Design Document Group 54 (PA4)

Members: William Kerr, Nathan Lakritz, Ali Hooman

Goals:

Augment our code from Assignment 3 by adding virtual memory support.
Include a Translation Lookaside Buffer (TLB), and virtual to physical address translation to fill the TLB on a TLB miss.

Tips:

Watch the Lecture from November 21st.
Use the get_ptbr() function. (Page Table Base Register)
Consider the following when designing our code:

1.) How can my code access and modify the TLBs?
2.) How can my code translate a virtual to a physical address using the page tables?
3.) How do I stall the pipeline on a TLB miss?
4.) How do I check to see when the pipeline can resume, and how do update the cache?
5.) How do TLB misses and hits interact with cache misses and hits?

All page tables will require that the low-order 14 bits of a virtual page number must match those of the corresponding physical page number. (From the assignment handout)

The Setup:

Virtual addresses: 32 bits
Physical addresses: 32 bits
Page size: 4 KiB
Page offset: 12 bits
Virtual page number: 20 bits
Physical frame number: 20 bits

TLB:

- Need TLB-I for instructions and TLB-D for data (both direct-mapped)
- Use array implementation, each 8 entries long

- Break virtual page number into an index and tag
    - Index should be 9 bits, and tag 11 bits for TLB-I
    - Index should be 11 bits, and tag 9 bits for TLB-D
    - Index is used to identify TLB block
    - Tag is stored in TLB block and used to determine TLB hit or miss
- We can create a struct for entries
    - Valid bit, dirty bit, virtual page number tag, virtual page number index, physical frame number, page offset
        - Page offset is the same for virtual and physical addresses

We can create a struct for entries:

```
struct TLB_entry {
      bool valid;
      bool dirty;
      uint32_t vpn_tag;
      uint32_t vpn_index;
      uint32_t physical_frame_number;
      uint16_t page_offset;
}
```

Both I-TLB and D-TLB will use this struct.

We will declare a global array for I-TLB and D-TLB, something like this:

```
#define I_TLB_SIZE 8
#define D_TLB_SIZE 8

struct TLB_entry I_TLB[I_TLB_SIZE];
struct TLB_entry D_TLB[D_TLB_SIZE];
```

Instruction and data caches:

- Need I-cache for instructions and D-cache for data, as before
- Now we need virtually indexed, physically tagged caches
- Break down our virtual address into tag, index, and page offset
    - Index should be 9 bits, and tag 11 bits for instructions
    - Index should be 11 bits, and tag 9 bits for data
    - Page offset is 12 bits for both instructions and data addresses
    - Use the index bits to find the block we want in the cache
- Use TLB to get the corresponding physical address from our virtual address
    - Break down our physical address into tag, index, and page offset (process identical to virtual address breakdown)

- Use the tag bits to check our block in cache (previously determined by the virtual address index) for a hit or miss

To simulate a I-cache and a D-cache entry, we will use the same struct, defines and arrays as before:

```
#define I_CACHE_SIZE 512
#define D_CACHE_SIZE 2048
struct cache_block {
     bool valid_bit;
     uint64_t tag;
     uint16_t index;
     uint8_t offset;
     uint64_t data;
}
struct cache_block i_cache[I_CACHE_SIZE];
struct cache_block d_cache[D_CACHE_SIZE];
```

Page Tables:

- 20 bit virtual page number => 2^20 total PTEs
- Two levels of page tables
- Root table has 1024 entries, each 32 bits long
    - Page table base register points to this root table
    - Entries contain the physical page number, with bit 31 (the high order bit) as the valid bit
    - Virtual page number serves as the index to the root table
    - Root table points to 2nd level table such that each root page table entry points to an entirely new 2nd level table
    - First 10 bits of virtual page number represents which PTE in the root table to use.
- Each 2nd level table has 1024 entries.
    - Last 10 bits of virtual page number represents which PTE in the 2nd level table to use.
- The total number of PTEs is calculated as such: (1024*1024=2^20)

To simulate a page table entry, we could use a struct like this:

```
struct PTE_entry {
     bool valid;
     uint32_t pointer;     // points to a physical address.
                           // in the root table,
                           // points to the second table address.
}
```

```
#define ROOT_SIZE 1024
#define SECOND_SIZE 1024

struct PTE_entry root_PTE[ROOT_SIZE];
struct PTE_entry second_PTE[SECOND_SIZE];
int root_index;
int second_index;
```

Simulator Initialization:

- Use the Python script to generate the page tables
- Load the page tables into a memory location
- setptbr() to the memory location for the root table

Reading from Memory:

There are 2 types of reads: Instruction Read (Fetch stage), and Data Read (Memory stage).
When we read, we are looking in both a TLB and a cache simultaneously.
Instruction fetches look through the I-TLB and the I-cache.
Data fetches look through the D-TLB and the D-cache.

TLB search:

First, search the I-TLB (for instruction fetches) or D-TLB (for data reads) for the virtual address using a for loop.

Loop through all the valid cache entries. Compare the virtual address with the TLB and find the physical address tag from TLB. Compare this tag with the tag in cache. If the tag and the index match the current tag and index, then read the physical frame number from there. We have a TLB hit.

If the virtual address isn't in the TLB, we have a TLB miss. Stall the pipeline.
To handle misses, we must walk the page table to find our physical frame number.
We're doing this part in a separate C file.

Page walking and Address Translation:

Read the top level page table. Find the entry we're looking for.
The address is located at get_ptbr() + first 10 bits of VPN.
If the valid bit is not set at the root table PTE, throw an error. Professor Miller says he won't test for PTE errors.
Otherwise, go to the second level page table.
Set a variable ptbsr (page table base second register) to the contents of the pointer in the PTE entry.

The address of the second level page table PTE entry is located at ptbsr + last 10 bits of VPN.
Check the valid bit again of that PTE entry. If not set, then throw an error.
If the valid bit is set, then:
To translate it, simply shift the pointer in the PTE left by the offset calculated earlier.
Then, put it in the I-TLB or D-TLB respectively. If all pages are filled, evict the nearest one to the index.

While the TLB is searching, we can search the cache simultaneously.

Cache Search:

Use the offset of the virtual address to look in the cache.
Check the valid bit first.
If we have a hit, retrieve the physical tag from there.

If we don't hit, we have a cache miss.
Read from memory once the address translation is done.

Once we retrieve the address from the TLB or translate our physical address, and if we retrieve our physical tag from our cache, we compare the physical tag to our physical address.

If they match, retrieve the contents of the cache.

If they don't match, *then* we can utilize a memory_read() call to read from memory.
We stall the pipeline here to allow the memory to read.
Use the code from last time to do that.

Once the memory is done reading, we update the TLB and the cache.

Updating the TLB and cache:

If we have to read from memory, then we should update the cache and the TLB.

TLB:

Find a spot in the TLB, then write to it.
We do this by using a TLB index, starting from 0. Since we have 8 TLB entries, we simply use a TLB_index variable. We retrieve TLB data like this:

```
int foo = TLB[TLB_index % TLB_SIZE]
```

Every time we have a memory read, we increase the TLB index. This way, the TLB index loops around when we run out of space. We're using spatial locality in our program this way.
When we read from memory, set dirty bit to 0, valid bit 1, and put in vpn_index, vpn_tag, physical frame number and the offset into our TLB entry, so when we call it again, we can reconstruct the physical address.

TLB hit:

If we're writing on a TLB hit, set the dirty bit high on the TLB.

Cache:

Same as TLB for writing to the cache and evicting a cache entry.
But, we're still using write-through caches. So, no dirty bit needed.