IC-UNICAMP

MO401

IC/Unicamp

Prof Mario Côrtes

Apêndice C:
Conceitos básicos de pipelining

IC-UNICAMP

Tópicos

- Funcionamento básico
- Hazards: estrutural, dados, controle
- Dificuldades na implementação de pipelines
- Extensão: operações multi-ciclo

Pipeline

- Objetivo: aumentar o throughput
  - se balanceado: speedup do throughput = nº estágios
- No Ap_C: ISA do MIPS64
- Dependendo da referência (baseline)
  - reduzir o CPI das instruções: meta → CPI =1
  - reduzir o cycle time: mais estágios menores e mais simples → menor cycle time

IC-UNICAMP

Instruções no pipeline: visão de tempo

	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

Figure C.1 Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write-back.

Pipeline: módulos no tempo

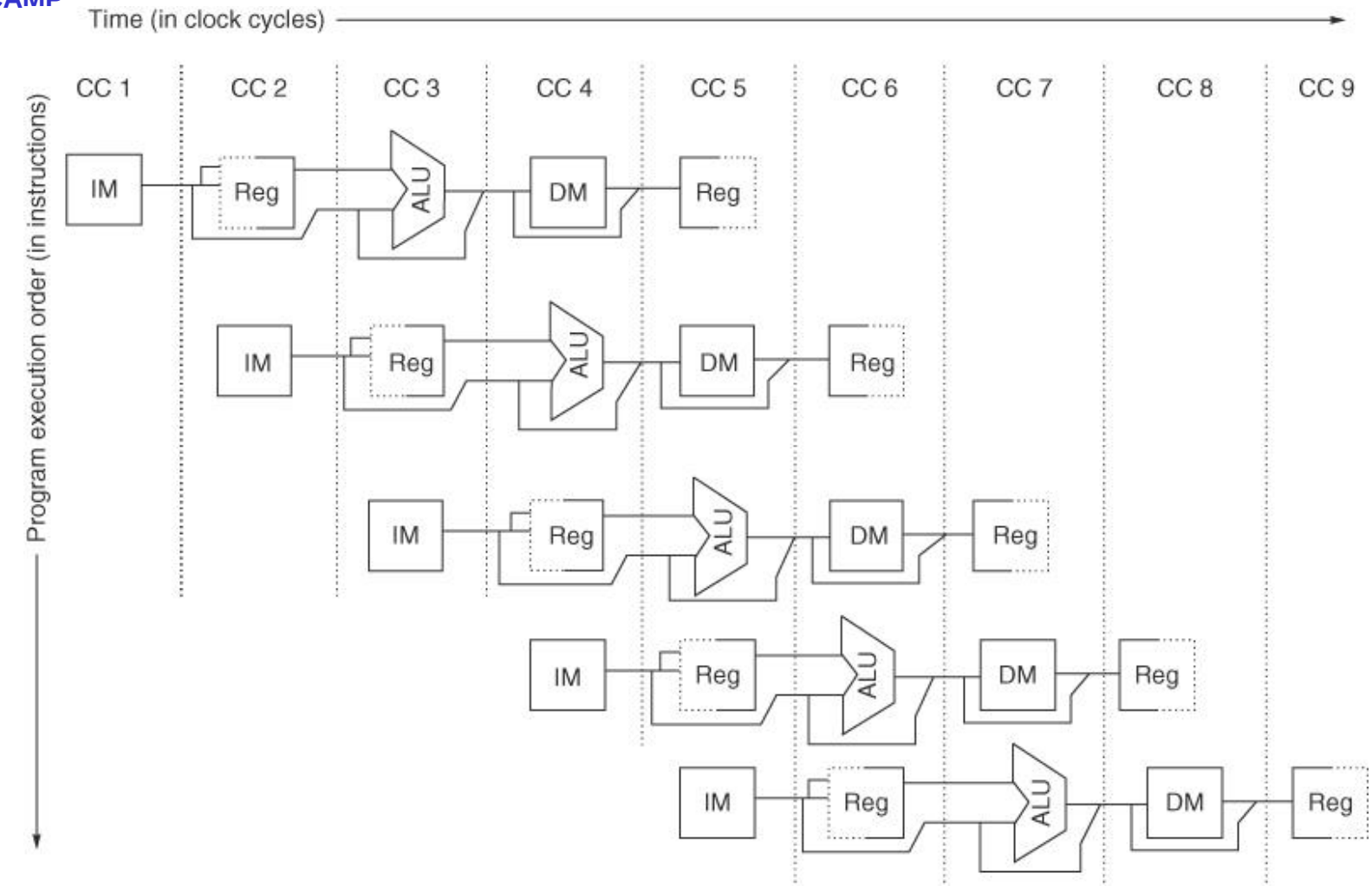


Figure C.2 The pipeline can be thought of as a series of data paths shifted in time. This shows the overlap among the parts of the data path, with clock cycle 5 (CC 5) showing the steady-state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one part of the stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

O pipeline básico do MIPS: registradores

IC-UNICAMP

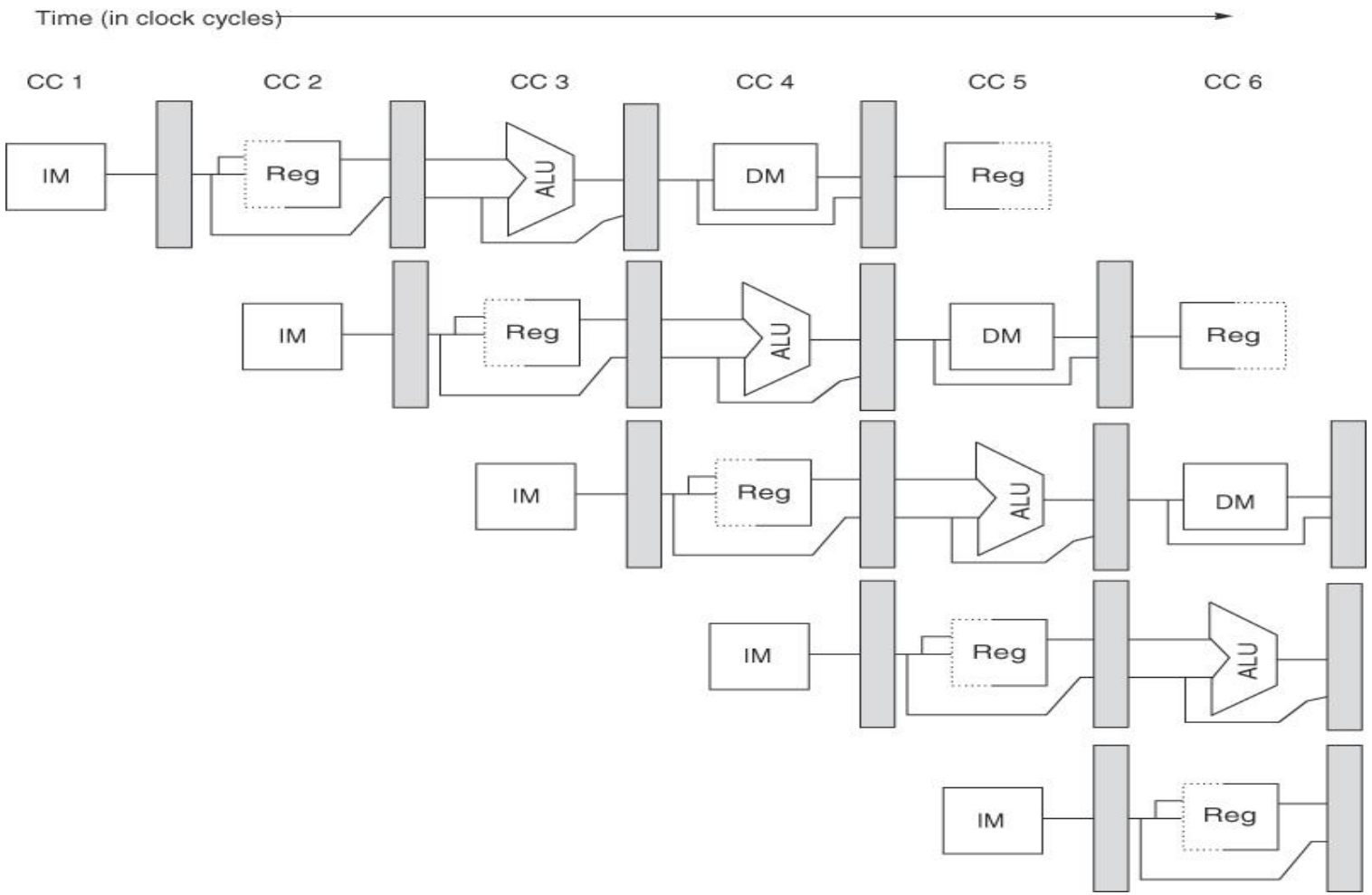


Figure C.3 A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

Exmpl C-10: Desempenho do pipeline

IC-UNICAMP

Consider the unpipelined processor in the previous section. Assume that it has a 1 ns clock cycle and that it uses 4 cycles for ALU operations and branches and 5 cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.2 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

Answer The average instruction execution time on the unpipelined processor is

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI}\\ &= 1\text{ ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5]\\ &= 1\text{ ns} \times 4.4\\ &= 4.4\text{ ns}\end{aligned}$$

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 1 + 0.2 or 1.2 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

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

$$= \frac{4.4\text{ ns}}{1.2\text{ ns}} = 3.7\text{ times}$$

The 0.2 ns overhead essentially establishes a limit on the effectiveness of pipelining. If the overhead is not affected by changes in the clock cycle, Amdahl's law tells us that the overhead limits the speedup.

C-2: Limites de Pipelining

- Hazards: impedem que a próxima instrução seja executada no ciclo de clock “previsto” para ela
  - Structural hazards: O HW não suporta uma dada combinação de instruções (falta de recurso)
  - Data hazards: Uma Instrução depende do resultado da instrução anterior que ainda está no pipeline
  - Control hazards: Causado pelo delay entre o fetching de uma instrução e a decisão sobre a mudança do fluxo de execução (branches e jumps).

