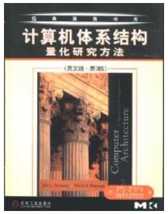
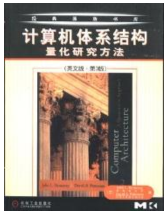


Lecture 4 for pipelining

- What makes pipelining hard to implement ?
  (execeptions)

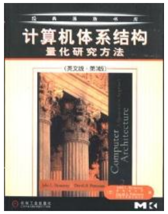
- Extending the MIPS Pipeline to Handle Multicycle Operations

- MIPS R4000 integer pipeline



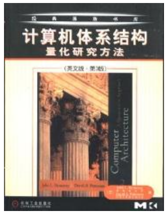
What Makes Pipelines Hard to Implement?

- Detecting and resolving hazards
  - OK. We have solved this problem.
- Exceptions and Interrupts
- Instruction Set complications
  - Very complex multicycle instructions are difficult to pipeline
  - Example:
  - stringMov from 0x1234, to 0x4000, 0x1000 bytes



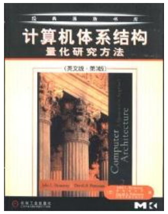
Exception causes

- I/O device requests
- User OS service requests
- Breakpoints
- Integer arithmetic overflow/underflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory protection violations
- Hardware malfunctions
- Undefined instructions



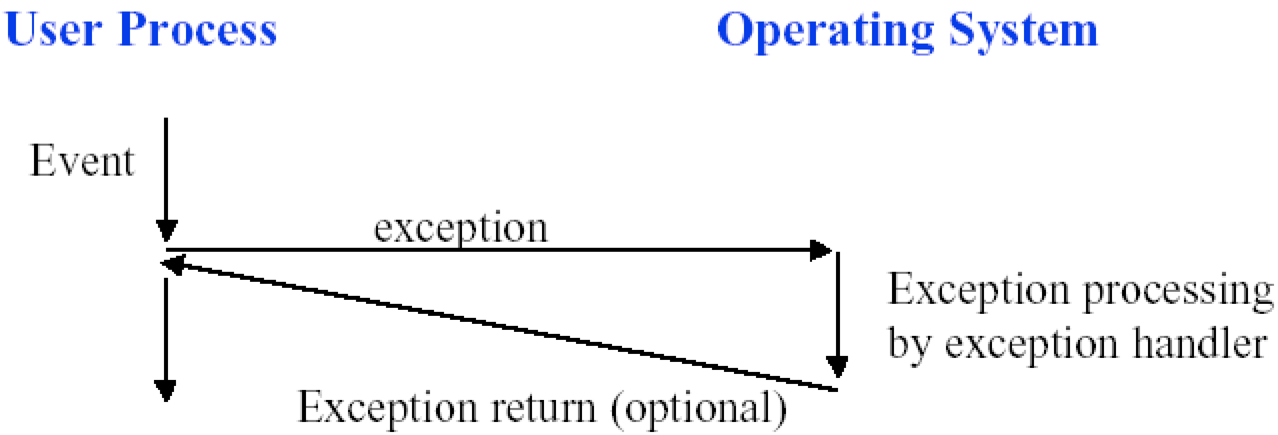
Exceptions and Interrupts

- Exceptions are *exceptional* events that *disrupt* the normal flow of a program
- Terminology varies between different machines
- Examples of Interrupts
  - User hitting the keyboard
  - Disk drive asking for attention
  - Arrival of a network packet
- Examples of Exceptions
  - Divide by zero
  - Overflow
  - Page fault



