MPCA unit-2

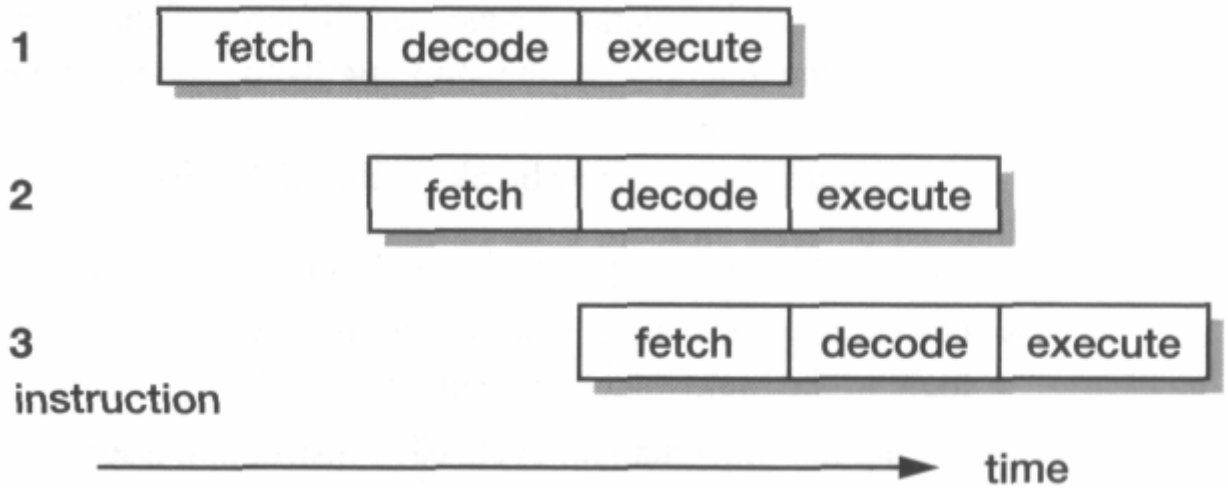
T2-Arm System-On-Chip Architecture

4.1 3-stage pipeline ARM organization

• *Fetch*;
the instruction is fetched from memory and placed in the instruction pipeline.

• *Decode*;
the instruction is decoded and the datapath control signals prepared for the next cycle. In this stage the instruction 'owns' the decode logic but not the datapath.

• *Execute*;
the instruction 'owns' the datapath; the register bank is read, an operand shifted, the ALU result generated and written back into a destination register.



The simplest way to view breaks in the ARM pipeline is to observe that:
• All instructions occupy the datapath for one or more adjacent cycles.

• For each cycle that an instruction occupies the datapath, it occupies the decode logic in the immediately preceding cycle.

• During the first datapath cycle each instruction issues a fetch for the next instruction but one.

• Branch instructions flush and refill the instruction pipeline.

PC Behaviour :
One consequence of the pipelined execution model used on the ARM is that the program counter, which is visible to the user as r!5, must run ahead of the current instruction. If, as noted above, instructions fetch the next instruction but one during their first cycle, this suggests that the PC must point eight bytes (two instructions) ahead of the current instruction.

4.2 5-stage pipeline ARM organization

The time T , required to execute a given program is given by:

$$T_{prog} = \frac{N_{inst} \times CPI}{f_{clk}}$$

where N_{inst} is the number of ARM instructions executed in the course of the program, CPI is the average number of clock cycles per instruction and f_{clk} is the
processor's clock frequency.

there are only two ways to increase performance:
• Increase the clock rate, f clk
• Reduce the average number of clock cycles per instruction, CPI

Memory bottleneck :
The fundamental problem with reducing the CPI relative to a 3-stage core is related to the von Neumann bottleneck - any stored-program computer with a single instruction and data memory will have its performance limited by the available memory bandwidth

The 5-stage pipeline

The ARM processors which use a 5-stage pipeline have the following pipeline stages:

• *Fetch*;
the instruction is fetched from memory and placed in the instruction pipeline.

• *Decode*;
the instruction is decoded and register operands read from the register file. There are three operand read ports in the register file, so most ARM instructions can source all their operands in one cycle.

• *Execute*;
an operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU.

• *Buffer/data*;
data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock cycle to give the same pipeline flow for all instructions.

• Write-back;
the results generated by the instruction are written back to the register file, including any data loaded from memory.

