University of Connecticut
CSE 4302: Computer Architecture
Fall 2022

Programming Assignment 3:
5-Stage Pipeline w/ Dynamic Branch Prediction and Data Cache

Due December 6th, 2022 (Tuesday) @ 11:59 PM on HuskyCT

Introduction

In this programming assignment, you will add dynamic branch prediction and a set associative data cache to the 5-stage pipelined *riscy-uconn* simulator.

First, ensure that your git repository is up-to-date by executing `git pull` within the `cse4302` directory. This will create a new `pa3` directory in the repository root that contains the materials for this programming assignment. The directory and file structure of `pa3` is the same as `pa2`. **As before, you may only modify `sim_stages.c`.**

To receive full credit for this assignment, your simulator implementation must be fully functional and correct **(i) with and without forwarding, (ii) with and without the data cache, and (iii) with and without dynamic branch prediction** for all the unit tests in the PA3 `unittests` directory as well as for the unit tests of all previous assignments.

Although only a subsest of the above may be used for grading, the ability to enable and disable certain features should aid debugging, allow you to periodically test the code, and isolate parts as needed.

Once you have completed this programming assignment, submit your `sim_stages.c` file via HuskyCT by the posted deadline. **Note that unlike PA1 and PA2, an in person code review is not necessary to receive credit for PA3. Only a submission of your file is needed.**

1 5-Stage Pipelined Simulator with Dynamic Branch Prediction and Data Cache

The objective of this programming assignment is to extend the 5-stage pipelined simulator implementation of PA2 with (i) a BTB and direction predictor for BEQ and BNE instructions and (ii) A two-way set associative data cache with multi-cycle memory operations.

1.1 Simulator Structure

The simulator structure is mostly the same as the pipelined simulator of PA2 with the following differences:

1. When instructions are fetched from memory, they are now stored in a `State` structure, like the other stages. The arguments and return values of the `fetch()` and `decode()` functions have been updated accordingly. The only fields set in the fetch stage are `inst` and the new `inst_addr` field (described below). All other fields are set after fetch (as with PA1 and PA2).
2. The `State` structure has the new field `inst_addr`. This field will be set to the memory address of the instruction in the fetch stage (the `pc` value that fetches the instruction from memory). In the following stages, instructions must use this new field instead of `pc` directly if needed for calculating branch/jump addresses.
3. Multi-cycle operation control variables for the data cache have been added to `sim_core.h` and are listed in Figure 1. The subsequent sections detail when and how to use these new control variables.

Multi-cycle Memory Control Variables
`dmem_access_cycles` (read-only)
`dmem_busy`
`dmem_cycles`

Figure 1: Multi-cycle operation control variables.

4. An incomplete single-cycle data cache implementation has been added to the simulator. The data cache may be enabled or disabled based on the new program argument described in the next section. More details regarding the data cache implementation can be found in the subsequent sections.
5. Data cache control variables have been added to `sim_core.h` and are listed in Figure 2. The subsequent sections detail when and how to use these new control variables.

Data Cache Control Variables
`dcache_enabled`
`dcache_accesses`
`dcache_hits`

Figure 2: Data cache control variables.

6. Two data cache functions `dcache_lookup()` and `dcache_update()` have been added to `sim_stages.c`. The subsequent sections contain more details regarding data cache functionality.
7. The simulator now creates a `cdump.txt` file that outputs the status of the cache at the end of simulation. The simulator also prints data cache accesses and hit statistics (corresponding to `dcache_accesses` and `dcache_hits` respectively) at the end of each simulation.
8. An incomplete dynamic branch prediction implementation has been added to the simulator. The dynamic branch predictor may be enabled or disabled based on the new program argument described

in the next section. More details regarding the dynamic branch prediction implementation can be found in the subsequent sections.

9. Branch prediction control variables have been added to `sim_core.h` and are listed in Figure 3. The subsequent sections detail when and how to use these new control variables.

Branch Prediction Control Variables
`branch_prediction_enabled`
`total_branches`
`correctly_predicted_branches`

Figure 3: Branch prediction control variables.

