University of Connecticut
CSE 4302 / CSE 5302 / ECE 5402: Computer Architecture
Fall 2024

Graduate Student Project:
RISC-V Simulator: Dynamic Branch Prediction and Data Cache

Due December 13, 2024 @ 11:59 PM on HuskyCT

Ensure that your git repository is up-to-date by executing `git pull` within the `cse4302` directory. This will create a new `project` directory in the repository root that contains the materials for this programming assignment. There are several source code files in the `src` directory, but you will **only** modify `sim_stages.c` for this assignment; **you are not allowed to modify any other files in the `src` and `unittests` directories or the two bash scripts.**

You will modify `sim_stages.c` to implement a fully functional pipelined processor simulator that supports in-order and out-of-order execution with multi-cycle latency for the *riscv-uconn* ISA. Your simulator implementation must be functional and terminate for all unit tests in the `unittests` directory. The `dump_all.sh` script will automatically execute each assembled unit test and gather the required outputs in a single `project_out.txt` file. A `project_out_gold.txt` is provided to you to test the register and memory state output of your assignment. This file does not provide the performance counters data and thus must be ignored using -I '<NAME OF PERFORMANCE COUNTER FLAGS> when using the `diff` utility for comparisons.

The simulator executable for the project accepts following arguments:

```
$ ./simulator OUT_FILE FORWARDING_ENABLE OOO_ENABLE DP_ENABLE DCACHE_ENABLE
```

- `FORWARDING_ENABLE` enables/disables data forwarding in the pipline. For the project, this flag is enabled (set to 1) and implements all logic needed for `PA4`.
- `OOO_ENABLE` enables/disables the Out-of-Order execution of Instructions. For the project, this flag is enabled (set to 1) and implements all logic needed for `PA4`.
- `DP_ENABLE` uses a value of 0 to indicate the baseline not-taken static predictor used in `PA4`. When set to 1, it implements the 1-level dynamic branch predictor, and when set to 2, it implements the 2-level dynamic branch predictor.
- `DCACHE_ENABLE` uses a value of 1 to indicate the use of direct-mapped data cache in the **execution_ld_st** and **execution_2nd_ld_st** stages. When set to 2, it implements the 2-way set associative cache, and when set to 3, it implements the 4-way set-associative cache. Note, that a value of 0 indicates the baseline operation of `PA4`, where no data cache is implemented.

1 Dynamic Branch Prediction

Previously, B-Type instructions utilized a static not-taken (`pc` + 4) branch predictor in the fetch stage, which invoked a large penalty in the case of a taken branch. This project extends the capability of branch prediction by introducing Branch Target Buffer (BTB) and a Branch History Table (BHT) for a 1-level dynamic branch predictor, and with the addition of a Branch History

Shift Register (BHSR) for the 2-level dynamic branch predictor. These components work together to enable branch predictions during the fetch stage. Specifically, the BTB calculates the branch target address in both dynamic prediction schemes. However, for 1-level direction prediction uses the BHT, whereas the 2-level use the BHSR to index into the BHT for branch direction prediction. When the branch resolves in the execute stage, the BTB, BHT, and BHSR are updated.

Each entry of the BTB includes the following metadata stored in the **BranchTargetBuffer** Struct. The `inst_addr` is the instruction address and `inst_addr` is the branch target address. The `valid` indicates whether the BTB entry is valid.

The BHSR is a 5-bit shift register that is used to record branch pattern history for branch instructions.

Each BHT entry contains a two-bit predictor, and has 4 states: $\{00\} \leftrightarrow$ N, $\{01\} \leftrightarrow$ NT, $\{10\} \leftrightarrow$ TN, and $\{11\} \leftrightarrow$ T. These mappings are provided in the `sim_core.h` file as the `PREDICTION enum`. The state machine scheme that you must use is given in 1.1.

