Programming Assignment 3:
5-Stage Pipelined Simulator with Multi-cycle Operations and Data Cache

Due 19-Nov-2021 (Fri) @ 11:59 PM on HuskyCT

Introduction

In this programming assignment, you will add (i) multi-cycle ALU and memory operations and (ii) a 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`.**

The objective of this programming assignment is to modify `sim_stages.c` to implement multi-cycle ALU and memory operations and a data cache for the 5-stage pipelined *riscy-uconn* simulator. The necessary modifications to your PA2 simulator are described in the subsequent sections of this document.

To receive full credit for this assignment, your simulator implementation must be fully functional and correct **both (i) with and without forwarding and (ii) with and without the data cache** for all the unit tests in the PA3 `unittests` directory as well as for the unit tests of all previous assignments. Explicitly, the simulator must work correctly for the following four (4) configurations:

<table>
<thead>
<tr>
<th>Forwarding</th>
<th>Data Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>Disabled</td>
<td>Enabled</td>
</tr>
<tr>
<td>Enabled</td>
<td>Disabled</td>
</tr>
<tr>
<td>Enabled</td>
<td>Enabled</td>
</tr>
</tbody>
</table>

Once you have completed this programming assignment, submit your `sim_stages.c` file via HuskyCT by the posted deadline. **To receive credit for this assignment, you must also schedule a 10–15 minute code review meeting with the TA. You have up until 2 weeks after the HuskyCT deadline to complete the code review.**
1 5-Stage Pipelined Simulator with Multi-cycle Operations and Data Cache

The objective of this programming assignment is to extend the 5-stage pipelined simulator implementation of PA2 with (i) multi-cycle ALU operations for R-Type instructions, (ii) multi-cycle memory accesses, and (iii) a single-cycle direct-mapped data cache.

1.1 Simulator Structure

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

1. Multi-cycle operation control variables 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.

<table>
<thead>
<tr>
<th>Multi-cycle Operation Control Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dmem_access_cycles</code> (read-only)</td>
</tr>
<tr>
<td><code>rtype_execute_cycles</code> (read-only)</td>
</tr>
<tr>
<td><code>dmem_busy</code></td>
</tr>
<tr>
<td><code>dmem_cycles</code></td>
</tr>
<tr>
<td><code>exe_busy</code></td>
</tr>
<tr>
<td><code>exe_cycles</code></td>
</tr>
</tbody>
</table>

Figure 1: Multi-cycle operation control variables.

2. 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.

3. 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.

<table>
<thead>
<tr>
<th>Data Cache Control Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dcache_enabled</code></td>
</tr>
<tr>
<td><code>dcache_accesses</code></td>
</tr>
<tr>
<td><code>dcache_hits</code></td>
</tr>
</tbody>
</table>

Figure 2: Data cache control variables.

4. 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.

5. The simulator now creates a `cdump.txt` file on each invocation 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.

6. 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`. This function must be modified to hold the output of the pipeline stages when appropriate. The subsequent sections contain more details regarding this function.
1.2 Data Cache Program Argument