10. Four dynamic branch prediction functions have been added: `BTB_lookup()`, `BTB_target()`, `predict_direction()`, and `predictor_update()`. The subsequent sections contain more details regarding dynamic branch prediction implementation.
11. The simulator now creates a `bpdump.txt` file that outputs the status of the BTB and direction predictor at the end of the simulation. The simulator also prints the total number of conditional branches, and the number of those branches predicted correctly (corresponding to `total_branches` and `correctly_predicted_branches` at the end of each simulation.
12. The logic responsible for updating the pipeline state has been moved from `sim_core.c` into a new `update_simulator_state()` function located in `sim_stages.c`. You **SHOULD NOT** modify it, but it will be essential to understand how it stalls the pipeline for memory operations. The subsequent sections contain more details regarding this function.

1.2 Extra Program Arguments

The simulator executable now accepts 3 program arguments that independently control whether (i) forwarding (ii) the data cache and (iii) dynamic branch prediction are enabled:

```
$ ./simulator OUT_FILE FORWARDING_ENABLED DATA_CACHE_ENABLED DYNAMIC_BP_ENABLED
```

where `OUT_FILE` is an assembled *riscy-uconn* program, `FORWARDING_ENABLED` is either 0 or 1 to indicate that forwarding is disabled or enabled respectively, `DATA_CACHE_ENABLED` is either 0 or 1 to indicate that the data cache is disabled or enabled respectively, and `DYNAMIC_BP_ENABLED` is either 0 or 1 to indicate that dynamic branch prediction is enabled or disabled respectively.

The data cache and branch prediction status is visible in the `sim_stages.c` file through the `dcache_enabled` and `branch_prediction_enabled` variables declared in `sim_core.h`.

1.3 Overview of All Changes

At a high level, this programming assignment involves 3 changes to the simulator: (i) Adding dynamic branch prediction for `BNE` and `BEQ` instructions; (ii) changing LW and SW instructions to take 5 total cycles in the memory stage to simulate memory access latency; and (iii) implementing a single-cycle two-way set assosiative data cache to improve the memory access latency of LW and SW instructions in the memory stage. The following sections describe the required modifications to the simulator. Each stage should use your PA2 implementation as its starting point. It is assumed that your PA2 implementation is correct.

1.4 Multi-cycle Memory Stages

1.4.1 Overview

The memory stage must be modified to model the memory access latency of LW and SW instructions. The memory access latency is determined by the `dmem_access_cycles` control variable. Since this variable is set to 5 cycles, the memory stage must stall for this many cycles before allowing LW or SW instructions to progress to the write-back stage.

1.4.2 Memory Related Pipeline Changes

The decode stage must be modified so that `pipe_stall` is asserted and a `NOP` is returned when `dmem_busy` (discussed in the next paragraph) is asserted. This ensures that the decode and fetch stages stall when the memory stage stalls. Moreover, when `dmem_busy` is asserted, the decode function must return before any modifications are made to the `pc_n` or `j_taken` variables. This ensures that the program counter does not update and the jump logic does not introduce any side effects when the pipeline is stalled.

The 2 new control variables `dmem_busy` and `dmem_cycles` have been added to manage stalling in the memory stage. `dmem_busy` must be de-asserted at the beginning of the memory stage so that is only asserted for cycles where it evaluates true as described below. `dmem_cycles` is used to count the number of cycles the memory stage has stalled. When it is less than `dmem_access_cycles`, `dmem_busy` must be asserted and the memory stage must return the structure. Asserting `dmem_busy` ensures that the pipeline stalls whenever the memory stage is stalled waiting for a data access to complete. When `dmem_cycles` is equal to `dmem_access_cycles`, `dmem_busy` and `dmem_cycles` are cleared and the memory access can proceed as usual. The memory stage must also track data cache accesses and hits for statistics purposes. LW and SW instructions must update the `dcache_accesses` variable when `dmem_cycles` is equal to `dmem_access_cycles` so that each memory access is tracked. When the data cache is disabled, `dcache_hits` should always be 0.

