Arquitecturas de Alto Desempenho

Instruction-Level Parallelism (Principles)

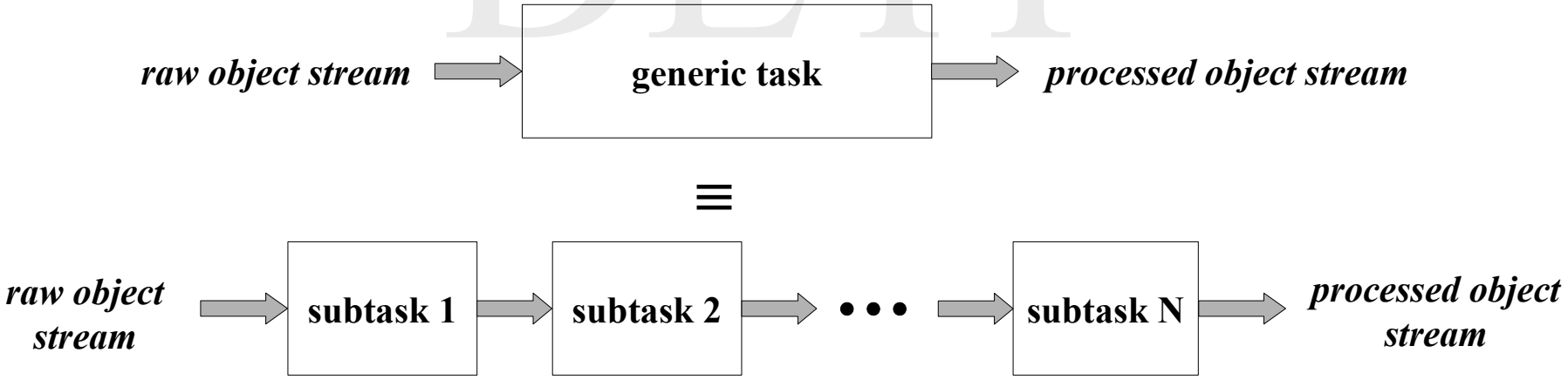
António Rui Borges

Summary

- *What is pipelining?*
- *Instruction-level parallelism*
- *Simplified RISC instruction set*
- *Non-pipelined implementation of RISC instruction set*
- *Classical 5-stage pipeline for a RISC processor*
- *Major hurdles of pipelining*
  - *Performance of pipelines with stalls*
  - *Structural hazards*
  - *Data hazards*
  - *Control hazards*
- *Implementation of classical 5-stage pipeline*
- *Exceptions*
- *Multicycle operations in classical 5-stage pipeline*
- *MIPS R4000 pipeline*
- *Suggested reading*

What is pipelining? - 1

Pipelining is an implementation technique where the execution of a generic task on the objects of a stream is converted into a tandem of independent subtasks which operate simultaneously on successive objects of the stream. Each of the individual subtasks, called *pipe stages* or *segments*, is performed in sequence and represents a definite fraction of the whole task. Their combined ordered execution is equivalent to the execution of the original task on every object of the stream.



What is pipelining? - 2

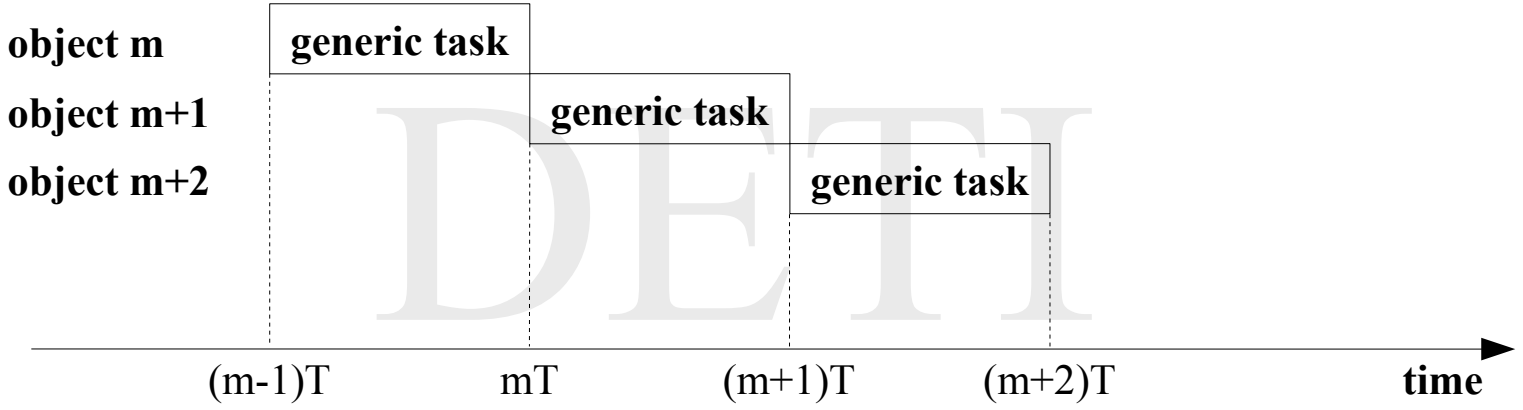
The concept of pipelining was invented in the context of the automobile industry by Ford Motor Company in the beginning of the 1900s, inspired by the operating procedures of the Chicago cattle slaughterhouses. The first factory which introduced a fully developed moving assembly line, started operations by the end of 1913 in Detroit for the production of Ford Model T.

The goal of a moving assembly line was to speed up the process of building a car from its parts and, consequently, lower its market price and make it available to a much wider consumer audience. In fact, the time to assemble a car passed from about 12h and 30m, prior to the introduction of the moving assembly line, to just 1h and 33m afterwards.

The impact of moving assembly lines was so great for the automobile industry as whole that Ford Motor Company was producing 50% of all cars sold in the US and about 40% of the cars sold in the British Commonwealth by 1920.

What is pipelining? - 3

non-pipelined version

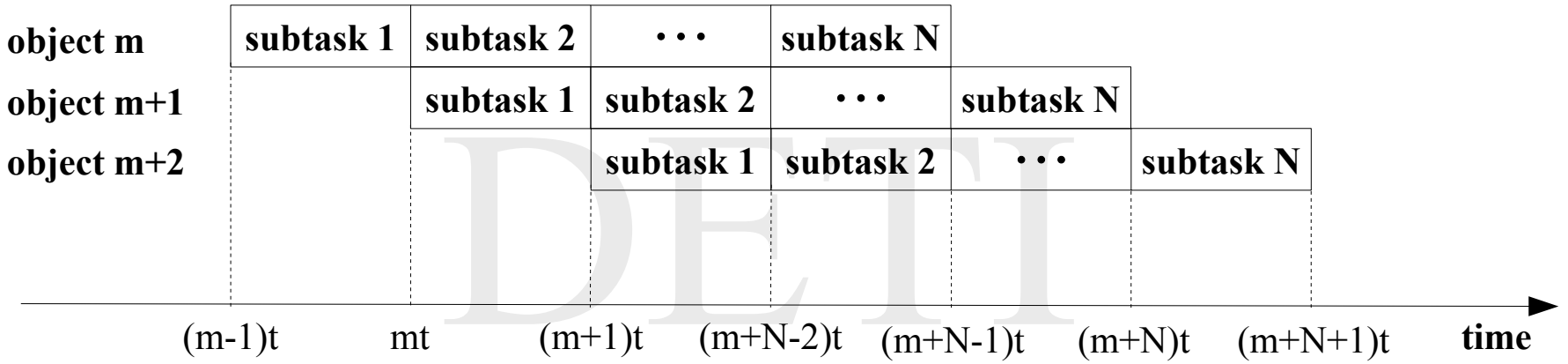


$$\text{throughput} = \frac{1}{T}$$

$$\text{execution time for a M object stream} = M \cdot T$$

What is pipelining? - 4

N-stage pipelined version



t_n , with $n = 1,2, \cdots, N$ – execution time for subtask n

throughput $= \frac{1}{t}$, where $t = max(t_1, t_2, \cdots, t_N)$

execution time for a M object stream $= (N - 1 + M) \cdot t$

What is pipelining? - 5

The *speed up* obtained from the execution of a N-stage pipeline implementation over the non-pipelined execution of the same task is expressed by

$$\text{speed up}_{\text{N-stage pipeline}} = \frac{\text{execution time of the non-pipelined implementation}}{\text{execution time of the N-stage pipelined implementation}} =$$

$$= \frac{M \cdot T}{(N - 1 + M) \cdot t} = \frac{M \cdot N \cdot t^*}{(N - 1 + M) \cdot t} \approx N \cdot \frac{t^*}{t} \quad ,$$

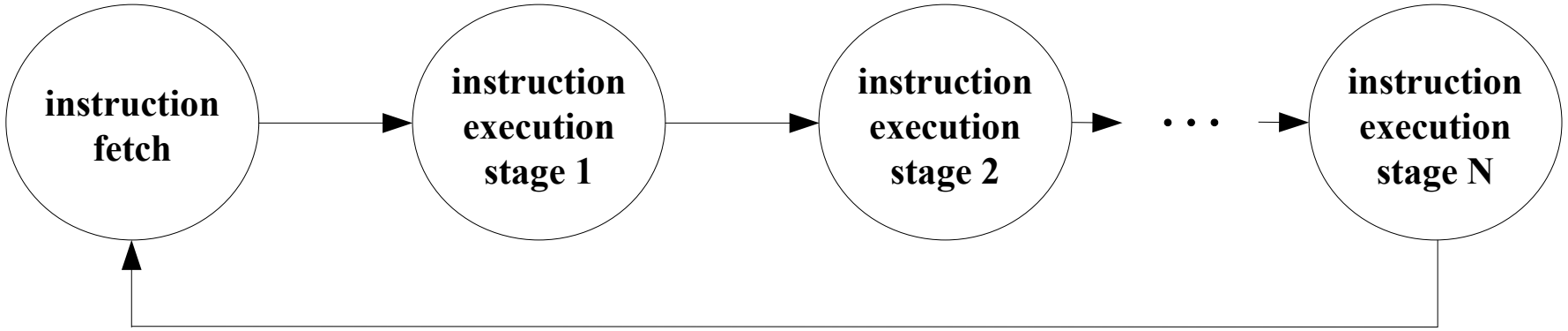
where $T = M \cdot t^*$ and $N << M$.

Ideally, if the pipe stages are almost perfectly balanced and the overhead involved in pipelining is negligible, then the ratio t^*/t approaches unity and the speed up is approximately equal to the number stages used on the partition of the original task into subtasks.

Instruction-level parallelism

All processors since 1985 have adopted pipelining as a means for overlapping the execution of instructions and for improving performance. This potential overlap of instructions during their own execution is called *instruction-level parallelism* (ILP) because the instructions are in some sense processed in parallel through the activity decomposition that takes place.

A common approach for doing this is by decomposing the *instruction execution* phase of the processor cycle into several stages.



Simplified RISC instruction set - 1

Part of the core architecture of a typical RISC (Reduced Instruction Set Computer), MIPS64, is going to be used to illustrate the basic concepts of pipelining.

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

- the only operations that affect memory are the *load* and *store* instructions that move data from memory to a register of the register bank, or from a register of the register bank to memory; instructions that transfer less than the contents of a full register are often also available
- the only operations on data are the *arithmetic* and *logic* instructions that apply to data on registers of the register bank; they change in principle the whole register
- the instruction formats are few in number, having typically the same size.

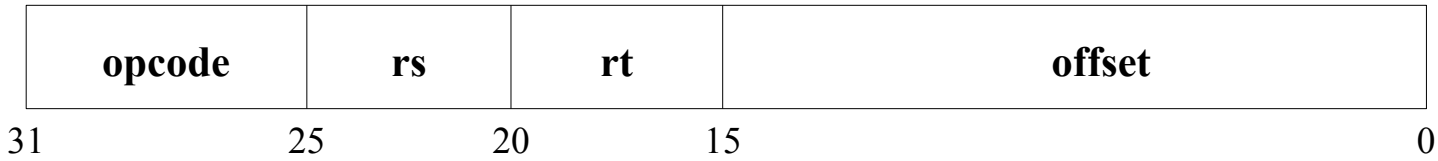
These properties also lead to dramatic simplifications in the implementation of pipelining, which is one of the reasons why the instruction sets are so designed.

Simplified RISC instruction set - 2

A brief description of the 64-bit version of MIPS instruction set follows. All instructions are 32 bits in length. The data types consist of 8-bit *bytes*, 16-bit *half words*, 32-bit *words* and 64-bit *double words*. The extended 64-bit instructions are generally denoted by having a `D` at the start, or at the end, of the mnemonic. The general-purpose register bank contains 32 64-bit registers, of which *register* 0 is read-only and has value zero.

The core instruction set has three classes of instructions

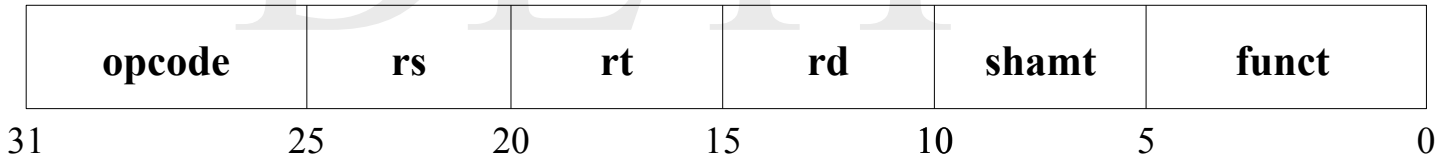
- *load and store instructions* – these instructions take as operands a register source, the *base register*, and an immediate 16-bit field, the *offset*; the sum of the contents of the base register and the sign-extended offset is used as a memory address, the *effective address*; the instructions LD and SD load or store the entire 64-bit register contents



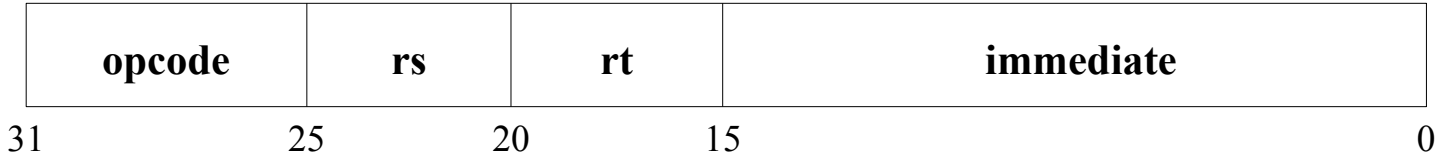
I-type instruction

Simplified RISC instruction set - 3

• *arithmetic / logic instructions* – these instructions take either two registers or a register and a sign-extended immediate value, operate on them and store the result in another register; typical *arithmetic* instructions include add (`DADD`) and subtract (`DSUB`) for the 64-bit extensions; *logical* instructions, however, do not differentiate between 32-bit and 64-bit extensions; *immediate* versions use the same mnemonics with suffix `I`; there are signed and unsigned forms of the *arithmetic* instructions: the unsigned forms do not generate overflow exceptions and have a suffix `U` at the end (`DADDU`, `DSUBU`, `DADDIU`)



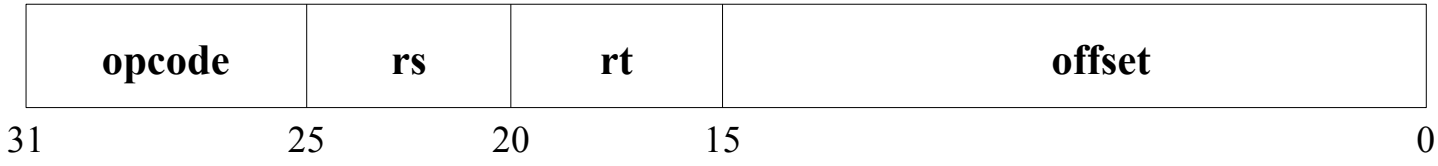
R-type instruction



I-type instruction

Simplified RISC instruction set - 4

- *branch instructions* – these instructions are conditional modifications of the strict sequentiality of instruction execution; the branch *condition* is typically specified either by a set of *condition bits*, also called *condition code*, a limited set of comparisons between a pair of registers (*greater*, *greater or equal*, *less*, *less or equal*, *equal* and *non-equal*) or between a register and zero (*equal* and *non-equal*); the branch *destination* is obtained by adding a sign-extended offset, shifted left by two bits to make it a word offset, to the current contents of the *program counter* (PC).



I-type instruction

Non-pipelined implementation of a RISC instruction set - 1

A simple non-pipelined implementation of part of the MIPS64 core instruction set is presented and will be used as a framework to fully understand the inner workings of pipelining. It is not the most economical or the highest performant implementation, but it is designed to lead naturally to pipelining.

Implementing the instruction set requires the introduction of several temporary registers that are not part of the architecture; they are introduced just to simplify pipelining. This implementation will focus only on the integer subset of MIPS64 core instruction set that consists of the load / store, integer ALU and branch operations. They will be implemented in at most 5 clock cycles.

The operation at each clock cycle is as follows

1. *Instruction fetch cycle* (IF)

   The contents of the program counter (PC) is used as an address to fetch the next instruction from memory, its value is stored in the instruction register (IR). The PC is next updated to point the next instruction by adding 4 to the current contents.

Non-pipelined implementation of a RISC instruction set - 2

2. *Instruction decode / register fetch cycle* (ID)

The instruction is decoded leading to the operations

- the contents of the registers corresponding to the register source specifiers are read from the register bank
- the equality test is carried out on the contents of the registers as they are read, to be later used in a possible branch
- the offset field is sign-extended in case it will be needed
- the possible branch target address is computed by adding the incremented PC contents to the 2 bit left- shifted sign-extended offset field.

The *branch* instruction is completed at the end of this stage by storing the branch target address into the PC if the test condition yields true, or by leaving it unchanged if the test condition yields false.

Non-pipelined implementation of a RISC instruction set - 3

3. *Execution* / *effective address cycle* (EX)

   The ALU takes the operands computed in the previous cycle and, depending on the instruction type, performs one of the actions bellow

   - *memory reference* – the base register and the offset are added together to form the effective address
   - *register-to-register* ALU *instruction* – the operation specified by the opcode is carried out on the values read from the register bank
   - *register-immediate* ALU *instruction* – the operation specified by the opcode is carried out on the first value read from the register bank and the sign-extended immediate value.

Non-pipelined implementation of a RISC instruction set - 4

4. *Memory access* (MEM)

   When the instruction is a *load*, the effective address computed in the previous cycle is the address of the memory location whose data is to be read; when the instruction is a *store*, the second value read from the register bank is to be written into the memory location whose address is the effective address.
   The *store* instruction is completed at the end of this stage.

5. *Write-back cycle* (WB)

   The result from an instruction of one of the types *register-to-register*, *register-immediate*, or *load*, is written into a register of the register bank.

Non-pipelined implementation of a RISC instruction set - 5

In this implementation, *branch* instructions are executed in 2 clock cycles, *store* instructions in 4 clock cycles and all other instructions in 5 clock cycles. Assuming that a given benchmark has a branch frequency of 12% and a store frequency of 10%, what is the average CPI for running this application on the non-pipelined implementation just described?

$$\text{overall CPI} = \sum_i \frac{\text{instruction count}_i}{\text{instruction count}} \cdot \text{CPI}_i =$$

$$= 0.12 \cdot 2 + 0.10 \cdot 4 + 0.78 \cdot 5 = 4.54 \quad .$$

Classical 5-stage pipeline for a RISC processor - 1

The non-pipelined implementation can be converted into a pipelined implementation with almost no structural changes by fetching a new instruction every clock cycle.