Data forwarding
A major source of complexity in the 5-stage pipeline (compared to the 3-stage pipeline) is that, because instruction execution is spread across three pipeline stages, the only way to resolve data dependencies without stalling the pipeline is to introduce forwarding paths.

T1 - C1

What Is Pipelining?

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.

The throughput of an instruction pipeline is determined by how often an instruction exits the pipeline

The time required between moving an instruction one step down the pipeline is a processor cycle.

the time per instruction on the pipelined processor—assuming ideal conditions—is

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

the speedup from pipelining equals the number of pipe stages

RISC architectures are characterized by a few key properties, which dramatically simplify their implementation:

■ All operations on data apply to data in registers and typically change the entire register (32 or 64 bits per register).

■ The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, respectively.
Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.

■ The instruction formats are few in number, with all instructions typically being one size.

Every instruction in this RISC subset can be implemented in at most 5 clock cycles. The 5 clock cycles are as follows.

1. *Instruction fetch cycle (IF):*
   Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 (since each instruction is 4 bytes) to the PC.
2. *Instruction decode/register fetch cycle (ID):*
   Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers
   as they are read, for a possible branch. Sign-extend the offset field of the
   instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC.
3. *Execution/effective address cycle (EX):*
   The ALU operates on the operands prepared in the prior cycle, performing
   one of three functions depending on the instruction type.
   ■ Memory reference—The ALU adds the base register and the offset to form the effective address.
   ■ Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.
   ■ Register-Immediate ALU instruction—The ALU performs the operation
   specified by the ALU opcode on the first value read from the register file
   and the sign-extended immediate.
4. Memory access (MEM):
   If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory
   writes the data from the second register read from the register file using the effective address.

5. Write-back cycle (WB):
   ■ Register-Register ALU instruction or load instruction:
   Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

There are three observations on which this fact rests :

1. we use separate instruction and data memories, which we would typically implement with separate instruction and data caches
2. the register file is used in the two stages: one for reading in ID and one for writing in WB.
3. To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage in preparation for the next instruction

Although it is critical to ensure that instructions in the pipeline do not attempt to use the hardware resources at the same time, we must also ensure that instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing pipeline registers between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle

throughput—the number of instructions completed per unit of time

pipelining usually slightly increases the execution time of each instruction due to overhead in the control of the pipeline. The increase in

instruction
throughput means that a program runs faster and has lower total execution time, even though no single instruction runs faster!

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

C.2 The Major Hurdle of Pipelining—Pipeline Hazards

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining.
There are three classes of hazards:

1. Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
2. Data hazards arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. Control hazards arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to stall the pipeline. Avoiding a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed

Performance of Pipelines with Stalls

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}} \\ &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

Structural Hazards

When a processor is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have a structural hazard.

- To resolve this hazard, we stall the pipeline for 1 clock cycle when the data memory access occurs. A stall is commonly called a pipeline bubble or just bubble, since it floats through the pipeline taking space but carrying no useful work.