1.4.3 Simulator State

When `dmem_busy` is asserted the `update_simulator_state()` function will update the `pc`, `fetch_out`, `decode_out`, `ex_out`, and `mem_out` state variables so that the pipeline holds the state of the fetch, decode, execute, and memory stages respectively. This is done so that the pipeline stages are able to re-execute their current instruction when the memory stages are stalled. Thus, it important that `dmem_busy` is asserted correctly to avoid unnecccesary stalling.

1.5 Data Cache

1.5.1 Overview

The simulator now features an incomplete data cache implementation that is enabled using the relevant program argument. The cache is two-way set associative and addressed by a 32-bit data memory address. Each set contains 2 cache blocks (lines), each cache block contains 4 data words (or 16 bytes), and the cache has a total size of 256 bytes. This leaves 8 total cache sets. The cache hit latency is fixed at 1 cycle while cache misses incur `dmem_access_cycles` (defined as `5`) cycles. Note that the cache size is fixed, and should not be changed.

The following structures are introduced to represent the data cache:

CacheSet Field	Description
`block[2]`	the set's two cache blocks of type `CacheBlock`
`lru`	the current LRU block for this set

CacheBlock Field	Description
`tag`	stores the tag of the cache block
`valid`	indicates if the block is valid

Each set is represented by the `CacheSet` type defined in `sim_core.h`, and contains (i) an array of type `CacheBlock` representing the 2 ways of the set and (ii) the variable `lru`. The `CacheBlock` type contains the valid and tag bits for each block in the set. `lru` holds the index of the "least recently used" block in the set, which consequently is the block to be evicted in the event of a collision. **Do 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 variables are zero; in other words, the cache is empty on start-up.

The cache implementation must be completed by implementing the `dcache_lookup()` and `dcache_update()` functions in `sim_stages.c`. Each cache block contains 4 words (16 bytes), and each memory address contains 1 word (4 bytes) of data. Consequently, you must decode 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. The below figure illustrates how the data cache structure is implemented:

LRU block?	Way 0		Way 1	
	Valid?	Tag	Valid?	Tag
dcache[0].lru	dcache[0].block[0].valid	dcache[0].block[0].tag	dcache[0].block[1].valid	dcache[0].block[1].tag
…	…	…	…	…
dcache[i].lru	dcache[i].block[0].valid	dcache[i].block[0].tag	dcache[i].block[1].valid	dcache[i].block[1].tag
…	…	…	…	…
dcache[7].lru	dcache[7].block[0].valid	dcache[7].block[0].tag	dcache[7].block[1].valid	dcache[7].block[1].tag

*dcache_lookup()
*dcache_update()

*Extract index bits *i* from the memory address to access the cache set

Figure 4: Visualization of data cache and function accesses.

1.5.2 Data Cache Functions

The `dcache_lookup()` function takes a memory address as input and extracts the cache set index bits to determine which set to access. Within the set, if (i) the tag bits in either of the two blocks match the tag bits of the input memory address and (ii) that cache block is valid, the function must return 1. Otherwise, the function returns 0. In other words, the function returns 1 for a cache hit or 0 for a cache miss.

The `dcache_update()` function takes a memory address as input and extracts the cache set index bits to determine which set to access. Then, you must check if block 0 (i) contains the decoded tag or (ii) is invalid. If either of these two conditions are true, then the valid bit of the block is set to 1 (if invalid), the tag bits are set appropriately (if invalid), and the LRU bit must be set to the index of the *other* block. If those conditions are not true for block 0, a similar check is done for block 1. If these conditions are untrue for both blocks, then a valid cache block must be evicted for the new one. The cache set identified by `lru` is the victim, and should be replaced with the new tag. Remember to also update the `lru` variable to the block opposite of the victim.

1.5.3 Data Cache Related Pipeline Changes

The `dcache_update()` and `dcache_lookup()` functions are used in the memory stage for LW and SW instructions. When the cache is enabled, LW and SW instructions must perform a cache lookup on the effective memory address to determine if the memory address is a cache hit or miss. If the memory address is a hit, the memory stage does not stall and the memory access proceeds as usual. In this case, `dcache_hits` must be incremented to track the number of cache hits. If the address is a miss, the memory stage stalls as described in the previous section. Regardless of a cache hit or miss, `dcache_accesses` should be incremeneted, and the memory address must be updated in the cache with `dcache_update()` before it is accessed. This is done so that the address is cached for future memory lookups.

