Introduction

In this programming assignment, you will add dynamic branch prediction and a set associative data cache to the 5-stage pipelined riscv-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 20 unit tests in the PA3 unittests directory (15 from PA2, along with 5 new tests).

Once you have completed this programming assignment, submit your sim_stages.c file via HuskyCT by the posted deadline.

The sections are as follows:

- Section 1 gives an overview of the new simulator features and additions.
- Section 2 explains the additional program arguments introduced to enable / disable the new features.
- Section 3 gives an overview of the data cache, new structures, and functions to be implemented.
- Section 4 gives an overview of dynamic branch prediction, new structures, and functions to be implemented.
- Section 5 explains the necessary pipeline modifications you must make, as well as how to use the control variables listed in section 1 and the structures and functions listed in sections 3 and 4.
1 Overview

1.1 Summary of All Changes

At a high level, this programming assignment involves 3 changes to the simulator: (i) Adding dynamic branch prediction for B-Type instructions; (ii) changing LW/SW instructions to take multiple cycles in the memory stage to simulate memory access latency; and (iii) implementing a four-way set associative 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.

1.2 Simulator Structure

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

1. The State structure has a new field: br_predicted. In the fetch stage, this field will be set to ‘1’ if a branch-taken is predicted, and is otherwise set to ‘0’. This field is used later in execute stage to determine branch mispredictions. Details on using this field are given in section 5.

2. New variables for the data memory and cache have been added to sim_core.h and are listed in Figure 1.1. Section 5 details how an when to use these new control variables.

<table>
<thead>
<tr>
<th>New Memory Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dcache_enabled</td>
<td>Set when the data cache is enabled</td>
</tr>
<tr>
<td>dmem_accesses</td>
<td>Counts the total number of LW/SW instructions processed</td>
</tr>
<tr>
<td>dcache_hits</td>
<td>Counts the total number of LW/SW cache hits</td>
</tr>
<tr>
<td>dmem_access_cycles (read-only)</td>
<td>Number of cycles required to read data from memory</td>
</tr>
<tr>
<td>dcache_access_cycles (read-only)</td>
<td>Number of cycles required to read data from cache</td>
</tr>
<tr>
<td>dmem_cycles</td>
<td>Number of cycles spent processing a LW/SW instruction</td>
</tr>
<tr>
<td>dmem_busy</td>
<td>Set when more cycles are required to process a LW/SW instruction</td>
</tr>
</tbody>
</table>

Figure 1.1: Global variables used to control the pipeline.

3. Two data cache functions dcache_lookup() and dcache_update() have been added to sim_stages.c. Section 3 details how to implement these functions.

4. 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.

5. Branch prediction variables have been added to sim_core.h and are listed in Figure 1.2. Section 5 details when and how to use these new control variables.

6. Five dynamic branch prediction functions have been added: BTB_lookup(), BTB_target(), BTB_update(), predict_direction(), and direction_update(). Section 4 details how to implement these new functions.

7. The simulator now creates a bdump.txt file that outputs the status of the BTB, BHSR, and BHT at the end of the simulation. The simulator also prints the total number of conditional branches and correctly predicted branches (corresponding to total_branches and...
<table>
<thead>
<tr>
<th>New Branch Prediction Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch_prediction_enabled</td>
<td>Set when dynamic branch prediction is enabled</td>
</tr>
<tr>
<td>total_branches</td>
<td>Counts the total number of B-type instructions processed</td>
</tr>
<tr>
<td>correctly_predicted_branches</td>
<td>Counts the number of correctly predicted branch directions</td>
</tr>
</tbody>
</table>

Figure 1.2: Global variables used to control the pipeline.

correctly_predicted_branches) at the end of each simulation. Also, when dynamic branch prediction is enabled, additional BHSR and BHT info is displayed in pipe_trace.txt.

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.

3 Data Cache and Multi-Cycle Latency

3.1 Overview

The simulator now features an incomplete data cache implementation that is enabled using the relevant program argument. The cache is four-way set associative and addressed by a 32-bit data memory address. Each set contains 4 ways for cache blocks (or lines), and each cache block contains 4 data words (or 16 bytes). The cache has a total size of 256 bytes, which leaves 4 total cache sets to access. Cache hits incur dcache_access_cycles (set to 2) cycles, while cache misses incur dmem_access_cycles (set to 6) cycles. Note: when the data cache is disabled, every memory access will behave as a “cache miss,” taking 6 cycles every time. When a memory instruction is not finished processing, then the dmem_busy variable is asserted to cause instructions in previous stages to stall. More details about pipeline changes needed to accommodate the data cache latencies are given in section 5.