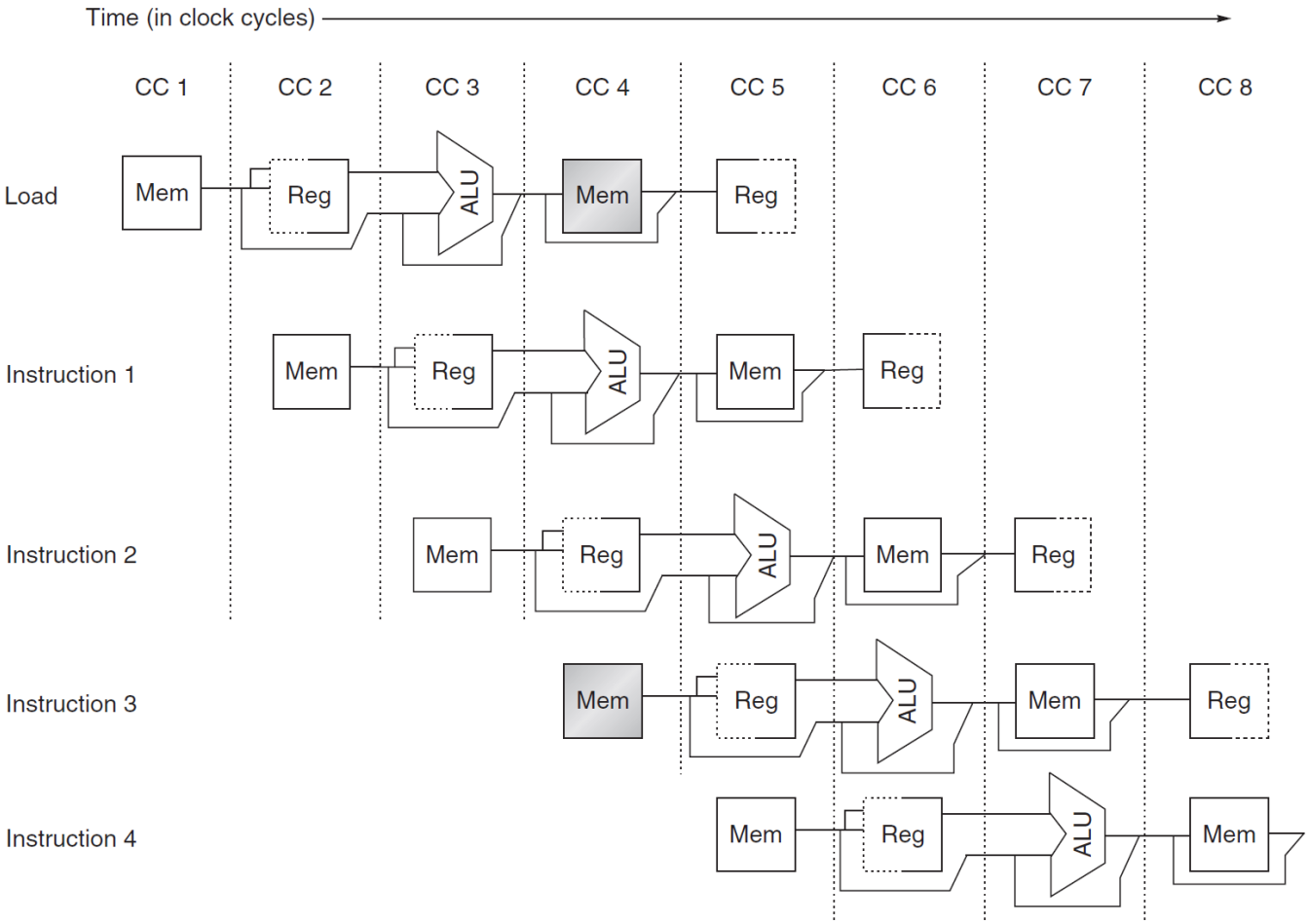


Figure C.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

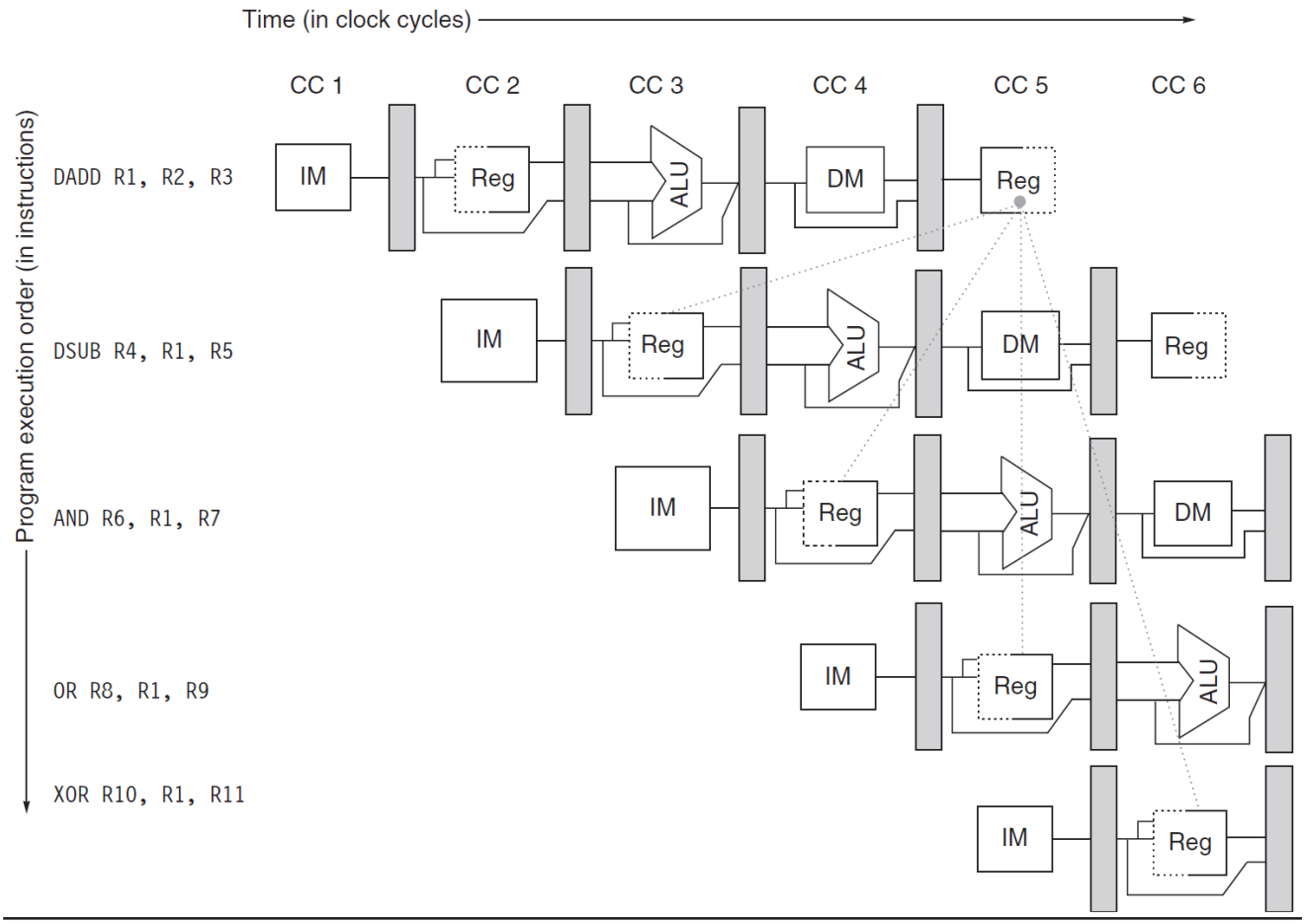
	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				Stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Data Hazards

Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.

eg 1 :

```
DADD        R1,R2,R3
DSUB        R4,R1,R5
AND         R6,R1,R7
OR          R8,R1,R9
XOR         R10,R1,R11
```



- DSUB instruction will read the wrong value and try to use it. In fact, the value used by the DSUB instruction is not even deterministic
- The AND instruction is also affected by this hazard.
- The XOR instruction operates properly because its register read occurs in clock cycle 6, after the register write.

- The OR instruction also operates without incurring a hazard because we perform the register file reads in the second half of the cycle and the writes in the first half.

Minimizing Data Hazard Stalls by Forwarding
(also called bypassing and sometimes short-circuiting)

The key insight in forwarding is that the result is not really needed by the DSUB until after the DADD actually produces it. If the result can be moved from the pipeline register where the DADD stores it to where the DSUB needs it, then the need for a stall can be avoided.

forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.
2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