IC-UNICAMP

Desempenho pipelines com stalls

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

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

- Objetivo do Pipeline: diminuir CPI ou cycletime
- Primeiro caso: diminuir CPI (CPI ideal com pipeline = 1)

$$\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$= 1 + \text{Pipeline stall clock cycles per instruction}$$

- Assumindo overhead=0, pipeline balanceado, cycletimes iguais

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

- Caso especial (frequente), latência de todas instruções = estágios

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

- Intuição OK: se stall = 0, speedup = pipeline depth

Desempenho pipelines com stalls (cont)

IC-UNICAMP

- Segundo caso: diminuir cycletime → CPI = 1 com ou sem pipeline

$$\text{Speedup from pipelining} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

- Se pipeline balanceado e overhead = 0 → ganho no cycletime = pipeline depth

$$\text{Clock cycle pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

$$\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\text{Speedup from pipelining} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}$$

- Intuição OK: se stall = 0, speedup = pipeline depth

Hazard estrutural

IC-UNICAMP

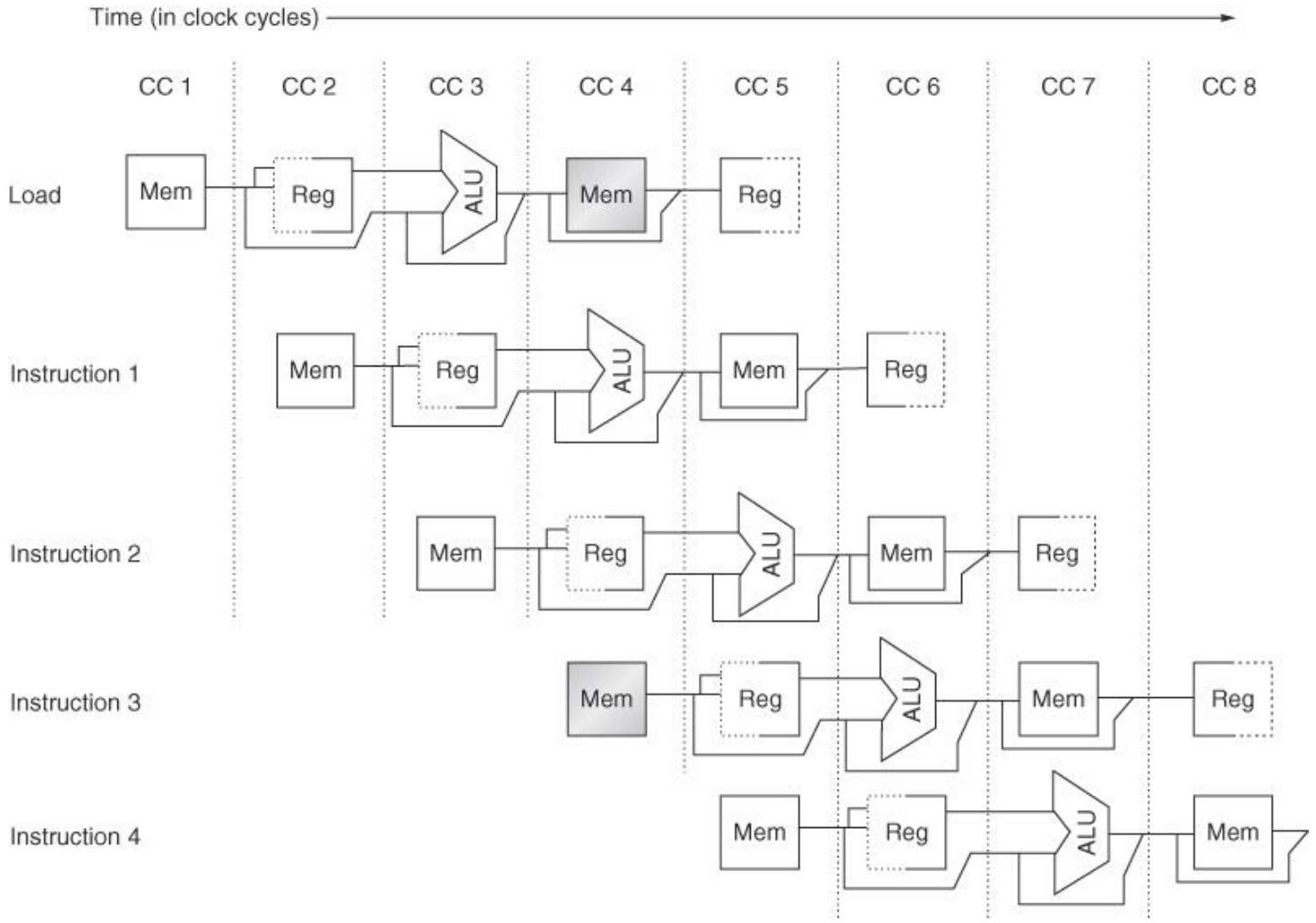


Figure C.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

IC-UNICAMP

Hazard estrutural

	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

Figure C.5 A pipeline stalled for a structural hazard—a load with one memory port. As shown here, the load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall—no instruction is initiated on clock cycle 4 (which normally would initiate instruction $i + 3$). Because the instruction being fetched is stalled, all other instructions in the pipeline before the stalled instruction can proceed normally. The stall cycle will continue to pass through the pipeline, so that no instruction completes on clock cycle 8. Sometimes these pipeline diagrams are drawn with the stall occupying an entire horizontal row and instruction 3 being moved to the next row; in either case, the effect is the same, since instruction $i + 3$ does not begin execution until cycle 5. We use the form above, since it takes less space in the figure. Note that this figure assumes that instructions $i + 1$ and $i + 2$ are not memory references.

Hazard de dados: RAW

- Dado produzido por uma instrução é lido pelas subsequentes
  - `DADD  R1, R2, R3`
  - `DSUB  R4, R1, R5`
  - `AND  R6, R1, R7`
  - `OR  R8, R1, R9`
  - `XOR  R10, R1, R11`

IC-UNICAMP

Hazard de dados

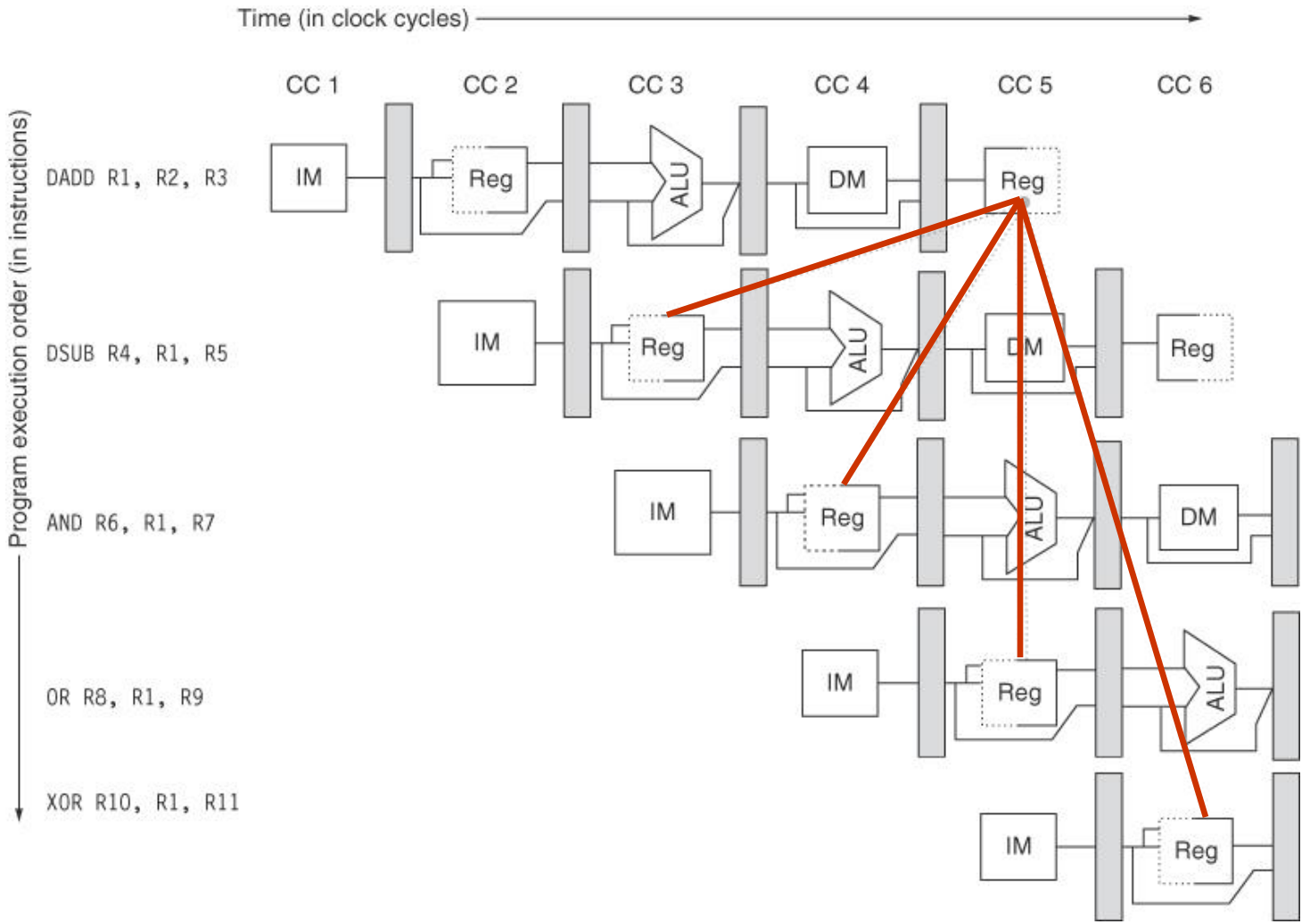


Figure C.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

IC-UNICAMP

Solução por forwarding

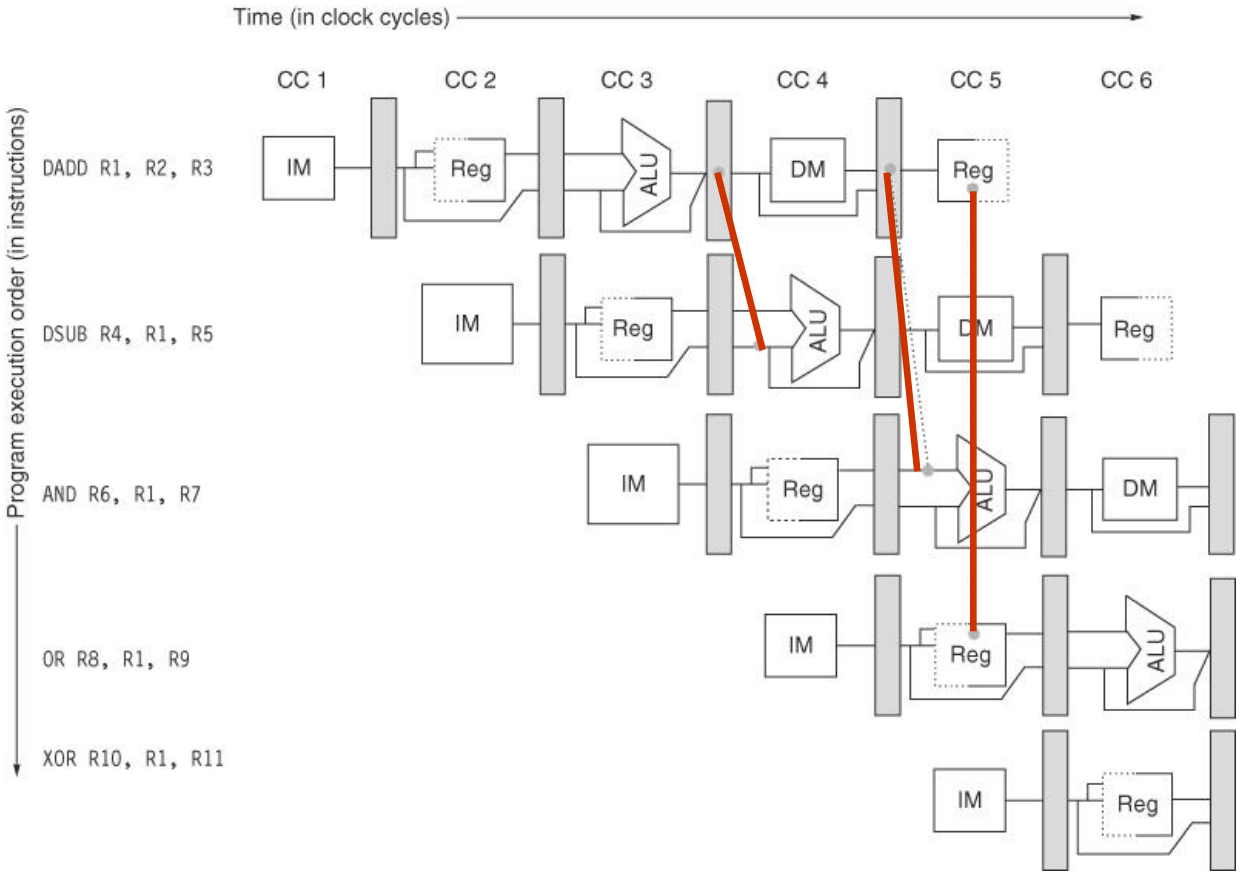


Figure C.7 A set of instructions that depends on the DADD result uses forwarding paths to avoid the data hazard. The inputs for the DSUB and AND instructions forward from the pipeline registers to the first ALU input. The OR receives its result by forwarding through the register file, which is easily 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 forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the AND instruction was AND R6,R1,R4.