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

Although each instruction takes 5 clock cycles to complete, during each cycle the hardware will be processing some part of five consecutive instructions.

Classical 5-stage pipeline for a RISC processor - 2

However, for this transformation to work, one has to determine what happens at every processor clock cycle and make sure that no two operations are using the same data path resource at the same time, that is, one needs to check carefully that the overlap of instructions in the pipeline is not causing such a conflict.

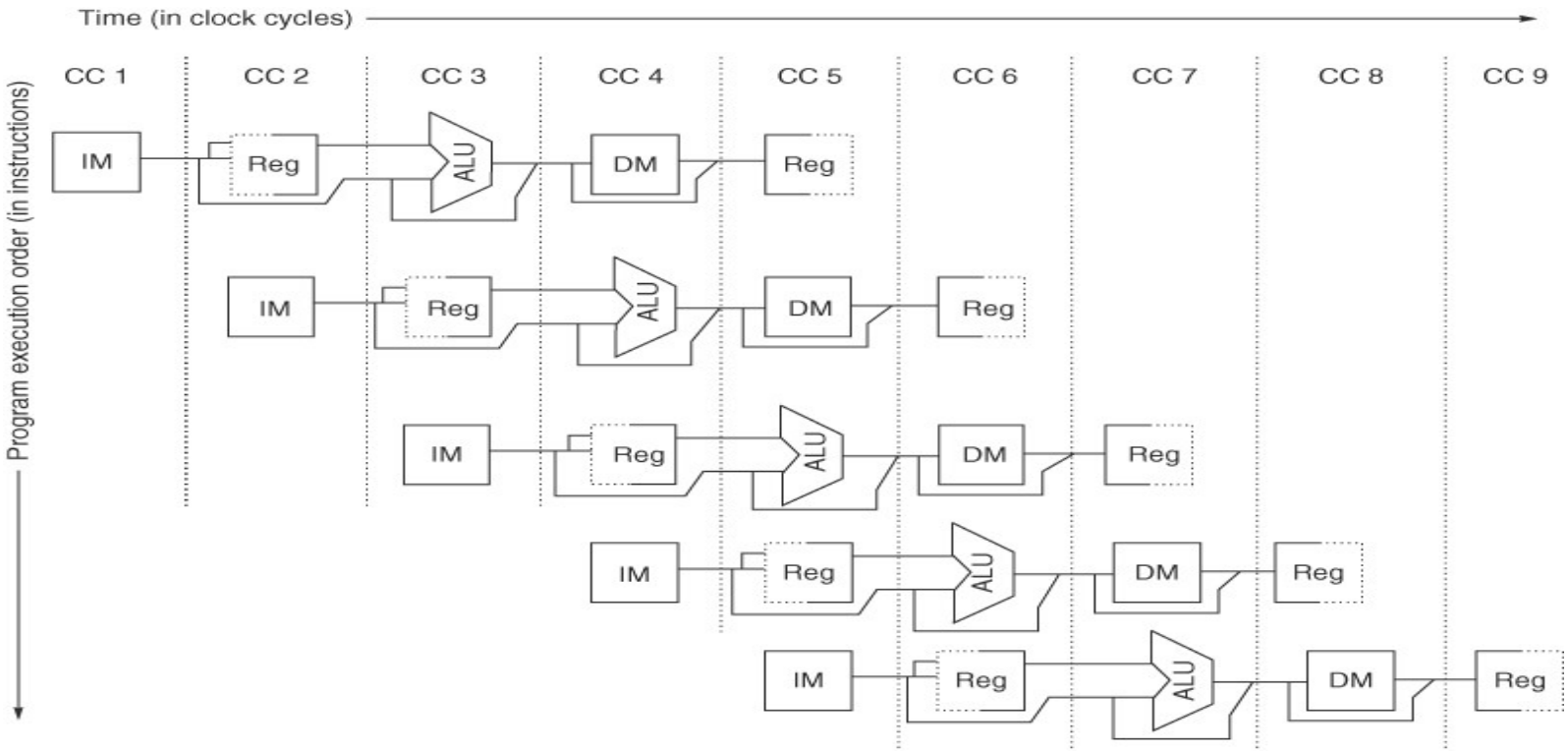
The number of details to consider are directly proportional to the number of pipeline stages and the number of instructions that are in overlapping execution. So it is important to devise a graphical representation of the execution that makes apparent what is happening.

A popular way of doing this is by describing the flow of operations as a series of the data path resources in use at the different stages of the pipeline.

Classical 5-stage pipeline for a RISC processor - 3

Pipeline seen as a series of data path resources shifted in time

Source: Computer Architecture: A Quantitative Approach



CC – clock cycle
DM – data memory
ALU – arithmetic/logic unit
IM – instruction memory
Reg – register bank

Classical 5-stage pipeline for a RISC processor - 4

A few observations are in order

- there is a need for separate *instruction* and *data memories*, which means the implementation of separate instruction and data caches for eliminating the conflict of a single memory that would arise between the *instruction fetch* (IF) and *data memory access* (MEM) stages
- the *register bank* is accessed in two stages:for reading, in the *instruction decode* / *register fetch* stage (ID), and for writing, in the *write-back* stage (WB); one needs to perform two reads and one write every clock cycle – to handle reading and writing at the same register, register write is performed in the first half of the clock cycle and register read in the second half
- to fetch an instruction every clock cycle, the *program counter* (PC) must be updated every clock cycle in the *instruction fetch* (IF) stage in preparation for the next instruction; furthermore, the potential branch target address must be computed during the *instruction decode* / *register fetch* stage (ID); still, since the branch, when the test condition yields true, also changes the PC in the ID stage, there is a conflict that will be dealt with later on.

Classical 5-stage pipeline for a RISC processor - 5

One must also ensure that instructions in different stages of the pipeline are not interfering with one another. The solution is to add registers for temporary storage of relevant data between successive stages, so that at the beginning of a clock cycle all the results from a given stage are stored into the registers that are used as inputs for the next stage. The registers are named for the two stages that are separated by that register: for instance, the registers between the IF and the ID stages are called IF/ID.

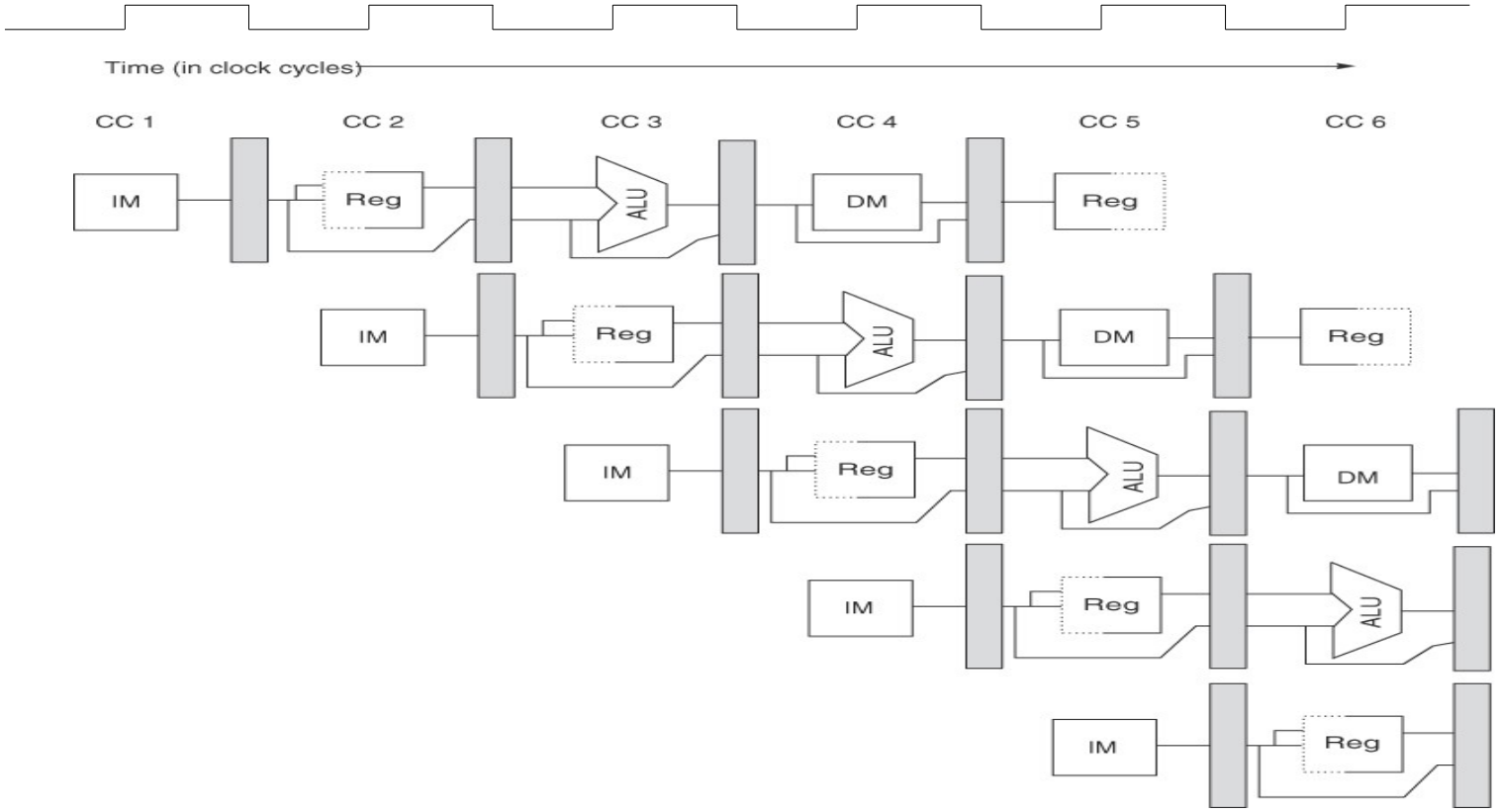
Notice that no registers are required at the end of the WB stage. All instructions must update in some particular manner the computer state (the register bank, the memory or the program counter) so a separate register at the end of the pipeline would be redundant to the state being updated.

The program counter itself may be thought of as a pipeline register: one feeding the IF stage of the pipeline. Unlike the other registers, however, the program counter is part of the visible architectural state; its contents must be saved when an exception is serviced, while the contents of the other pipeline registers is discarded.

Classical 5-stage pipeline for a RISC processor - 6

Pipeline with registers between successive stages for temporary storage

Source: Adapted from Computer Architecture: A Quantitative Approach



Classical 5-stage pipeline for a RISC processor - 7

Pipelining increases the instruction throughput, but does not reduce the execution time of an individual instruction. In fact, it slightly increases it due to the required overhead to control the pipeline. Notice that the increase in instruction throughput means that a program runs faster and has a lower execution time, even though no single instruction runs faster.

Consider the non-pipelined processor previously described. Assume it has a 1ns clock cycle and that it runs a benchmark with a branch frequency of 20% and a store frequency of 10%. Suppose that due to the overhead to control the pipeline, the clock cycle of the pipelined version of the processor is 1.2ns. What is the ideal speed up in the instruction execution rate gained from pipelining?

$$\text{average inst exec time}_{\text{nonpipelined}} = \text{overall CPI}_{\text{nonpipelined}} \cdot \text{clock cycle time} =$$

$$= (0.2 \cdot 2 + 0.1 \cdot 4 + 0.7 \cdot 5) \cdot 1.0 = 4.3\,\text{ns}$$

$$\text{speed up} = \frac{\text{average inst exec time}_{\text{nonpipelined}}}{\text{average inst exec time}_{\text{pipelined}}} = \frac{4.3}{1.2} = 3.6 \ .$$

Major hurdles of pipelining - 1

Pipelining would work as described if the instructions were independent from one another. In reality, this is not so. There are situations, called *hazards*, which prevent the next instruction in the stream to be executed during its designated clock cycle. Thus, reducing the performance from the ideal speed up gained by pipelining.

There are three classes of hazards

- *structural hazards* – they arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution
- *data hazards* – they arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of the instructions in the pipeline
- *control hazards* – they arise from the pipelining of branch instructions and other instructions that affect the PC.

Major hurdles of pipelining - 2

Hazards in pipelines may require to *stall* the pipeline, that is, in order to avoid the hazard it is often needed that some instructions in the pipeline are forced to remain in the same stage, while others are allowed to proceed to the next stage. When an instruction is stalled, all the instructions that were fetched later than the stalled instruction, are also stalled, and all the instructions fetched earlier must continue, since otherwise the hazard would never disappear. As a result, no new instructions are fetched during a stall and an execution delay takes place.

Performance of pipelines with stalls

$$\text{speed up} = \frac{\text{average inst exec time}_{\text{nonpipelined}}}{\text{average inst exec time}_{\text{pipelined}}} =$$

$$= \frac{\text{overall CPI}_{\text{nonpipelined}}}{\text{overall CPI}_{\text{pipelined}}} \cdot \frac{\text{clock cycle time}_{\text{nonpipelined}}}{\text{clock cycle time}_{\text{pipelined}}} =$$

$$= \frac{\text{overall CPI}_{\text{nonpipelined}}}{1 + \text{pipeline stall cycles per instruction}_{\text{pipelined}}} \cdot \frac{\text{clock cycle time}_{\text{nonpipelined}}}{\text{clock cycle time}_{\text{pipelined}}} .$$

Structural hazards - 1

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

The most common instance of structural hazards arises when some functional unit is not fully pipelined. Then, a sequence of instructions where some of them use that unit, cannot proceed through the pipe stages at the rate of one instruction per clock cycle. Another common cause is when some resource is not replicated enough to allow all combinations of instructions to execute at the nominal rate.

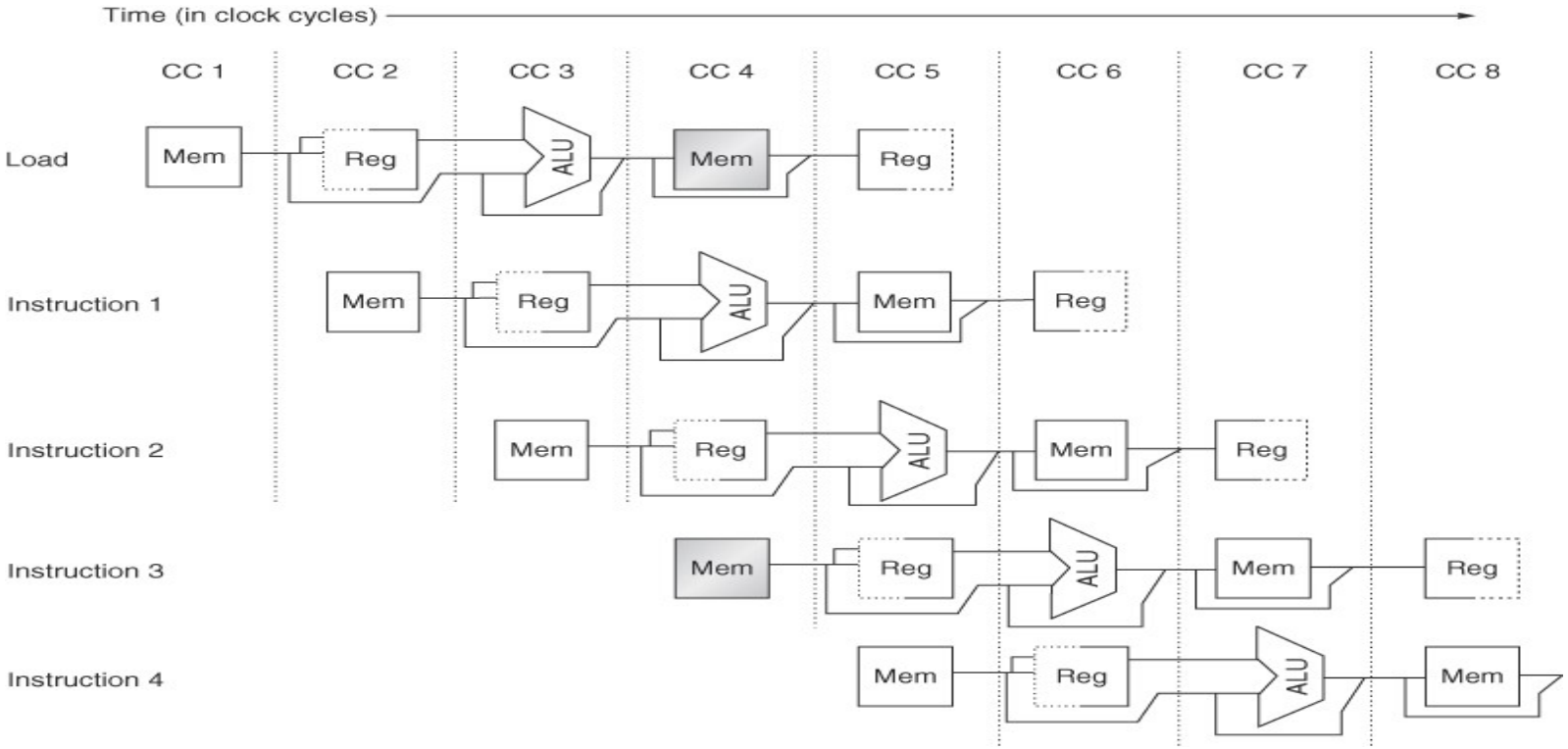
When this type of hazard is encountered, the pipeline will *stall* the latest coming instruction until the required unit is available. The stall will generate what is usually called a *bubble*, since this condition may be thought of flowing through the pipe stages taking space, but not producing any useful work.

Structural hazards - 2

Some processors have a single memory to access both instructions and data. This will generate a potential conflict between the instruction fetch (IF) and the data memory access (MEM) stages.

Processor with only one memory port

Source: Computer Architecture: A Quantitative Approach



Structural hazards - 3

Processor with only one memory port: the effect of a *load* instruction

Source: Adapted from Computer Architecture: A Quantitative Approach

	Clock number									
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
load	IF	ID	EX	MEM	WB					
m+1		IF	ID	EX	MEM	WB				
m+2			IF	ID	EX	MEM	WB			
m+3				***stall***	IF	ID	EX	MEM	WB	
m+4						IF	ID	EX	MEM	WB
m+5							IF	ID	EX	MEM
m+6								IF	ID	EX

	Clock number									
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
load	IF	ID	EX	MEM	WB					
m+1		IF	ID	EX	MEM	WB				
m+2			IF	ID	EX	MEM	WB			
bubble				*IF*	*ID*	*EX*	*MEM*	*WB*		
m+3					IF	ID	EX	MEM	WB	
m+4						IF	ID	EX	MEM	WB
m+5							IF	ID	EX	MEM
m+6								IF	ID	EX

Structural hazards - 4

Suppose that data reference instructions (*load* and *store*) constitute 40% of the instruction mix for a given benchmark and that the ideal CPI for the pipelined processor, ignoring the structural hazard, is 1. Assume that the processor with the structural hazard has a clock rate that is 1.05 times higher than the processor without the hazard. Disregarding any other performance losses, which processor runs the benchmark faster?

$$\text{average inst exec time}_{\text{without haz}} = \text{overall CPI}_{\text{without haz}} \cdot \text{clock cycle time}_{\text{ideal}} =$$

$$= 1 \cdot \text{clock cycle time}_{\text{ideal}}$$

$$\text{average inst exec time}_{\text{with haz}} = \text{overall CPI}_{\text{with haz}} \cdot \text{clock cycle time}_{\text{ideal}} =$$

$$= (1 + 0.4 \cdot 1) \cdot \frac{\text{clock cycle time}_{\text{ideal}}}{1.05} \approx$$

$$\approx 1.3 \cdot \text{clock cycle time}_{\text{ideal}} \ .$$

Structural hazards - 5

If all factors are equal, a processor without structural hazards will always have a lower CPI. Why, then, would a designer allow structural hazards?