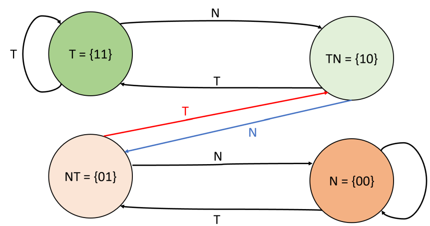
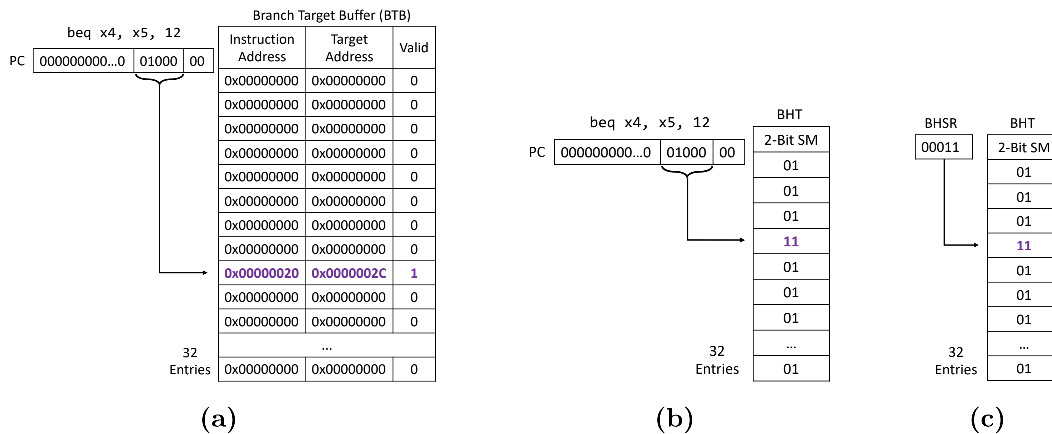


Figure 1.1: Stage machine logic for the two-bit predictor entries of `bht[]`.

Figure 1.2 shows the data structures used for the dynamic branch predictors.



(a) (b) (c)

Figure 1.2: Data structures used in the dynamic branch predictors. (a) 32-entry BTB, (b) 32-entry BHT in 1-level Branch Predictor, and (c) 5-bit BHSR and 32-entry BHT in 2-level Branch Predictor.

Each structure is implemented as follows: `btb[]` is an array that holds 32 entries of the BTB. `bht[]` is also a 32-entry array that holds the 2-bit direction predictor for each entry. The `btb[]` entries are initialized to zero. The `bht[]` predictors are initialized to $\{00\}$ (N), or the "strongly not

taken" state. For 1-level predictor both `btb[]` and `bht[]` are both indexed by the lower bits of an instruction's address (direct mapping). Keep in mind that the two lowest bits of each instruction address will always be 0 (since PC is only incremented in multiples of 4). You should not use these lowest two bits in the index calculation. To index into the 32-entry `btb[]` and `bht[]`, bits 2-6 need to be used from the program counter.

For 2-level predictor, the `btb[]` is indexed using the lowest 5 bits in the `bhsr` variable. **Note: Since the `bhsr` variable itself is larger than 5 bits, you will always truncate the upper bits to '0' to avoid indexing outside of the `bht[]` capacity.**

1.1 Branch Prediction Functions

The dynamic branch predictor is used when `branch_prediction_enabled` flag is set to 1 for 1-level and 2 for 2-level predictor. The dynamic branch predictors must use the following functions: `BTB_lookup()`, `BTB_target()`, `BTB_update()`, `predict_direction()`, and `direction_update()`.

`BTB_lookup()` takes an instruction's address as the input. The input instruction address determines if the indexed BTB entry's instruction address matches the input instruction address, and if the entry is valid, 1 is returned (BTB hit). Otherwise, it must return 0 (BTB miss).

`BTB_target()` takes an instruction's address as the input and returns the branch target address stored by that BTB entry.

`BTB_update()` takes an instruction's address and branch target address as inputs. It sets the instruction address, branch target address, and valid bit to '1' in the corresponding entry.

`predict_direction()` For a 1-level predictor the function takes an instruction address to index into the BHT, whereas for a 2-level predictor BHSR is used to index into the BHT and the corresponding prediction bits are checked as shown in 1.1. If the bits indicate "T" or "TN", then this function returns '1' (for predict branch taken). Otherwise, this function returns '0' (for branch not taken).

`direction_update()` The function takes the instruction address and the branch direction as the input ('1' for taken, '0' for not taken). For 1-level predictor the instruction address is used to index into the BHT for update, whereas for 2-level predictor the BHSR is used to determine which BHT entry is being updated. The state machine logic performs state transitions based on the actual branch direction using 1.1.

