Design Document Group 54 (PA3)

Members: William Kerr, Nathan Lakritz, Ali Hooman

Goals:

Augment our code from Assignment 2 by adding a split instruction and data cache to our pipelined RISC-V CPU. ISA is same the from PA1 and PA2. Must be capable of executing simple assembly language programs in RISC-V assembly. Must handle variable pipeline delays.

Changes from Assignment 2:

We shall be using the memory_status() function. Before, the memory_status() was always ignored.

We shall specify what memory location to read from.

We shall be implementing two caches for this assignment: an instruction cache (I-cache) and a data cache (D-cache). We will implement them as arrays. These will be arrays of structs. The struct we will be using will contain the cache data, tag, valid bit, index, offset, and dirty bit (if using write back caches) for that particular entry.

I-cache is 8KiB, 16 byte block size. (*never* overwrite instructions).

D-cache is 16KiB, 8 byte block size, write-through.

To implement these changes, memory_read() and memory_write() now return true or false. If false, the memory isn't done reading yet. Data copied into the value pointer will be incorrect. If we're doing an *invalid* read, some incorrect value will be copied into the value pointer and **the function will actually return true** (so, we need to test loads extensively).

memory_status() can be used to check the status of a previously made memory_read() or memory_write() call. Just feed it the address of the read or write in question. If the status of a read shows as completed, the value will be obtained.

While we write our design, we must consider the following questions:

1.) How can my code access and modify the caches?
2.) How do I stall the pipeline on a cache miss?
3.) How do I check to see when the pipeline can resume, and how do update the cache?
4.) What happens if there are two cache misses on the same cycle?

Extra credit:

If we want to do extra credit, we could write a two-way set-associative write-back D-cache. It should also be write-allocate: on a write miss, it should read the cache line into the cache before modifying it by writing the new value.

We could also write a unified Level Two cache. 128KiB, two-way set associative, with 32-byte blocks, and use a writeback scheme. Write allocate on write misses.

CODE:

Cache Organization:

Type	Capacity	# of blocks	Block Size	Tag	Index	Offset
I-cache	8KB	512	16 bytes	[63:13] (51)	[12:4] (9)	[3:0] (4)
D-cache	16KB	2048	8 bytes	[63:14] (50)	[13:3] (11)	[2:0] (3)

Both caches will have a valid bit, but neither will have a dirty bit (since no writeback). All caches are direct-mapped.

I-cache:
Has 512 blocks, each 16 bytes. We will implement it using structs, as follows:

Each entry will have a type like this:

```
struct i_cache_block {
     bool valid_bit;
     uint64_t  data; // actual data being stored
     uint64_t  tag;
     uint16_t  index;
     uint8_t   offset;
```

```
}
```

The I-Cache will look like this:

```
struct i_cache_block i_cache[I_CACHE_SIZE];
```

Where I_CACHE_SIZE is defined somewhere in a header, like this:

```
#define I_CACHE_SIZE 512
```

D-Cache:
Has 2048 blocks, each 8 bytes.

Each entry will have a type like this.

```
struct d_cache_block {
     uint64_t  data; // actual data being stored
     uint64_t  tag;
     uint16_t  index;
     uint8_t   offset;
     bool      valid_bit;
}
```

The D-Cache will look like this:

```
struct d_cache_block d_cache[D_CACHE_SIZE];
```

Where D_CACHE_SIZE is defined somewhere in a header, like this:

```
#define D_CACHE_SIZE 2048
```

If we need to make these caches writeback, we can add a dirty bit by adding a boolean field “dirty”.

Fetch:

This time, we need to specify to fetch from I-cache. If the data is not in the cache (or the valid bit is 0), we will need to fetch from memory. We will have a boolean variable in the decode stage register telling us if the fetch stage is currently stalling. That way, we will be able to stall the fetch stage.

To fetch from the I-cache, read from the I-cache. If the entry is not valid, we read from memory. If the tag doesn't match, we read from memory.

To read from memory, we call memory_read(), then we stall the fetch stage.

To stall the fetch stage, we will set new_d_reg's stalling variable to true.
We will need to stall the fetch stage. We will have a variable in the decode function that tells us to stall. We will do this starting when memory_status() returns false, and stop stalling when memory_status() returns true.