The primary reason is to reduce the processor cost, since pipelining all functional units, and / or replicating them, may prove to be too costly. For example, processors that require an instruction and a data cache access every clock cycle need twice as much total memory bandwidth. Likewise, designing a fully-pipelined floating point multiplier consumes lots of gates and space may not be available.

If the structural hazard is rare, it may not be worth the cost to avoid it.

Data hazards - 1

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. Due to this, data hazards may arise. *Data hazards*, in fact, occur when the pipeline forces an actual order for read / write operations to operands different from the one perceived by sequentially executing the instructions on a non-pipelined processor.

Consider the pipelined execution of the code

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

Data hazards - 2

All the instructions after DADD use the result of the DADD instruction.

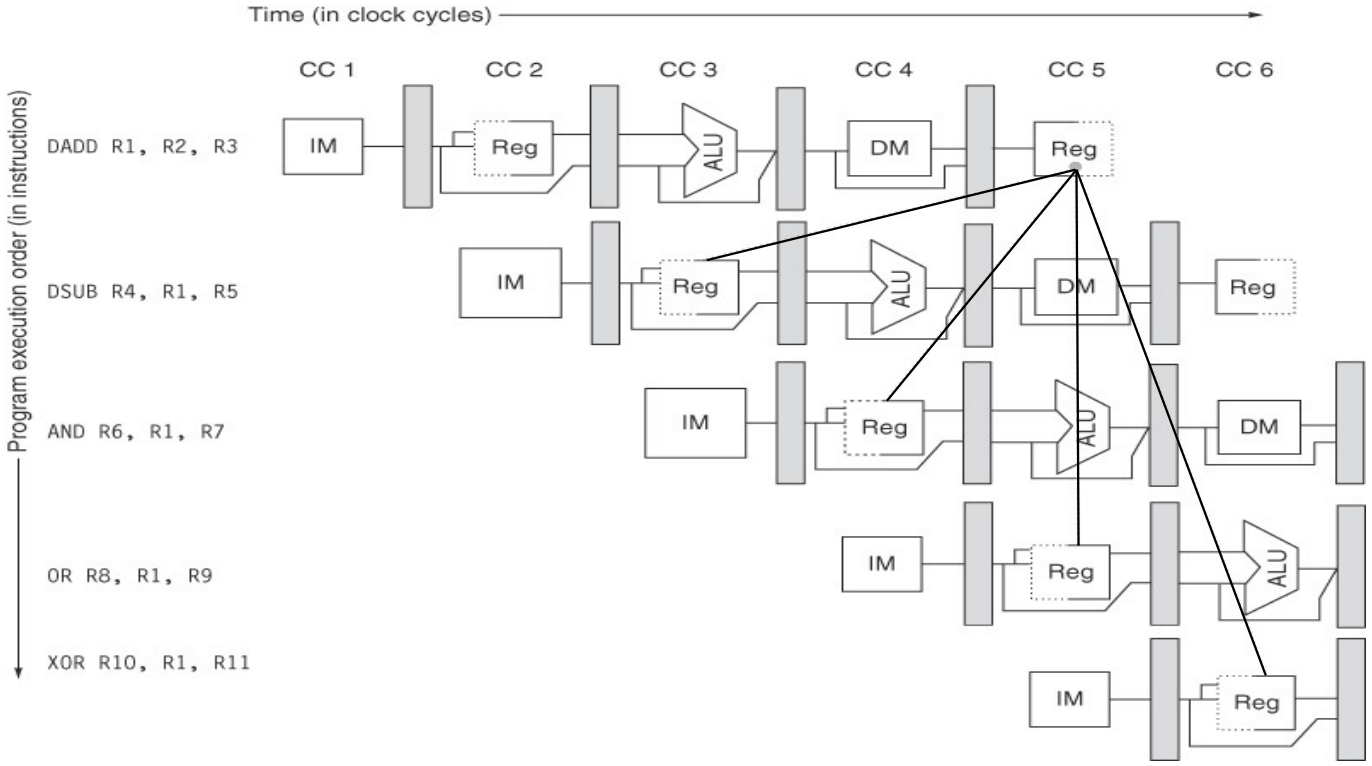
The DADD updates the value of R1 in the WB stage, but DSUB reads R1 prior to this in its ID stage. Unless precautions are taken to prevent it, DSUB will read a wrong value. The value read is not even deterministic. It seems logic to assume that DSUB will always get the value assigned to R1 by some instruction prior to DADD, but this may not be the case. If an interrupt is serviced between the DADD and the DSUB instructions, the WB stage of DADD will complete and the value of R1 at that point will be the result of DADD.

The AND instruction is also affected by this hazard. The writing of R1 does not complete until the first half of clock cycle 5. Thus, AND that reads the register at the second half of clock cycle 4, will receive a wrong result. The OR and XOR instructions, however, execute properly: the former will read R1 at the second half of clock cycle 5, after the writing has taken place; the latter reads R1 only at clock cycle 6.

Data hazards - 3

The effect of the DADD instruction on the instructions that follow

Source: Computer Architecture: A Quantitative Approach



Data hazards - 4

The problem can be solved through the use of a simple hardware technique called *forwarding*, *bypassing*, or *short-circuiting*. The key insight in *forwarding* is that the result is not really needed by `DSUB`, or by `AND`, until after `DADD` actually produces it. If the result can be moved from the register where `DADD` stores it to where `DSUB` and `AND` need it, then the introduction of stalls may be avoided.

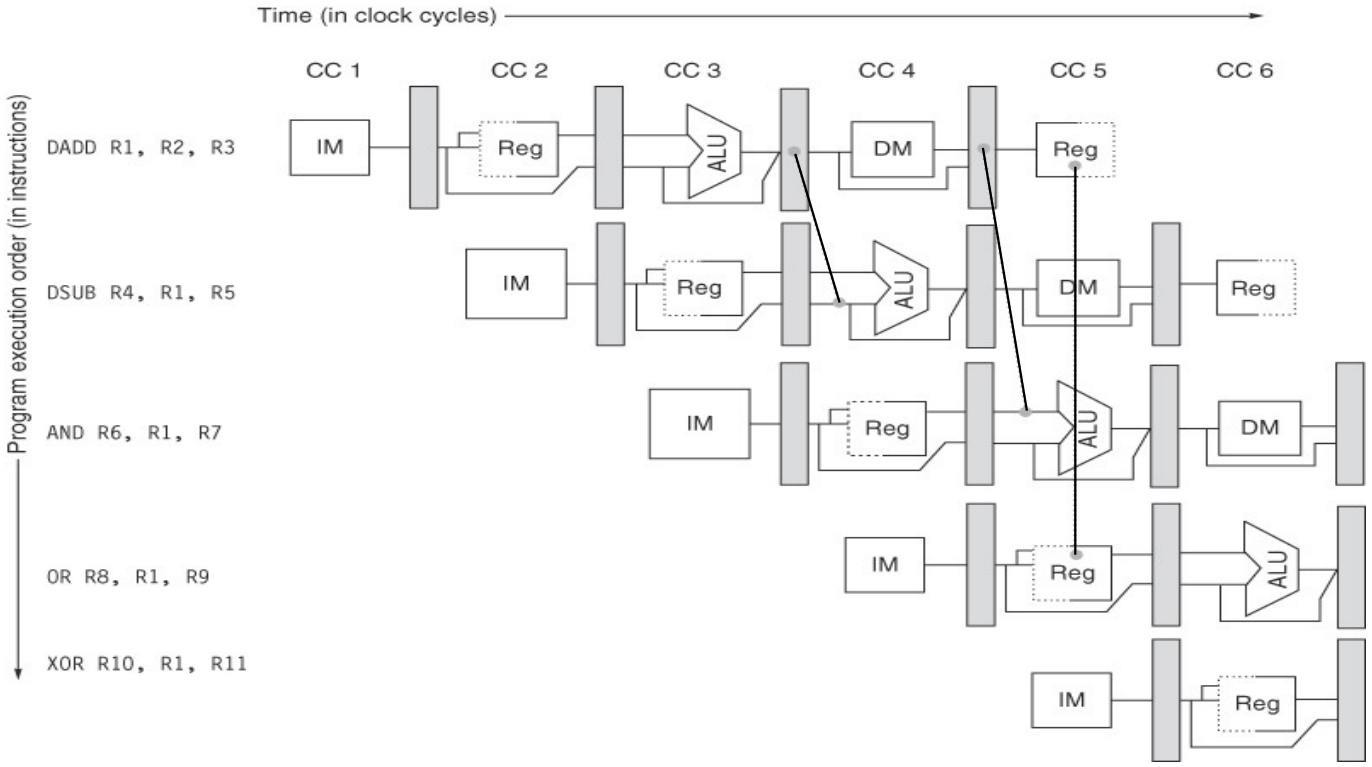
Using this observation, *forwarding* works as follows

- the ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs
- if the forwarding hardware detects that a previous ALU operation has modified the register corresponding to a source for the current ALU operation, control logic selects the forward result as the ALU input rather than the value read from the register bank.

Data hazards - 5

The effect of the DADD instruction on the instructions that follow solved by forwarding to avoid a data hazard

Source: Computer Architecture: A Quantitative Approach



Data hazards - 6

Forwarding can be generalized to include passing a result to the functional unit that requires it. A result is forwarded from the pipeline register corresponding to the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit.

Consider the pipelined execution of the code

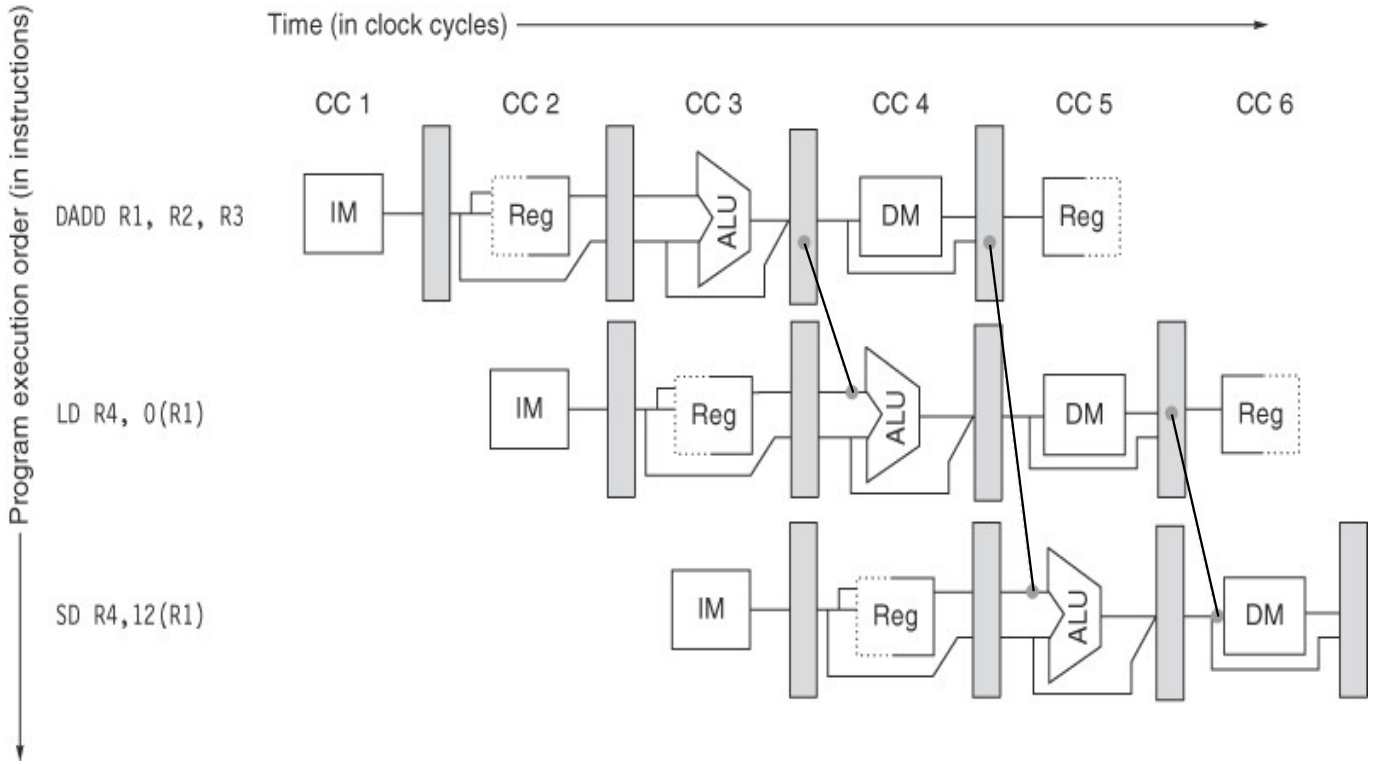
```
DADD    R1,R2,R3
LD      R4,0(R1)
SD      R4,12(R1)
```

To prevent a stall in this sequence, one needs to forward the values of the ALU output and the data memory output from the pipeline registers to the ALU and the data memory inputs.

Data hazards - 7

Forwarding of operand required by *store* during MEM

Source: Computer Architecture: A Quantitative Approach



Data hazards - 8

Unfortunately, not all potential data hazards can be handled by forwarding.

Consider the pipelined execution of the code

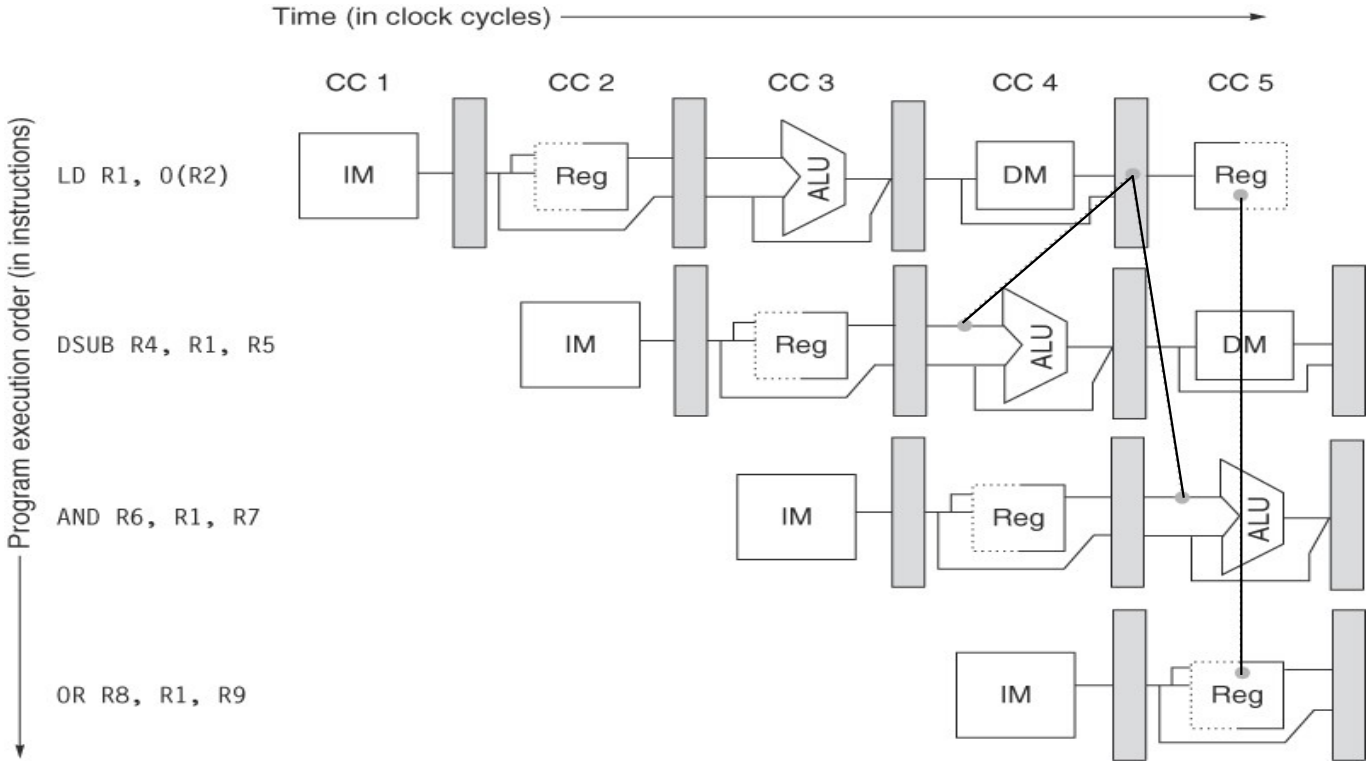
```
LD      R1,0(R2)
DSUB    R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

The `LD` instruction does not have the data until the end of clock cycle 4, while the `DSUB` instruction requires it at the beginning of this cycle. Thus, the hazard from using the result of a load instruction cannot be completely eliminated. The result can be forwarded immediately to the ALU for use by the `AND` instruction, which begins 2 clock cycles after `LD`. Likewise, the `OR` instruction has no problem since it gets the value through the register file.

Data hazards - 9

Data hazard that cannot be solved by forwarding

Source: Computer Architecture: A Quantitative Approach



Data hazards - 10

To solve the problem, one needs to add special hardware, called *pipeline interlock*, to preserve the correct execution pattern. In general, the *pipeline interlock* detects a hazard and stalls the pipeline until the hazard disappears. In this case, the *pipeline interlock* stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it and makes it available.

	Clock number								
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
`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

Control hazards - 1

When a branch is executed, it may or may not change the PC. It is said that if a branch instruction changes the PC to its target address, it is a *taken* branch; if it falls through, it is an *untaken* branch. When an instruction is a taken branch, the PC is not changed until the end of the ID stage.

The simplest method to deal with branches is to redo the fetch of the instruction following the branch, once the branch instruction is detected (during the ID stage). The first IF cycle is essentially a stall, because it never performs useful work. One should notice that if a branch is untaken, there is a penalty, the repetition of the IF stage is unnecessary since the correct instruction was indeed fetched.

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

Control hazards - 2

A higher-performance alternative, called *predicted-untaken* scheme, is to treat every branch as *untaken*, simply allowing the hardware to continue as if the branch instruction were not executed. The complexity of this scheme arises from the need to ascertain when the state have been changed by an instruction and to revert such a change. The pipeline looks as if nothing out of the ordinary is happening in this case. If the branch is *taken*, however, the fetched instruction is turned into a *no-op* instruction and the fetch stage is restarted at the target address.

	Clock number							
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
untaken branch	IF	ID	EX	MEM	WB			
successor		IF	ID	EX	MEM	WB		
successor+1			IF	ID	EX	MEM	WB	
successor+2				IF	ID	EX	MEM	WB

	Clock number							
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
taken branch	IF	ID	EX	MEM	WB			
successor		IF	***idle***	***idle***	***idle***	***idle***		
target			IF	ID	EX	MEM	WB	
target+1				IF	ID	EX	MEM	WB

Control hazards - 3

One may think the other way around and treat every branch as *taken*. This scheme is obviously called the *predicted-taken* scheme. It is only relevant for processors that use implicit set condition codes or more powerful, and hence slower, branch conditions – the branch target address is known before the branch outcome and, because of that, the *predicted-taken* scheme might make sense.

In either of the predicted schemes, the compiler plays an important role since it can improve performance by organizing the code so that the most frequent path matches the hardware choice.

Control hazards - 4

A last scheme, called *delayed branch* scheme, was heavily used in early RISC processors. The execution cycle with a branch delay of one is defined as