For a 2-level predictor, the BHSR is also updated after updating the BHT. The input direction bit is shifted into the BHSR by shifting the BHSR to the left by 1 and bitwise ORing the least significant bit with the direction bit. **Remember that only bits 4 to 0 are used to record branch history, as the BHT has only 32 entries. Upper bits beyond bit 4 must be cleared to '0'.**

1.2 Pipeline Modifications

The dynamic branch predictors are used when `branch_prediction_enabled` flag is set to 1 or 2. Otherwsie, the default not-taken static predictor must be used. The State structs are also modified to include a `br_predicted` flag that may be used to identify if the dynamic predictor is being used for a given instruction's branch prediction. Following modifications are needed in each stage to incorporate the dynamic branch predictors:

In the **fetch** stage, if `branch_prediction_enabled` is set to 1 or 2, then `BTB_lookup()` is used to

check if the current instruction has a valid entry in the BTB. If so, then `predict_direction()` is used to check the branch direction prediction. For a taken branch, the function `BTB_target()` is used to get the target address, and the `br_predicted` flag set to '1' to indicate a taken branch predicted instruction. Otherwise, this field should be '0'.

In case of the dynamic predictors being enabled, the BTB hit and the BHT direction of 'taken' uses the BTB target address to update the `pc_n` using advance_pc() function. Otherwise, the `pc_n` is incremented using the not-taken path.

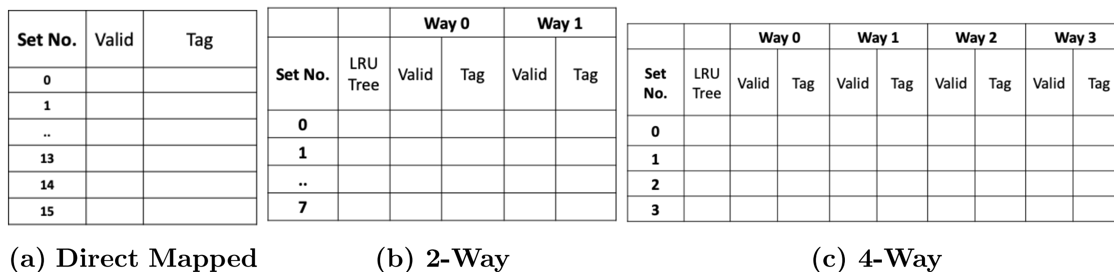
In the **execute** stage, for all B-Type instructions (BEQ, BNE, BLT, BGE), if the resolved branch direction ("taken" or "not taken") does not match what was predicted in fetch, then the branch is mispredicted, and sets the `br_mispredicted` variable. Also, if the branch is mispredicted in the fetch stage, then `pc_n` is set to the corrected address to recover the pipeline. In addition, if `branch_prediction_enabled` is set, then the `BTB_update()` and `direction_update()` functions are called to update the BTB and BHT.

Finally, all B-Type instructions must increment the `total_branches` variable, while only correctly predicted branches increment the `correctly_predicted_branches` variable. **Note: You should still be incrementing `correctly_predicted_branches` even when `branch_prediction_enabled` is 0; the prediction will just always be `pc_n` = `pc` + 4, like in previous assignments.**

2 Data Cache

For this project, the `dcache_enabled` flag indicates one of the three options for a data cache that is concurrently accessible in both **execution_ld_st** and **execution_2nd_ld_st** stages. A cache hit incurs `dcache_access_cycles` (fixed to 2) cycles, while cache misses incurs `dmem_access_cycles` (fixed to 6) cycles. **Note: when the data cache is disabled, every memory access will behave as a "cache miss", taking 6 cycles every time.**

The cache has a total size of 256 bytes and cache block contains 4 data words (or 16 bytes). This implies 16 cache sets for direct-mapped, 8 sets for 2-way set associative, and 4 sets for 4-way set-associative cache configuration. Figure 2.1 shows the structure layout of the data cache tag array for each configuration.



Set No.	Valid	Tag
0		
1		
..		
13		
14		
15		

(a) Direct Mapped

		Way 0		Way 1	
Set No.	LRU Tree	Valid	Tag	Valid	Tag
0					
1					
..					
7					

(b) 2-Way

		Way 0		Way 1		Way 2		Way 3	
Set No.	LRU Tree	Valid	Tag	Valid	Tag	Valid	Tag	Valid	Tag
0									
1									
2									
3									

(c) 4-Way

Figure 2.1: Data cache layout.