IC-UNICAMP

Forwarding: instruções LD e SD

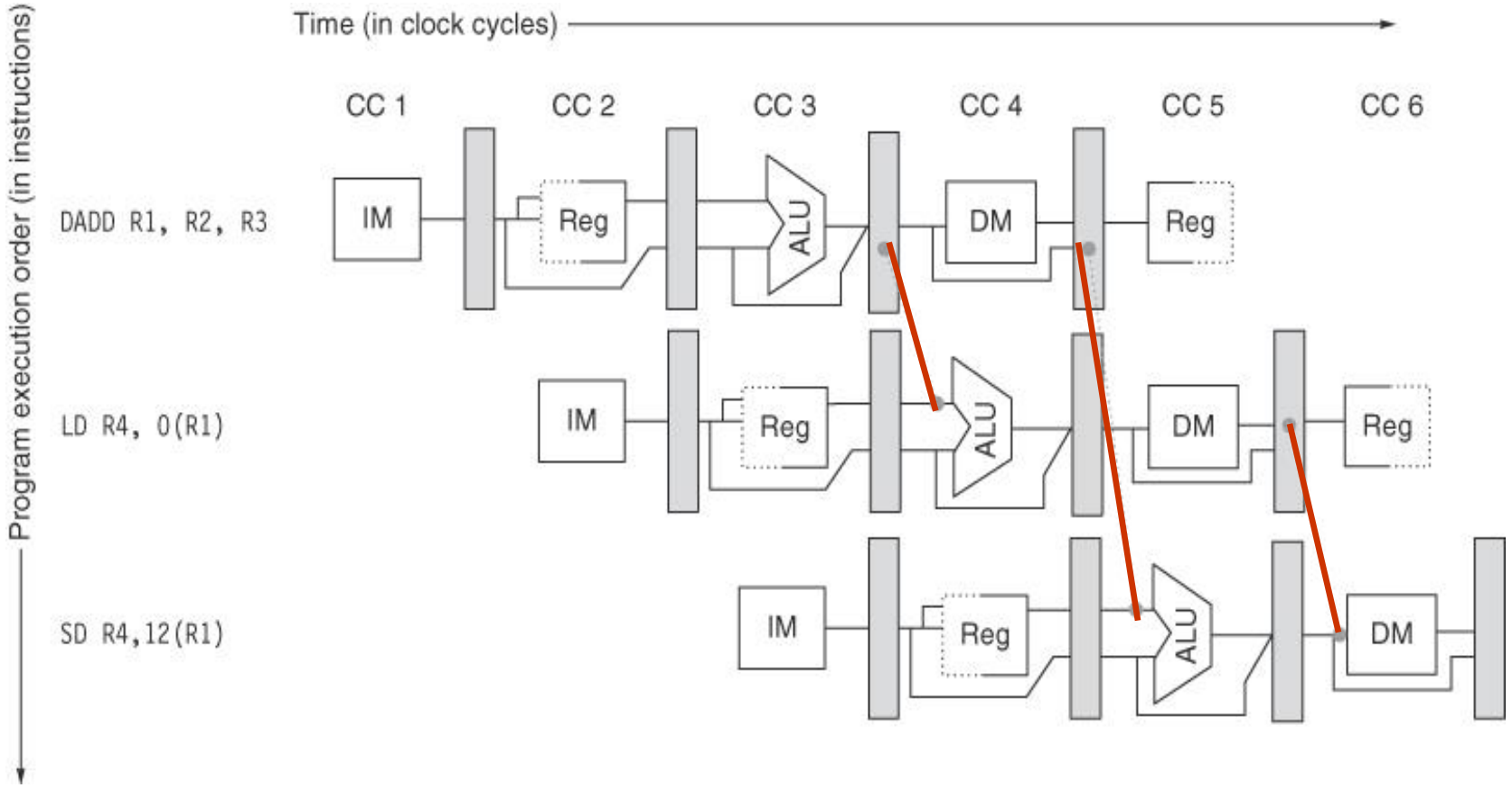


Figure C.8 Forwarding of operand required by stores during MEM. The result of the load is forwarded from the memory output to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than 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.

IC-UNICAMP

LD seguido de Arith: hazard

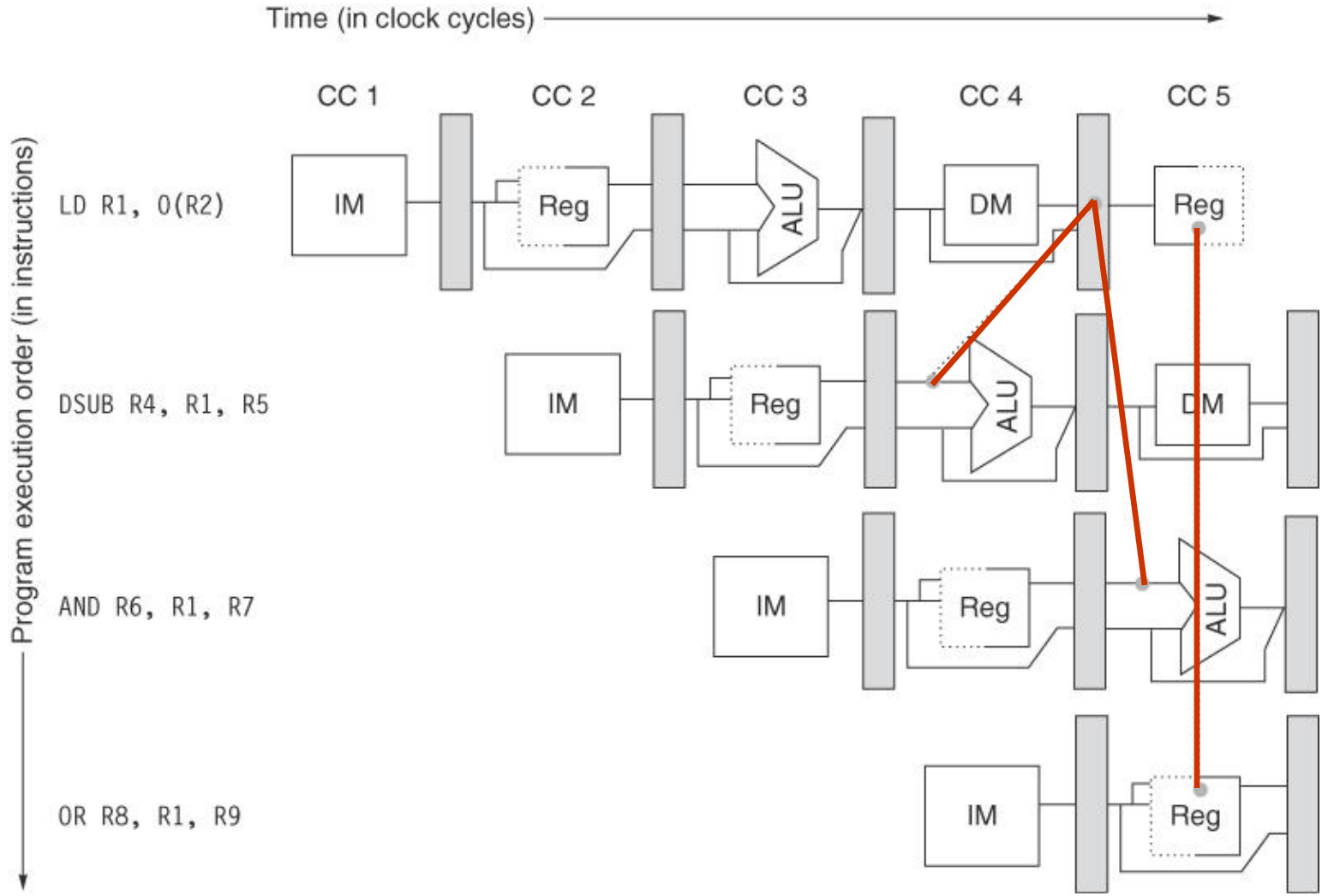


Figure C.9 The load instruction can bypass its results to the AND and OR instructions, but not to the DSUB, since that would mean forwarding the result in "negative time."

IC-UNICAMP

LD seguido de arith: 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

- Em desvio condicional, quando condição é calculada, já houve IF da próxima instrução
  - uma solução: refazer o IF

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.

IC-UNICAMP

Redução da penalidade do branch

- Neste Apêndice, quatro soluções simples no tempo de compilação
  - 1: freeze or flush the pipeline; não reduz penalidade (figura C-11)
  - 2: predict not taken
  - 3: predict taken
  - 4: delayed branch

IC-UNICAMP

Predict not taken / taken

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.

IC-UNICAMP

Delayed branch

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.

IC-UNICAMP

Delayed branch

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.

Delayed branch

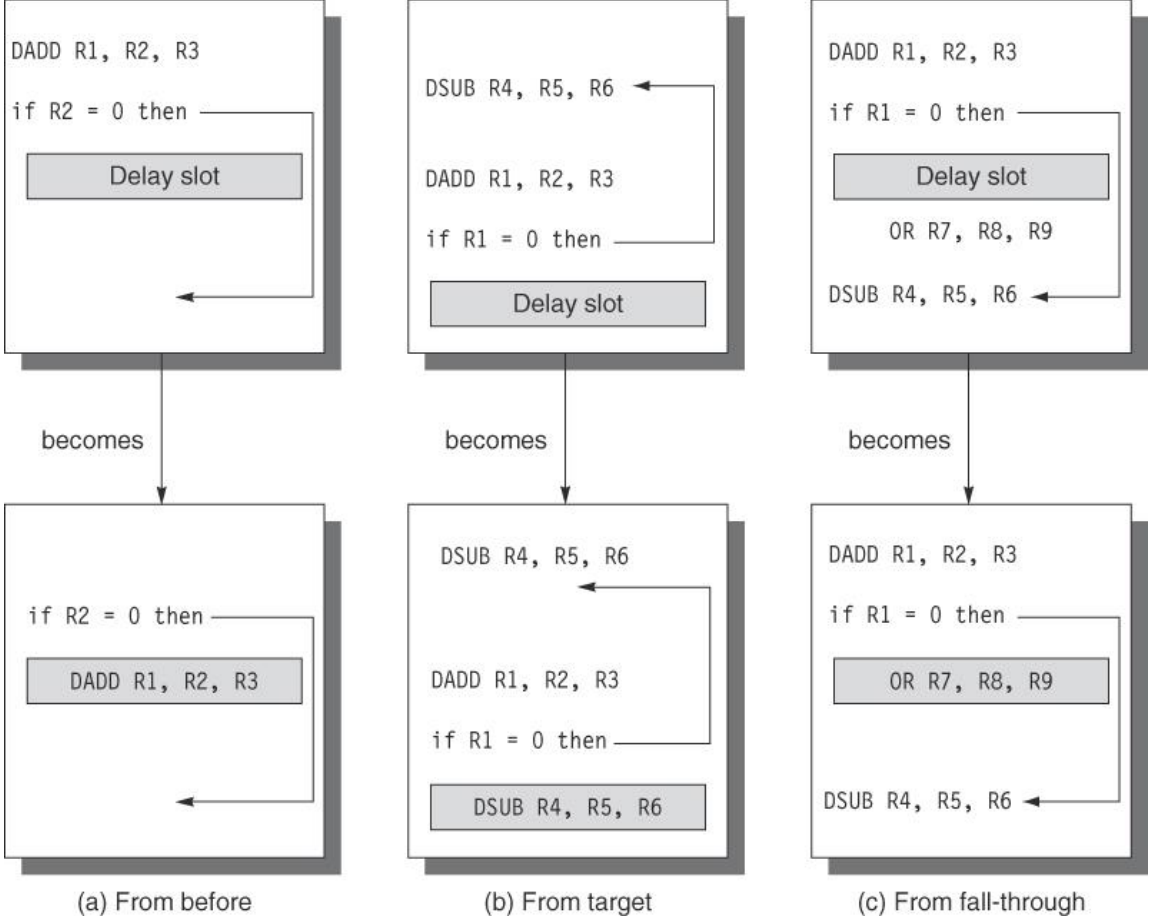


Figure C.14 Scheduling the branch delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) 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.