```
branch instruction
sequential successor
branch target if taken
```

The sequential successor is in the *branch delay slot*. This instruction is executed whether or not the branch is taken and can not be itself a branch instruction.

	Clock number							
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
untaken branch	IF	ID	EX	MEM	WB			
branch delay		IF	ID	EX	MEM	WB		
successor			IF	ID	EX	MEM	WB	
successor+1				IF	ID	EX	MEM	WB

	Clock number							
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
taken branch	IF	ID	EX	MEM	WB			
branch delay		IF	ID	EX	MEM	WB		
target			IF	ID	EX	MEM	WB	
target+1				IF	ID	EX	MEM	WB

Control hazards - 5

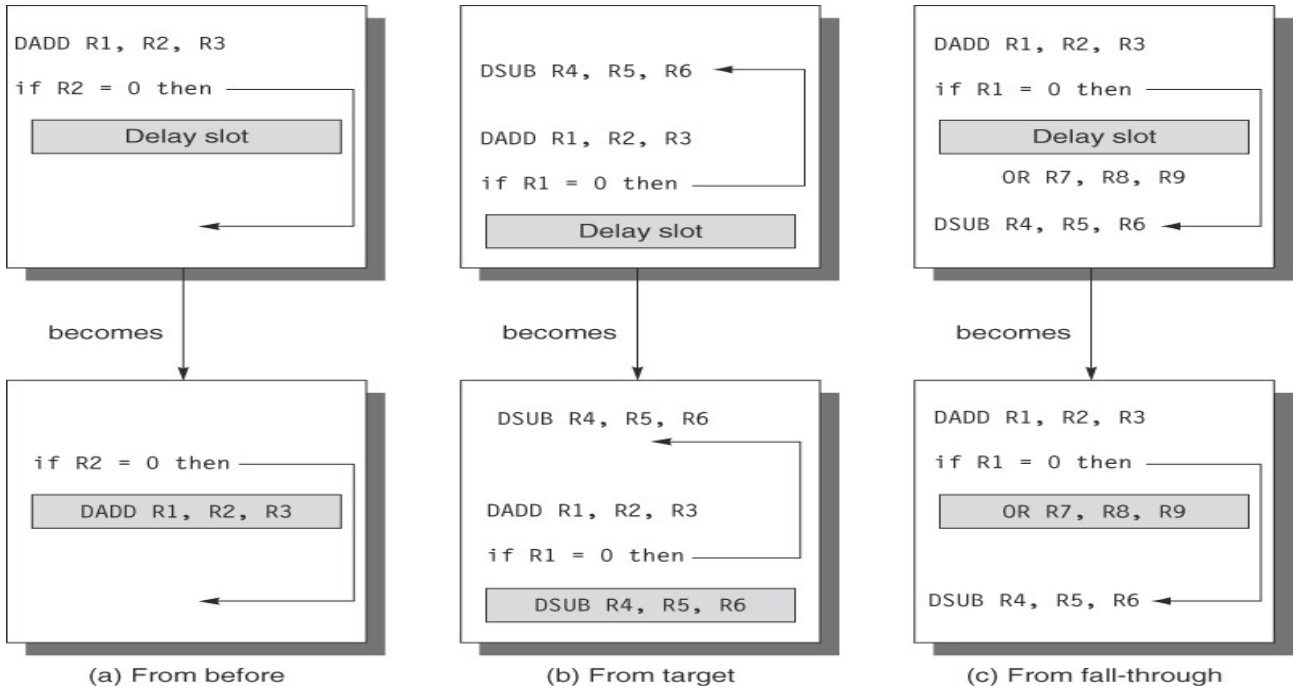
The job of the compiler is to make the sequential successor instruction valid and useful. Several alternatives are possible

- *from before* – insert in the *branch delay slot* an instruction supposed to be executed before the branch instruction and that is independent from it (the *ideal* situation since no side effects result from it application)
- *from target* – insert in the *branch delay slot* the instruction that the branch, if *taken*, points to and make the branch point to its successor (notice that when the branch is *not taken*, an extra instruction, not originally intended, is executed)
- *from fall-through* – insert in the *branch delay slot* the instruction that will be executed next if the branch is *not taken* (then this instruction is still being executed when the branch is *taken*).

Control hazards - 6

Branch delay slot compiler optimizations

Source: Computer Architecture: A Quantitative Approach



One has to bear in mind that the last two optimizations are only possible if they do not imply as side effect something which makes the program run incorrectly when the branch takes the unexpected direction!

Control hazards - 7

To improve the ability of the compiler to fill the branch delay slot, most processors with conditional branches have introduced a *cancelling* or *nullifying branch* instruction. It looks like the usual *delayed branch* instruction. When the branch is *taken*, the instruction in the *branch delay slot* is executed; however, when the branch is *not taken*, the instruction in the *branch delay slot* is turned into a *no-op* instruction and nothing happens.

The effective speed up of these schemes to deal with control hazards when the code is executed, can be expressed by

$$\text{speed up} = \frac{\text{overall CPI}_{\text{nonpipelined}}}{1 + \text{pipeline stall cycles per instruction}_{\text{pipelined}}} \cdot \frac{\text{clock cycle time}_{\text{nonpipelined}}}{\text{clock cycle time}_{\text{pipelined}}}$$

$$= \frac{\text{overall CPI}_{\text{nonpipelined}}}{1 + \left(\text{branch frequency} \cdot \text{branch penalty}\right)_{\text{pipelined}}} \cdot \frac{\text{clock cycle time}_{\text{nonpipelined}}}{\text{clock cycle time}_{\text{pipelined}}} .$$

Control hazards - 8

As pipelines go deeper and the potential penalty of branches increases, delayed branches and similar schemes are deemed to be insufficient. It is necessary to be more aggressive in predicting branches.

The problem is approached in two different ways

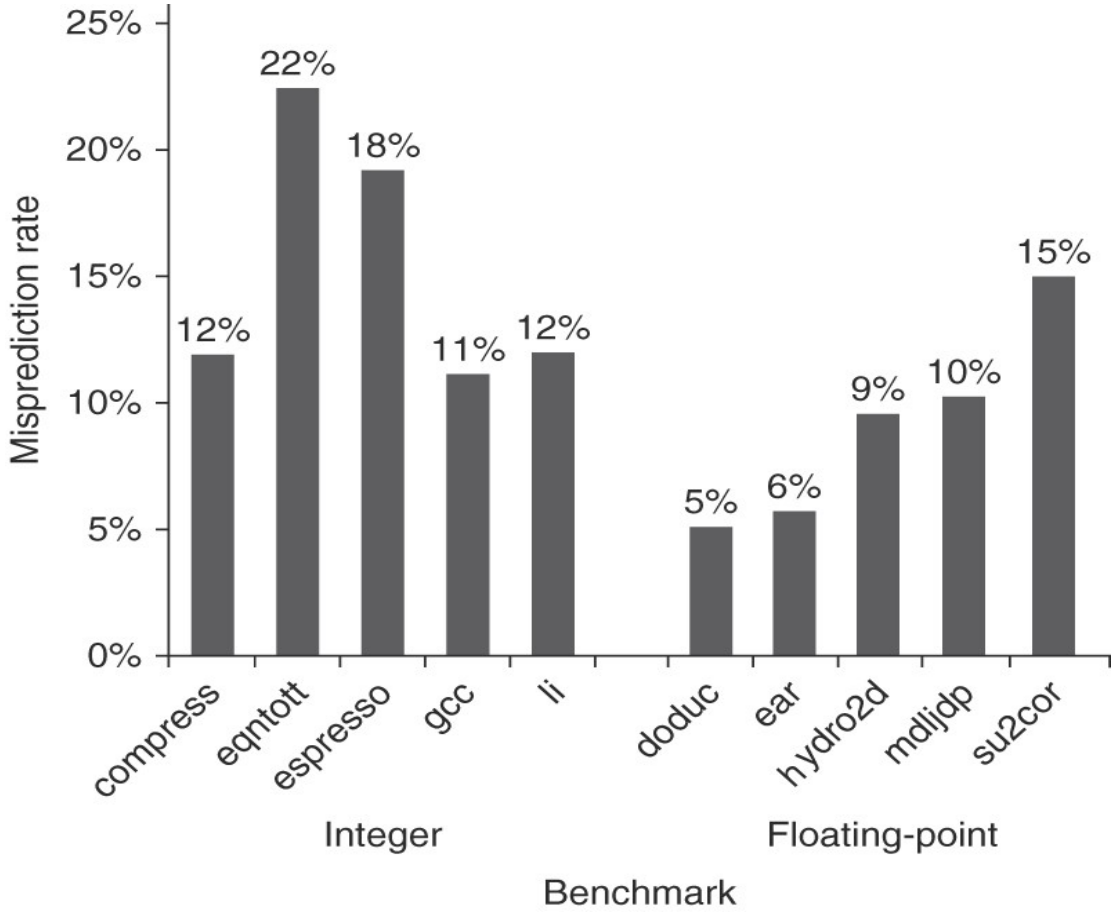
- *low-cost static schemes* – they rely on information available at compiler time and on profiling earlier runs of the same code; the key observation that makes this effort worthwhile is that branch behavior is often bimodally distributed, that is, each particular branch tends to be often highly biased towards *taken* or *not taken*
- *dynamic schemes* – they consist of methods to predict the branch direction during runtime.

In general, the misprediction rate for integer programs is higher than for floating-point programs, not only their branch frequency is higher, but also the branching direction is more random.

Control hazards - 9

Misprediction rate for SPEC92 using a profile-based predictor

Source: Computer Architecture: A Quantitative Approach



Control hazards - 10

The simplest dynamic branch-prediction scheme uses a *branch-prediction buffer* (BPB) or a *branch-prediction table* (BPT). A *branch-prediction buffer* is a small memory indexed by the lower part of the address of the branch instruction. The memory has a bit per position that states whether the branch was recently *taken* or *not taken*.

With such a device, one does not really know if the prediction is correct – it may have been set or reset by another branch instruction that shares the same low-order address bits, or now the branch behavior may be opposite to what was before. If one thinks a little about this, it is not determinant. A prediction is just a hint about the future one assumes to be true and, therefore, fetching proceeds in the predicted direction. If it is proved to be false later on, the prediction bit is complemented and fetching is restarted.

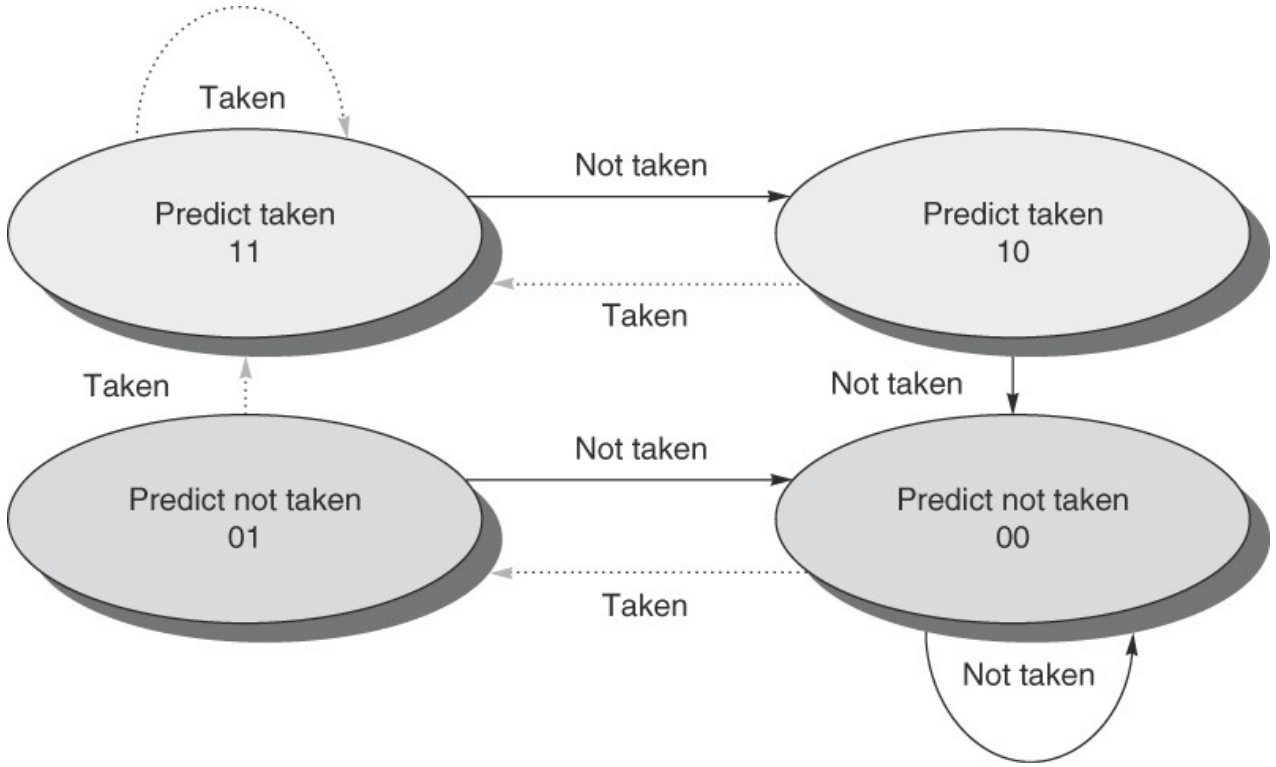
This simple 1-bit prediction scheme has a performance shortcoming: even when the branch is almost always taken, it will be likely to predict incorrectly twice, rather than once, when it was not taken. This can be remedied by using a 2-bit prediction scheme instead.

Control hazards - 11

The 2-bit prediction scheme introduces hysteresis in the prediction process.

2-bit prediction scheme state diagram

Source: Computer Architecture: A Quantitative Approach

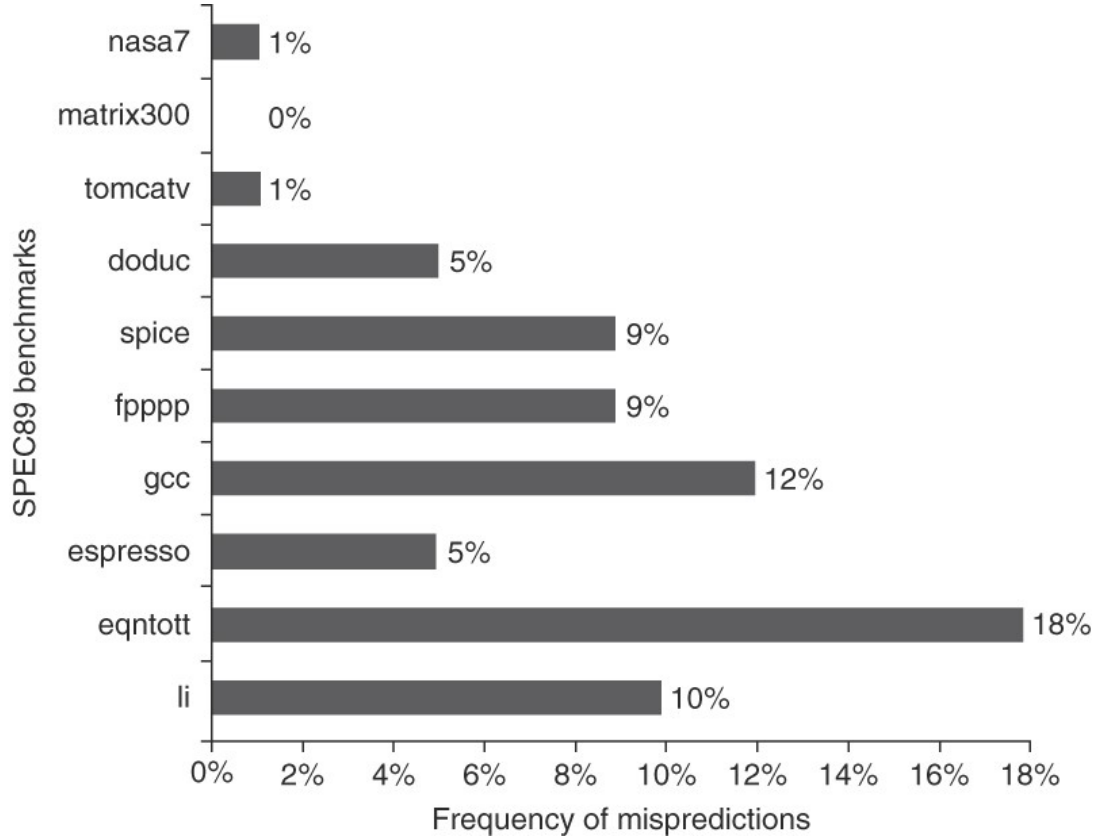


Control hazards - 12

