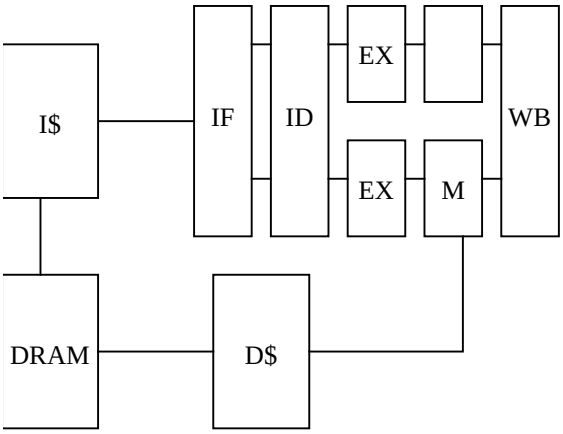
Some ADDITIONAL INFO FOR LAB 7

This document is intended to present a quick summary of the three main types of architectures in lab #7. It is not a complete reference, and is designed to help you decide which one of these architectures, if any, you wish to implement.

Super-Scalar: two pipelines, one instruction stream, and one cache.

A super-scalar machine looks at pairs of instructions and then executes them both, if possible. The IF stage needs to fetch two instructions, and then determine if both instructions can be issued in parallel (i.e., do they have a data hazard?) Additional complexity is added because data must be forwarded between the two pipes. In this implementation, notice that only one pipe has a real memory stage, so two lw/sw instructions can not be issued in parallel.

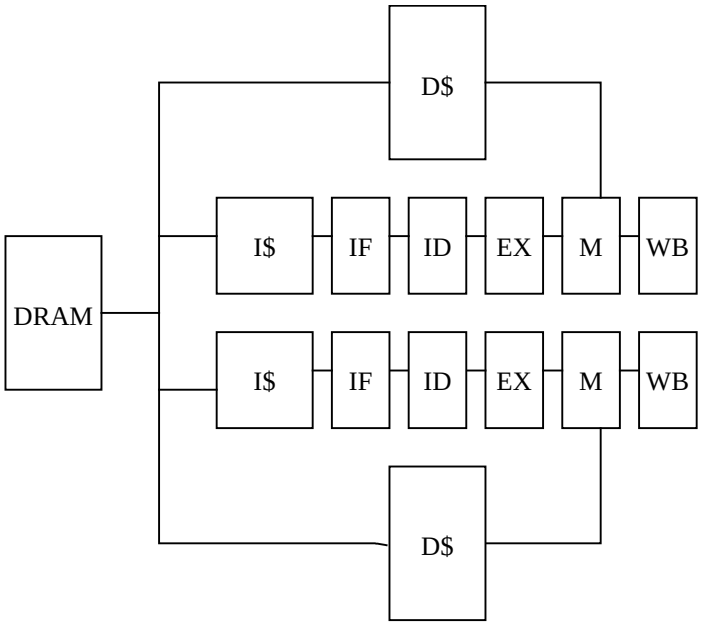


Ideally, super-scalar machines can issue 2 instructions every cycle. However, due to dependencies between instructions, typically only 1.3 instructions are issued per cycle, on average. Because it move instructions closer together (cycle wise), super-scalar architectures compound hazards between instructions. For example, instead of loads having one delay slot, super scalar machines typically have three load-delay slots (enforced in hardware). This increase in delay slots can really impact performance (which is why speed-ups are typically 1.3x instead of 2x). Techniques such as branch prediction and load-value prediction are useful in overcoming these difficulties.

For more information on super-scalar, please refer to H&P (look in the index).

Multi-Processing: two pipelines, two instruction streams, and two caches

Multiprocessing uses two complete pipelines (each with separate caches). The program is divided into two parts, and each is executed separately on a different processor. The two programs, however, share the same address space (common DRAM). To communicate between the two halves of the program, additional overhead must be added to the program. The difficulty with multi-processing implementations is cache-coherency: a write to one D-cache must be reflected in the other D-cache (if the line exists in the second D-cache) – the pipelines themselves are not changed.



Ideally, multi-processing machines can have a 2x speedup; also, they don't have many of the problems associated with super-scalar processors (increase in delay slots). However, the additional communication and synchronization overhead added at the program level will effectively reduce the speed-up. Also, since all the caches share one DRAM, there is also the issue of increased contention for the main memory. If the two executing threads are very independent, i.e. don't require a lot of communication, then multi-processing is a very effective way to achieve speed-up. It is expected that the communication and synchronization overhead will reduce the multi-processing speedup to be roughly on-par with that of a super-scalar implementation.

