Overview

In this project, you will: (1) integrate instruction and data caches to hide the latency of accessing main memory, (2) add 2-way fine-grain multithreading, and finally, (3) combine 2 multithreaded cores to form a multicore processor with 2x2 threads. This project has three checkpoints:

Checkpoint 1: Caches
You will add direct-mapped instruction and data caches to your processor. The pipeline will require changes to handle cache miss stalls by instruction and data fetches.

Checkpoint 2: Multithreading
You will add 2-way multithreading support to your processor. The threads will share the pipeline, but have separate architectural resources (e.g. register file, PC and address space). The interleaving will be dynamically scheduled such that instructions from a second (low-priority) thread are issued to take up any slack cycles unusable by the first (high-priority) thread.

Checkpoint 3: Multicore
You will instantiate a second (multithreaded) core. The two cores will operate independently, except for sharing access to memory (both instructions and data). You will arbitrate access fairly in cases of contention.

**Note**: For project 4, the processor should not implement branch or load delay slot semantics.

Checkpoint 1: Caches
In past projects, you have assumed an idealized single-cycle, fixed-latency main memory. In modern processors, main memory accesses can take hundreds of processor cycles. A cache allows fast access to memory locations that are repeatedly accessed (temporal locality) or are near recently-accessed locations (spatial locality). The cache you will use in Project 4 has an access latency of 1 cycle on a read hit and at least 10 cycles on a miss.
For this checkpoint, we will supply you with a direct-mapped, write-back data cache, a direct-mapped instruction cache, an address translation module, and a new memory module. In order to attach the caches to your processor, you will have to modify the testjig module to integrate address translation, an instruction cache, a data cache, memory, and your processor core (from Project 3). The caches require a new memory module that operates on a cache block (128-bit) granularity. As a result, there is a different memory module for this project with a wider data bus and narrower address bus. (Your interconnection wires should be adjusted accordingly. You will not be able to use the old memory modules.) In addition, instruction caches are typically read-only, so no write signals should be asserted in the instruction cache.¹

Your processor pipeline needs to stall if the cache indicates a miss has occurred (i.e., the requested memory location is not in the cache for reading or writing). For read requests, when the cache eventually becomes ready, the cache will return the requested data value on the output lines. For writes, you must wait until the cache block being written is in the cache. Thus, for all requests, you should verify that the \( p_{\text{data\ ready}} \) signal is asserted before continuing execution. For writes, you must wait until the cache block being written is in the cache. The write will occur in the cycle after the cache reports a hit. That is the cache is busy for 2 cycles even on a write hit. You should try to hide the additional write latency, as discussed in class.

The instruction and data cache use the same interface. The details of the inputs and outputs of the cache modules are documented in the Verilog comments within the files we provide. In your simulations, you should modify the cache to print out the number of hits, misses, and total accesses when your simulation finishes. This can be done in the testjig module.

Finally, to allow sharing of Verilog modules for all checkpoints in this project, the caches and memory are capable of distinguishing accesses from different threads (up to four threads in checkpoint 3). To accomplish this, all addresses are two bits wider than otherwise expected; the upper two bits are used to determine which thread is making the request. For checkpoints 2 and 3, you will use an address translation unit that will take a 32-bit per-thread address and return a 34-bit global address. For checkpoint 1, there is only a single thread. You can hardwire the upper two bits of the address to zero in the wires connecting your pipeline to the caches.

**Checkpoint 2: Multithreading**

In this checkpoint, you will add multithreading support to your Checkpoint 1 processor pipeline (after disabling data forwarding²). Multithreading is a technique that allows multiple hardware threads to execute on shared pipeline resources. To the user and operating system, each thread appears to be running on an independent “processor.” Sun’s CoolThreads and Intel’s HyperThreading are both examples of multithreading.

While there are many different types of multithreading, you will implement a fine-grain, dynamically scheduled variant. In fine-grain multithreading, instructions from different threads may be selected to issue in the decode stage on a cycle-by-cycle basis. Your pipeline will maintain two threads, Thread 0 and Thread 1. Thread 0 is considered higher priority than Thread 1. The decode stage should try to issue instructions from Thread 0 whenever possible. When thread 0 needs to stall due to data hazards or structural hazards, the decode stage will consider instructions from Thread 1 for issue. Thread 0 should achieve the full pipeline performance (even without forwarding) if its instruction stream does not encounter hazards. When Thread 0 must stall, the pipeline performance still can be reclaimed by giving the unusable issue cycles to Thread 1. In the worst case scenario when neither Thread 0 nor Thread 1 can be issued, the decode stage should stall.

¹We are implicitly forbidding self-modifying code here (not as if 18-447 students would write such code!)
²Be sure to make a copy first before modifying the pipeline for Checkpoint 2.
For this checkpoint, after removing the forwarding paths, you will modify the pipeline to maintain two sets of architectural states separately for Thread 0 and Thread 1. Each instruction in the pipeline should be tagged with its thread ID (0 or 1). When a pipeline stage access architectural states on behalf of an instruction, it uses the instruction’s thread ID to select which set of architectural states to use. You will need to reason about each piece of your design to determine whether it should be duplicated for or be shared by Thread 0 and Thread 1.

Your pipeline should not allow instructions from one thread to overtake instructions from the other thread. Therefore, when one thread must stall in one stage (e.g., data cache miss), all upstream pipeline stages stall. You must ensure your stall logic is updated so it only detects data hazards between instructions of same thread.