The improvement over a profile-based predictor is real, but not spectacular.

Misprediction rate for SPEC89 using a 4096-entry 2-bit prediction buffer

Source: Computer Architecture: A Quantitative Approach



Control hazards - 13

Studies show that the hit rate of the buffer is not the major limiting factor. In fact, the behavior of a 4096-entry buffer is very similar to the behavior of a buffer with unlimited number of entries. Increasing the number of bits of the predictor without changing the predictor structure also has little impact.

Current approaches favor the use of both local and global information to improve prediction accuracy.

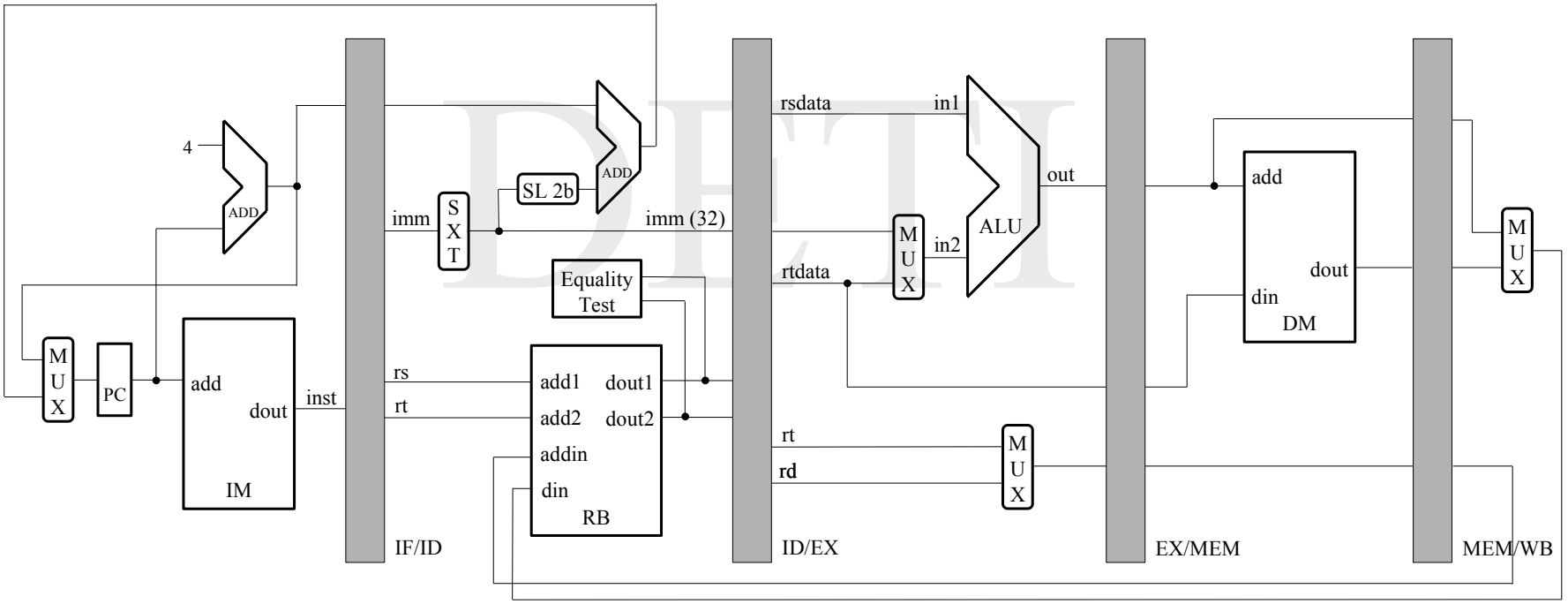
In the simplest case, a *correlating predictor* may consist of a 2-bit predictor for each branch, reporting its past history in different global situations.

More recently, *tournament predictors* have been used. A *tournament predictor* uses multiple predictors, tracking which yields the best result for each branch and taking the winner suggestion as the next prediction.

Implementation of classical 5-stage pipeline - 1

5-stage pipeline datapath

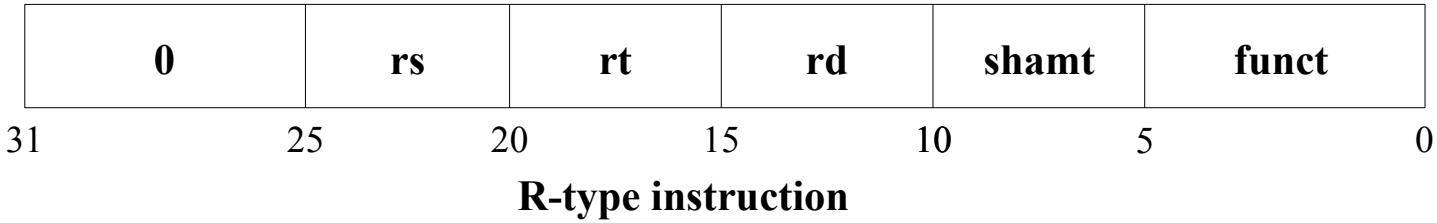
Source: Adapted from Computer Architecture: A Quantitative Approach



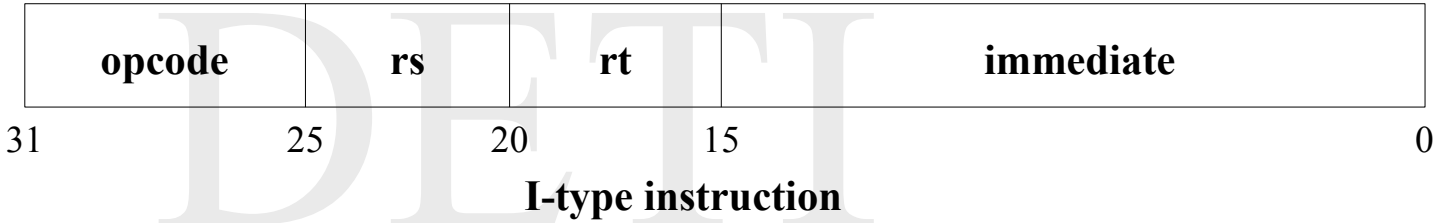
Implementation of classical 5-stage pipeline - 2

Classes of instructions to be considered

arithmetic / logic instructions

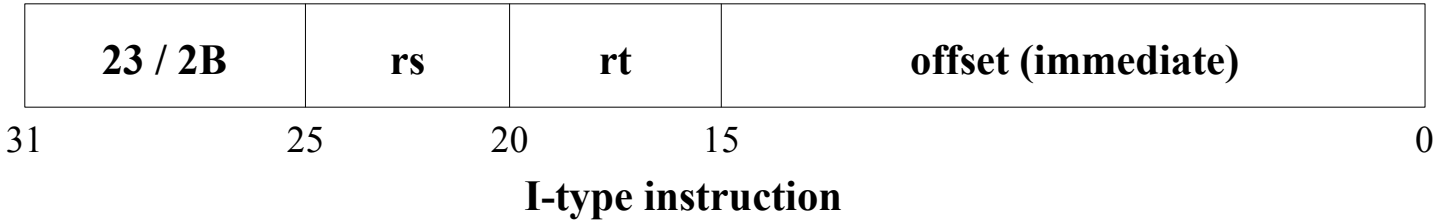


R-type instruction



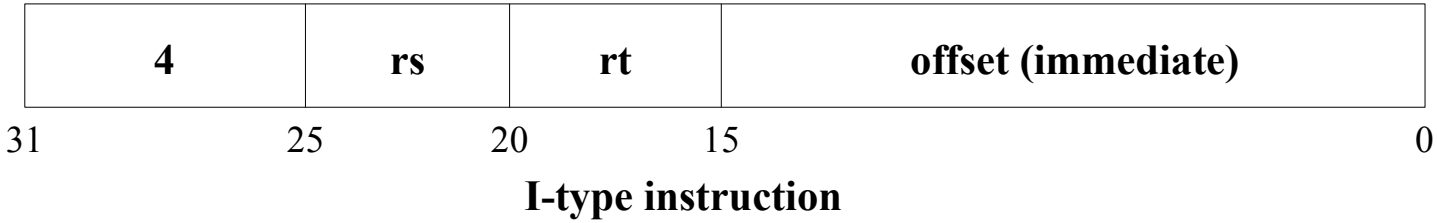
I-type instruction

load and store instructions



I-type instruction

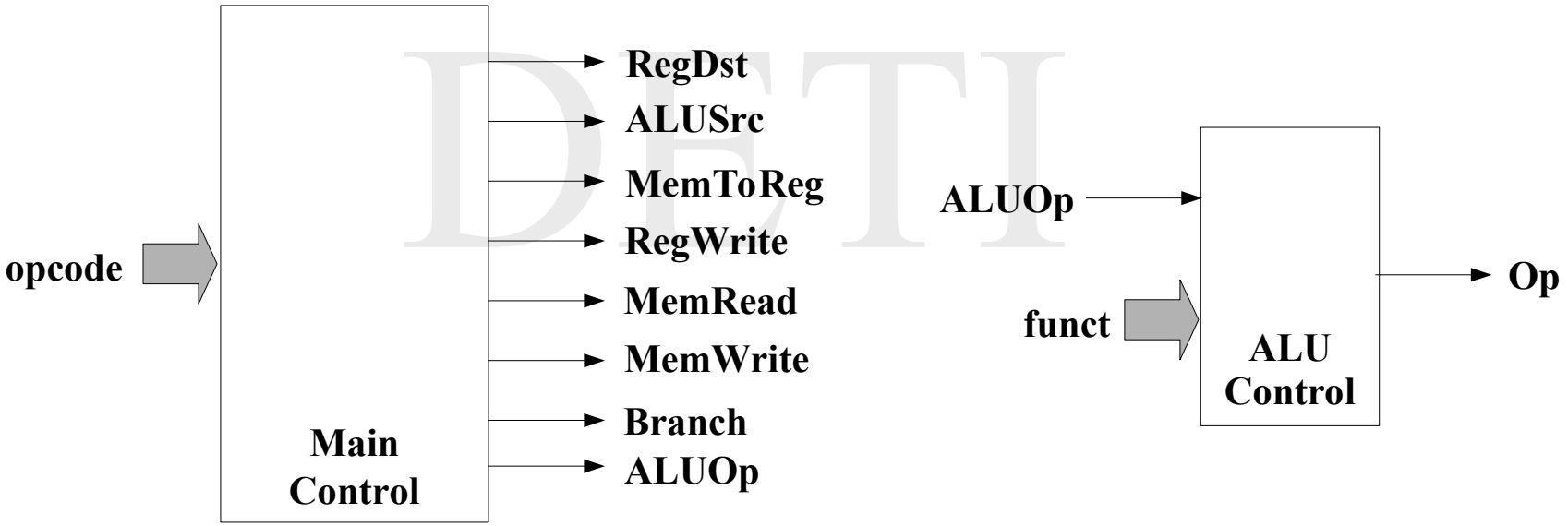
branch instructions



I-type instruction

Implementation of classical 5-stage pipeline - 3

Pipeline Control Unit



Implementation of classical 5-stage pipeline - 4

Main Control Functional Description

Source: Adapted from Computer Organization and Design: The Hardware/Software Interface

		Instruction Decode / Register Fetch Stage	Execution / Effective Address Stage				Memory Access Stage		Write Back Stage	
Instruction Type	**OpCode**	**Branch**	**RegDst**	**ALUSrc**	**ALUOp1**	**ALUOp2**	**MemRead**	**MemWrite**	**RegWrite**	**MemToReg**
R-format	0x00	0	1	0	1	0	0	0	1	0
lw	0x23	0	0	1	0	0	1	0	1	1
sw	0x2B	0	x	1	0	0	0	1	0	x
beq	0x04	1	x	0	0	0	0	0	0	x

Signal Name	Effect when deasserted (0)	Effect when asserted (1)
RegDst	the register destination number for the write register comes from the rt field (bits 20:16)	the register destination number for the write register comes from the rd field (bits 15:11)
RegWrite	none	the register whose number is in addin is written with the value in din
ALUSrc	the ALU second operand comes from the contents of the register whose number is in rt field (bits 20:16)	the ALU second operand is the immediate field (bits 15:0) after sign-extending it to 32 bits
PCSrc	the PC is replaced by the output of the adder that computes the value of PC+4	the PC is replaced by the output of the adder that computes the branch target address
MemRead	none	the contents of the memory location referenced by add is placed at dout
MemWrite	none	the contents of the memory location referenced by add is replaced by the value at din
MemToReg	the value fed to the data input of the write register comes from the ALU	the value fed to the data input of the write register comes from the dout of data memory

Implementation of classical 5-stage pipeline - 5

ALU Control Functional Description

Source: Adapted from Computer Organization and Design: The Hardware/Software Interface

Instruction	ALUOp	Function	Desired ALU action	Operation
lw	00	xxxxxx	add	0010
sw	00	xxxxxx	add	0010
add	10	100000	add	0010
subtract	10	100010	subtract	0110
and	10	100100	and	0000
or	10	100101	or	0001
slt	10	101010	set on less than	0111

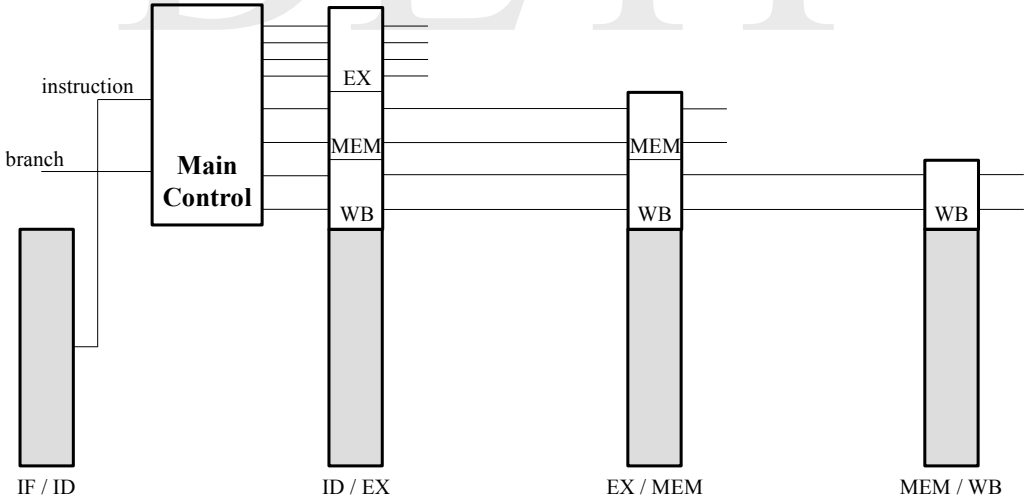
Implementation of classical 5-stage pipeline - 6

In order control circuitry to work properly, the control signals must be set to the right values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.

Since control signals start to be applied at the EX stage, the control information can be created during the ID stage.

Deployment of control signals through the different pipe stages

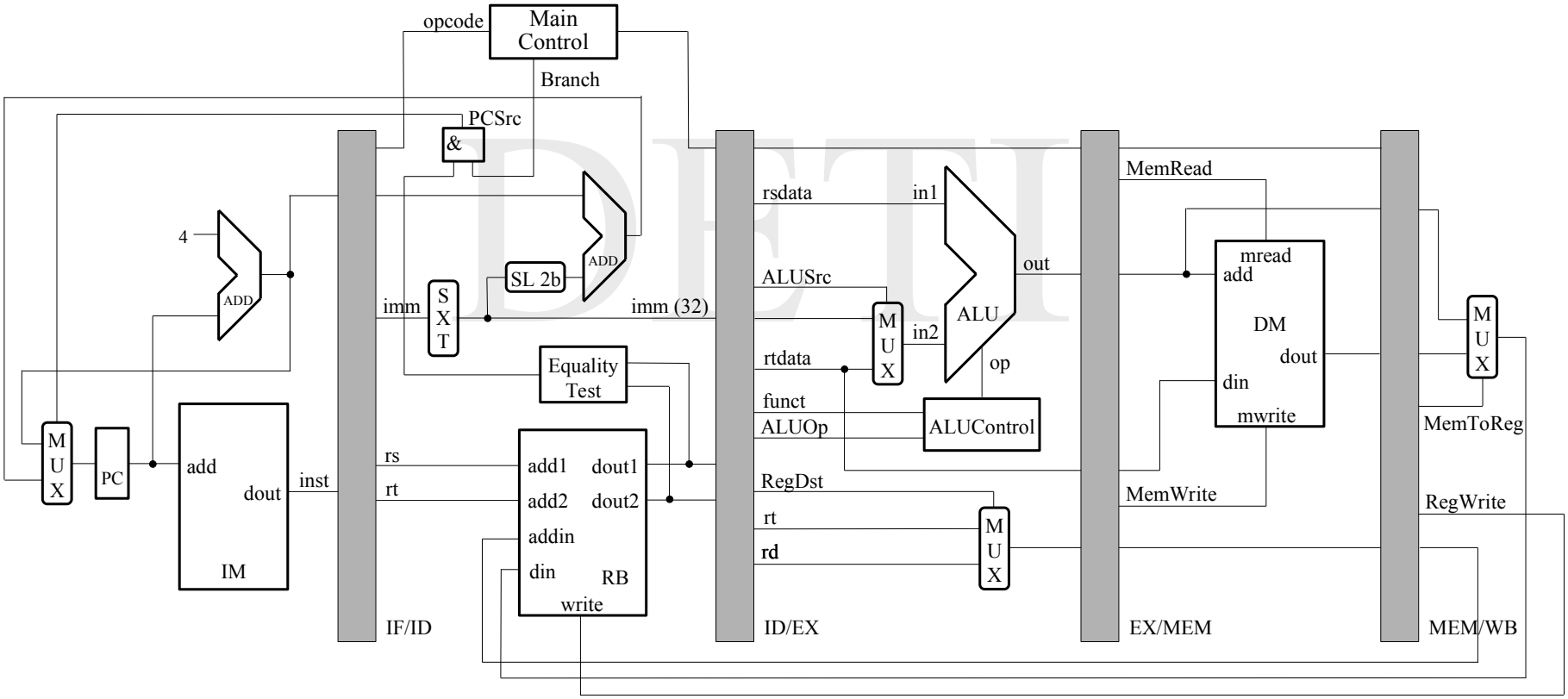
Source: Adapted from Computer Organization and Design: The Hardware/Software Interface



Implementation of classical 5-stage pipeline - 7

Integrating control in the 5-stage pipeline datapath

Source: Adapted from Computer Architecture: A Quantitative Approach



Implementation of classical 5-stage pipeline - 8

Data hazards that can be solved by *forwarding* for this specific 5-stage pipeline implementation, may be classified in the following classes

- ID *hazard* – the contents of a register of the register bank is required by a `beq` instruction for equality test and has been modified by a prior instruction and its updated value is presently stored in a ID/EX, EX/MEM or MEM/WB pipeline register
- EX *hazard* – the contents of a register of the register bank is required by an instruction and has been modified by a prior instruction and its updated value is presently stored in a EX/MEM or MEM/WB pipeline register
- MEM *hazard* – the contents of a register of the register bank is required by an instruction and has been modified by a prior instruction and its updated value is presently stored in a MEM/WB pipeline register.

Implementation of classical 5-stage pipeline - 9

Detecting ID data hazards