1.6 Branch Prediction

1.6.1 Overview

Previously, `BNE` and `BEQ` instructions utalized a static `PC+4` predictor, which invoked a large penalty in the case of of a branch taken. This assignment extends the capability of branch prediction by introducing a branch target buffer (BTB) and a single-level branch history table (BHT). The table below shows the fields of the BTB structure:

BranchTargetBuffer Field	Description
`inst_addr`	the instruction address of the entry
`branch_target`	the branch address of the entry
`valid`	indicates if the entry is valid

The BHT contains a single-bit predictor, and has 2 states: 0 for branch not taken, and 1 for branch taken. `btb[]` is the array that holds each entry of the BTB, and `bht[]` is the array that holds each 1-bit predictor of the BHT. Both the BTB and BHT have 32 entries, and are indexed by the lower bits of an instruction's address. Keep in mind, howevever, that the two lowest bits of each instruction address will always be 0 (since PC is only incremented in multiples of 4). You should avoid using these in the index, as they will not yeild the maximum unique indicies into the BTB/direction predictor. Therefore, you should use the *next* lowest bits besides bits 0 and 1 . Determine which bits you need for the specified number of BTB/BHT entries, and extract those bits from the intruction's address to index those arrays when necessary. Note that the number of entries in the BTB/BHT are fixed and should not be modified. The below figure illustrates the implementation of the BTB and BHT structures, and what functions access each structure:

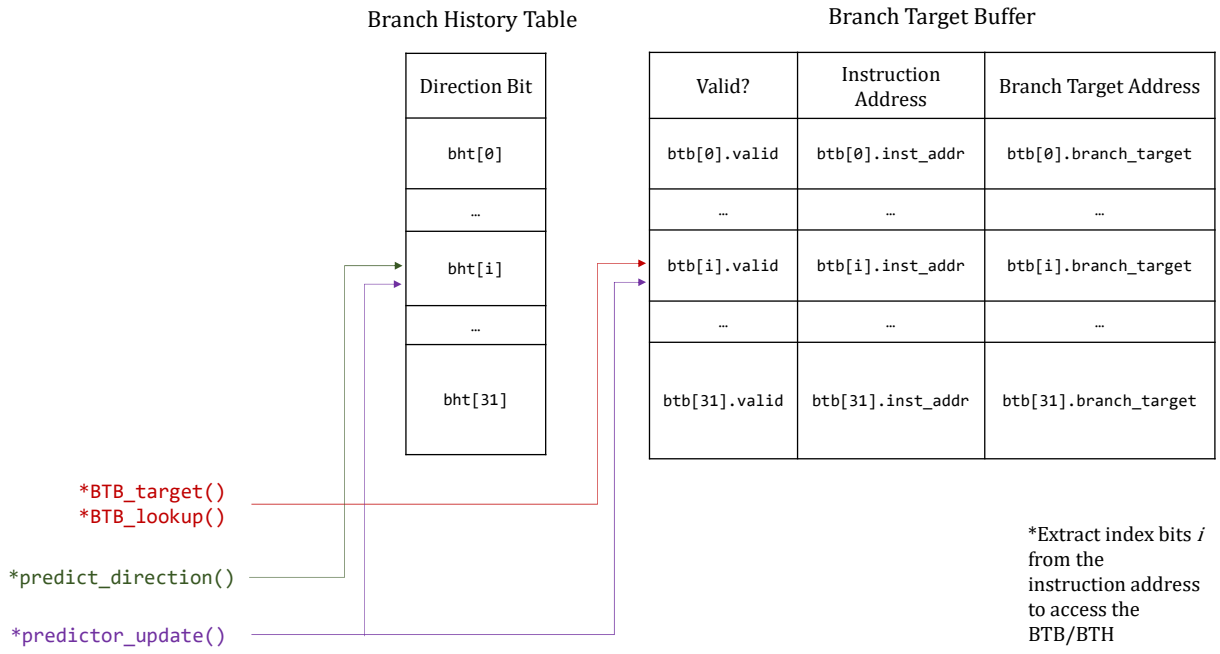

Figure 5: Visualization of branch prediction structures.