The simulator executable now accepts two (2) program arguments that independently control whether (i) forwarding and (ii) the data cache are enabled:

```
$ ./simulator OUT_FILE FORWARDING_ENABLED DATA_CACHE_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, and `DATA_CACHE_ENABLED` is either 0 or 1 to indicate that the data cache is disabled or enabled respectively.

The data cache enabled status is visible in the `sim_stages.c` file through the `dcache_enabled` variable declared in `sim_core.h`.

Details regarding the data cache are provided in the subsequent sections.

1.3 Modifications

At a high level, this programming assignment involves three (3) changes to the simulator: (i) changing R-Type instructions to take five (5) cycles in the execute stage to simulate ALU operation latency; (ii) changing LW and SW instructions to take ten (10) cycles in the memory stage to simulate memory access latency; and (iii) implementing a single-cycle direct-mapped 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.

1.3.1 Multi-cycle Execute and Memory Stages

The execute stage must be modified to add an ALU operation latency for the ADD, SUB, AND, OR, SLL, SRL, and SLT R-Type instructions. The ALU operation latency is determined by the `rtype_execute_cycles` control variable. Since this variable is set to five (5) cycles, the execute stage must stall for this many cycles before allowing the aforementioned instructions to proceed to the memory stage. **The pipeline must not be stalled for NOOP instructions.** Two (2) new control variables `exe_busy` and `exe_cycles` have been added to manage stalling in the execute stage. `exe_busy` must be de-asserted at the beginning of the execute stage so that it is only asserted for cycles where it evaluates true as described below. `exe_cycles` is used to count the number of cycles the execute stage has stalled. When `exe_cycles` is less than `rtype_execute_cycles`, `exe_busy` must be asserted and the execute stage must return a NOOP. Asserting `exe_busy` ensures that the entire pipeline stalls whenever the execute stage is stalled waiting for an ALU operation to complete. When `exe_cycles` is equal to `rtype_execute_cycles`, `dmem_busy` (explained below) is checked. If `dmem_busy` is asserted, the execution stage must also stall, so `exe_busy` must be asserted until the memory stage can proceed forward. If `dmem_busy` is not asserted, `exe_busy` and `cycles_to_exe` are cleared.

The memory stage must also 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 ten (10) cycles, the memory stage must stall for this many cycles before allowing LW or SW instructions to progress to the write-back stage. Two (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. `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 a NOOP. 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 zero (0).

When `dmem_busy` and/or `exe_busy` are asserted, the `update_simulator_state()` function must 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 execute and/or memory stages are stalled.

The decode stage must also be modified so that `pipe_stall` is asserted and a NOOP is returned when `dmem_busy` and/or `exe_busy` are asserted. This ensures that the decode and fetch stages stall when the memory and/or execute stages stall. Moreover, when `dmem_busy` and/or `exe_busy` are asserted, the decode function must return before any modifications are made to the `pc_n` or `br_taken` variables. This ensures that the program counter does not update and the branch logic does not introduce any side effects when the pipeline is stalled.

### 1.3.2 Data Cache

The simulator now features an incomplete data cache implementation that is enabled using the relevant program argument. The cache is direct-mapped and addressed by a 32-bit memory address. The cache contains thirty two (32) cache blocks (cache lines) of thirty two (32) bytes, or eight (8) words, each resulting in a total cache size of 1,024 bytes. The cache hit latency is fixed at one (1) cycle while cache misses incur `dmem_access_cycles` (defined as 10 cycles).

Each cache block is represented by the `CacheBlock` type defined in `sim_core.h`, and contains the valid and tag bits for each cache block. **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 cache. On simulator start-up, the cache is initialized such that the valid and tag fields of all thirty two (32) cache blocks are zero (0); in other words, the cache is empty on start-up. The cache itself is represented by the `dcache[]` array declared in `sim_core.h` and consists of thirty two (32) cache blocks.

The cache implementation must be completed by implementing the `dcache_lookup()` and `dcache_update()` functions in `sim_stages.c`. Each cache block contains eight (8) words (thirty two (32) bytes), and each memory address contains one (1) word (32-bits, or four (4) bytes) of data. Consequently, you must decode the appropriate bits from the memory address to determine the cache block index and offset into the cache block. The remaining most significant bits of the memory address should be tracked as tag bits.

The `dcache_update()` function takes a memory address as input and extracts the cache block index bits to determine which cache block to access. Then, the valid bit of the corresponding cache block is set to one (1) and the tag bits are set appropriately.

The `dcache_lookup()` function takes a memory address as input and extracts the cache block index bits to determine which cache block to access. If the tag bits in the cache block match the tag bits of the input memory address, and the cache block is valid, the function must return one (1). Otherwise, the function returns zero (0). In other words, the function returns one (1) for a cache hit or zero (0) for a cache miss on a given memory address.

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, the memory address must be updated in the cache before it is accessed. This is done so that the address is cached for future memory lookups.