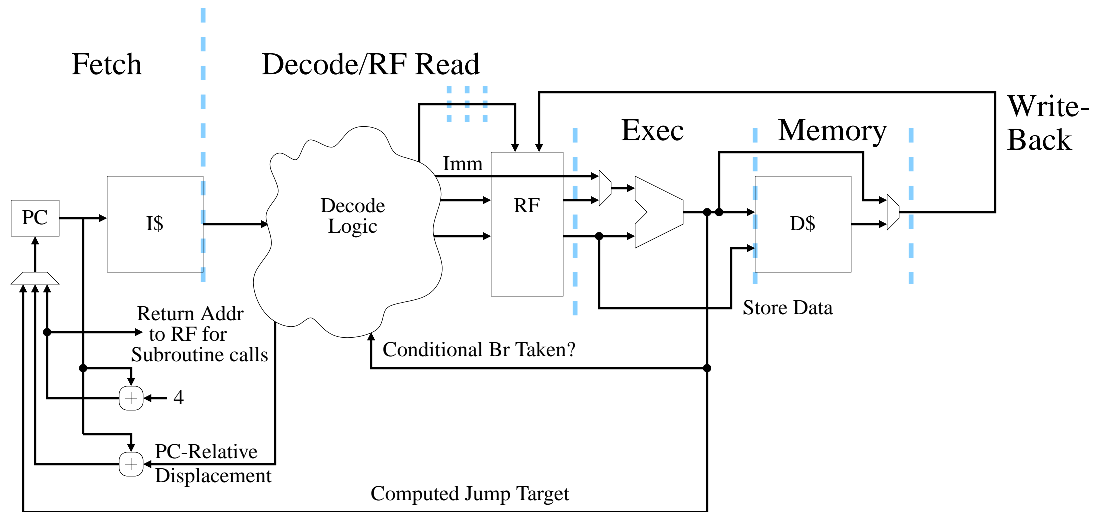
Table-Based Pipeline Control Logic

Prof. Loh
CS3220 - Processor Design - Spring 2005

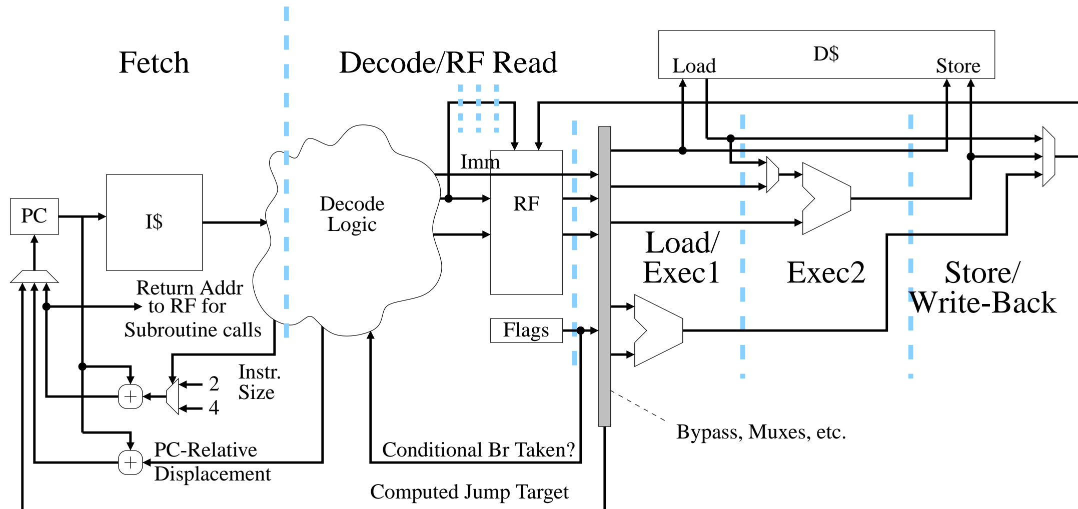
April 5, 2005

1 Basic w75 Pipeline

The traditional RISC five-stage pipeline has the following stages: Fetch, Decode, Execute, Memory Acess, and Write-Back. This is illustrated here:



The w75 ISA allows for more complex instructions that, for example, may require multiple memory accesses. An instruction such as `add *r1 r2` requires a load from memory for the value pointed to by `r1`, adding that value to `r2`, and then writing the result back to the original memory location. This requires some changes to the pipeline organization, but we can still fit this all into a five-stage pipeline that looks like this:



We effectively have two pipelines. The upper pipeline handles loads, load-ops, stores, and load-op-stores. The bottom pipeline handles regular ops. Note that even though physically we have two pipelines, there will still only be a single instruction per pipeline stage. For example we will not have an `add` in the LOAD/EXEC$_1$ stage of the bottom pipeline and simultaneously have a `load` in the LOAD/EXEC$_1$ stage of the upper pipeline.