Each set is represented by `CacheSet_DM`, `CacheSet_2_way`, or `CacheSet_4_way` type Struct for direct-mapped, 2-way, and 4-way set associative configurations in `sim_core.h`.

The `CacheSet_DM` contains a single cache block of type `CacheBlock` that stores the valid and tag bits for the cache block. The `dcache_DM` is initialized for the 16 sets in `sim_core.c`.

The `CacheSet_2_way` contains two cache blocks of type `CacheBlock` that store the valid and tag bits for each cache block. The `lru_tree` uses 1 bit to indicate the "least recently used" block in the set based on the true LRU scheme for the 2-way set associative cache. The `dcache_2_way` is initialized for the 8 sets in `sim_core.c`.

The `CacheSet_4_way` contains four cache blocks of type `CacheBlock` that store the valid and tag bits for each cache block. The `lru_tree` uses 2 bits to indicate the "least recently used" block in the set based on a pseudo 2-bit LRU scheme for the 4-way set associative cache. The `dcache_4_way` is initialized for the 4 sets in `sim_core.c`.

Note that the `CacheBlock` type does not actually contain the cached data. Instead, cached data is returned directly from the `memory[]` array when the cache block valid and tag bits indicate that a given memory address is in the block. On simulator start-up, the cache is initialized such that the the lru, valid, and tag bits are all are zero.

The cache implementation must be completed by implementing the `dcache_lookup_DM()`, `dcache_lookup_2_way()` or `dcache_lookup_4_way()` and `dcache_update_DM()`, `dcache_update_2_way()` or `dcache_update_4_way()` functions for direct-mapped, 2-way, or 4-way set associative cache in `sim_stages.c`. You must parse the appropriate bits from the memory address to determine the cache set *index* and block *offset*. The remaining most significant bits of the memory address should be tracked as *tag* bits. Although the offset bits are not directly used, they are needed to compute the index and tag bits.

The instruction State struct is modified to support `cache_line_hit_way` variable to track the assigned cache line hit during execution for a 2-way and a 4-way cache. This flag value of -1 implies that there was a cache miss whereas 1,2,3,4 or 1,2 determines cache hit way for a 4-way or a 2-way cache respectively. This variable is passed to the `dcache_update_2_way()` or `dcache_update_4_way()` functions for 2-way or 4-way Set Associative Cache.

2.1 Data Cache Functions

For a set associative cache the `dcache_lookup_2_way()` or `dcache_lookup_4_way()` function takes a memory address as input and extracts the index bits to determine which set to access. Within the set, if for *any* of the blocks (i) it is valid and (ii) the tag bits in the block match the tag bits of the input memory address, the function must return the way within the set that is a hit (either 0, 1, 2, or 3 for 4-way or 0 or 1 for 2-way). Otherwise, the function returns -1, which indicates a cache miss.

For a Direct Mapped Cache, the `dcache_lookup_DM()` function takes a memory address as input and extracts the index bits to determine which set to access. Within a set if the block (i) is valid and (ii) the tag bits in the block match the tag bits of the input memory address, the function must return a hit (0). Otherwise, the function returns -1, which indicates a cache miss.

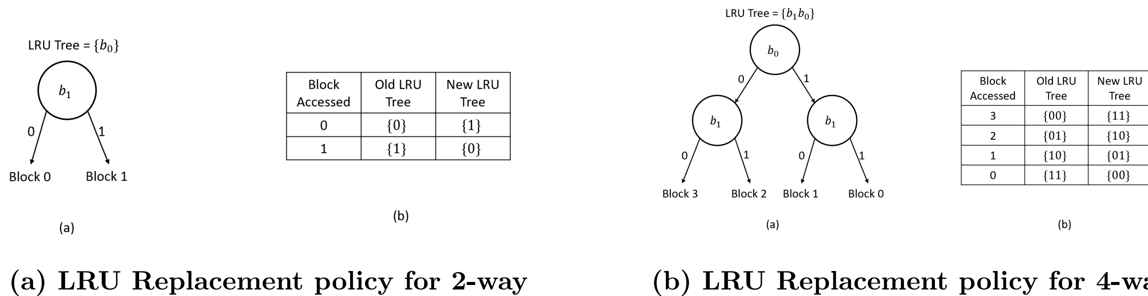
For a set associative cache, the `dcache_update_2_way()` or `dcache_update_4_way()` function takes a memory address and `cache_line_hit_way` as input and extracts the index bits from the address to determine which set to access. Then, it determines which block is accessed and needs to be updated. Using the `cache_line_hit_way`, it must check if the current memory address is already in the cache. If `cache_line_hit_way` indicates a hit, then that is the selected block to be updated. Otherwise, the current memory address is not in the cache, and the block it needs to access is determined as described below.

