EE480 Assignment 4: Coherently TACKY

Sam McCauley
Computer Engineering
University of Kentucky
Lexington, Ky
stmc225@uky.edu

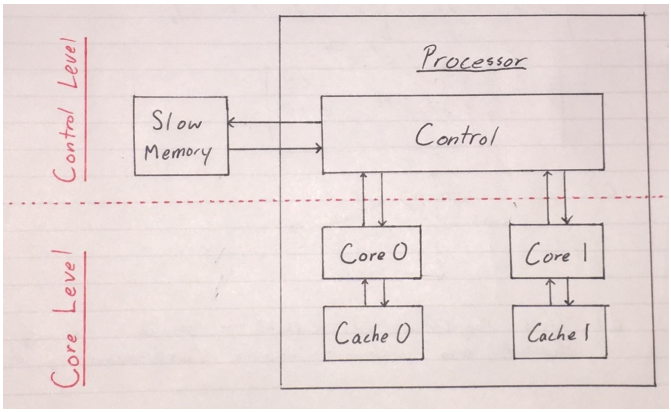
Andrew Blaney
Computer Engineering
University of Kentucky
Lexington, Ky
andrewblaney@uky.edu

Greg Schloemer
Computer Engineering
University of Kentucky
Lexington, Ky
greg.schloemer@uky.edu

***Abstract*—In this project, our team is attempting to build a pipelined, multiprocessor implementation of TACKY. It is a VLIW and our multiprocessor implementation is expected to be able to operate with two coherent data caches. Rather than operating with just one processor, our system is instantiated with two separate cores.**

I. General Approach

Our general approach to this project was to split the processor into two levels: a cache control level and a processor core level. The control level handled the read and write requests from the two processor cores. The control level also managed all of the higher level flow control that was necessary to maintain cache coherence. The processor core level of the hierarchy was the one sending out the read and write requests to the control level when necessary. The main challenge of the core level was properly maintaining cache coherence with the adjacent core and ensuring that the two cores stayed in sync with one another.



II. Core Level Implementation

The core level had two main segments. The first segment is handled in the first ALU stage of the pipeline. If an instruction in this stage is a memory read then the core checks to see if a clean version of the memory is already in the cache–in which case that value is immediately loaded. If a clean version of the memory is not in the cache, then the core issues a read request to the control level by pulling its read request signal high and communicating the requested memory address. On the other hand, if an instruction in this stage is a memory write request, the core updates its own cache and then issues a write request to the control level by pulling its write line high and communicating the memory address and write value.

The second segment of the core level implementation handles the control level's response to a memory request. There are a couple of scenarios for this:

- At the positive edge of the control level's busy line, an internal core register called pause is pulled high. This ensures that both cores stay in sync with one another by freezing all pipeline stage progression.
- The negative edge of the busy line indicates that a request has been completed. At this event, the core checks if it is the response to its own request or if it was for the adjacent core. If it is for itself, then it handles by placing the read data into the cache as well as updating the partialResult register from the ALU stage. It then clears up the control registers and allows the pipeline to continue.
- A positive edge of the lineChanged signal indicates the adjacent core changed its cache. In this event, the corresponding core checks if it needs to dirty something in its cache or not.

III. Control Level Implementation

The control-level implementation for this processor is responsible for governing the interface between the processor cores and the slowmem memory. There are two modules that comprise the control logic. The first is a declaration of cacheController, which details the logic for each scenario. Then, a separate processor module provides the interconnection between cachecontroller, each processor core, and slowmem; it contains no logic definitions. To adequately maintain the slowmem/core relationship, the processor/cacheController modules maintain these connections:

- **Inputs to slowmem:** strobe (to start memory operations), memory address, line to be written, rnotw (to signal read or write)
- **Outputs from slowmem:** memDone (to indicate when read finishes), value read from memory
- **Inputs to core0/core1:** *lineChanged* (for writes, indicates which cache line might be dirtied), *lineChangedSignal* (signal line for writes, indicates a line has been changed), value read from memory, busy line (indicates when read/write operation is underway)

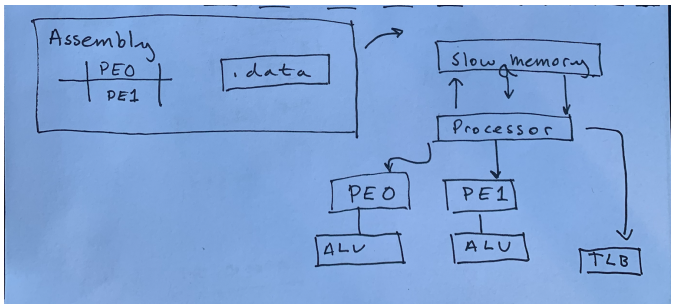
- **Outputs from core0/core1:** word to write to memory, address to write/read

The cache controller interface must be able to respond to each of the following scenarios:

- **Single Core Read:** Reads are fairly simple, even though they take 4 clock cycles to complete. First, set busy signal, then find line address from word address (div by 4), and finally, tell slowmem to read. When read is finished, the controller will clear busy, return read line to core, and halt memory operations in slowmem.
- **Single Core Write:** Since we perform operations on slowmem by entire words only, we must first read the line from memory before we can write a single word into it. Thus, a write performs, first, all of the "single core read" steps listed above. Once that is finished, it simply stores the word given by the processor core in the line read from memory 1 cycle ago, then tells slowmem to perform a write of the updated line at the specified line address. Additionally, the write operation must inform the non-writing core that a given memory line address has been changed (in other words- its cache is dirty). This technique is known as **write-invalidate**, and is used to ensure cache coherency across our multi-core processor implementation.
- **Simultaneous Reads on Both Cores:** In the event of a simultaneous attempt to read from both cores, the processor will start by initiating the read request on core 0, while storing the address of the core 1 request in a buffer for use later. Essentially, a simultaneous read request is handled the same as a single core read described above, except: 1) it will perform core 0 read, then set a "pendingOp" flag 2) when core 0 read completes, pendingOp is set 3) if pendingOp is set, it will perform a core 1 read on the address value stored in the buffer. The busy signal will only be cleared after both the core 0 and core 1 operations are complete and thus, the cores will not try to accept new instructions in any given pipeline stage.
- **Simultaneous Writes on Both Cores:** As with simultaneous reads described above, the processor will initiate core 0 write, save the core 1 address for later, then only start the core 1 write once the core 0 write has completed. The busy signal will not be cleared until both writes are complete, thus no pipeline stage will attempt to complete a new instruction.

IV. Testing/Implementation Problems

Though we were not able to thoroughly test the design of our multiprocessor, we made sure that the verilog file compiled and that our controller (theoretically) handles simultaneous reads/writes effectively. We know that the implementation of the pipeline processor from last project worked (though there were some issues with forwarding that we were not able to resolve). As mentioned, the testing of our multiprocessor design was limited. However, we have a strategy that we believe would work fairly well and we believe our design would be effective. An image of the general assembly strategy is below.



Part of the difficulty with testing our design was generating assembly code that would effectively test the main components of our multiprocessor design. For example, we needed to have instructions that would trigger the busy/read lines in our design. By ensuring that these busy lines were triggered, we would know that both caches aren't accessing the same memory. Our general strategy was to create two instances (PE0 and PE1) that would perform instructions at the same time. By comparing the theoretical values to those that were generated, we would be sure that there were not any errors in the multiprocessor design. Below is an example of what we planned on implementing (though the code written below would need to be turned into machine code).