For more information on multi-processing, please refer to H&P, chapter 9, specifically section 9.3.

To allow execution of two threads in one memory space, we have declared that one processor should start execution at address 0x00, while the other should start execution at address 0x08. The reason for this is show by the piece of code below. When it finishes executing, one thread will be executing with the value 0 in register $1, and the other thread will be executing with the value 1 in register $1. This value can then be used to dynamically alter the program flow.

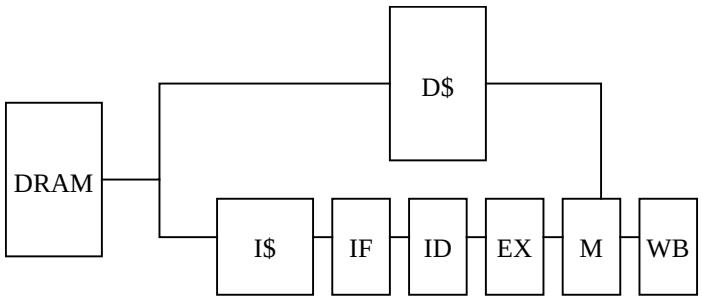
```
0x0000  b       0x000C
0x0004  addi    $1, $0, #0
0x0008  addi    $1, $0, #1
0x000C
```

Additionally, the two pieces of code must be able to communicate effectively. For example, the merge-sort algorithm divides a list to be sorted into two halves, and then combines the two sorted portions together into one sorted whole. In order for this to work cache coherency is necessary, else Thread2 would never see that Thread1 had completed. Cache coherency involves the detection of shared data between processors, as well as moving the data from one cache to the other (most likely by way of the DRAM).

```
            B     Thread2
            NOP
Thread1:    Sort 1st half
            SW    $0, [$0]          ; store 0 to indicate completion
Done1:      B     Done1
            NOP
Thread2:    Sort 2nd half
Again:      LW    $1, [$0]          ; check for completion
            BNE $1, $0, Again       ; loop until 1st part completion is detected
            NOP
            Merge the two halves
```

Multi-Threading: one pipeline, two instruction streams, and one cache

Multi-threading stores two thread contexts in the processor register file (requiring it to store 64 registers, instead of the standard 32). As can be seen in the diagram below, the datapath for a multi-threaded processor is almost identical to that of the standard MIPS pipeline (the major differences will be in the register-file and control logic). The idea being multi-threading is that because two threads are immediately available, a context switch can then be easily accomplished in only one cycle.



Given this ability to quickly switch threads, the system can immediately switch threads and continue processing when one thread misses in the cache. An example code sequence is given below. Note that there are two register \$2… one for Thread A and one for Thread B (this is why there are 64 registers). Effectively, there is a register \$2a and \$2b (i.e., use 6 bits to store the register number internally). At any given time, there may be instructions from two different threads in the pipeline (i.e., you may be fetching and decoding instructions from thread A, while executing an instruction from thread B).

Thread	Instruction	
A	add \$1, \$2, \$3	
A	lw \$2, [\$3]	*cache miss*
B	add \$4, \$5, \$2	
B	ori \$4, \$5, \$2	
B	lw \$4, [\$4]	*cache miss*
A	lw \$2, [\$3]	*restart instruction that caused miss in Thread A*

The critical decision for multi-threaded processors is *when* to switch contexts. In the previous example, the context was switched every time there was a cache miss. Another possibility is to switch every cycle; the advantage of this is that there no longer needs to be a forwarding path from the execute stage to the decode stage (there are never dependencies between instructions of different threads)– which would decrease the cycle-time of many processors. (It is not recommended to switch threads unconditionally every cycle for subtle reasons that I do not want to go into here). Other ideas are to switch contexts on branches, on every memory access (again, reduce cycle time), or every time there would be a stall in the pipeline for whatever reason (load-delay slots, etc…). Care must be taken to ensure fairness of the two threads… i.e., is it possible for the system to never context switch? (consider the case of switching on a cache miss when there are never any cache misses…)

Multi-threading uses efficiency to gain performance, not raw hardware. Although there is not the potential 2x speedup, problems such as cache stalls, contention, branch delays, etc… are avoided. The benefits of multi-threading will be specific to each pipeline implementation. The code-level synchronization issues with multi-threading are identical to those of multi-processing. However, they are much easier to implement because all accesses go through one cache, so cache-coherency is not a problem. Again, the expected speed-up from multi-threading is on-par with super-scalar or multi-processing.