IC-UNICAMP

Desempenho das alternativas

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

$$\text{Stall cycles from branches} = \text{Branch frequency} \quad x \quad \text{Branch penalty}$$

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

IC-UNICAMP

Exmpl p C-25: desempenho das altern.

Example For a deeper pipeline, such as that in a MIPS R4000, it takes at least three pipeline stages before the branch-target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls on the registers in the conditional comparison. A three-stage delay leads to the branch penalties for the three simplest prediction schemes listed in Figure C.15.

Find the effective addition to the CPI arising from branches for this pipeline, assuming the following frequencies:

Unconditional branch	4%
Conditional branch, untaken	6%
Conditional branch, taken	10%

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predicted taken	2	3	2
Predicted untaken	2	0	3

Figure C.15 Branch penalties for the three simplest prediction schemes for a deeper pipeline.

IC-UNICAMP

Exmpl p C-25: desempenho das altern.

Branch scheme	Additions to the CPI from branch costs			
	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predicted taken	0.08	0.18	0.20	0.46
Predicted untaken	0.08	0.00	0.30	0.38

Figure C.16 CPI penalties for three branch-prediction schemes and a deeper pipeline.

Answer We find the CPIs by multiplying the relative frequency of unconditional, conditional untaken, and conditional taken branches by the respective penalties. The results are shown in Figure C.16.

The differences among the schemes are substantially increased with this longer delay. If the base CPI were 1 and branches were the only source of stalls, the ideal pipeline would be 1.56 times faster than a pipeline that used the stall-pipeline scheme. The predicted-untaken scheme would be 1.13 times better than the stall-pipeline scheme under the same assumptions.

Predição: redução dos custos

- Static branch prediction
  - menor custo: decisões tomadas no tempo de compilação com base em dados estatísticos (profile from earlier runs)
  - ver desempenho: fig C-17
    - desempenho ruim para benchmarks inteiros (além disso, frequencia de desvios é maior nestes benchmarks) →solução pouco usada
- Dynamic branch prediction
  - decisões tomadas dinamicamente em tempo de execução, com base no comportamento do programa

Erro na predição estática

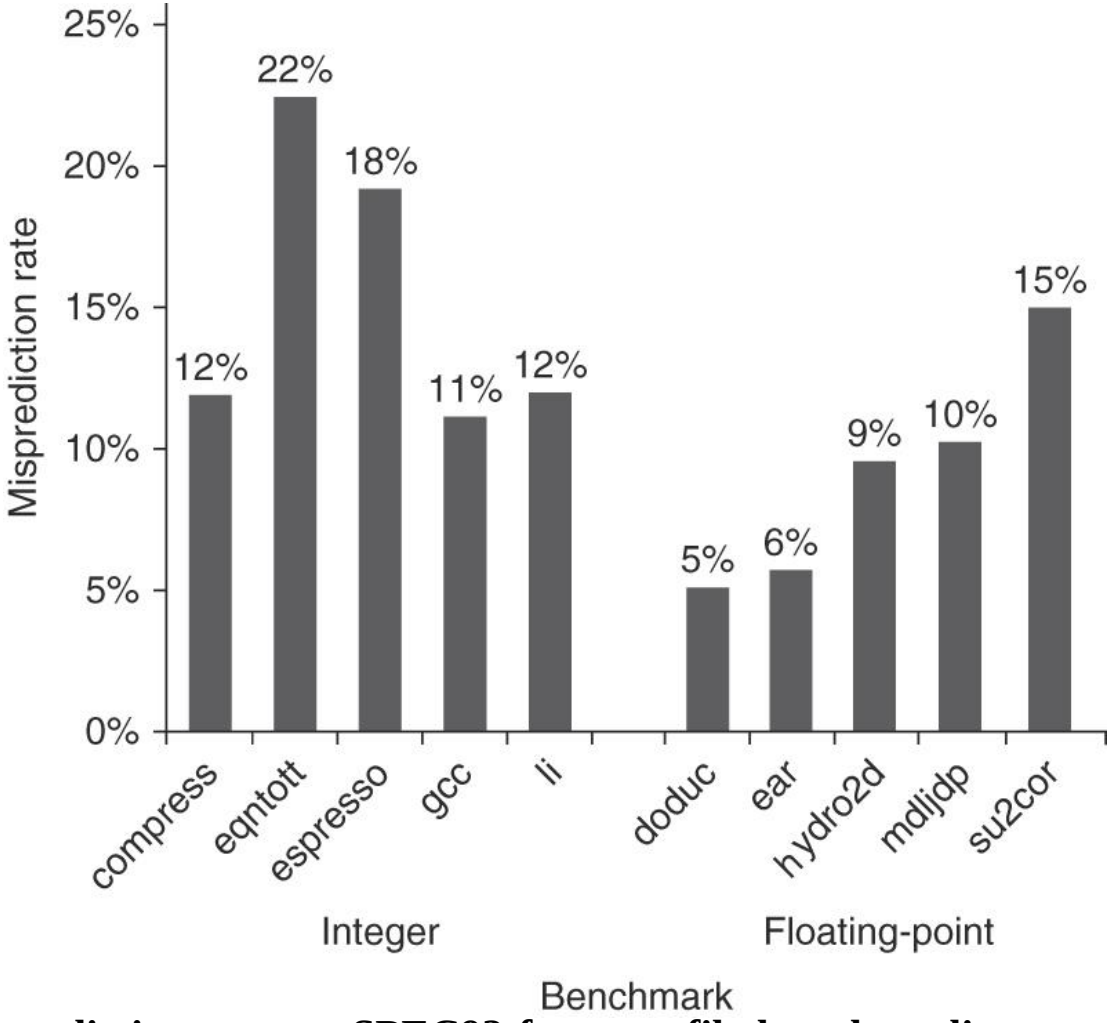


Figure C.17 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the floating-point programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which vary from 3% to 24%.

IC-UNICAMP

Dynamic Branch Prediction

- Branch prediction buffer or table
- Solução1
  - pequena memória indexada pelos bits inferiores do endereço da instrução de desvio contém bit indicando se o desvio foi tomado na última vez
  - pode ter registro de outro branch (mesmo índice)
- Deficiência: mesmo que um desvio seja quase sempre tomado→ dois erros de predição: ao sair do loop e ao entrar

IC-UNICAMP

Dynamic Branch Prediction com 2 bits

- Solução: 2 bits. Predição deve errar duas vezes para causar a inversão
- Implementação
  - “cache” indexada pela instrução ainda em IF ou 2 bits anexados a cada bloco na cache de instrução
- Se instrução é decodificada como branch e predição é “taken”, novo fetch usa endereço de desvio assim que é conhecido