With multiple functional units (ALUs, load ports from the data cache) spread out over multiple pipeline stages, there are now several possible locations where an in-flight register value may resides (in-flight means the value has been produced, but has not yet written back to the register file). Each one of these points represents a location where we must be able to bypass values *from*, and these will be routed *to* all of the possible locations where the value may be consumed.

The bypass paths represent a significant amount of wiring (each line in the figure represents 32 bits of data (or 64 for x86-64, or 128 for SSE2). The bypass multiplexors at the inputs of the functional units are correspondingly wide (how many gates are needed to implement a 7-to-1, 128-bit multiplexor?). Besides all of these wires and all of this logic, there is implicitly some additional control logic that provides the control signals to make sure that the data values get bypassed from the correct producer to the correct consumer at the correct point in time. In this set of notes, we will make this control logic explicit.

2 What Do We Need to Control?

Describing execution and dataflow can occur from multiple perspectives. From an instruction's point-of-view, we need to know:

- Where am I? (What pipe stage or resource)
- Where am I reading from? (RF, immediate or bypass source)
- Where am I writing to? (RF)

From a functional unit's point-of-view:

- What instruction am I executing? (opcode)
- Where am I reading from? (RF, immediate or bypass source)
- Where am I writing to? (RF)

And from the register file's point-of-view:

- Who's reading me?
- Who's writing me?

To provide all of the control signals for the datapath of our pipelined processor, we basically need to keep track of all of this information. Note that some of the information listed above ends up being redundant with some of the other points, but one way or another we need to track the union of all of this data. One could "simply" implement the logic as a traditional finite-state machine, but the number of states is *very* large and the corresponding state transition diagram would be a mess. Another alternative is to use a table or *scoreboard* to track all of the information in a more organized fashion.

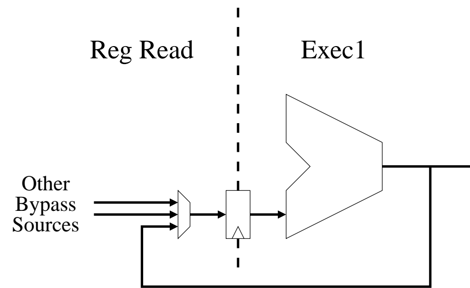
In our pipelined processor resource scoreboard, we will track the location and readiness of registers and functional units. Considering only integer resources for now (no FP registers, no FP functional units), the table looks like:

	Load Port	Exec1	Exec2	Store port
Busy/In-Use				
Available Next Cycle				

	R0	R1	R2	R3	R4	R5	R6	R7
Location								
Available Next Cycle								

For each execution resource (functional unit), the table keeps track of whether the unit is currently in use, and how many cycles until the resource will be ready - 1 (therefore 0 means the resource will be ready next cycle). In the case of multi-cycle latency instructions (such as division), the functional unit may be in use for many cycles before it becomes available for the next instruction. Similarly for the registers, the table tracks how many cycles until the register will be ready and where in the pipeline the value is located (so that you can correctly bypass it to any consumers).

We assume that bypassing occurs at the very end of a cycle such that (for example) an ALU output will traverse the bypass paths, go through the bypass mux, and then stop at the inter-stage pipeline latch:



3 Basic Register Forwarding

Consider the following set of four w75 instructions:

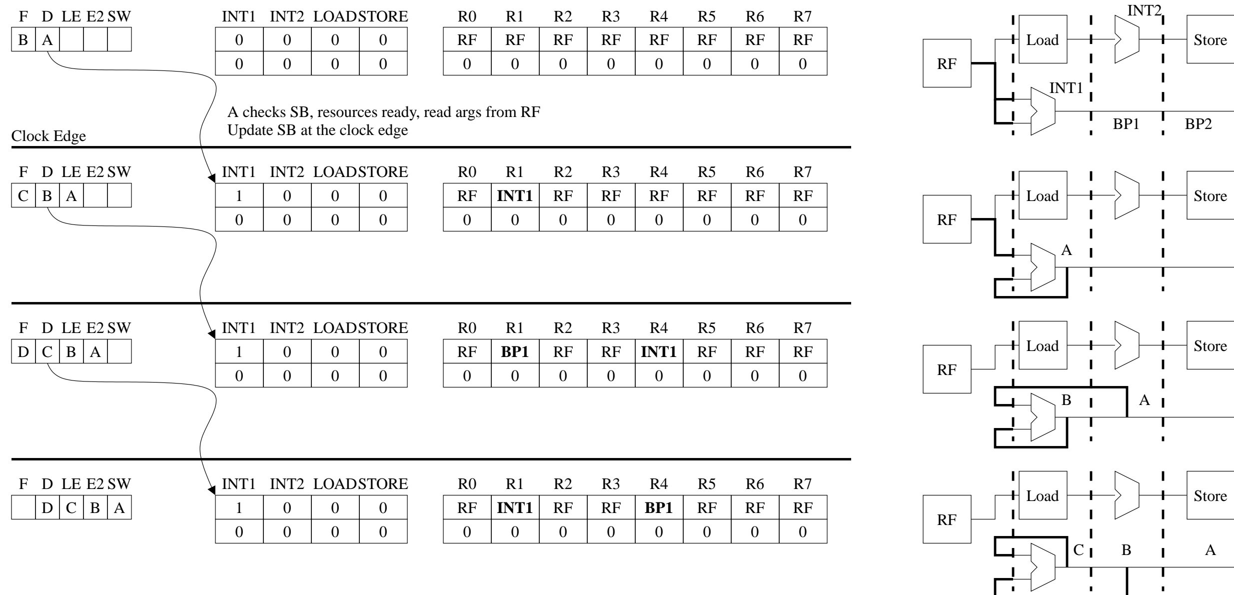
A. ADD R1 R2 (R1 = R1 + R2)

B. SUB R4 R1 (R4 = R4 – R1)

C. ADD R1 R4 (R1 = R1 + R4)

D. XOR R1 R4 (R1 = R1 $\oplus$ R4)

Initially, all register values are located in the register file, and all execution resources are ready and available. The following illustrate how the state of the scoreboard gets updated as each instruction proceeds to execution. It also illustrates how each instruction reads the scoreboard to determine where to find its operands in the pipeline.



When instruction A reaches the decode stage, it reads its arguments from the regsiter file. In parallel, it consults the scoreboard to discover if (a) its registers are actually available because it could be possible that an earlier instruction has not yet written its result back to the RF, and (b) where it is located if in fact it is available. In this case, the scoreboard indicates that all arguments are in fact ready and can be selected from the RF outputs. This means instruction A can

proceed to execution and it will update the scoreboard to reflect this. When the clock edge occurs, the scoreboard will load in instruction A's updated scoreboard information.

In this second cycle, instruction B consults the scoreboard. It sees that all of its arguments are ready (or will be ready by the end of the cycle such as R1 which is being produced during the earlier part of the cycle) and that INT1 (Integer ALU 1) will be available next cycle (it's busy right now, but A only needs it for this current cycle). The scoreboard indicates that its argument R4 is in the register file, and so the RF output gets routed to the first ALU input. The scoreboard also indicates the its second argument R1 is currently located at the output of INT1. The bypass mux (not shown in the figure) chooses the bypass path that originates from INT1's output (the path is shown in the figure). Since all of B's resources will be ready, it may proceed to execute in the next cycle. Correspondingly, it, too, updates the scoreboard.

In the third cycle illustrated, instruction C reaches the decode stage and consults the scoreboard. Again, the scoreboard indicates that all of C's resources are ready. This time, both of C's parents are still in the pipeline and have not yet written their results back to the register file. This is not a problem because the scoreboard explicitly tracks the locations of these "in-flight" register values. Using this information, C knows its first argument R1 is located at location BP1, and its second argument R4 is located at INT1. C also accessed the register file (instructions always will), but in this instance the outputs of the RF are simply ignored. The figure shows these result being forwarded to the latches at the end of the decode/register-read cycle for instruction C. C then proceeds to update the scoreboard.