First, the function must check if there are any invalid blocks (starting with block 0). If an invalid block is found, then that is the selected block to be updated. Otherwise, a valid cache block needs to be evicted.

For a 4-way set associative cache the `lru_tree` bits are used to make this decision. First, bit 0 (the LSB) is checked. If it is '0', then bit 1 is checked. If bit 1 is '0' then block 3 is selected as the block to be updated, otherwise block 2 is selected. If bit 0 is '1', then bit 1 is checked instead. If bit 1 is '0', block 0 is the selected block to be updated, otherwise block 1 is selected.

For a 2-way set associative cache the `lru_tree` bits are used to make this decision. Bit 0 (the LSB) is checked. If it is '0', then block 0 is selected as the block to be updated, otherwise block 1 is selected.

After doing these checks, the `lru_tree` must be updated regardless of if the memory address was already in the cache or not. Figure 2.2 illustrates the process of replacing a valid block and updating the LRU bits.



Block Accessed	Old LRU Tree	New LRU Tree
0	{0}	{1}
1	{1}	{0}

Block Accessed	Old LRU Tree	New LRU Tree
3	{00}	{11}
2	{01}	{10}
1	{10}	{01}
0	{11}	{00}

(a) LRU Replacement policy for 2-way **(b) LRU Replacement policy for 4-way**

Figure 2.2: LRU Replacement policy

2.2 Pipeline Modifications

Both the **execute** stages of Load Store (`execute_ld_st() and execute_2nd_ld_st()`) must be modified to model the memory and cache access latency of load and store instructions, along with the implementation of the data cache. The memory access latency is determined by the `dmem_access_cycles` control variable and is set to 6, while cache access latency is determined by the `dcache_access_cycles` control variable and is set to 2. These variables determine **the total number of cycles spent in the memory stage** for a cache miss or hit, respectively. The function `dcache_lookup_*()` (based on cache configuration) is used when cache is enabled to determine if a given memory address is in cache.

The `dmem_busy` and/or `dmem_busy2` flags are only asserted for cycles where it evaluates true as described below. When the cache is disabled, or there is a cache miss with cache enabled, the `dmem_access_cycles` variable will be used as the latency to model reading from main memory. Otherwise when there is a cache hit, `dcache_access_cycles` is used to model the latency of accessing the cache.

Regardless of a cache hit or miss, `dmem_cycles` and/or `dmem_cycles2` are used to count the number of cycles the memory stage has been accessing data. Whenever `dmem_cycles` and/or `dmem_cycles2` is less than the incurred latency, `dmem_busy` and/or `dmem_busy2` must be asserted and the memory stage must return the structure. Asserting `dmem_busy` and/or `dmem_busy2` ensures that the pipeline stalls whenever the memory stage is stalled waiting for a data access to complete.

When `dmem_cycles` and/or `dmem_cycles2` are equal to the incurred latency, dmem busy and cycles

flags are cleared and the memory / cache access can proceed as usual. Also, `dcache_update_*()` must be called at this point if cache is enabled, so that the set `lru_tree` is updated, and the corresponding block's tag and valid bits are updated.

The load/store execution stages must also track data cache accesses and hits for statistics purposes. Load and store instructions must increment the `dmem_accesses` variable for all load and store instructions after the incurred latency has passed and the instruction can proceed. Similarly, load and store instructions must increment the `dcache_hits` variable on a cache hit. **Note: load and store instructions should only increment `dcache_hits` and `dmem_accesses` at most once; stalls due to memory latency should not count as a cache hit / memory access.** When the data cache is disabled, `dcache_hits` should always be 0.

3 Deliverable

In addition to the `sim_stages.c` and `project_out.txt` files, prepare and submit a PDF report explaining the branch misprediction rate and the cache hit rate for the different test cases provided to you for different configurations of Branch Predictor and Cache. Explain why a certain configuration works better for each test case in your report.