```
if (Branch && ID/EXE.RegWrite && (ID/EXE.RegDest ≠ 0) &&
     (ID/EXE.RegDest == IF/ID.RegRs) )
    fwdIDA = 01
    else if (Branch && EX/MEM.RegWrite && (EX/MEM.RegDest ≠ 0) &&
              (EX/MEM.RegDest == IF/ID.RegRs) )
             fwdIDA = 10
             else if (Branch && MEM/WB.RegWrite) &&
                      (MEM/WB.RegDest ≠ 0) &&
                      (MEM/WB.RegDest == IF/ID.RegRs) )
                     fwdIDA = 11
                     else fwdIDA = 00
```

Implementation of classical 5-stage pipeline - 10

Detecting ID data hazards (cont.)

```
if (Branch && ID/EXE.RegWrite && (ID/EXE.RegDest ≠ 0) &&
     (ID/EXE.RegDest == IF/ID.RegRt) )
    fwdIDB = 01
    else if (Branch && EX/MEM.RegWrite && (EX/MEM.RegDest ≠ 0) &&
              (EX/MEM.RegDest == IF/ID.RegRt) )
             fwdIDB = 10
             else if (Branch && MEM/WB.RegWrite) &&
                      (MEM/WB.RegDest ≠ 0) &&
                      (MEM/WB.RegDest == IF/ID.RegRt) )
                     fwdIDB = 11
                     else fwdIDB = 00
```

Implementation of classical 5-stage pipeline - 11

Detecting EX data hazards

```
if (EX/MEM.RegWrite && (EX/MEM.RegDest ≠ 0) &&
     (EX/MEM.RegDest == ID/EX.RegRs) )
    fwdEXA = 10
    else if (MEM/WB.RegWrite && (MEM/WB.RegDest ≠ 0) &&
              (MEM/WB.RegDest == ID/EX.RegRs) )
             fwdEXA = 11
             else fwdEXA = 00

if (EX/MEM.RegWrite && (EX/MEM.RegDest ≠ 0) &&
     (EX/MEM.RegDest == ID/EX.RegRt) && !ID/EX.ALUSrc )
    fwdEXB = 10
    else if (MEM/WB.RegWrite && (MEM/WB.RegDest ≠ 0) &&
              (MEM/WB.RegDest == ID/EX.RegRt) && !ID/EX.ALUSrc )
             fwdEXB = 11
             else if (ID/EX.ALUSrc)
                      fwdEXB = 01
                      else fwdEXB = 00
```

Implementation of classical 5-stage pipeline - 12

Detecting MEM data hazards


```
if (EX/MEM.MemWrite && (EX/MEM.RegDest ≠ 0) &&
     MEM/WB.RegWrite &&
     (EX/MEM.RegDest == MEM/WR.RegDest))
    fwdMEM = 1
    else fwdMEM = 0
```

Implementation of classical 5-stage pipeline - 13

Selection values for the forwarding multiplexors at the ID stage

MUX Selection	Source	Explanation
fwdIDA= 00	IF/ID	The first ALU operand comes from the register file
fwdIDA= 01	ID/EX	The first ALU operand is forwarded from the last ALU result
fwdIDA= 10	EX/MEM	The first ALU operand is forwarded from first to last ALU result
fwdIDA= 11	MEM/WB	The first ALU operand is forwarded from data memory or second to last ALU result
fwdIDB = 00	IF/ID	The second ALU operand comes from the register file
fwdIDB = 01	ID/EX	The second ALU operand is forwarded from the last ALU result
fwdIDB = 10	EX/MEM	The second ALU operand is forwarded from first to last ALU result
fwdIDB = 11	MEM/WB	The second ALU operand is forwarded from data memory or second to last ALU result

Implementation of classical 5-stage pipeline - 14

Selection values for the forwarding multiplexors at the EX stage

MUX Selection	Source	Explanation
fwdEXA= 00	ID/EX	The first ALU operand comes from the register file
fwdEXA= 10	EX/MEM	The first ALU operand is forwarded from the last ALU result
fwdEXA= 11	MEM/WB	The first ALU operand is forwarded from data memory or first to last ALU result
fwdEXB = 00	ID/EX	The second ALU operand comes from the register file
fwdEXB = 01	ID/EX	The second ALU operand is the immediate field sign-extended to 32 bits
fwdEXB = 10	EX/MEM	The second ALU operand is forwarded from the last ALU result
fwdEXB = 11	MEM/WB	The second ALU operand is forwarded from data memory or first to last ALU result

Implementation of classical 5-stage pipeline - 15

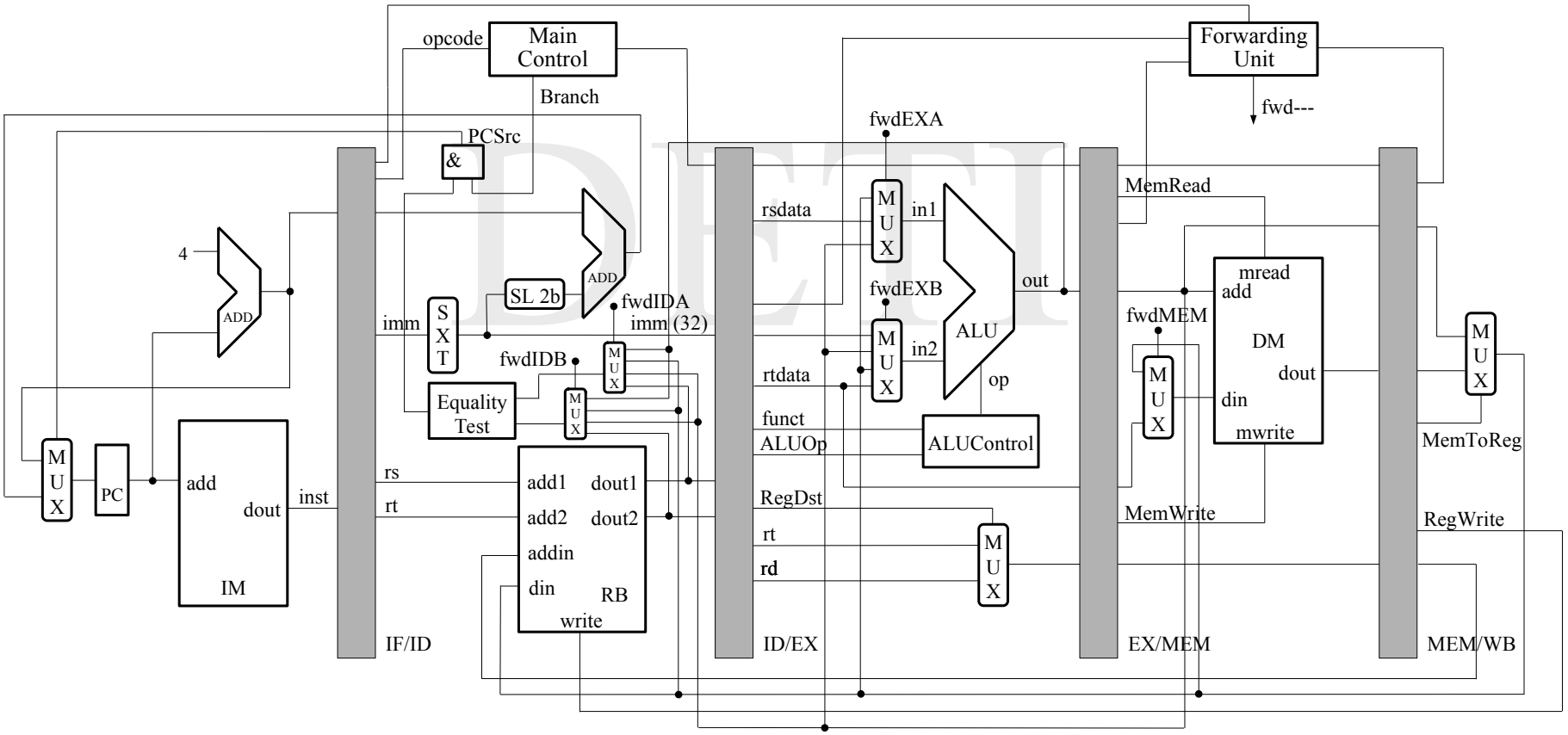
Selection values for the forwarding multiplexors at the MEM stage

MUX Selection	Source	Explanation
fwdMEM = 0	EX/MEM	The value to be possibly written in data memory comes from the contents of register rt
fwdMEM = 1	MEM/WB	The value to be written in data memory is the value that has just been produced

Implementation of classical 5-stage pipeline - 16

Integrating forwarding in the 5-stage pipeline datapath

Source: Adapted from Computer Architecture: A Quantitative Approach



Implementation of classical 5-stage pipeline - 17

Not all the *data hazards* for this specific 5-stage pipeline implementation can be solved by *forwarding*. As it was seen, if an instruction immediately following a `load` tries to read the same register which is written by it, there is no way for this instruction to proceed until the new value is effectively retrieved from memory. So the progress of this instruction in the pipeline must be *stalled*.

Furthermore, and as far as *control hazards* are concerned, if a *delayed branch* scheme is assumed, no new abnormal situations that would require the stalling of instruction progress, need to be considered.

When the instruction that follows the `load` is a ALU instruction, one stall cycle is required; when it is a `beq` instruction, two stall cycles are required.

Implementation of classical 5-stage pipeline - 18

Detecting the need for the insertion of stall cycles

```
if (ID/EX.MemRead && !MemWrite &&
    ((RegDest == IF/ID.RegRs) ||
     (RegDest == IF/ID.RegRt && !MemRead))
    stall the pipeline
    else if (Branch && EX/MEM.MemRead &&
               ((EX/MEM.RegDest == IF/ID.RegRs) ||
                (EX/MEM.RegDest == IF/ID.RegRt))
              stall the pipeline
```

Implementation of classical 5-stage pipeline - 19

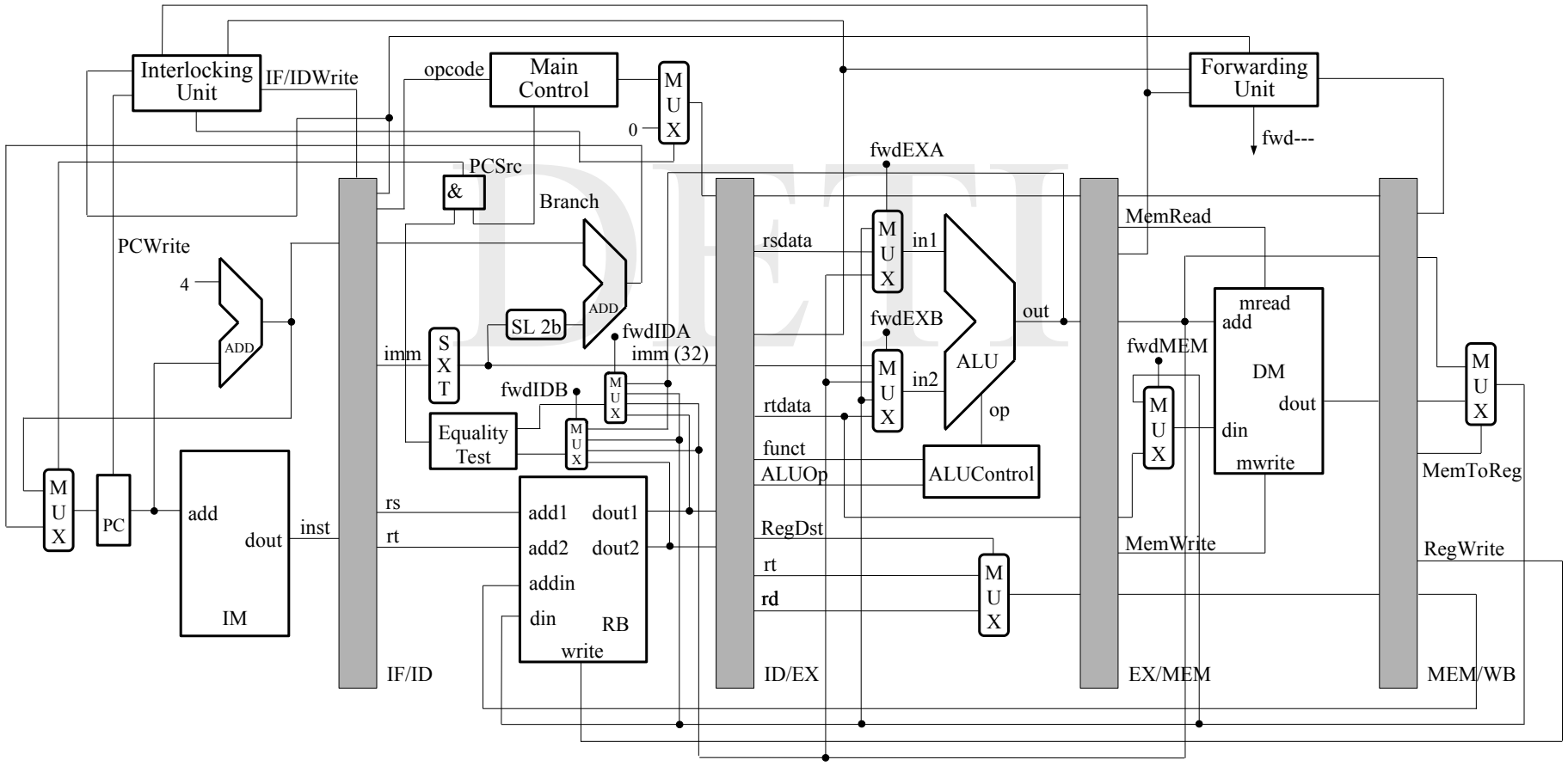
In order for the instruction at the ID stage to be stalled, the instruction at the IF stage must also be stalled. A simple way to achieve this is to prevent both the PC and the IF/ID pipeline registers from changing. Provided the contents of these registers is preserved, the instructions processed at the IF and ID stages are kept the same.

On the other hand, the instructions at the other stages must proceed as if nothing had happened. This means that a *no-op* instruction must be generated and inserted in the EX stage at the next clock cycle. This can be achieved by disabling the control signals when they are written to the ID/EX pipeline registers.

Implementation of classical 5-stage pipeline - 20

Integrating interlocking in the 5-stage pipeline datapath

Source: Adapted from Computer Architecture: A Quantitative Approach



Exceptions - 1

Control is the most challenging aspect of processor design: it is both the hardest part to get right and the hardest part to make go fast. One of the hardest parts of control is allowing for *exceptions* or *interrupts* – events that cause the breaking of the strict sequentiality of instruction execution, other than branches and jumps.

Typical exceptions include

- requests of attention by a controller associated to a I/O device
- invoking a system call from the user level
- tracing instruction execution or defining breakpoints
- integer overflow or floating-point anomaly
- page fault in a virtual memory organization
- misaligned memory access
- memory protection violation
- trying to execute an undefined or unimplemented instruction
- hardware malfunction
- power failure.

Exceptions - 2

Individual exceptions have important characteristics that determine what action the hardware is required to perform. These requirements can be classified in five semi-independent categories

- *synchronous* vs. *asynchronous* – unlike asynchronous events, synchronous events occur at the same place every time the program is executed with the same data and memory allocation; asynchronous events, in contrast, may occur anywhere within the program and can be usually handled after the completion of the current instruction
- *user requested* vs. *coerced* – user requested exceptions, being predictable, are not really in some sense true exceptions; they are only treated as such because the same mechanism, which is used to save and restore the program state, is also applied to them; since the only function of the instruction that triggers this kind of exception is to cause the exception itself, they can always be handled after the instruction is completed; coerced exceptions, in contrast, are caused by some hardware event that the program does not control and are totally unpredictable

Exceptions - 3

- *maskable* vs. *non-maskable* – some exceptions allow the program to choose the moment the hardware responds to them
- *within* vs. *between instructions* – an event may prevent the completion of an instruction in execution, when it is triggered by it, because some malfunction or anomaly has occurred at the software / hardware level; the corresponding exceptions are usually synchronous and hard to implement because the instruction must be stopped and, eventually, restarted later; asynchronous exceptions that occur within instructions, in contrast, arise from catastrophic situations and always cause program termination
- *resume* vs. *terminate* – exceptions that do not require the program to resume after they are handled, are easier to implement because there is no need to restart the program.

Exceptions - 4

How common exceptions stand in the 5-category classification scheme

Source: Adapted from Computer Architecture: A Quantitative Approach

Exception Type	Synchronous vs. Asynchronous	User requested vs. Coerced	Maskable vs. Non-maskable	Within vs. Between instructions	Resume vs. Terminate
I/O device request	asynchonous	coerced	maskable	between	resume
operating system invocation	synchronous	user requested	maskable	between	resume
tracing instruction execution or defining breahpoints	synchronous	user requested	maskable	between	resume
integer overflow or FP anomaly	synchronous	coerced	maskable	within	resume
page fault (virtual memory)	synchronous	coerced	non-maskable	within	resume
misaligned memory access	synchronous	coerced	non-maskable	within	terminate
memory protection violation	synchronous	coerced	non-maskable	within	terminate
undefined or unimplemented instruction execution	synchronous	coerced	non-maskable	within	terminate
hardware malfunction	asynchonous	coerced	non-maskable	within	terminate
power failure	asynchonous	coerced	non-maskable	within	terminate

Exceptions - 5

As in non-pipeline organizations, the difficult task is implementing exceptions occurring within instructions, typically at the EX or the MEM stages, which have to be resumed. The implementation supposes that another program, in principle the operating system, is invoked to save the state of the executing program, correct the cause of the exception and restore the state of the program, before the instruction that caused the exception is executed again – a mechanism that has obviously to be totally transparent to the executing program.

If the pipeline has the ability to handle the exception, save the state and restart without affecting program execution, the processor is said to be *restartable*. While early supercomputers and microprocessors often lacked this property, almost all processors today support it, at least for the integer pipeline, because it rests at the base of making operational virtual memory organization.

Exceptions - 6

For the operating system to be able to handle an exception, it must know the reason that has triggered it, in addition to the instruction that has caused it or that would next be executed if the exception was not serviced. There two main methods to communicate the reason for an exception

- *cause register* – it is a status register that holds a field describing the cause for an exception that has occurred, its value being set by the hardware upon its detection; when the exceptions are serviced by the same entry point address, this is the means the operating system has to determine the cause of the exception
- *vectored exceptions* – there are multiple entry point addresses for service of the exceptions; in general, each entry point address is associated with a particular exception so the identification of the cause by the operating system becomes trivial.

Exceptions - 7

When an exception is serviced, the pipeline control must take the following steps to save the program state and allow the program to resume later on, if this is to be the case.

1. The current value of PC is saved and is replaced by the address of the entry point corresponding to the exception for the next IF cycle. The processor is placed at a privileged mode where a wider instruction set is available and all maskable exceptions of the same or lower priority level are disabled.
2. All succeeding instructions currently in the pipeline, and the instruction itself if the exception is within, have to be turned into *no-op* instructions for the remaining pipe stages. All preceeding instructions, if any, on the other hand, must be allowed to complete so that the program state is consistent at the time of the processing of the exception.

Exceptions - 8

3. After the exception service routine in the operating system starts executing, it immediately updates the saved PC value to the correct value, which depends on the cause of the exception (within or in between, and if the former case is true, on the pipe stage that has triggered it).