IC-UNICAMP

FSM de 2 bits: predição dinâmica

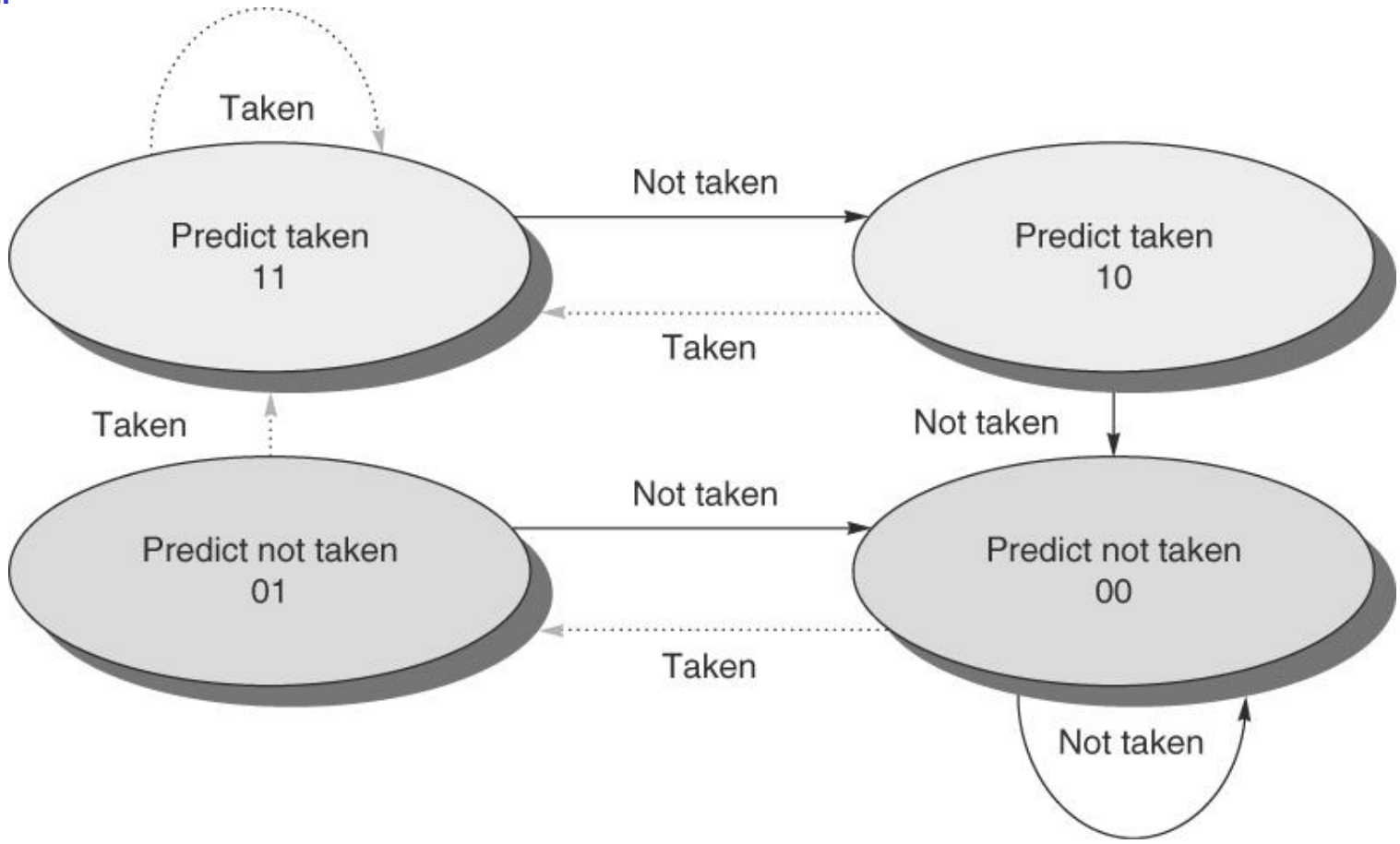
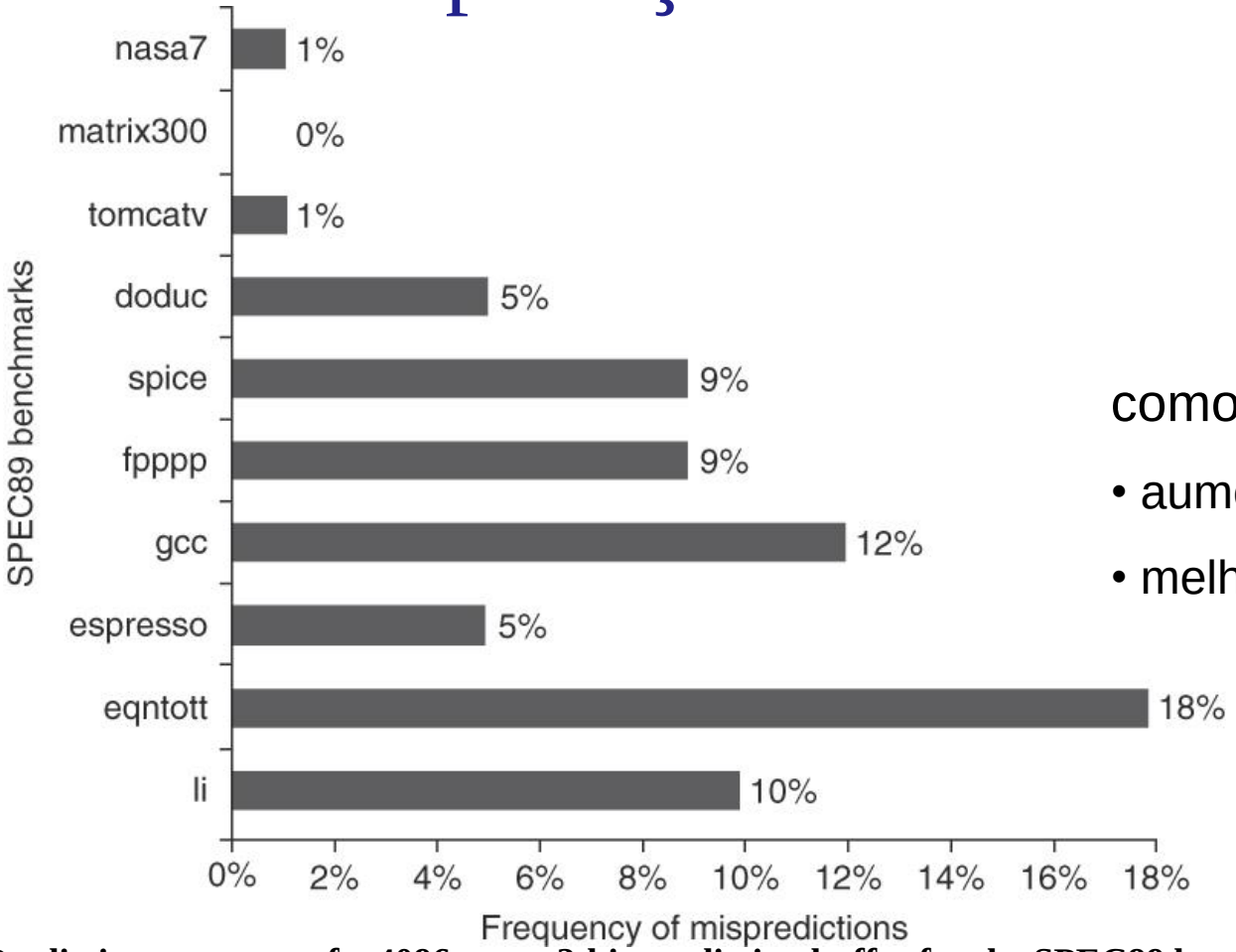


Figure C.18 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a 1-bit predictor. The 2 bits are used to encode the four states in the system. The 2-bit scheme is actually a specialization of a more general scheme that has an *n*-bit saturating counter for each entry in the prediction buffer. With an *n*-bit counter, the counter can take on values between 0 and 2n – 1: When the counter is greater than or equal to one-half of its maximum value (2n – 1), the branch is predicted as taken; otherwise, it is predicted as untaken. Studies of *n*-bit predictors have shown that the 2-bit predictors do almost as well, thus most systems rely on 2-bit branch predictors rather than the more general *n*-bit predictors.

IC-UNICAMP

Erro de predição: buffer de 4K



como melhorar?

- aumento do buffer?
- melhor estratégia?

Figure C.19 Prediction accuracy of a 4096-entry 2-bit prediction buffer for the SPEC89 benchmarks. The misprediction rate for the integer benchmarks (gcc, espresso, eqntott, and li) is substantially higher (average of 11%) than that for the floating-point programs (average of 4%). Omitting the floating-point kernels (nasa7, matrix300, and tomcatv) still yields a higher accuracy for the FP benchmarks than for the integer benchmarks. These data, as well as the rest of the data in this section, are taken from a branch-prediction study done using the IBM Power architecture and optimized code for that system. See Pan, So, and Rameh [1992]. Although these data are for an older version of a subset of the SPEC benchmarks, the newer benchmarks are larger and would show slightly worse behavior, especially for the integer benchmarks.

IC-UNICAMP

Efeito do tamanho do buffer

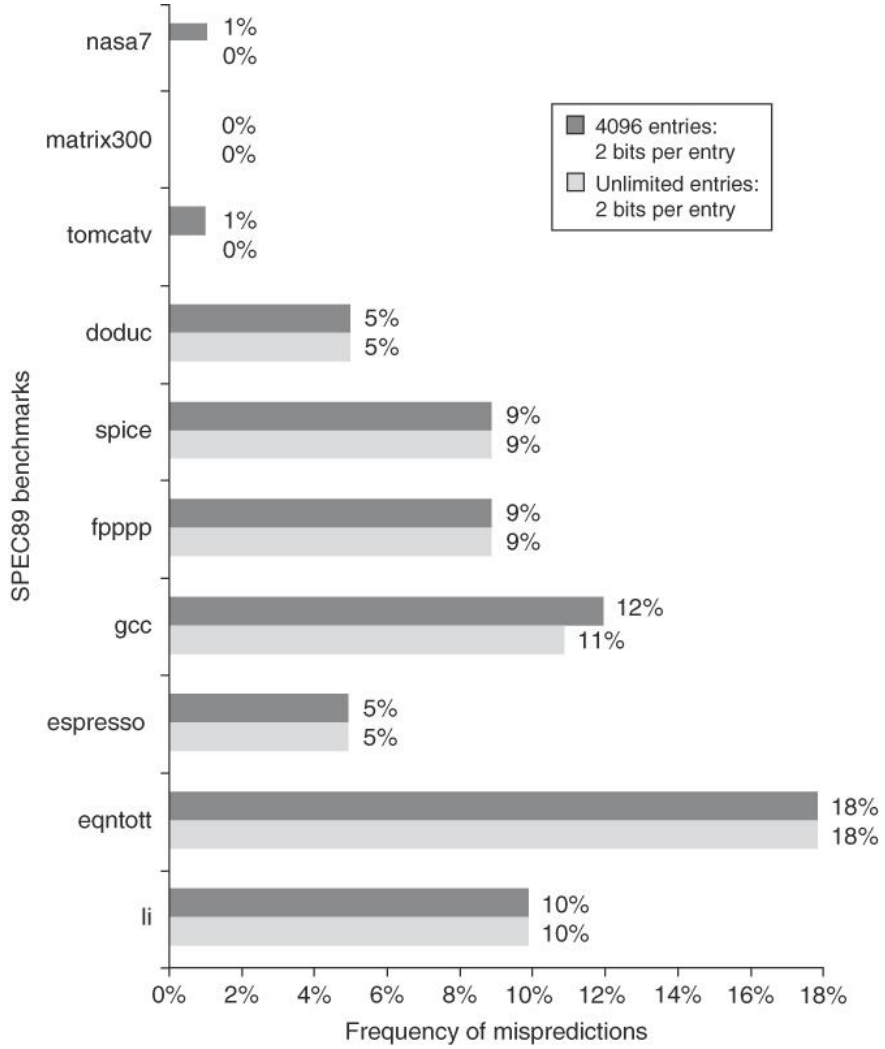


Figure C.20 Prediction accuracy of a 4096-entry 2-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks. Although these data are for an older version of a subset of the SPEC benchmarks, the results would be comparable for newer versions with perhaps as many as 8K entries needed to match an infinite 2-bit predictor.

IC-UNICAMP

C-3: Implementação do pipeline

- ver seção C-3 (pags C30 – C42)
  - material semelhante ao visto em HW/SW Interface

IC-UNICAMP

C-4: Complicações adicionais

- Exceções
- Dificuldades causadas pelo ISA

Pipelines e exceções

- Exceções: interrupções, traps etc
- Aguns tipos
  - IO request
  - System call from user program
  - Tracing instruction execution
  - Breakpoint (programmer requested interrupt)
  - Integer arithmetic overflow
  - FP arithmetic anomaly
  - Page fault
  - Misaligned memory access (if alingment is required)
  - Memory protection violation
  - Using na undefined or unimplemented instruction
  - Hardware malfunctions
  - Power failure

IC-UNICAMP

Nomes típicos

Exception event	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O device request	Input/output interruption	Device interrupt	Exception (L0 to L7 autovector)	Vectored interrupt
Invoking the operating system service from a user program	Supervisor call interruption	Exception (change mode supervisor trap)	Exception (unimplemented instruction)—on Macintosh	Interrupt (INT instruction)
Tracing instruction execution	Not applicable	Exception (trace fault)	Exception (trace)	Interrupt (single-step trap)
Breakpoint	Not applicable	Exception (breakpoint fault)	Exception (illegal instruction or breakpoint)	Interrupt (breakpoint trap)
Integer arithmetic overflow or underflow; FP trap	Program interruption (overflow or underflow exception)	Exception (integer overflow trap or floating underflow fault)	Exception (floating-point coprocessor errors)	Interrupt (overflow trap or math unit exception)
Page fault (not in main memory)	Not applicable (only in 370)	Exception (translation not valid fault)	Exception (memory-management unit errors)	Interrupt (page fault)
Misaligned memory accesses	Program interruption (specification exception)	Not applicable	Exception (address error)	Not applicable
Memory protection violations	Program interruption (protection exception)	Exception (access control violation fault)	Exception (bus error)	Interrupt (protection exception)
Using undefined instructions	Program interruption (operation exception)	Exception (opcode privileged/reserved fault)	Exception (illegal instruction or break-point/unimplemented instruction)	Interrupt (invalid opcode)
Hardware malfunctions	Machine-check interruption	Exception (machine-check abort)	Exception (bus error)	Not applicable
Power failure	Machine-check interruption	Urgent interrupt	Not applicable	Nonmaskable interrupt

IC

Categorias de exceção

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

Figure C.31 Five categories are used to define what actions are needed for the different exception types shown in Figure C.30. Exceptions that must allow resumption are marked as resume, although the software may often choose to terminate the program. Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement. We might expect that memory protection access violations would always result in termination; however, modern operating systems use memory protection to detect events such as the first attempt to use a page or the first write to a page. Thus, CPUs should be able to resume after such exceptions.