The structures in figure 3.1 are introduced to represent the data cache:
<table>
<thead>
<tr>
<th>CacheSet Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>block[4]</td>
<td>The different ways within the cache set of type CacheBlock</td>
</tr>
<tr>
<td>lru_tree</td>
<td>Three bits used to select the LRU block</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CacheBlock Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tag</td>
<td>Stores the tag of the cache block</td>
</tr>
<tr>
<td>valid</td>
<td>Indicates if the block is valid</td>
</tr>
</tbody>
</table>

Figure 3.1: Structures used to represent the data cache.

Each set is represented by the CacheSet type defined in sim_core.h, and contains (i) an array of type CacheBlock representing the 4 ways of the set and (ii) the variable lru_tree. The array dcache[4] implements the four sets in the cache itself.

The CacheBlock type contains the valid and tag bits for each block in the set. lru_tree holds 3 bits to indicate the “least recently used” block in the set, according to the pseudo-LRU algorithm. **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 lru, valid, and tag bits are all are zero. Figure 3.2 shows the structure of the data cache.

```
<table>
<thead>
<tr>
<th>Set No.</th>
<th>LRU Tree</th>
<th>Way 0</th>
<th>Way 1</th>
<th>Way 2</th>
<th>Way 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Valid</td>
<td>Tag</td>
<td>Valid</td>
<td>Tag</td>
<td>Valid</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 3.2: Data cache layout.

The cache implementation must be completed by implementing the dcache_lookup() and dcache_update() functions 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, it is necessary
to determine the offset bits to correctly find the index and tag.

### 3.2 Data Cache Functions

The \texttt{dcache\_lookup()} function takes a memory address as input and extracts the index bits to determine which set to access. Within the set, if for \textit{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). Otherwise, the function returns -1, which indicates a cache miss. This function should be implemented before \texttt{dcache\_update()}.

The \texttt{dcache\_update()} function takes a memory address as input and extracts the index bits to determine which set to access. Then, you must determine which block is accessed and needs to be updated. Using the \texttt{dcache\_lookup()} function, you must check if the current memory address is already in the cache. If \texttt{dcache\_lookup()} 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 you need to access is determined as described below.

First, you must first check if there are any invalid blocks (starting with block 0, and ending with block 3). If an invalid block is found, then that is the selected block to be updated. Otherwise, a valid cache block needs to be evicted. The \texttt{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 0 is selected as the block to be updated, otherwise block 1 is selected. If bit 0 is ‘1’, then bit 2 is checked instead. If bit 2 is ‘0’, block 2 is the selected block to be updated, otherwise block 3 is selected.

After doing these checks, the \texttt{lru\_tree} must be updated regardless of if the memory address was already in the cache or not. If block 0 or 1 is selected, then bits 0 and 1 need to be flipped (while leaving bit 2 unchanged). Likewise, if block 2 or 3 is selected, then bits 0 and 2 must be flipped (while leaving bit 1 unchanged). Figure 3.3 illustrates the process of replacing a valid line and updating the LRU bits.
Figure 3.3: The 3-bit LRU tree used to (a) select the evicted cache line given a cache miss with no invalid lines and (b) the LRU bit update logic regardless of a cache hit or miss.

4 Branch Prediction

Previously, B-Type instructions utilized a static $pc + 4$ predictor in the fetch stage, which invoked a large penalty in the case of a branch taken. This assignment extends the capability of branch prediction by introducing a branch target buffer (BTB), a branch history shift register (BHSR), and a branch history table (BHT) to make a branch prediction in the fetch stage. The BTB is used to determine the branch target address, while the BHSR and BTB are used together to determine the direction of the branch. After a prediction is made, these structures are updated and the pipeline needs to recover given an incorrect prediction in the execute stage. The necessary pipeline modifications needed to support dynamic branch prediction are discussed in detail in section 5.

The table below shows the fields of the new BTB structure:

<table>
<thead>
<tr>
<th>BranchTargetBuffer Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>inst_addr</td>
<td>the instruction address of the entry</td>
</tr>
<tr>
<td>branch_target</td>
<td>the branch address of the entry</td>
</tr>
<tr>
<td>valid</td>
<td>indicates if the entry is valid</td>
</tr>
</tbody>
</table>

The BHSR is a 4 bit value that records branch directions for particular branch patterns, capturing spatial and temporal locality for branches.

The BHT 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 an enum for your use. The state machine scheme that you must use is given in 4.1.
Figure 4.1: Stage machine logic for the two-bit predictor entries of bht[].

Each structure is implemented as follows: btb[] is the array that holds 32 entries of the BTB. The bhsr is a variable used to record branch pattern histories to make direction predictions. bht[] is the array that holds each entry of the 2-bit predictors. The btb[] fields and bhsr variables are initialized to zero. The bht[] predictors are initialized to {01} (NT), or the “weak not taken” state.

There are 32 entries in btb[] indexed by the lower bits of an instruction’s address (direct mapping). Keep in mind, however, 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.