1.6.2 Branch Prediction Functions

Branch prediction itself is implemented with the 4 functions: `BTB_lookup()`, `BTB_target()`, `predict_direction()`, and `predictor_update()`.

`BTB_lookup()` takes an instruction's address as the input. The index is extracted from the input instruction address to determine which BTB entry to look up. If (i) the indexed BTB entry's instruction address matches the input instruction address and (ii) the entry is valid, 1 is returned (BTB hit). Otherwise, you must return 0 (BTB miss).

`BTB_target()` takes an instruction's address as the input. The index is extracted from the input instruction address to determine which BTB entry to look up. Then, the function returns the branch target address being stored by that BTB entry.

`predict_direction()` takes an instruction's address as the input. The index is extracted from the input instruction address to determine which direction predictor entry to look up. Then, the corresponding entry is looked up in the BHT. If the prediction bit is a 0, the function should return 0 (branch not taken), and if it is a 1, the function returns 1 (branch taken).

`predictor_update()` takes 3 inputs: the instruction's address, a branch target address, and a branch direction. The index is extracted from the input instruction address to determine which BTB and direction predictor entry to update. The corresponding BTB entry is updated with the instruction address and branch

target address passed as arguments, and should be marked as valid. The corresponding BHT entry should also be updated with the resolved branch direction passed in as an argument.

1.6.3 Branch Prediction Related Pipeline Changes

During the fetch stage, the current instruction (`memory[pc / 4]`) and it's address (`pc`) is saved to the `State` structure in the `inst` and `inst_addr` fields, respectively. Control flow instructions that previously used `pc` to calculate branch/jump addresses in PA2 should now use `inst_addr + 4` instead. The same checks done in PA2 are done for jumps, mispredicted branches, and stalls. After these checks, a branch prediction is made if dynamic branch prediction is enabled. Using the instruction's address, you should use the `BTB_lookup()` function to check if the current address is a BTB hit. Given a BTB hit, the `predict_direction()` function should be used to index the BHT and check what direction is predicted for that instruction address. In the case of a BTB hit and branch taken predicted, `BTB_target()` is used to obtain the next predicted address and is set to `pc_n`. When dynamic branch prediction is disabled, the fetch stage operates the same as it did in PA2.

In the execute stage, only `BNE` and `BEQ` are affected by branch prediction. Once all hazards are resolved for these instructions, if dynamic branch prediction is enabled you must determine (i) if the instruction resolved to a branch taken or not and (ii) if branch taken was predicted in the fetch stage or not. `BTB_lookup()` and `predict_direction()` should be used to determine what prediction was made in the fetch stage. If there is a mismatch between the resolved branch direction and the predicted branch direction, `br_mispredicted` should be set, which will cause instructions in decode and fetch to be flushed as they were in PA2. If a branch was mispredicted, you must set `pc_n` to the correct address depending on what direction the mispreciction was. You should then make a call to `predictor_update()` using the instruction's address, the branch target address, and the resolved direction of the branch. This is to ensure that future fetches of this instruction have an updated entry in the BTB and branch predictor. **It is important that only `BEQ` and `BNE` instructions create an entry in the BTB.**

If dynamic branch prediction is disabled, then `PC+4` will always be statically predicted for conditional branches. In this case, conditional branches behave exactly as they did in PA2. `br_mispredicted` should be set whenever the branch is taken, and `pc_n` should be corrected with the branch target address.

Finally, make sure to increment `correctly_predicted_branches` whenever there was no branch mispredict, and that `total_branches` is incremented for all branches. **Note that a predicted branch outcome can be correct even when dynamic branch prediction is disabled. If a static `PC+4` prediction was correct, you should still increment `correctly_predicted_branches` .**