IC-UNICAMP

Stopping and restarting execution

- Maior dificuldade:
  - exceção ocorre no meio da instrução
  - execução precisa ser retomada após tratamento
- No pipeline:
  - forçar IF=trap no próximo IF
  - “desabilitar” todas instruções já no pipeline (desligar writes), inclusive a que gerou a exceção
  - a rotina de tratamento de exceção salva o PC
- Se delayed branch é usado
  - só medidas acima: não é possível retomar execução
  - necessário salvar vários PCs desde o desvio

IC-UNICAMP

Exceções no pipeline do MIPS

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

Figure C.32 Exceptions that may occur in the MIPS pipeline. Exceptions raised from instruction or data memory access account for six out of eight cases.

IC-UNICAMP

Impacto do ISA nas exceções

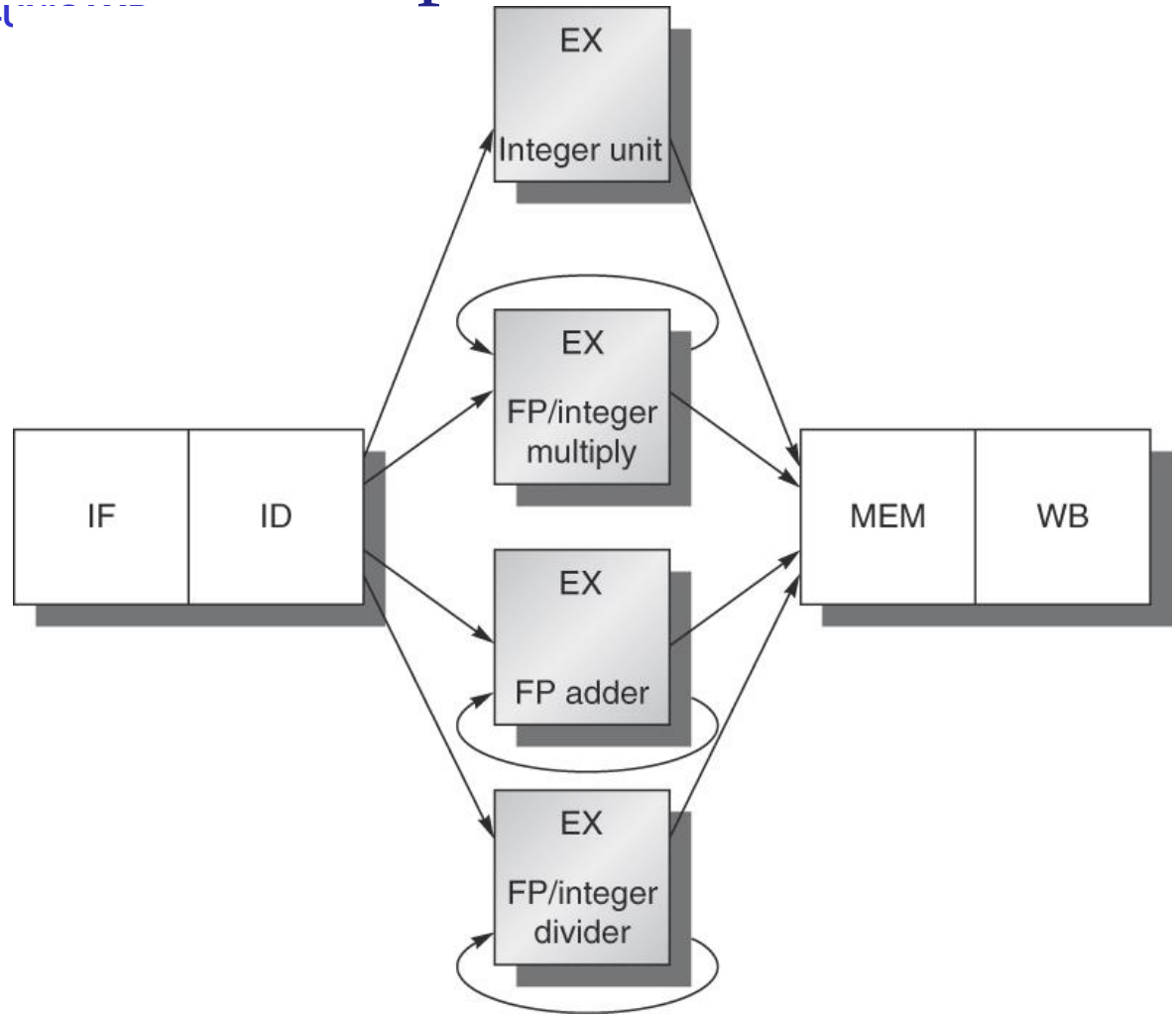
- Def: Precise exception – quando é possível retomar o fluxo de execução após o seu tratamento
- No MIPS, é simples garantir:
  - instruções só tem um resultado
  - estado do processador (memória e regs) só alterado no final (WB) → commit
- Há processadores em que é mais complicado:
  - Exemplo: AutoIncremento (VAX, IA-32, IBM) altera um registrador no meio do pipeline; antes de garantir que essa e instruções predecessoras "commit"
  - Exemplo: StringCopy (VAX, IBM) muda estado da memória no meio do pipeline
- Pipeline MIPS multiciclo (adiante) → problemas

IC-UNICAMP

C-5 Operações multi-ciclo no MIPS

- Estágio EX para FP exige múltiplos ciclos
  - ou aumento inaceitável do cycle time
- Modelo de concepção, pipeline int = fp, mas:
  - EX pode ser repetido inúmeras vezes
  - Múltiplas unidades funcionais de FP
- Decisão de projeto → unidades disponíveis
  1. Unidade principal de inteiros: loads/stores, ALU inteiro e branches
  2. Multiplicador(es) inteiro e FP
  3. Somador FP: add, sub, conversion
  4. Divisor(es) inteiro e FP

Pipeline do MIPS multiciclo



EX não pipelined: instruções posteriores → stall

Figure C.33 The MIPS pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The FP operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

IC-UNICAMP

Para EX → pipeline

- Latência da unidade: nº ciclos entre uma instrução que produz um resultado (input do pipe) e outra que usa o resultado (output do pipe)
- Initiation interval: nº ciclos entre dois inputs ao pipe

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Figure C.34 Latencies and initiation intervals for functional units.

Latências e intervalos

- Latência ≅ nº estágios do pipeline - 1
  - exceção stores: consome valor 1 ciclo depois
- Para obter menor clock cicle → maior nº de estágios no pipeline → maior latência

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Figure C.34 Latencies and initiation intervals for functional units.

Pipeline do MIPS: suporte a múltiplas operações FP pendentes

IC-UNICAMP

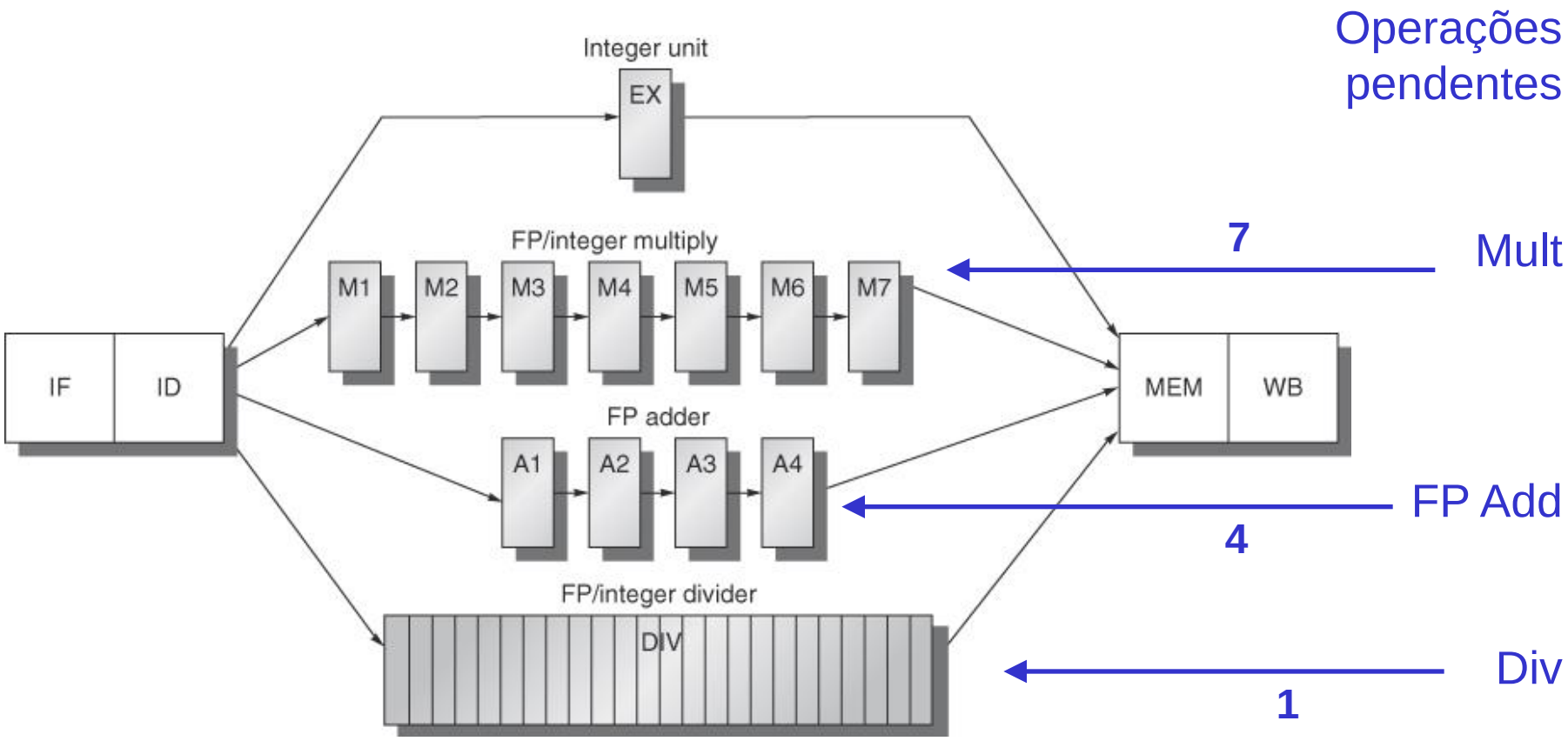
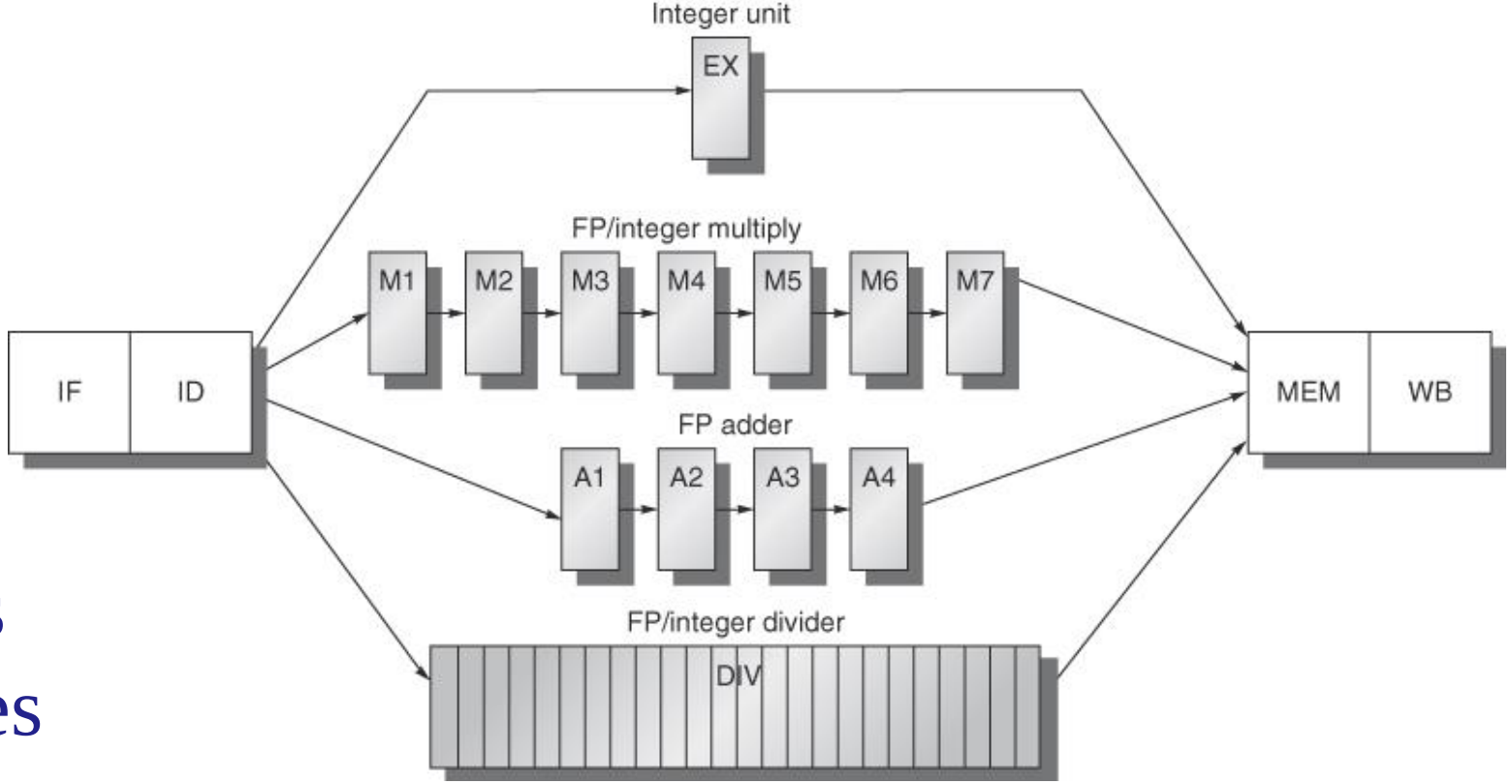