The fourth cycle is similar to the previous several cycles. Instruction D checks the scoreboard and discovers that its arguments are ready and finds their locations. Notice that the pipeline now contains two *instances* of register R1. Instruction A produces a value for R1 and is located at position BP2. Instruction C also produces a version of R1, but it is located at INT1. D needs to read the most recent version, but because the instructions update the scoreboard in-order, the most recent producer of a register will always be the most recent instruction to update the scoreboard. Hence, the scoreboard will point to the newest instance of the register. In this case, the scoreboard tells D that it should read its R1 argument from location INT1, which is correct.

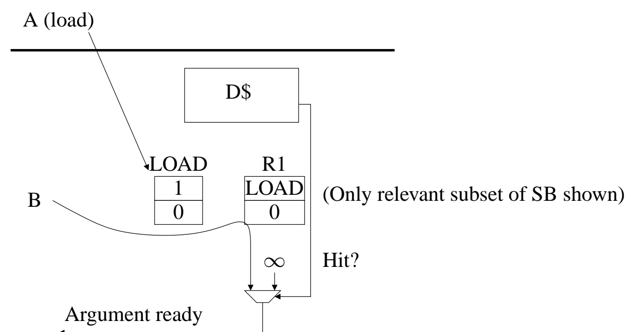
4 Load Miss Stalls

The previous example was fairly straight-forward since it only contained simple ALU instructions (although the data-dependencies were still interesting). Consider the following instructions:

A. MOV R1 *R2 (LOAD R1 = [R2])

B. ADD R4 R1 (R4 = R4 + R1)

When instruction A proceeds to execute, it also updates the scoreboard indicating that its result will be available at the LOAD unit. At the time A updates the scoreboard (at the clock edge between decode and load), it hasn't yet even started accessing the cache. This means the load does not know if it will hit in the cache or not. So how can the load set the scoreboard entry for whether the value will be available next cycle or not? To get around this, the load can set the scoreboard to indicate that the value will be ready, and then provide a late override if a cache miss occurs.



When B enters the decode stage, A proceeds to the load stage and accesses the cache. B reads the scoreboard information for its sources (only the R1 lookup is illustrated) although the readiness information in this case assumes

that A will hit in the cache. Toward the end of the cycle when the cache access has completed, A will know whether there was a hit or a miss. If there was a hit, then the value from the scoreboard is passed back to B. If there was a miss, then a non-zero value gets passed back to B which indicates that there will be more than one cycle before the value is ready (and therefore B must stall until the value is actually available).

5 Multi-Cycle Instructions

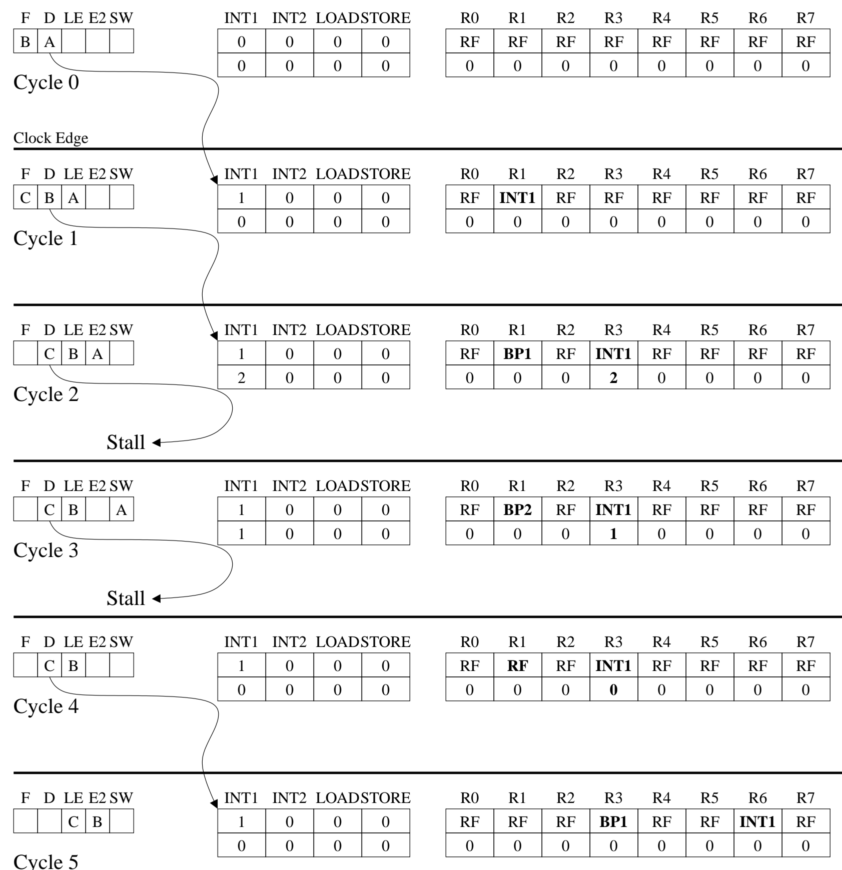
Some instructions such as multiplication and division may take multiple cycles to execute. In the presence of such instructions, dependent instructions may need to wait multiple cycles before reading their arguments and proceeding to execution. Consider the following instructions:

A. ADD R1 R2 (R1 = R1 + R2)

B. MUL R3 R1 (R3 = R3 × R1)

C. SUB R6 R3 (R6 = R6 - R3)

For an instruction like multiply, the latency of execution (number of cycles required to produce the result) is constant as a result of the implementation of the functional unit. In this case, when the multiply updates the scoreboard, it can load in its latency - 1 into the field that indicates when the result will be ready. The following example assumes a three cycle multiply latency.

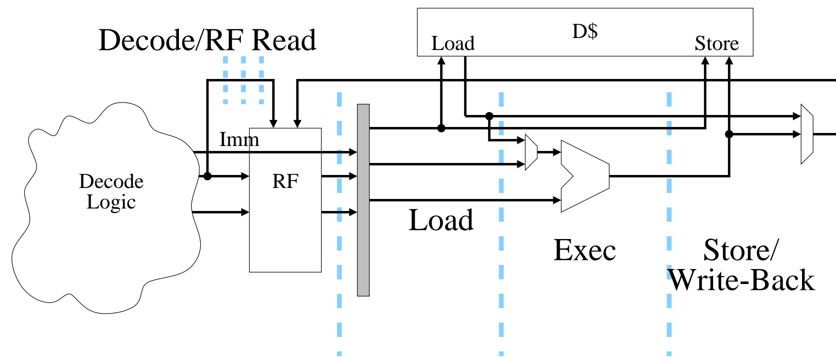


Instruction A is an add and proceeds as usual. When instruction B enters execution (cycle 2), the scoreboard entries corresponding to its resources (R3 and INT1) get updated with the number of cycles remaining until the resources will be available. Since the multiply latency is three, the resources will be available in another *two* cycles. The multiply

executes during cycles 2, 3 and 4 for a total of its three cycles of latency. In each cycle, the counters associated with B's resources get decremented. Instruction C continues to poll the scoreboard entries awaiting its resources to become ready. Finally in cycle 4 in B's last cycle of execution, the scoreboard counters reach zero, which tells instruction C that it can proceed to execute in the next cycle (cycle 5). Similar to the previous examples, the value of R3 will be ready toward the end of cycle 4, at which point C will select it using the bypass muxes located in the decode/register read stage (refer to the pipeline figure from the first example).

6 Consolidation of Execution Resources — Pipeline Uniformity

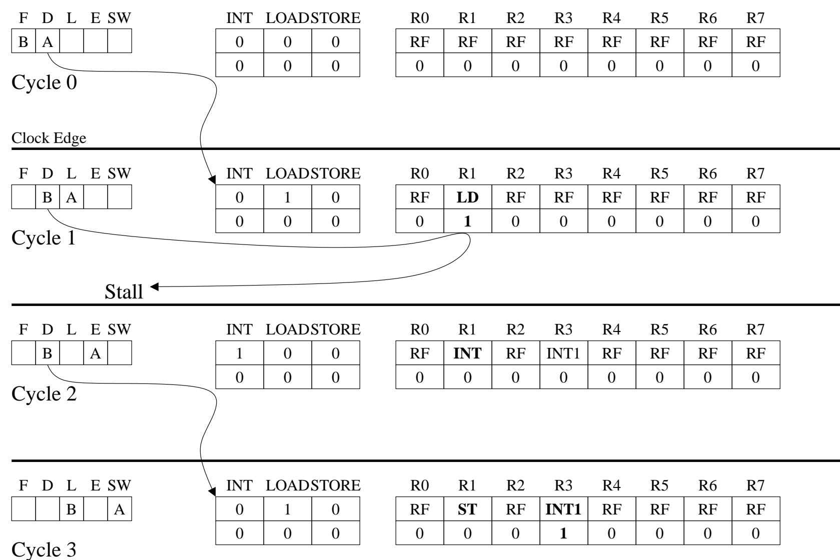
Up to this point, we have assumed that there are two execution units available. One to perform simple instructions that do not involve memory operations, and a second to support load-op and load-op-store instructions. Note that the w75 instructions have forms for a stand-alone multiply, as well as a load-multiply-store. This implies that *both* ALUs INT1 and INT2 must be able to support multiplication. The same is true for division, which means we need to implement *two* SRT dividers! The hardware cost for having two of everything is quite high (in terms of both chip area, complexity, design time, and power), and so this is not a desireable situation. One way to eliminate the second set of execution resources is to force all instructions through the load-op-store datapath:



Instructions that do not require memory accesses simply do not access the cache in the load and store stages. Now consider the following instructions:

A. ADD R1 R2 (R1 = R1 + R2)

B. SUB R3 R1 (R3 = R3 - R1)



When instruction A leaves the decode stage, it updates the scoreboard entry for register R1 with a value of 1. This is because the add instruction must spend a cycle in the load stage before proceeding on to the execution stage, thus requiring two cycles (from decode) to produce a result. At the end of cycle 1, instruction B is not allowed to proceed to the next stage because the scoreboard shows a non-zero count for one of its resources (R1) since instruction A is still only at the load stage (where it doesn't actually do anything since it doesn't need to access the cache). On cycle 2, instruction A proceeds to execute, and by the end of this cycle will have produced the value for R1. Instruction B grabs the value at the end of the cycle, and on cycle 3 proceeds to the load stage.

The way we have reorganized the pipeline, any two dependent instructions can no longer execute in back-to-back cycles. On the other hand, if instruction B did not require A's output, then it could have read its arguments from the register file and proceeded to the load stage during cycle 2. While it is theoretically possible to try to find independent instructions to place inbetween dependent instructions to prevent bubbles, it is difficult to fill all of these slots in practice.

An alternative is to allow B to access the scoreboard more than once. First in the decode stage, B checks the scoreboard and reads whatever register values it needs from the register file and other bypass locations (for arguments that have been produced but not written back). If the load stage is available, B then proceeds to the load stage, *regardless of whether all of its operands are ready*. In the load stage, B checks the scoreboard again to determine if any of its remaining operands are ready. At this point, if any resources required by B for execution are not ready, it must stall. On the other hand, if all resources are available, then it can proceed to the execute stage without having had a bubble injected into the pipeline.

The hardware cost to perform an additional read from the load stage is that the scoreboard will require an extra read port. Note that the scoreboard is basically an SRAM-like structure and it actually requires quite a few read and write ports since many resources in different stages of the pipeline all need to update the state of the scoreboard. If not carefully designed, the scoreboard can easily become the slowest component which ends up limiting the clock frequency of the pipeline.

7 Memory Dependencies

In a traditional RISC pipeline, there is only a single point to access the data cache. In the w75 pipeline, a load-op-store instruction may access the cache once in the load stage and then a second time in the store stage. This creates the possibility of a new type of hazard or dependency between instructions: memory dependencies. A memory dependency is basically the same as a data dependency, but in this case the data is communicated through a memory location rather than a register.

Consider the following instructions:

A. MOV R5 R1 (R5 = R1)

B. ADD *R1 R2 ([R1] = [R1] + R2)

C. SUB R3 *R5 (R3 = R3 - [R5])

The first instruction sets R5 equal to R1. Instruction B uses the address stored in R1 to load a value, add R2 to it, and then write it back to memory. Instruction C uses the address stored in R5 to load a value, and then subtracts that value from R3. Note that since R1 = R5, both B and C refer to the same memory location. In this case, we must stall instruction C until B has completed the store operation. If C performs its load too early, then it will read a stale value from the cache.

For register data dependencies, it is easy to detect the existance of a depedency by simply comparing the register specifier fields or by using our scoreboard data structure. Directly inspecting the instructions does not help. For example, instructions B and C in our example have absolutely no register fields in common. The problem with scoreboarding the memory locations is that it is not practical to track the readiness of every possible location in memory. A load in the decode stage must stall so long as there exists a store (include load-op-stores) somewhere further down the pipeline that has not yet written its value back to the cache.