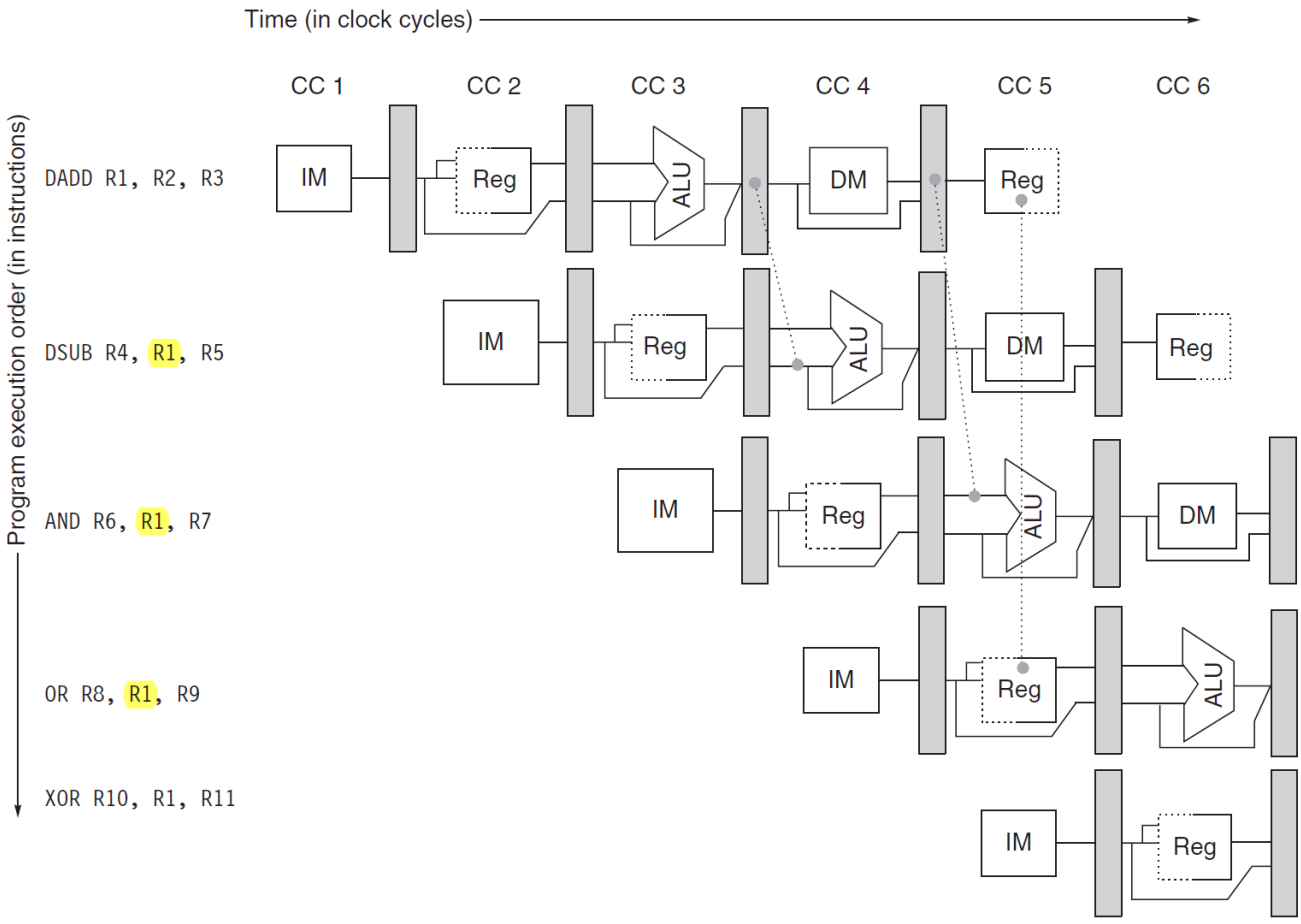


Figure C.7 A set of instructions that depends on the DADD result uses forwarding paths to avoid the data hazard. The inputs for the DSUB and AND instructions forward from the pipeline registers to the first ALU input. The OR receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. ~~Notice that the~~