Figure C.35 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages. For example, the fourth instruction after an FP add can use the result of the FP add. For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.

Algumas instruções



MUL.D	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADD.D		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
L.D			IF	ID	*EX*	**MEM**	WB				
S.D				IF	ID	*EX*	*MEM*	WB			

Figure C.36 The pipeline timing of a set of independent FP operations. The stages in italics show where data are needed, while the stages in bold show where a result is available. The ".D" extension on the instruction mnemonic indicates double-precision (64-bit) floating-point operations. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

IC-UNICAMP

Harzards em um pipeline mais longo

- Hazard estrutural possível
- Nº de reg wr em um ciclo ≥ 1
- Instruções → WB fora de ordem → WAW possível
- WB fora de ordem → precise exception difícil
- Stalls devido RAW são mais frequentes

		Clock cycle 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

Figure C.37 A typical FP code sequence showing the stalls arising from RAW hazards. The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The S.D must be stalled an extra cycle so that its MEM does not conflict with the ADD.D. Extra hardware could easily handle this case.

IC-UNICAMP

Desempenho do pipeline MIPS FP

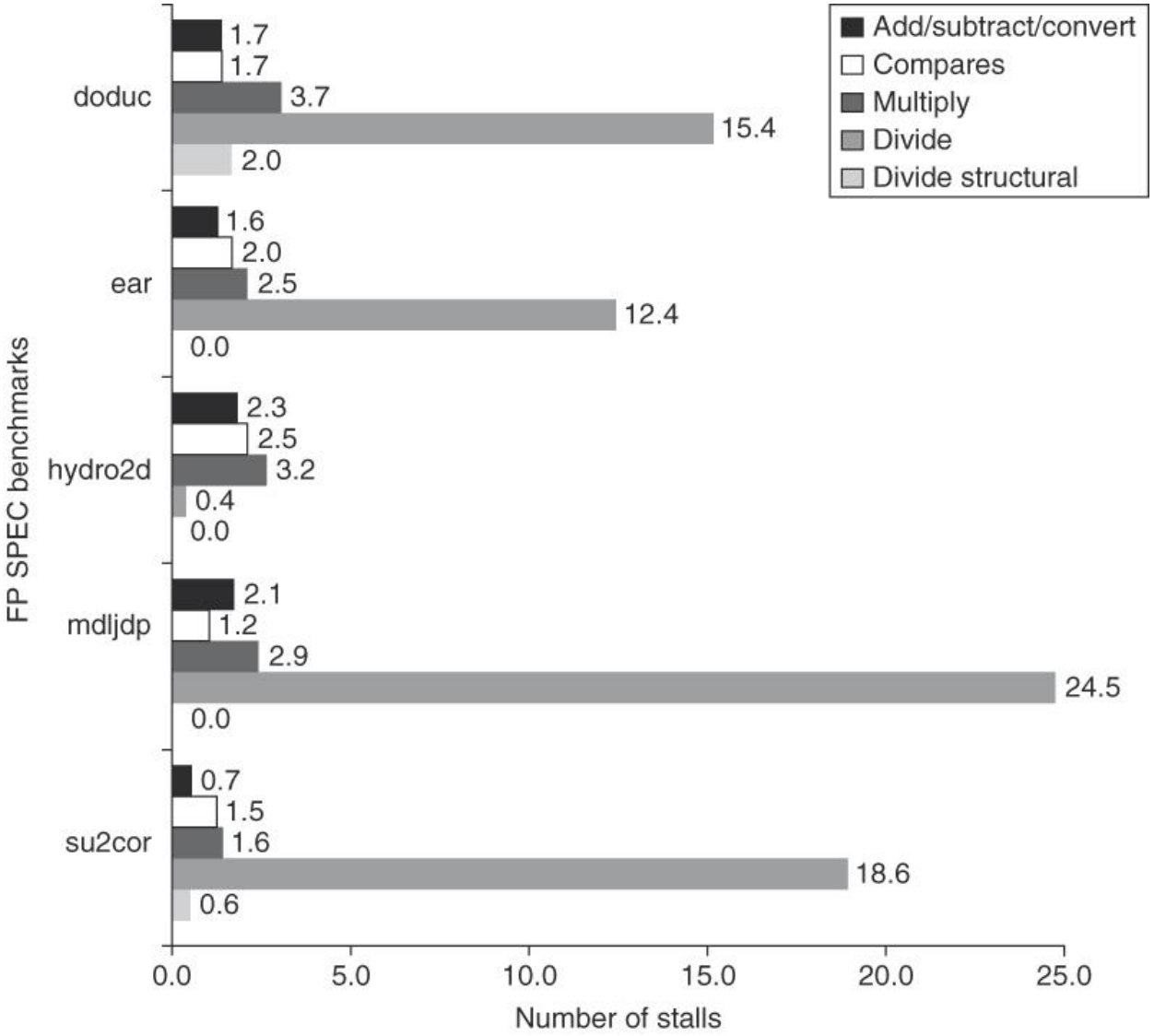


Figure C.39 Stalls per FP operation for each major type of FP operation for the SPEC89 FP benchmarks. Except for the divide structural hazards, these data do not depend on the frequency of an operation, only on its latency and the number of cycles before the result is used. The number of stalls from RAW hazards roughly tracks the latency of the FP unit. For example, the average number of stalls per FP add, subtract, or convert is 1.7 cycles, or 56% of the latency (3 cycles). Likewise, the average number of stalls for multiplies and divides are 2.8 and 14.2, respectively, or 46% and 59% of the corresponding latency. Structural hazards for divides are rare, since the divide frequency is low.

IC-UNICAMP

Desempenho do pipeline MIPS FP

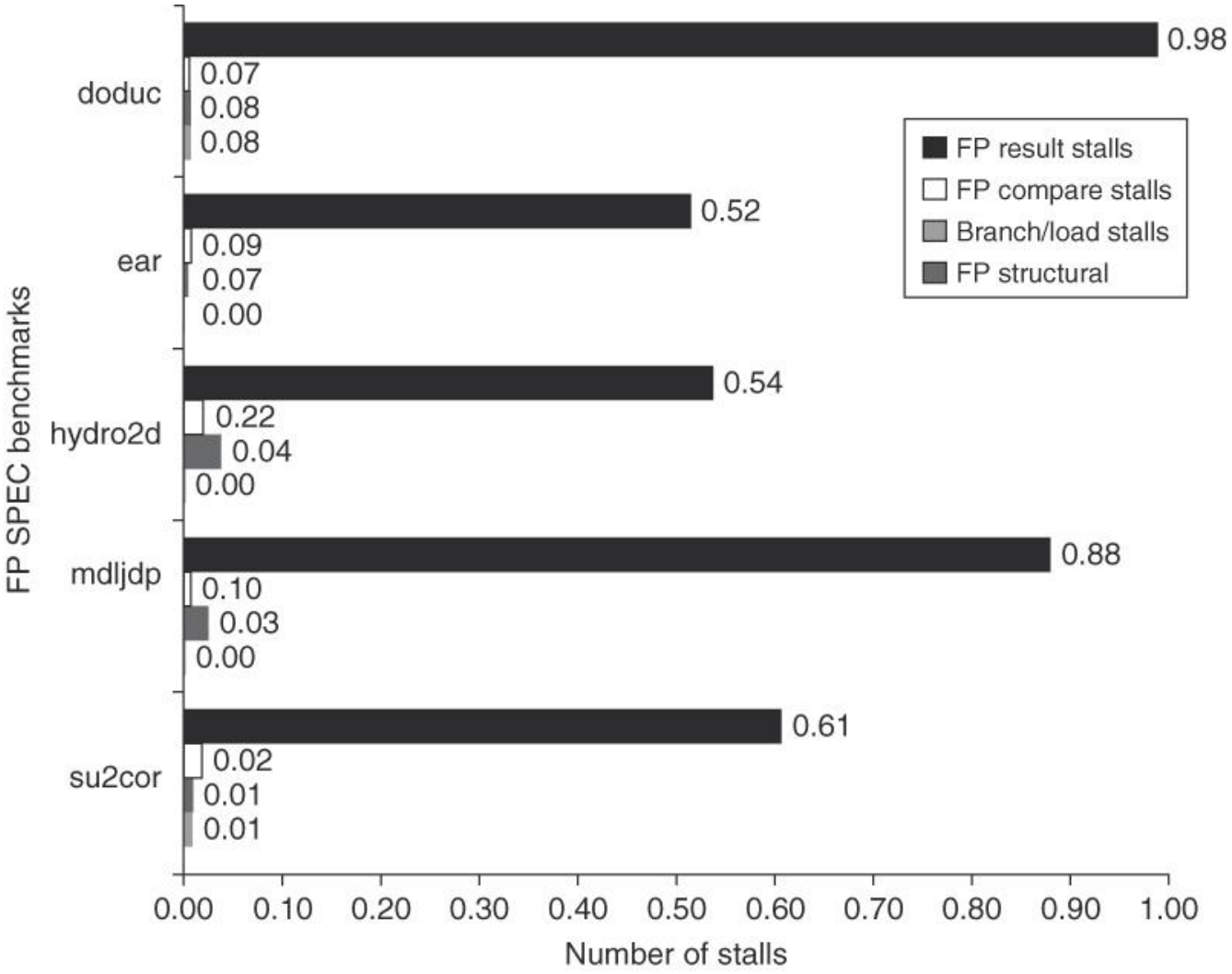


Figure C.40 The stalls occurring for the MIPS FP pipeline for five of the SPEC89 FP benchmarks. The total number of stalls per instruction ranges from 0.65 for su2cor to 1.21 for doduc, with an average of 0.87. FP result stalls dominate in all cases, with an average of 0.71 stalls per instruction, or 82% of the stalled cycles. Compares generate an average of 0.1 stalls per instruction and are the second largest source. The divide structural hazard is only significant for doduc.