To stall the fetch stage, we will set new_d_reg's stalling variable to true. At the beginning of the fetch stage, we will check if the memory is still reading. If so, we continue stalling. If not, we will set new_d_reg to false, and send the instruction to decode.

If the fetch stage is stalling, we check the memory_status(). If memory_status() is false, we continue stalling. If it's true, we send the instruction to the first non-valid I-cache block.

Decode:

Decode the instruction. Send an enumerated type to the execute stage to tell what instruction is being called. Same as last time.

We will need to stall the decode stage. To reiterate from last time:

We will stall the decode stage if a variable used is dependent on a memory value being loaded. Once it's loaded, we can forward that loaded variable to the decode stage with a forwarding variable. This forwarding variable will be in the writeback stage register.

This time, we will also need to stall the decode stage if the memory stage is taking too long, and the execute stage is also stalling. We will do this by checking if the execute stage is stalling.

We will check the execute stage register for a stalling variable. If it's set true, then we will stall the decode stage.

Execute:

Compute addresses, execute the instruction, etc. Same as last time.

We will need to stall the execute stage. If it's taking a while for memory to complete, we will need to stall the execute stage.

In the memory stage register, we will have a stalling variable for execute. If it's set high, we stall the execute stage.

To check to see if we need to stall the execute stage, we check if the writeback stage register is set high. If it is, then we stall the execute stage by setting new_m_reg's stalling variable to high.

Memory:

This time, we need to specify to fetch from the D-cache.

We must account for stalls in memory fetches. To do this, we will add a field to new_w_reg. This is a boolean stalling variable that tells if the memory will stall.

At the beginning of the function, we check if the current writeback register has the stalling field set high. If so, we check if the memory status has returned true. If so, we continue to the writeback stage. We set new_w_reg's stalling field false, return the memory value, and send it to writeback.

Writeback:

Write to the registers, same as last time.
This time, we will stop writing back if it's a store instruction.
There will be no stalls in writeback.

Implementation Flow Chart:

Fetch:

- If instruction is in the cache, fetch immediately and move to Decode
- If not in the cache, check "memory read flag" from previous cycle in decode stage register
    - If we tried to read already, do memory_status()
        - If status is true, unset the "memory read flag" and send the data to Decode. Also, add the data to the cache.
        - If status is false, keep the "memory read flag" set and send a NOP to Decode
    - If we haven't tried to read yet, do a memory_read().
        - If the status is true, send the data to Decode. Also, add the data to the cache.

        - If the status is false, send a NOP to Decode and set the “memory read flag.”

Memory:

- Need to check D-cache during loads
    - If in the cache, fetch immediately and move to Writeback.
    - If not in the cache, check “memory read flag” from previous cycle in writeback stage register
        - If we tried to read already, do memory_status()
            - If status is true, unset the “memory read flag” and send the data to Writeback. Also, add the data to the cache.
            - If status is false, keep the “memory read flag” set and send a NOP to Writeback
        - If we haven’t tried to read yet, do a memory_read()
            - If the status is true, send the data to Writeback. Also, add the data to the cache.
            - If the status is false, send a NOP to Writeback and set the “memory read flag”.
- Need to check D-cache during stores
    - If in the cache
        - Update the cache by writing the data immediately
        - Write the data to memory using memory_write() (time to complete this doesn’t matter, since we have the data in the cache for future loads)
        - Send a NOP to Writeback as usual for stores
    - If not in the cache, check “memory read flag” from previous cycle in writeback stage register
        - If we tried to store the data already, do memory_status()
            - If status is true, unset the “memory read flag” and send a NOP to Writeback. Do not add the data to the cache since we had a cache miss to begin with.
            - If status is false, keep the “memory read flag” set and send a NOP to Writeback
        - If we haven’t tried to store the data yet, do a memory_write()
            - If the status is true, send a NOP to Writeback. Do not add data to the cache.
            - If status is false, set the “memory read flag” and send a NOP to Writeback.

Fetch, Decode, and Execute stages always need to check the “memory read flag” at the writeback stage register and if set, not do anything. Everything before the Memory stage needs to know to stall from a pending memory access. These stages shouldn’t do anything, they should just terminate from the start.