   One should also notice that, if a *delayed branch* scheme is assumed, it is no longer possible to recreate the state of the processor with the storage of a single PC value – the instructions in the pipeline may not be sequentially related. In order for the exception service routine to be able to update the saved PC to the correct value, one needs as many PC addresses as the length of the *branch delay slot* plus 1.
4. Finally, after the termination of the exception service routine, special instructions of the *return from exception* type restart the former instruction stream by restoring the PC value and the former mode of execution.

Exceptions - 9

If the pipelined can be stopped so that the instructions just preceeding the faulting instruction are completed and itself, together with those succeeding it, can be restarted from scratch, the pipeline is said to have *precise exceptions*.

Ideally, the faulting instruction should not change the state of execution and, therefore, for correctly handling some exceptions, it is required that the faulting instruction produces no effects. For other exceptions, however, such as floating-point exceptions, the faulting instruction in some processors writes its result before the exception can be handled. In such cases, the hardware must be prepared to retrieve the source operands, even if the destination is the same as one of the source operands.

To overcome this problem, many recent high-performance processors have introduced two modes of operation: one mode has precise exceptions and the other, to be more performant, does not. Indeed, the precise exception mode must be slower because it has to allow a lot less overlapping among floating-point instructions.

Exceptions - 10

With pipelining, multiple exceptions may occur in the same clock cycle because there are multiple instructions in simultaneous execution.

Typical *within* exceptions that may occur in the classical 5-stage pipeline

Source: Computer Architecture: A Quantitative Approach

Pipeline Stage	Exceptions
IF	page fault on instruction fetch misligned memory access memory protection violation
ID	undefined or unimplemented instruction
EX	arithmetic exception
MEM	page fault on data fetch misligned memory access memory protection violation
WB	none

Exceptions - 11

Consider the instruction sequence

LD	IF	ID	EX	MEM	WB	
DADD		IF	ID	EX	MEM	WB

This pair of instructions can cause a data page fault and an arithmetic exception in the same clock cycle. The way to deal with this problem is to respond to the data page exception and then restart execution. The arithmetic exception will reoccur and only then can be handled independently.

In general, things are not so straightforward. Exceptions may occur *out of order*, that is, an instruction may trigger an exception before a preceeding one, which is in simultaneous execution, also triggers its own. To visualize this, take for instance the situation where the *load* instruction generates a page fault on data fetch at the MEM stage and the *add* instruction generates a page fault on instruction fetch at the IF stage.

To develop a *precise exceptions* pipeline implementation, the exception triggered by the *load* instruction must be handled first. How can that be done?

Exceptions - 12

The pipeline cannot handle exceptions as they occur in time, since by doing this the exceptions risk to be processed out of the non-pipelined order.

Instead, the hardware posts all exceptions caused by a given instruction in a status vector associated with that instruction. The status vector keeps moving along the pipe stages with the instruction. As soon as there is an exception indication in the status vector, any control signal that may cause a data value to be written, is deactivated (this means the register write and the memory write signals).

When the instruction enters the WB stage, the status vector is checked. If any exception is posted, it is handled in the order it would occur in time in a non-pipelined implementation.

Multicycle operations in classical 5-stage pipeline - 1

It is impractical to assume that all FP operations and integer multiplications and divisions will complete in one clock cycle, or even in two. Doing so would mean extending the clock period to allow the execution of the operations within it, or using an enormous amount of logic in the implementation of the functional units, or both together. Instead, the FP pipeline will contemplate a longer latency for the operations.

The best way to grasp the idea is imagining the FP instructions as having the same pipeline as the core integer instructions, with two important differences

- the EX cycle may be repeated as many times as needed to complete the operation – the number of repetitions can vary with the operation
- there may be multiple functional units.

A stall will occur if the instruction to be issued will cause either a structural hazard for the required functional unit, or a data hazard.

Multicycle operations in classical 5-stage pipeline - 2

For the sake of discussion, four separate functional units will be considered in the pipeline implementation

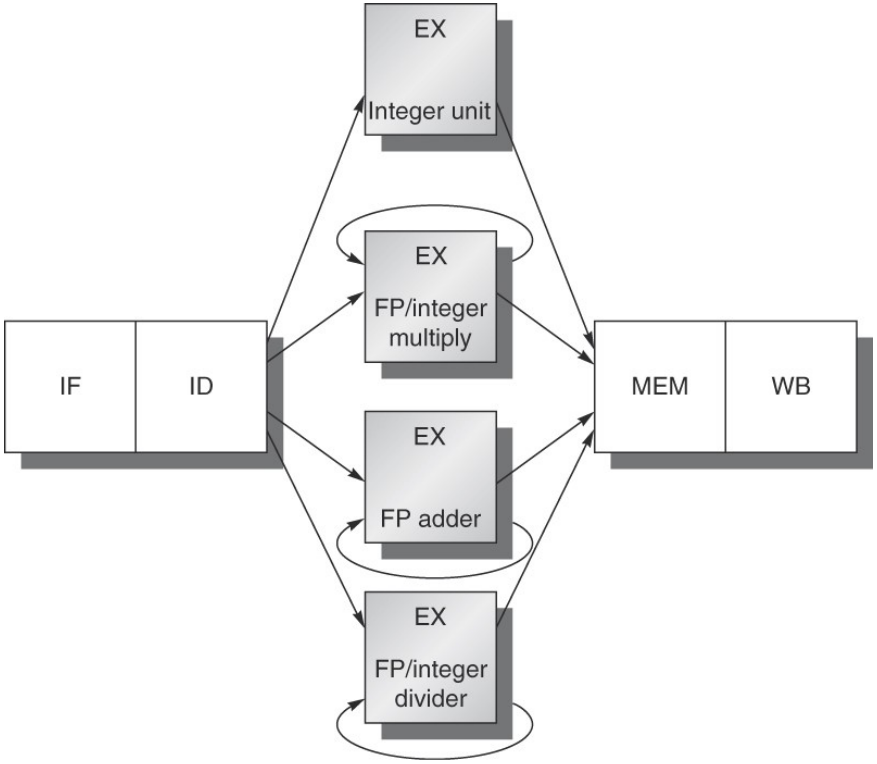
- the main integer unit, which handles loads, stores and the basic integer ALU operations
- the FP and integer multiplier
- the FP adder, which handles addition, subtraction and data type conversions
- the FP and integer divider.

Assuming these functional units are not pipelined, no new instruction using a specific functional unit may be issued if a previous instruction using the same unit is still in operation, or which is the same, has not yet left the EX stage. Moreover, if an instruction cannot proceed to the EX stage, the entire pipeline up to this point will be stalled.

Multicycle operations in classical 5-stage pipeline - 3

The classical 5-stage pipeline with three additional non-pipelined FP functional units

Source: Computer Architecture: A Quantitative Approach



Multicycle operations in classical 5-stage pipeline - 4

The structure of the whole pipeline may be generalized to allow pipelining of some FP functional units and enable multiple ongoing operations. To ease the way the description is made, two definitions are introduced for the EX functional units

- *latency* – is the number of intervening clock cycles between an instruction that produces a result and an instruction that uses the result
- *initiation* or *repetition interval* – is the number of clock cycles that must elapse between issuing two operations of the same type.

Integer ALU operations have a latency of zero since the results can be used on the next clock cycle. *Loads*, on the other hand, have a latency of one since the results can be used after one intervening clock cycle. Futhermore, since most operations consume their operands at the beginning of the EX stage, latency is usually the number of stages after EX that an instruction needs to produce the result: zero stages for the integer ALU operations and one stage for the load operations. The exception is *stores* where the latency is minus one.

Multicycle operations in classical 5-stage pipeline - 5

Latencies and repetition intervals for the operations in the functional units

Source: Computer Architecture: A Quantitative Approach

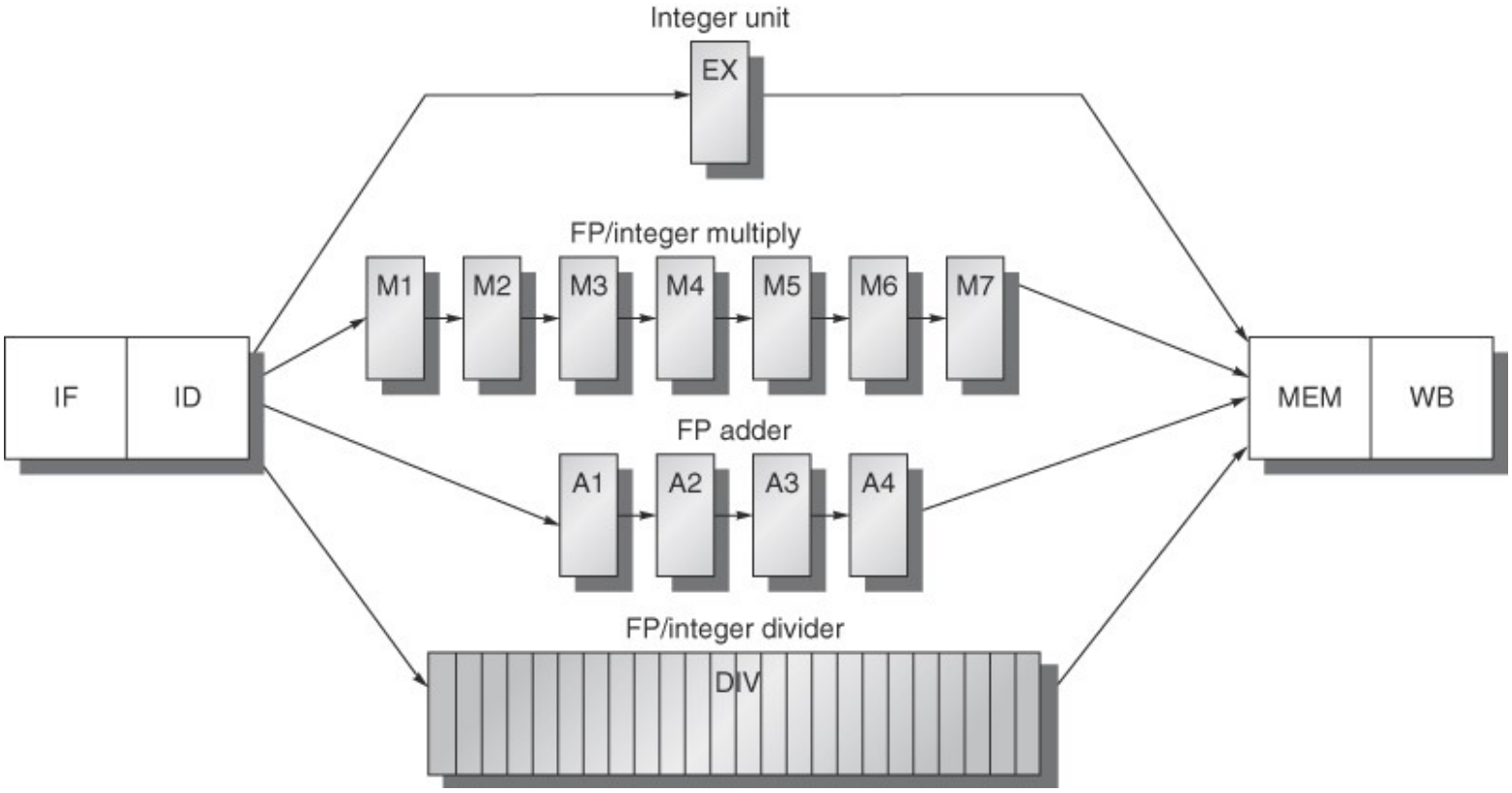
Functional Unit	Latency	Repetition Interval
Integer ALU	0	1
Data memory (Integer and FP loads)	1	1
FP addition	3	1
FP multiplication Integer multiplication	6	1
FP division Integer division	24	25

Notice that only the FP addition and the FP multiplication / Integer multiplication units are pipelined, The FP division / Integer division unit, on the other hand, is not pipelined and requires 24 clock cycles to produce a result.

Multicycle operations in classical 5-stage pipeline - 6

The classical 5-stage pipeline supporting multiple outstanding FP operations

Source: Computer Architecture: A Quantitative Approach



Multicycle operations in classical 5-stage pipeline - 7

The following observations are in order

- since the FP division / Integer division unit is not pipelined, *structural hazards* can occur; they have to be detected and the issuing instructions needing to use the unit have to be stalled
- since now not all instructions have the same execution time, the number of register writes in the same clock cycle may be larger than one
- *data hazards*, where an instruction tries to write a register before it has been written by another instruction, are possible – which implies that the non-pipelined execution order is no longer maintained
- the fact that instructions can complete in a different order from the one they were issued, will give rise to problems when dealing with exceptions
- *data hazards*, where an instruction tries to read a register before it has been written by another instruction, will be more frequent.

Multicycle operations in classical 5-stage pipeline - 8

Data hazards, where an instruction tries to read a register before it has been written by another instruction, follow a pattern that is fundamentally the same found for the integer pipeline.

	Clock number																
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***	***12***	***13***	***14***	***15***	***16***	***17***
`L.D F4,0(R2)`	IF	ID	EX	MEM	WB												
`MUL.D F0,F4,F6`		IF	ID	*stall*	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
`ADD.D F2,F0,F8`			IF	*stall*	ID	*stall*	*stall*	*stall*	*stall*	*stall*	*stall*	A1	A2	A3	A4	MEM	WB
`S.D F2,0(R2)`					IF	*stall*	*stall*	*stall*	*stall*	*stall*	*stall*	ID	EX	*stall*	*stall*	*stall*	MEM

Notice that the *store* instruction has been stalled an extra cycle to prevent the structural hazard that would arise from the access conflict to the MEM stage. However, since only one of the instructions accesses data memory, they can be made to proceed to the MEM stage in the same clock cycle, if extra hardware is added.

Multicycle operations in classical 5-stage pipeline - 9

If one assumes that the FP register bank has a single write port, sequences of FP operations, as well as a FP *load* instruction together with other FP operations, can give rise to access conflicts to the register write port.

	Clock number										
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***11***
`MUL.D F0,F4,F6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
. . .		IF	ID	EX	MEM	WB					
. . .			IF	ID	EX	MEM	WB				
`ADD.D F2,F4,F6`				IF	ID	A1	A2	A3	A4	MEM	WB
. . .					IF	ID	EX	MEM	WB		
. . .						IF	ID	EX	MEM	WB	
`L.D F0,0(R2)`							IF	ID	EX	MEM	WB

In clock cycle 11, all three instructions reach the WB stage and need to write to the FP register bank. With a single write port, the processor must serialize instruction completion. The number of write ports could be increased to solve the problem, but this approach may not be very attractive since the extra write ports may be seldom used.

Multicycle operations in classical 5-stage pipeline - 10

Instead, one can choose to detect the structural hazard and to schedule the access to the write port.

One solution is to track the use of the write port at the ID stage and to stall, if it is necessary, the current instruction before it issues, just as it is done for any other structural hazard. The implementation of this approach needs a shift register, called the *reservation register*, whose contents moves in the opposite direction to the instruction flow in the pipeline at every clock cycle. Its purpose is to signal the cycles where instructions already issued will write at the FP register bank. Thus, when the instruction at the ID stage will have to write to a register of the FP register bank, the corresponding stage of the shift register is checked. If the bit is set, meaning that another instruction already issued will also be doing it in that clock cycle, the instruction is stalled. Otherwise, that bit of the shift register is set and the instruction progresses.

This approach has the advantage of keeping all interlock detection and stall insertion restricted to the ID stage. The cost is the extra shift register and all the logic for determining the potential write conflict.

Multicycle operations in classical 5-stage pipeline - 11

Another solution is to stall a conflicting instruction as it tries to enter either the MEM or the WB stage. Any of the conflicting instructions may be chosen to stall. A simple, though sometimes suboptimal, heuristic is to give priority to the instruction exiting the unit with the highest latency, since it is the one most likely to have caused another instruction to be stalled.

This approach has the advantage of postponing the conflict detection until the moment where it becomes trivial to assert it. The disadvantage is that it makes pipeline control more complex as stalls can now arise from two places.

Multicycle operations in classical 5-stage pipeline - 12

Data hazards, where an instruction tries to write to a register before it has been written by another instruction, are apparently not dramatic and could be discarded. The rationale is that they should never occur in a well written code sequence because no compiler would generate two writes to the same register without an intervening read. However, if the sequence is unexpected, they can indeed occur and produce wrong results.