eg 2 :

```
DADD    R1,R2,R3
LD      R4,0(R1)
SD      R4,12(R1)
```

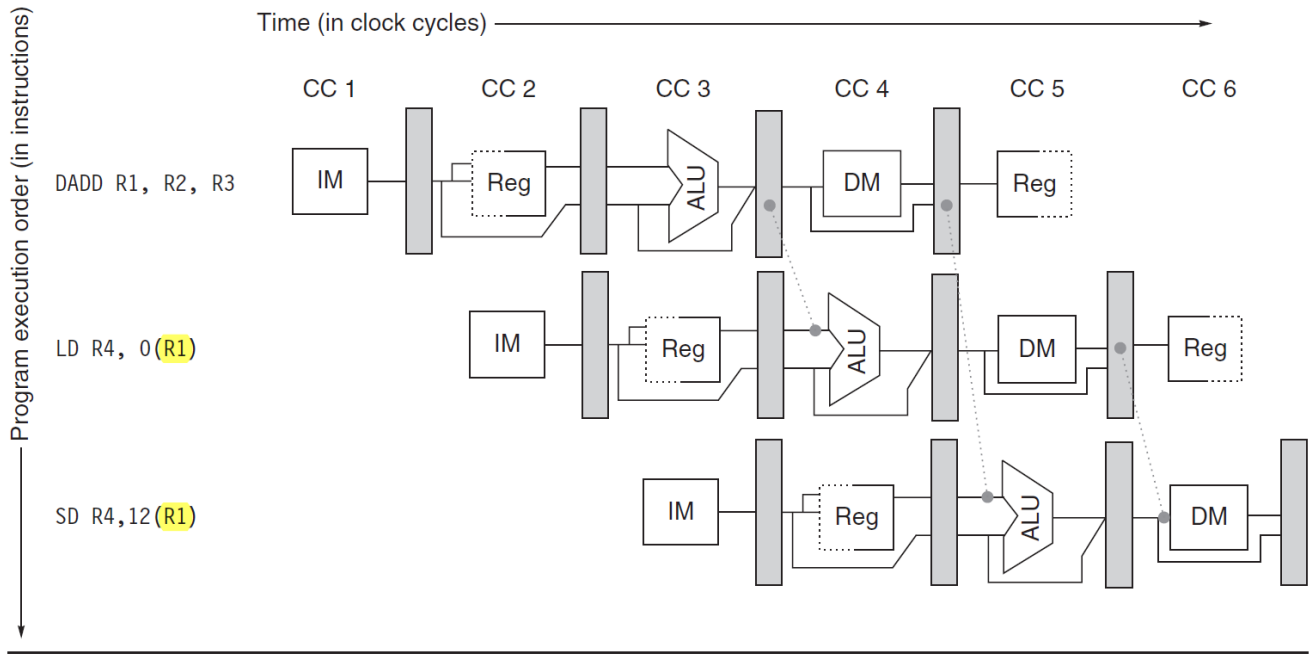


Figure C.8 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown above), the result would need to be forwarded to prevent a stall.

