = 0.
    return 0;
}

// Update the branch target buffer.
void br_target_buffer_t::update(uint64_t m_pc, uint64_t m_target_addr) {
}
```

- The first function is the C++ constructor of the class, which is called on creating a BTB; `br_target_buffer = new br_target_buffer_t(16)` in `proc.cc`.
- `buffer = new uint64_t[num_entries]` creates a `uint64_t` array, which is the same as `buffer = (uint64_t*) malloc(num_entries * sizeof(uint64_t))` in the C language. The only thing you have to add to this function is to initialize `buffer` entries.
- The destructor function is called when the BTB is destroyed at the end of the Kite simulation. The function deallocates `buffer`, which is the same as `free(buffer)` in the C language. This function is complete, and you do not have to work on it.
- `get_target()` predicts a branch target address. This process looks up the `buffer` array based on an instruction's PC (i.e., `m_pc >> 2`). It returns a branch target address stored in the identified BTB entry.
- Since the branch predictor of the skeleton code is the always-not-taken predictor, the BTB is not really used. The skeleton code meaninglessly returns 0 to avoid a compiler error. You should update the function to implement a properly working BTB.
- The `update()` function updates the BTB based on a branch instruction's PC (i.e., `m_pc`) with the actual branch target (i.e., `m_target_addr`). Unlike the branch predictor, BTB indexing is not affected by other branch instructions. It can be indexed in the same way as the `get_target()` function.

3 Submission

- After implementing the branch predictor and BTB, run the program code in Kite. If correctly implemented, the global predictor should give 99.9% branch prediction accuracy for the program code.

```
$ ./kite program_code
...

Start running ...
Done.

======== [Kite Pipeline Stats] =========
Total number of clock cycles = ...
Total number of stalled cycles = 0
Total number of executed instructions = ...
Cycles per instruction = ...
Number of pipeline flushes = ...
Number of branch mispredictions = ...
Number of branch target mispredictions = ...
Branch prediction accuracy = 0.999 (.../...)
...
```

- When the assignment is done, execute the `tar.sh` script in the `kite/` directory. It creates a tar file named after your student ID, e.g., `2024143530.tar`. Upload the tar file on LearnUs. Do not rename the file.

```
$ ./tar.sh
$ ls *.tar
2024143530.tar
```

4 Grading Rules

- The following is the general guideline for grading. A 30-point scale will be used for this assignment. The minimum score is zero, and negative scores will not be given. Grading rules are subject to change; a grader may add a few extra rules without notice for a fair evaluation of students' efforts.

  -5 points: The tar file includes redundant tags such as a student name, `hw4`, etc.

-5 points: The code has insufficient comments. Comments in the skeleton code do not count. You must clearly explain what each part of your code does.

-2 points per 1% branch prediction accuracy drop: Every 1% deficit loses 2 points. If your branch predictor achieves 99.9% prediction accuracy, you get full credit. Between 99.0% and 99.9%, 2 points will be taken out, and prediction accuracy between 98.0% and 98.9% will lose another 2 points.

-30 points: No or late submission.

Final grade = F: The submitted tar file is copied from someone else. All students involved in the incidents will get Fs for the final grade.

Final grade = F: The code is tweaked to deceive a grader as if it is a working implementation. Such an attempt is considered unethical and will be seriously penalized.

- Your teaching assistant (TA) will grade your assignments. If you think your assignment score is incorrect, discuss your concerns with the TA. Always be courteous when contacting the TA. If you and the TA do not reach an agreement, elevate the case to the instructor for review of your assignment. Refer to the course website for the contact information of the TA and instructor: `https://casl.yonsei.ac.kr/eee3530`
- Arguing for partial credits for no valid reasons will be regarded as a cheating attempt; such a student will lose the assignment scores.