```
        BNZ     R1,foo
        DIV.D   F0,F2,F4
        . . .
foo:    L.D     F0,0(R3)
```

If the branch is taken (assuming a delayed branch scheme), the `L.D` instruction will reach the WB stage before the `DIV.D` instruction can complete. Thus, originating an inconsistency in the program state from this point on.

Multicycle operations in classical 5-stage pipeline - 13

There are two possible ways to handle this hazard. The first approach is to delay the issue of the second writing instruction until the first enters the MEM stage. The second is to stamp out the first instruction by detecting the hazard and disabling its ability to change the register – the second instruction can then be immediately issued.

Due to its rarity to appear in the code, either approach is acceptable.

Multicycle operations in classical 5-stage pipeline - 14

A critical problem caused by long running instructions is illustrated by the code sequence bellow

```
DIV.D   F0,F2,F4
ADD.D   F10,F10,F8
SUB.D   F12,F12,F14
```

Although there are no data dependancies, the first instruction will complete after the next two. A situation that is known as *out of order completion*.

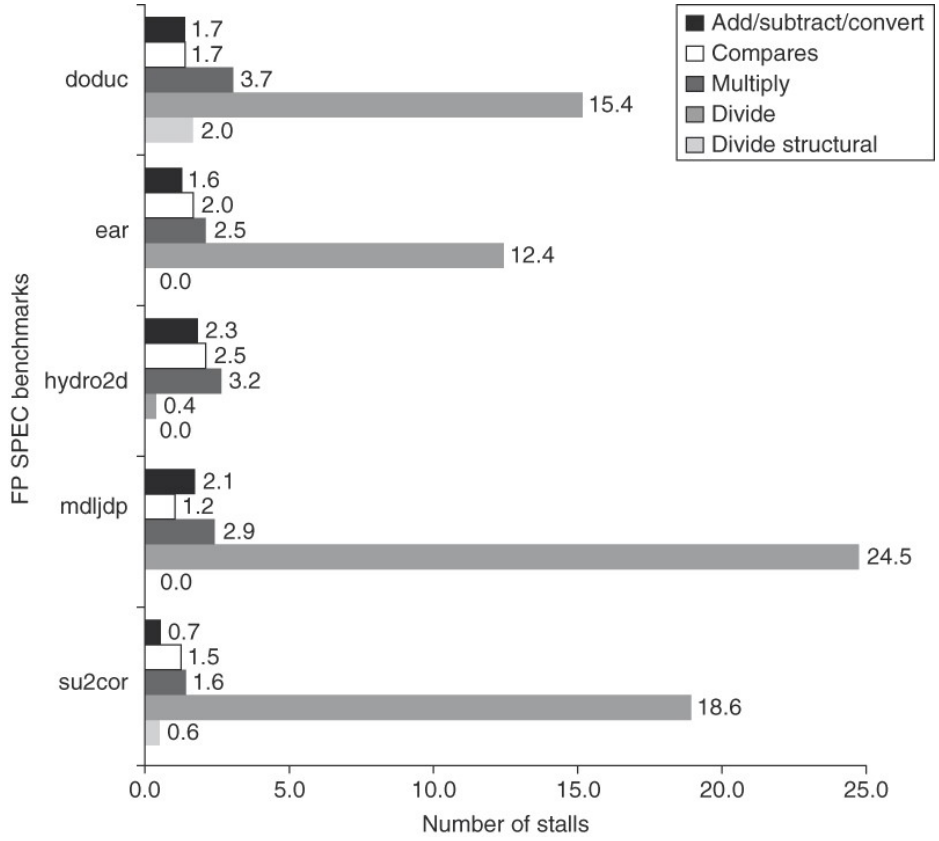
This condition may lead to *imprecise exceptions*. To understand how, consider that after the completion of the `ADD` and the `SUB` instructions, the `DIV` instruction will generate an exception. Since both `ADD` and `SUB` have changed each one of their operands, the program state has been modified and cannot be recovered, not even with software help. So restarting instruction `DIV` is not possible!

There are ways to deal with this problem, but they will not be discussed here.

Multicycle operations in classical 5-stage pipeline - 15

Stalls per FP operation for each major type of FP operation for SPEC89 FP benchmark

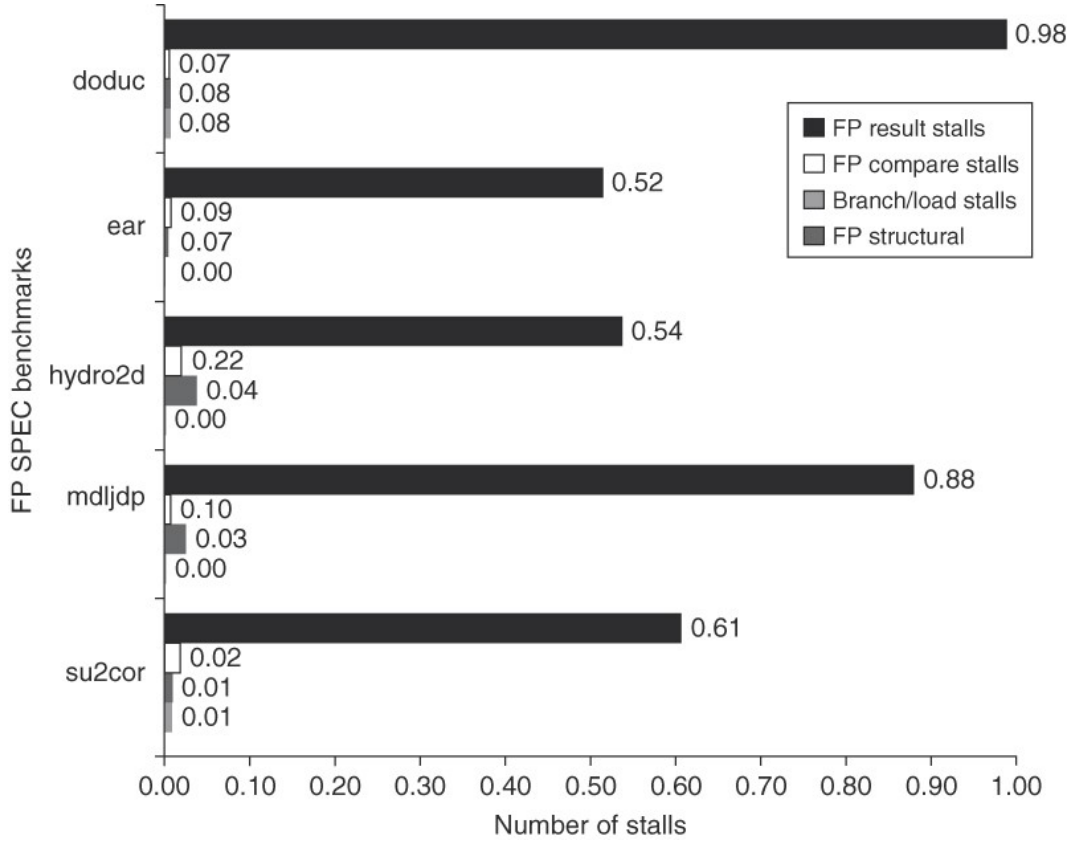
Source: Computer Architecture: A Quantitative Approach



Multicycle operations in classical 5-stage pipeline - 16

Stalls per instruction for the MIPS FP pipeline for SPEC89 FP benchmark

Source: Computer Architecture: A Quantitative Approach



MIPS R4000 pipeline - 1

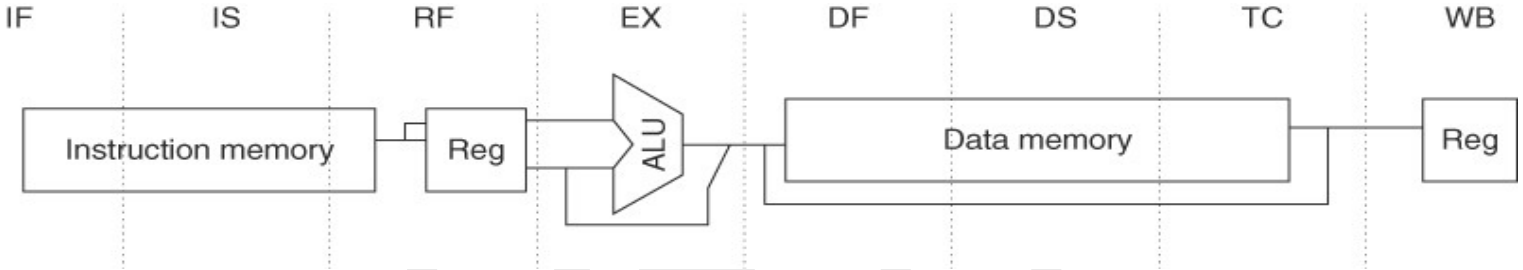
The MIPS R4000 processor family implements MIPS64 instruction set, but implements a deeper pipeline than the classical 5-stage pipeline that has been described, both for integer and FP programs.

The deeper pipeline allows the processor to achieve a higher clock rate by making a decomposition into eight stages, instead of five. Since memory access, even by the use of caches, is particularly time critical, the extra pipe stages are concerned with memory access decomposition. This type of deeper pipelining is sometimes called *superpipelining*.

MIPS R4000 pipeline - 2

R4000 eight-stage pipeline organization

Source: Computer Architecture: A Quantitative Approach



IF – first half of instruction fetch
IS – second half of instruction fetch
RF – instruction cache hit detection + instruction decode and register fetch + hazard checking
EX – execution, including ALU operation, effective address and branch target computation and condition evaluation
DF – first half of data fetch
DS – second half of data fetch
TC – data cache hit detection
WB – register write back

MIPS R4000 pipeline - 3

Although instruction and data memory accesses occupy multiple clock cycles, they are fully pipelined and, therefore, a new instruction can start on every clock cycle. In fact, as it can be noticed, since cache hit detection takes place on the last stage of memory access, the pipeline tries to use data even before hit detection is completed. If a miss occurs, the pipeline is backed up a cycle when the correct data are available.

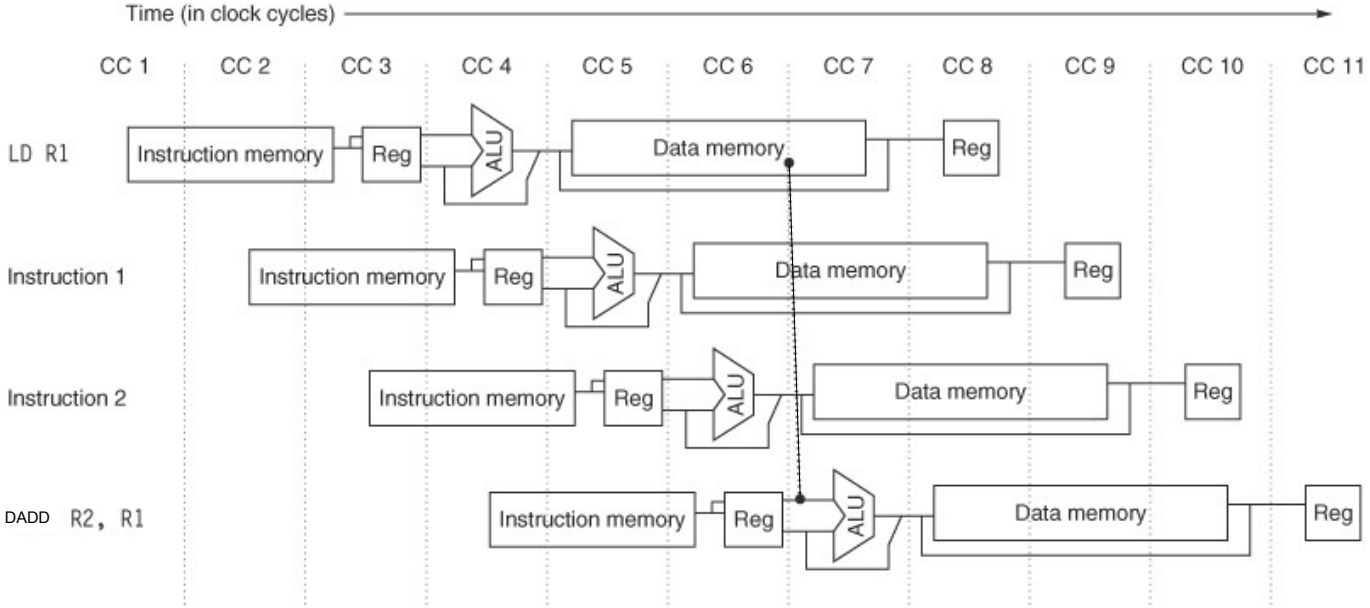
One should notice that this longer latency pipeline not only increases the required *forwarding* logic, but also increases the *load* and *branch* delays. The *load* delay is two cycles long, since data are available at the end of the DS stage. The *branch* delay is 3 cycles long, since the branching condition is computed here during the EX stage. R4000 architecture, however, uses a single cycle *delayed branch* scheme, together with a *predicted-untaken* strategy, which produces no idle cycles, when the branch is not taken, and two idle cycles, when the branch is taken.

Pipeline *interlocks* enforce both the 2-cycle branch stall penalty, when the branch is taken, and any data hazard that arises from the occurrence of a load instruction.

MIPS R4000 pipeline - 4

Two cycle load instruction delay on the eight-stage pipeline organization

Source: Computer Architecture: A Quantitative Approach



MIPS R4000 pipeline - 5

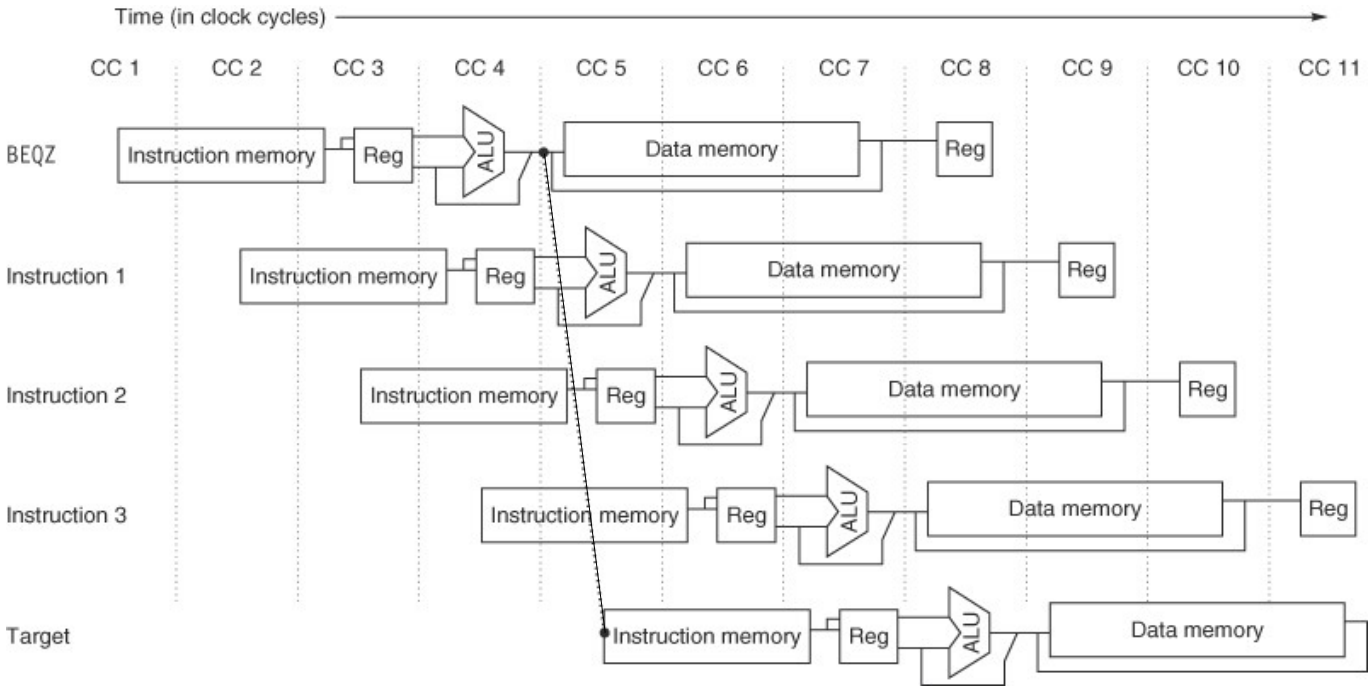
Two cycle stall produced by a load instruction, followed by the immediate use of the loaded value, on the eight-stage pipeline organization

	Clock number								
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
`LD R1,0(R10)`	IF	IS	RF	EX	DF	DS	TC	WB	
`DAAD R2,R1,R2`		IF	IS	RF	*stall*	*stall*	EX	DF	DS
`DSUB R3,R1,R3`			IF	IS	*stall*	*stall*	RF	EX	DF
`OR R4,R1,R4`				IF	*stall*	*stall*	IS	RF	EX

MIPS R4000 pipeline - 6

Three cycle basic branch delay on the eight-stage pipeline organization

Source: Computer Architecture: A Quantitative Approach



MIPS R4000 pipeline - 7

Delayed branch behavior both for the *taken* and *untaken* cases on the eight-stage pipeline organization

	Clock number								
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
delay slot		IF	IS	RF	EX	DF	DS	TC	WB
idle cycle			*stall*	*stall*	*stall*	*stall*	*stall*	*stall*	*stall*
idle cycle				*stall*	*stall*	*stall*	*stall*	*stall*	*stall*
branch target					IF	IS	RF	EX	DF

	Clock number								
Instruction	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***
branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
delay slot		IF	IS	RF	EX	DF	DS	TC	WB
branch instruction + 2			IF	IS	RF	EX	DF	DS	TC
branch instruction + 3				IF	IS	RF	EX	DF	DS

MIPS R4000 pipeline - 8

The MIPS R4000 floating-point unit consists of three nominal functional units: a floating-point adder, a floating-point multiplier and a floating-point divider. The adder logic is used in the final stages of a multiply or a divide operation.

Double precision operations may take from 2 cycles (for a negate) to 112 cycles (for a square root).

Each functional unit can be thought of as having up to eight different processing stages. There is a single copy of each of these processing stages and different instructions may use a particular stage zero or more times and combine them in their own order.

MIPS R4000 pipeline - 9

The eight processing stages in the R4000 floating point pipelines

Source: Computer Architecture: A Quantitative Approach

Processing stage	***Elemental functional unit***	***Description***
A	FP adder	mantissa ADD stage
D	FP divider	divide pipeline stage
E	FP multiplier	exception test stage
M	FP multiplier	multiplier first stage
N	FP multiplier	multiplier second stage
R	FP adder	rounding stage
S	FP adder	operand shift stage
U		unpack floating point numbers

MIPS R4000 pipeline - 10

Latencies, repetition intervals and stage composition for floating point operations in R4000

Source: Computer Architecture: A Quantitative Approach

FP instruction	Latency	Repetition interval	Processing Stages
add – subtract	4	3	U – S+A– A+R – R+S
multiply	8	4	U – E+M – M – M – M – N – N+A– R
divide	36	35	U – A– R – D^{28} – D+A – D+R – D+A – D+R – A– R
square root	112	111	U – E – $(A+R)^{108}$ – A– R
negate	2	1	U – S
absolute value	2	1	U – S
compare	3	2	U – A– R

MIPS R4000 pipeline - 11

Effect of a FP multiply instruction issued at clock cycle 0 on a FP add instruction issued between clock cycles 1 to 7

Source:Adapted from Computer Architecture: A Quantitative Approach

		Clock number										
Instruction	***Decision***	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
multiply	issue	U	E+M	M	M	M	N	N+A	R			
add	issue		U	S+A	A+R	R+S						
	issue			U	S+A	A+R	R+S					
	issue				U	S+A	A+R	R+S				
	stall					*stall*	*stall*	U	S+A	A+R	R+S	
	stall						*stall*	U	S+A	A+R	R+S	
	issue							U	S+A	A+R	R+S	
	issue								U	S+A	A+R	R+S

MIPS R4000 pipeline - 12

Effect of a FP divide instruction issued at clock cycle 0 on a FP add instruction issued between clock cycles 26 to 36

Source: Adapted from Computer Architecture: A Quantitative Approach

		Clock number										
Instruction	***Decision***	***26***	***27***	***28***	***29***	***30***	***31***	***32***	***33***	***34***	***35***	***36***
divide	issue at cc 0	D	D	D	D	D	D+A	D+R	D+A	D+R	A	R
add	issue		U	S+A	A+R	R+S						
	issue			U	S+A	A+R	R+S					
	stall				*stall*	*stall*	*stall*	*stall*	*stall*	*stall*	U	S+A
	stall					*stall*	*stall*	*stall*	*stall*	*stall*	U	S+A
	stall						*stall*	*stall*	*stall*	*stall*	U	S+A
	stall							*stall*	*stall*	*stall*	U	S+A
	stall								*stall*	*stall*	U	S+A
	stall									*stall*	U	S+A
	issue										U	S+A
	issue											U

MIPS R4000 pipeline - 13

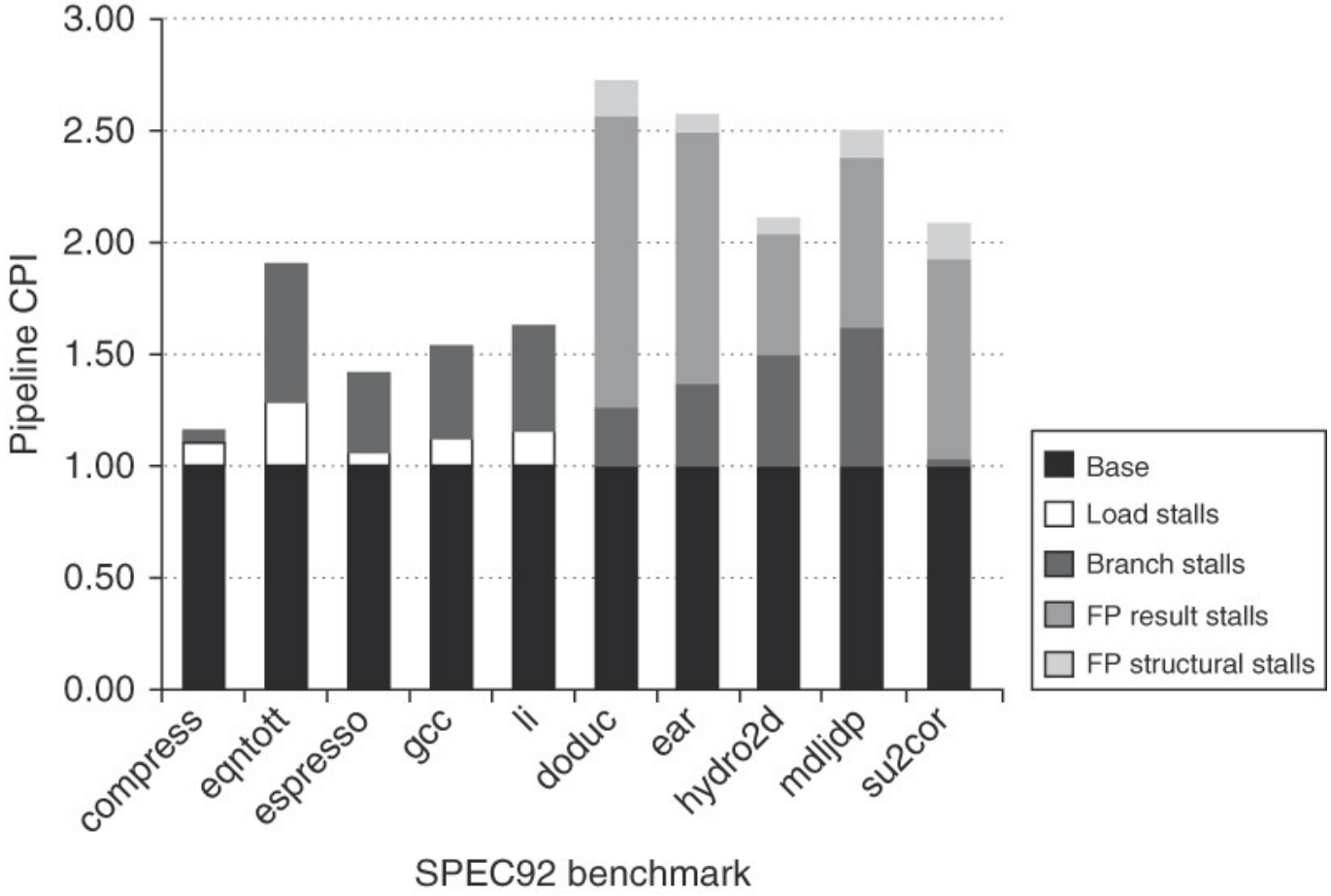
There are four major causes for pipeline stalls or losses which prevent that instruction issuing at the nominal rate be attained

- *load stalls* – delays arising from the use of the load value one or two clock cycles after load execution
- *branch stalls and losses* – two clock cycle delays after every *taken* branch and one clock cycle delay if the branch delay slot cannot be filled with an useful instruction
- *FP result stalls* – delays arising because an operand is required and is not yet computed
- *FP structural stalls* – delays arising because the required processing stages in the FP pipeline are not available when needed.

MIPS R4000 pipeline - 14

The pipeline CPI for 10 of the SPEC92 benchmarks assuming a perfect cache

Source: Computer Architecture: A Quantitative Approach



MIPS R4000 pipeline - 15

The R4000 pipeline has much longer branch delays than the classical 5-stage pipeline. The longer branch delay substantially increases the clock cycles spent on branches, specially for integer programs with a high branch frequency.

An interesting effect for FP programs is that the latency of the FP functional units leads to more result stalls than those produced by the structural hazards, which are due both to the repetition interval limitations and to conflicts from the use of specific processing stages in different FP instructions.

Thus, reducing the latency of FP operations should be the first task to be taken care of on an optimization effort, rather than increasing pipelining or replicating the functional units processing stages.

Suggested reading

- *Computer Architecture: A Quantitative Approach*, Hennessy J.L., Patterson D.A., 6th Edition, Morgan Kaufmann, 2017
  - Appendix A: *Instruction Set Principles* (Sections 1 to 8)
  - Appendix C: *Pipelining: Basic and Intermediate Concepts* (Sections 1 to 7)
- *Computer Organization and Architecture: Designing for Performance*, Stallings W., 10th Edition, Pearson Education, 2016
  - Chapter 12: *Instruction sets: Characteristics and Functions*
  - Chapter 13: *Instruction sets: Addressing Modes and Formats*
  - Chapter 14: *Processor Structure and Function*
  - Chapter 15: *Reduced Instruction Set Computers*