IC-UNICAMP

C-7 Crosscutting issues
Dynamically scheduled pipeline

- Pipeline schedulling:
  - statically: pelo compilador
  - dynamically: pelo HW, durante execução
    - scoreboarding (CDC 6600), introdução ao Alg. Tomasulo (cap3)
- Neste apêndice
  - In-order issue
  - Out-of-order execution (and completion)
- ID quebrado em dois estágios
  - Issue: decode, check for structural hazards
  - Read operands: wait for hazards, read operands

IC-UNICAMP

Dynamically schedule w/ scoreboarding

- In-order issue
- Problemas podem aparecer → WAR
  - `DIV.D  F0,  F2,  F4`
  - `ADD.D  F10,  F0,  F8`
  - `SUB.D  F8,  F8,  F14`
- Com out-of-order execution(commit), SUB.D pode ser executado antes de ADD.D → erro. Anti-dependência
- Scoreboarding:
  - stalls a segunda instrução envolvida na anti-dependência
  - objetivo: manter taxa de execução em 1 instrução / ciclo (se não houver hazard estrutural), inserindo (issue) nova instrução se ocorrer hazard

Scoreboarding

- Requesito: múltiplas unidades funcionais e/or pipelining
- CDC 6600: 16 unidades = 4 FPU + 5 memória +7 inteiros
- No MIPS foco em FP: 2 multiplicadores, 1 somador, 1 divisor, 1 unidade inteira (para ALU, referências a memória e branch)
  - scoreboard determina quando
    - uma instrução pode ler operandos e iniciar execução
    - uma instruçao pode escrever resultados no registrador
- Hazard detection and resolution centralizado no scoreboarding

Scoreboarding: MIPS

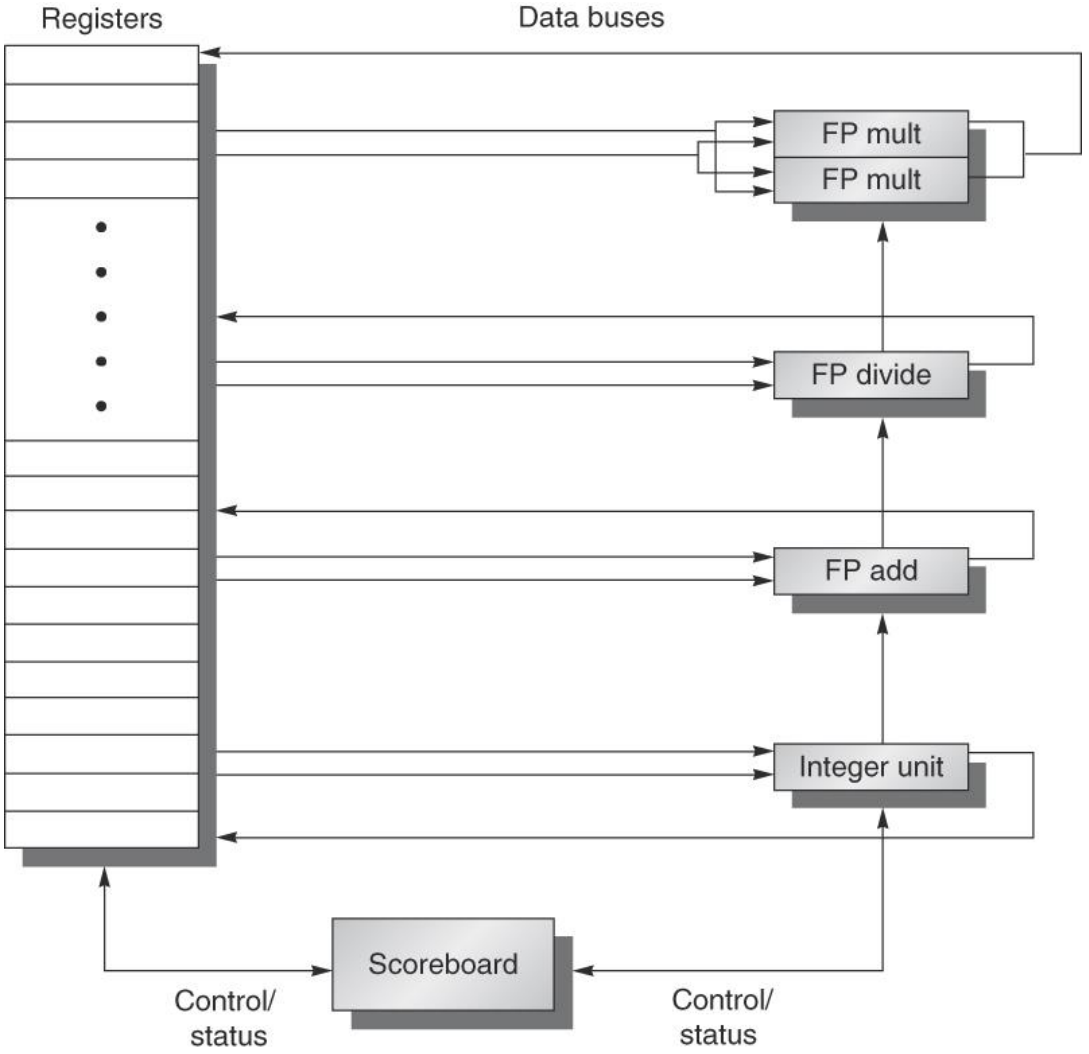


Figure C.54 The basic structure of a MIPS processor with a scoreboard. The scoreboard's function is to control instruction execution (vertical control lines). All of the data flow between the register file and the functional units over the buses (the horizontal lines, called *trunks* in the CDC 6600). There are two FP multipliers, an FP divider, an FP adder, and an integer unit. One set of buses (two inputs and one output) serves a group of functional units. The details of the scoreboard are shown in Figures C.55 to C.58.

IC-UNICAMP

Scoreboard: 4 passos (substitui estágios ID, EX e WB)

- 1- Issue (substitui parcialmente ID)
  - se existe unidade desocupara → emite instrução e atualiza sua estrutura de dados interna
    - garante que não há outra unidade tentando escrever no mesmo registrador → evita WAW
    - se necessário (hazard estrutural, por ex.) → stall
- 2- Read operands (substitui ID juntamente com 1)
  - verifica se há operandos disponíveis (nenhuma instrução anterior vai escrever nele). Resolve RAW, permitindo execução fora da ordem

IC-UNICAMP

Scoreboard: 4 passos (cont)

- 3- Execução (substitui EX)
  - Unidade executa e avisa o scoreboard no final
- 4- Write result (WB)
  - verifica WAR e se necessário, stall. No exemplo anterior, stalls SUB.D no estágio de escrita até que ADD.D leia seus operandos

- Observar que scoreboard não faz forwarding
- Recurso importante no scoreboard: barramentos de comunicação interna
  - falta deles pode causar hazard estrutural causado pelo próprio circuito de scoreboarding

IC-UNICAMP

Componentes do scoreboard: estrutura interna

Figure C.55 Components of the scoreboard. Each instruction that has issued or is pending issue has an entry in the instruction status table. There is one entry in the functional unit status table for each functional unit. Once an

		Instruction status			
Instruction		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	
MUL.D	F0,F2,F4	√			
SUB.D	F8,F6,F2	√			
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2				

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Add	Divide			

Campos de status: Unidades Funcionais

Functional unit status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

- Busy: ocupada ou não
- Op: operação
- Fi: registrador destino; Fj e Fk: regs operandos
- Qj, Qk: Unidades que produzem operandos para esta unidade
- Rj, Rk: flags indicando se os operandos Fj e Fk estão prontos ou não

IC-UNICAMP

Hazards no código

I1	L.D	F6,34(R2)d
I2	L.D	F2,45(R3)
I3	MUL.D	F0,F2,F4
I4	SUB.D	F8,F6,F2
I5	DIV.D	F10,F0,F6
I6	ADD.D	F6,F8,F2

- RAW:
  - de I2 para I3, I4, I6
  - de I3 para I5
  - de I4 para I6
- WAR:
  - de I4 para I6
  - de I5 para I6
- Estrutural na unidade funcional Add
  - I5 e I6

IC-UNICA

Fig C56

Instruction		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	√
MUL.D	F0,F2,F4	√	√	√	
SUB.D	F8,F6,F2	√	√	√	√
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2	√	√	√	

Functional unit status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult 1			Add		Divide			

Figure C.56 Scoreboard tables just before the MUL.D goes to write result. The DIV.D has not yet read either of its operands, since it has a dependence on the result of the multiply. The ADD.D has read its operands and is in execution, although it was forced to wait until the SUB.D finished to get the functional unit. ADD.D cannot proceed to write result because of the WAR hazard on F6, which is used by the DIV.D. The Q fields are only relevant when a functional unit is waiting for another unit.

IC-UNICA

Fig C57

Instruction		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)d	√	√	√	√
L.D	F2,45(R3)	√	√	√	√
MUL.D	F0,F2,F4	√	√	√	√
SUB.D	F8,F6,F2	√	√	√	√
DIV.D	F10,F0,F6	√	√	√	
ADD.D	F6,F8,F2	√	√	√	√

Functional unit status

Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6			No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult 1			Add		Divide			

Figure C.57 Scoreboard tables just before the DIV.D goes to write result. ADD.D was able to complete as soon as DIV.D passed through read operands and got a copy of F6. Only the DIV.D remains to finish.

IC-UNICAMP

Verificações em um scoreboard

Instruction status	Wait until	Bookkeeping
Issue	Not busy [FU] and not result [D]	Busy[FU]←yes; Op[FU]←op; Fi[FU]←D; Fj[FU]←S1; Fk[FU]←S2; Qj←Result[S1]; Qk← Result[S2]; Rj← not Qj; Rk← not Qk; Result[D]←FU;
Read operands	Rj and Rk	Rj← No; Rk← No; Qj←0; Qk←0
Execution complete	Functional unit done	
Write result	∀*f*((Fj[*f*] ≠ Fi[FU] or Rj[*f*] = No) & (Fk[*f*] ≠ Fi[FU] or Rk[*f*] = No))	∀*f*(if Qj[*f*]=FU then Rj[*f*]←Yes); ∀*f*(if Qk[*f*]=FU then Rk[*f*]←Yes); Result[Fi[FU]]← 0; Busy[FU]← No

Figure C.58 Required checks and bookkeeping actions for each step in instruction execution. FU stands for the functional unit used by the instruction, D is the destination register name, S1 and S2 are the source register names, and op is the operation to be done. To access the scoreboard entry named Fj for functional unit FU we use the notation Fj[FU]. Result[D] is the name of the functional unit that will write register D. The test on the write result case prevents the write when there is a WAR hazard, which exists if another instruction has this instruction's destination (Fi[FU]) as a source (Fj[*f*] or Fk[*f*]) and if some other instruction has written the register (Rj = Yes or Rk = Yes). The variable *f* is used for any functional unit.