Each thread will access a disjoint region of memory; you will not have to worry about sharing memory values. To accomplish this, we give each thread its own 32-bit address space. To direct accesses to the correct address space, we provide an address translation unit that must be consulted before every cache access. The translation unit takes a 32-bit memory address and a 1-bit thread identifier, and returns a 34-bit address. (The module also takes a “core” parameter on instantiation, but you will not need this until Checkpoint 3.) You must translate both instruction and data addresses. Because each thread has its own address space, all PCs and $gp’s can be initialized to the same value. Finally, translation occurs combinatorially, so it should not cause pipeline stalls.

Modify the syscall instruction logic to only exit when all threads have reached a syscall instruction, or any thread generates a decode exception. When only one of the two threads is running (whether Thread 0 or 1), the remaining thread should continue to execute to completion.

**Checkpoint 3: Multicore**

In this checkpoint, you will create a dual-core processor design (comprising two of your multithreaded pipelines from checkpoint 2). While each pipeline runs mostly independent of the other, they must share the memory hierarchy, and require arbitration when both pipelines try to access the memory or data cache port simultaneously.
Your design will have one instruction cache per core, but share a single data cache between both cores. This design will generate contention in two situations: (1) both I-caches accessing the single instruction port to memory, (2) and both pipelines accessing the port to the D-cache. In both cases, if the resource is busy when a core needs to initiate a new request, you must apply backpressure so the pipeline stalls, and only reinstate the request after the cache is finished with the outstanding request. (This includes waiting out an on-going cache miss processing.) Similarly, if both pipelines try to issue a memory request in the same cycle, backpressure must be applied to one so that it tries again when the cache is available.

You will create two arbitration modules, one for instructions and one for data. You should not need to change the underlying interfaces (the inputs to the data cache, and the instruction port to memory). The arbitration policy should be fair, so that neither pipeline always has priority. You will need to track which core has made outstanding requests to the cache so that the cache responses are routed to the correct core.

Add an instantiation of the address translation unit for the second core; make sure to set its “core ID” parameter to 1. Lastly, modify the syscall instruction logic to only exit when all (four) threads have reached a syscall instruction, or any thread generates a decode exception.

**Verilog Files**
The following Verilog files will be provided to you in the project4 handout directory (/afs/ece/class/ece447/handout/project4/). We will provide a few test programs; however, you should expect more difficult programs to be tested at check off. Therefore, you should create your own test cases as well.

- `mips_mem_sync_cache.v` – modified version of the memory module to work with caches
- `mips_icache.v` – instruction cache
- `mips_cache.v` – data cache
- `mips_addr_trans.v` – the address translation unit for multithreading/multicore

**SPIM**
In Project 4, you will still use the normal spim command to generate the memory data files that are read in by your processor for all threads. Note, however, that the naming convention has changed slightly to enable support for multiple threads:

- spim447 –vasm filename.s mem0 (will be executed on thread 0)
- spim447 –vasm filename.s mem1 (will be executed on thread 1)
- spim447 –vasm filename.s mem2 (will be executed on thread 2)
- spim447 –vasm filename.s mem3 (will be executed on thread 3)

Mips_mem_sync.v has been modified to load memory data for up to four threads. Using address translation as described above, each thread will automatically access its own memory segments.

**Pipeline Diagrams**

As with previous projects, you should include all control and datapath wires and keep careful track of which values are stored in pipeline registers. It may be helpful to use different colors or line weights to separate control and datapath wires. Use discretion when choosing what details to include on your diagram (for instance, the various units and control structures within your ALU are probably not relevant, but all of the inputs and outputs definitely are). All diagrams MUST be drawn on a computer.

One diagram is required for each checkpoint in this project. The first diagram should show your pipeline with connections to the caches. The second checkpoint diagram should show your multithreaded pipeline, including thread-specific state and forwarding paths. The third diagram should show the interfacing of both pipelines to the shared cache and to memory.

**Hand in**

You should electronically hand in all of your Verilog files through the course AFS space in your project4 directory. Place a complete copy of each checkpoint in an appropriate subdirectory. Hand in a paper copy of your diagrams at your project demo and an electronic copy in your handin directory.

There is a single deadline for handin and check-off of all three checkpoints: April 30. Since this project is due at the very end of the semester, no late work can be accepted. You may, however, check-off and handin individual checkpoints during any lab period—we especially recommend finishing with checkpoints 1 and 2 prior to the final week of this project. There are three opportunities for earning bonus points by checking off early.

**Grading**

- **Checkpoint 1: Caches** 90
  - Functionality: I-cache 40
  - Functionality: D-cache 40
  - Pipeline diagram 10
- **Checkpoint 2: Multithreading** 240
  - Functionality: Correct execution & performance 120
  - Efficient issue scheduling 50
  - Efficient fetch scheduling 50
  - Pipeline diagrams 20
- **Checkpoint 3: Multicore** 120
  - Functionality: Correct execution & performance 60
  - Fair port arbitration 40
• Pipeline diagrams 20
• Total 450 points
• Extra Credit (50 max)
  • Checkpoint 1 checked-off by 4/9 10
  • Checkpoint 1&2 checked-off by 4/21 20
  • Checkpoint 1&2&3 checked-off by 4/28 20