The bht[] array is indexed by the lowest 4 bits in the bhsr variable. Note: Since the bhsr variable itself is larger than 4 bits, you will always truncate the upper bits to ‘0’ to avoid indexing outside of the bht[] capacity.

Figure 4.2 illustrates an example a prediction being made in the fetch stage.
Figure 4.2: Example of dynamic branch prediction. An instruction is fetched, and select bits from the pc are used to index into the BTB. As the entry is valid and an address match, the BHSR is used to index into the BHT for a direction prediction. The bits {11} indicate that the branch should be predicted “taken.”

4.1 Branch Prediction Functions

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

`BTB_update()` takes an instruction’s address and branch target address as the input. The index is extracted from the input instruction address to determine which BTB entry to look up. Then, the function sets the corresponding instruction address, branch target address, and valid bit to ‘1’.

`predict_direction()` takes no argument. The BHSR is used to index into the BHT and check the state machine bits shown in Figure 4.2. If the bits indicate “T” or “TN”, then this function should return ‘1’ (for predict branch taken). Otherwise, this function should return ‘0’ (for branch not taken).

`direction_update()` takes a branch direction as the input (‘1’ for taken, ‘0’ for not taken). The BHSR is used to determine which BHT entry is being updated. The state machine logic must be
performed by transitioning the current state based on the actual branch direction based on the scheme in 4.1.

After updating the BHT entry, then the BHSR itself must be updated. 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 3 to 0 are used to record branch histories, as the BHT has only 16 entries. Upper bits beyond bit 3 must be cleared to ‘0’.

5 Pipeline Modifications

The following sections describe the required modifications for each stage of the simulator to support multi cycle memory latencies, dynamic branch prediction, and the data cache. Each stage should use your PA2 code as its starting point. It is assumed that your PA2 implementation is correct. The control variables in figures 1.2 and 1.1 will be used, along with your function implementations from sections 4.1 and 3.2.

5.1 Fetch Stage

Because dmem_busy can cause the pipeline to stall in addition to pipe_stall signal, you must ensure that pc_n is not changed when either stall is detected.

If the correct instruction was fetched (i.e. j_taken and br_mispredicted are ‘0’) and there are no stalls, then the next PC (pc_n) needs to be predicted. If branch_prediction_enabled is set to ‘1’, 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. For a taken branch, the function BTB_target() is used to get the target address, and the fetch_out.br_predicted field is set to ‘1’ to indicate a taken branch. Otherwise, this field should be ‘0’.

If branch_prediction_enabled is set to ‘0’ or the predicted branch direction is “not taken”, then the predicted pc_n direction is pc + 4 (as in previous assignments). In this instance, advance_pc(4) is called like normal.

5.2 Decode Stage

When dmem_busy is asserted, you must ensure that pc_n is not changed by JAL or JALR instructions so the pipe_trace.txt file accurately reflects the stall.

5.3 Execute Stage

First, when dmem_busy is asserted, you must ensure that pc_n is not changed by B-Type instructions so the pipe_trace.txt file accurately reflects the stall.

Next, the execute stage must be modified to support the resolution of branch instructions. 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 was mispredicted, and should set the br_mispredicted variable. Also, if the branch was mispredicted in the fetch stage, then pc_n should be 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, BHSR, and BHT.
Finally, all B-Type instructions should 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 PA2.

### 5.4 Memory Stage

The memory stage must be modified to model the memory and cache access latency of LW and SW 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()` is used when cache is enabled to determine if a given memory address is in cache.

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 it is 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 DRAM. Otherwise when there is a cache hit, `dcache_access_cycles` is used to model the latency of accessing cache.

Regardless of a cache hit or miss, `dmem_cycles` is used to count the number of cycles the memory stage has been accessing data. Whenever `dmem_cycles` is less than the incurred latency, `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 the incurred latency, `dmem_busy` and `dmem_cycles` 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 memory stage must also track data cache accesses and hits for statistics purposes. LW and SW instructions must increment the `dmem_accesses` variable for all LW and SW instructions after the incurred latency has passed and the instruction can proceed. Similarly, LW and SW instructions must increment the `dcache_hits` variable on a cache hit. **Note:** LW and SW 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.

### 5.5 Writeback Stage

No changes from PA2 for the writeback stage.

### 5.6 Simulator State

Similar to how `pipe_stall` held the fetch and decode stages in PA2, when `dmem_busy` is asserted the `update_simulator_state()` function will hold the state of the fetch, decode, execute, and memory respectively. This is done so that the pipeline stages are able to re-execute their current instruction when the memory stages are stalled. It is important that `dmem_busy` is asserted correctly in your memory stage to allow proper pipelining.