Data Hazards Requiring Stalls

Unfortunately, not all potential data hazards can be handled by bypassing. Consider the following sequence of instructions:

```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

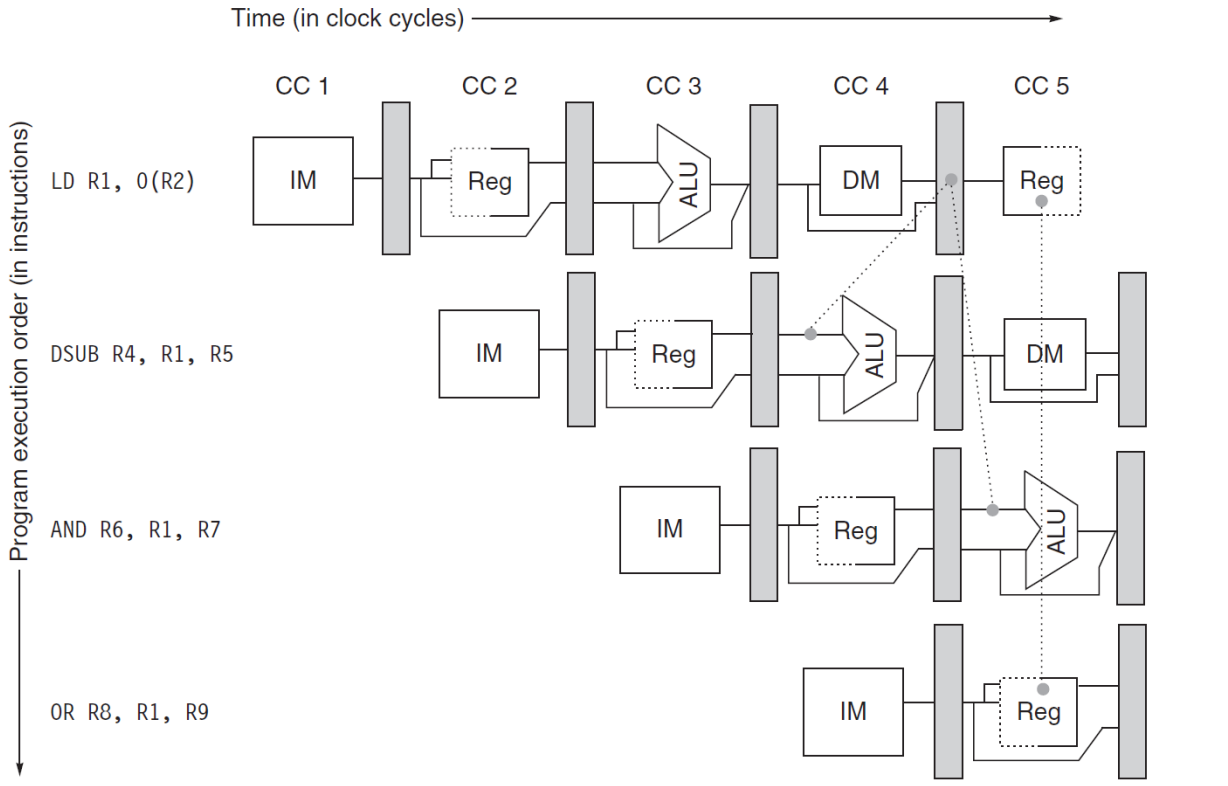


Figure C.9 The load instruction can bypass its results to the AND and OR instructions, but not to the DSUB, since that would mean forwarding the result in "negative time."

The pipelined data path with the bypass paths for this example is shown in Figure C.9. This case is different from the situation with back-to-back ALU operations. The LD instruction does not have the data until the end of clock cycle 4 (its MEM cycle), while the DSUB instruction needs to have the data by
the beginning of that clock cycle. Thus, the data hazard from using the result of a load instruction cannot be completely eliminated with simple hardware. As Figure C.9 shows, such a forwarding path would have to operate backward

in time—a capability not yet available to computer designers! We can forward
the result immediately to the ALU from the pipeline registers for use in the AND operation, which begins 2 clock cycles after the load. Likewise, the OR instruction has no problem, since it receives the value through the register file.
For the DSUB instruction, the forwarded result arrives too late—at the end of a
clock cycle, when it is needed at the beginning.

The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called a *pipeline interlock*, to preserve the correct execution pattern. In general, a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared. Basically **pipeline interlock -> stall**

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	

LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND	R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR	R8,R1,R9				stall	IF	ID	EX	MEM	WB

Figure C.10 In the top half, we can see why a stall is needed: The MEM cycle of the load produces a value that is needed in the EX cycle of the DSUB, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

Branch Hazards

Figure C.11 shows that the simplest method of dealing with branches is to redo the fetch of the instruction following a branch, once we detect the branch
during ID (when instructions are decoded). The first IF cycle is essentially a stall, because it never performs useful work. You may have noticed that if the branch is untaken, then the repetition of the IF stage is unnecessary

since the correct instruction was indeed fetched. We will develop several schemes to take advantage of this fact shortly

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Figure C.11 A branch causes a one-cycle stall in the five-stage pipeline. The instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for branch successor is redundant. This will be addressed shortly.

Reducing Pipeline Branch Penalties

- The simplest scheme to handle branches is to freeze or flush the pipeline, holding or deleting any instructions after the branch until the branch destination is known. The attractiveness of this solution lies primarily in its simplicity both for hardware and software. It is the solution used earlier in the pipeline shown in Figure C.11. In this case, the branch penalty is fixed and cannot be reduced by software.

- A higher-performance, and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed. Here, care must be taken not to change the processor state until the branch outcome is definitely known. The complexity of this scheme arises from having to know when the state might be changed by an instruction and how to “back out” such a change.
- In the simple five-stage pipeline, this predicted-not-taken or predicted untaken scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. The pipeline looks as if nothing out of the ordinary is happening. If the branch is taken, however, we

need to turn the fetched instruction into a no-op and restart the fetch at the target address. Figure C.12 shows both situations.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	ID	EX	MEM	WB			
Instruction $i + 2$			IF	ID	EX	MEM	WB		
Instruction $i + 3$				IF	ID	EX	MEM	WB	
Instruction $i + 4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i + 1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.12 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we fetch the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall 1 clock cycle.

- An alternative scheme is to treat every branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target. Because in our five-stage pipeline we don’t know the target address any earlier than we know the branch outcome, there is no advantage in this approach for this pipeline (although this might be useful is come processors which make use of condition code hence the branch target is known before the branch outcome).
- A fourth scheme in use in some processors is called delayed branch. This technique was heavily used in early RISC processors and works reasonably well in the five-stage pipeline. In a delayed branch, the execution cycle with a branch delay of one is

branch instruction
sequential successor$_1$
branch target if taken

The sequential successor is in the branch delay slot. This instruction is executed whether or not the branch is taken. The pipeline behavior of the five-stage pipeline with a branch delay is shown in Figure C.13. Although it is possible to have a branch delay longer than one, in practice almost all processors with delayed branch have a single instruction delay.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Figure C.13 The behavior of a delayed branch is the same whether or not the branch is taken. The instructions in the delay slot (there is only one delay slot for MIPS) are executed. If the branch is untaken, execution continues with the instruction after the branch delay instruction; if the branch is taken, execution continues at the branch target. When the instruction in the branch delay slot is also a branch, the meaning is unclear: If the branch is not taken, what should happen to the branch in the branch delay slot? Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot.

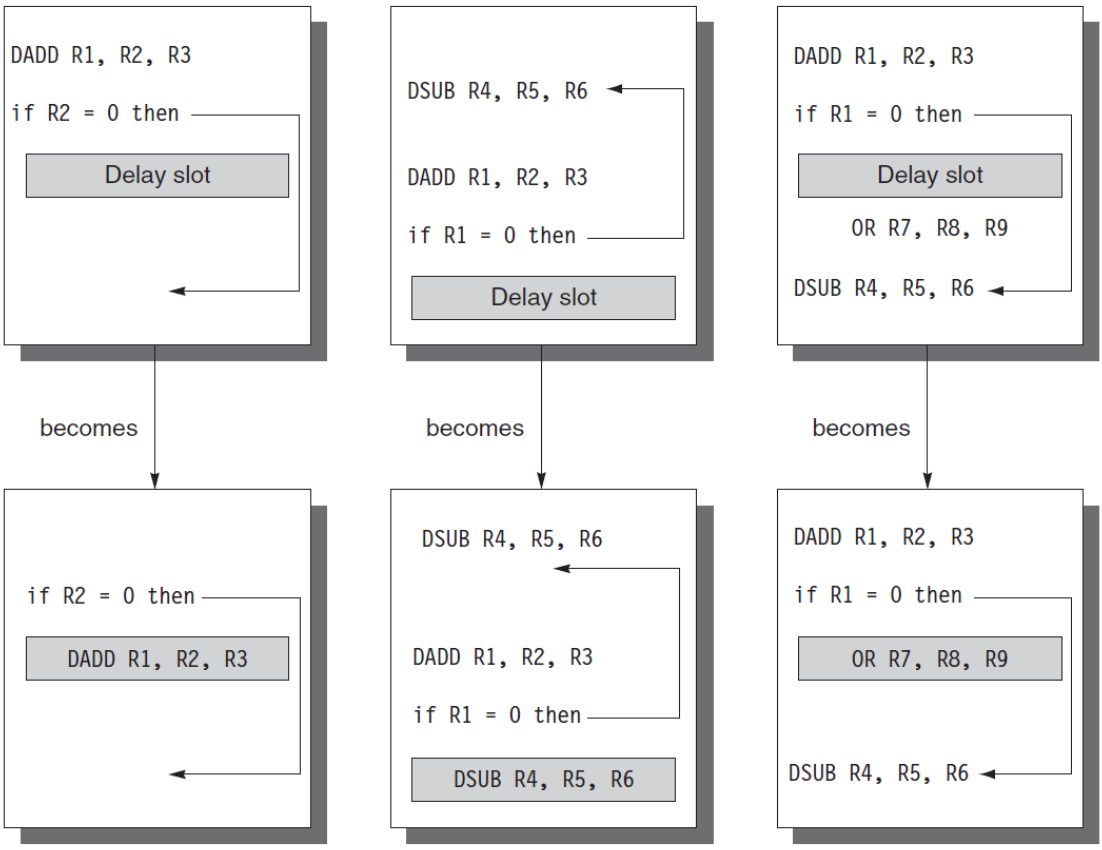


Figure C.14 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of R1 in the branch condition prevents the DADD instruction (whose destination is R1) from being moved after the branch. In (b), the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the moved instruction when the branch goes in the unexpected direction. By OK we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, in (c) if R7 were an unused temporary register when the branch goes in the unexpected direction.

The job of the compiler is to make the successor instructions valid and useful.

A number of optimizations are used. Figure C.14 shows the three ways in which the branch delay can be scheduled.

The limitations on delayed-branch scheduling arise from:

1. The restrictions on the instructions that are scheduled into the delay slots
2. Our ability to predict at compile time whether a branch is likely to be

taken or not.

- In a *canceling/nullifying* branch, the instruction includes the direction that the branch was predicted. When the branch behaves as predicted, the instruction in the branch delay slot is simply executed as it would normally be with a delayed branch. When the branch is incorrectly predicted, the instruction in the branch delay slot is simply turned into a no-op.

Performance of Branch Schemes

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$