Exception Flow

- When an exception (or interrupt) occurs, control is transferred to the OS



User Process

Operating System

Event

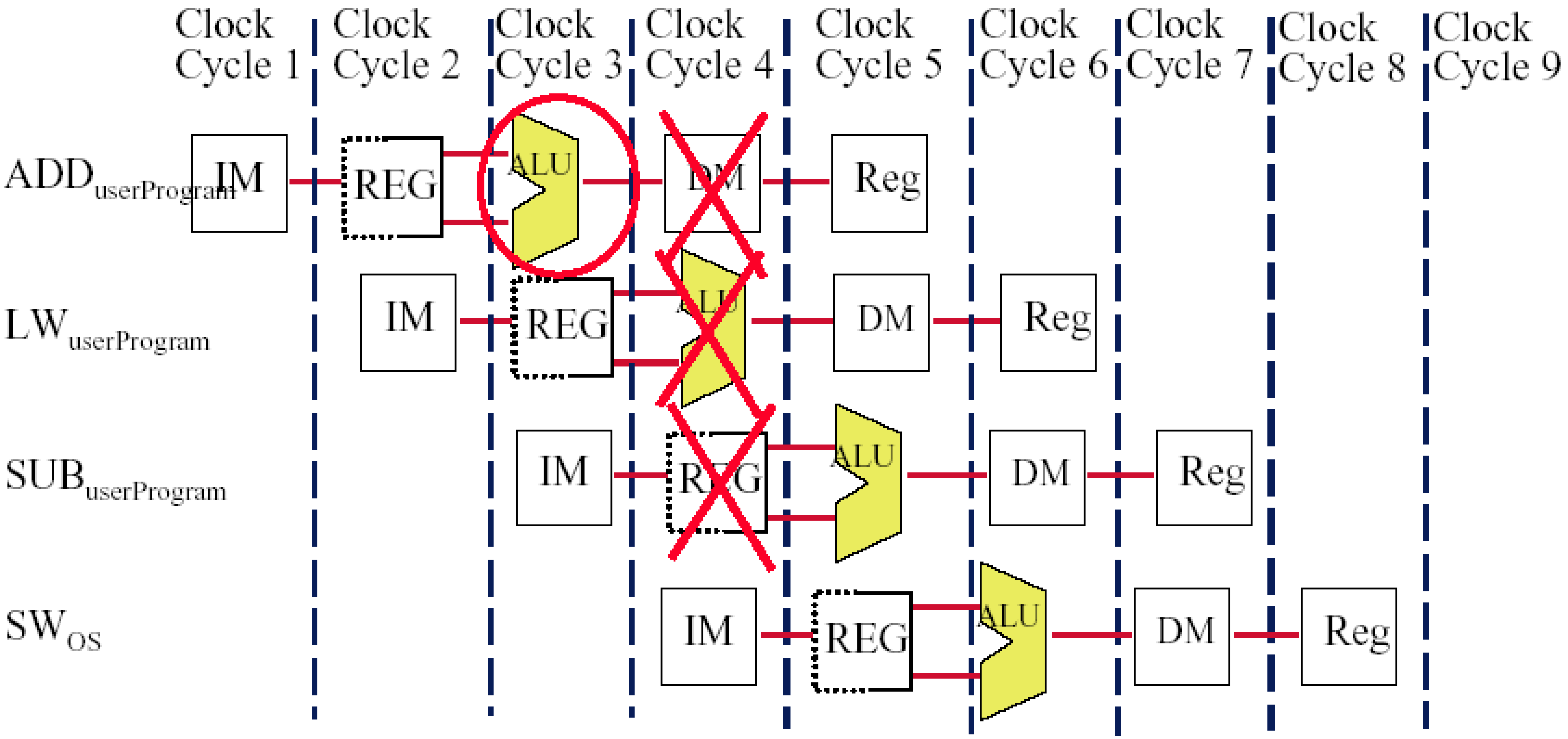
exception

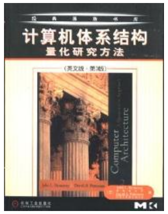
Exception processing by exception handler

Exception return (optional)

Flow of Instructions During Exception

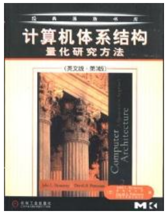
- ❑ Example: Add instruction overflows in clock cycle 3





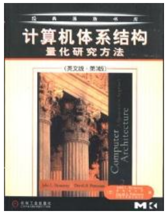
Characterizing Exceptions and Interrupts

- Synchronous vs asynchronous events
  - Synchronous events occur at the same place every time a program executes
  - Asynchronous events are caused by external devices such as a keyboard, disk drive or mouse
  - Asynchronous events can usually be handled after the completion of the current instruction, making them easier to handle
- User requested vs. coerced
  - If a user asks for it, it is user requested
  - Coerced are hardware events not under user control
  - Coerced exceptions are harder to implement since they are not predictable.



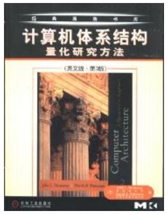
Characterizing Exceptions and Interrupts (continued)

- User maskable vs nonmaskable
  - Can a user disable an exception from being detected?
- Within vs. between instructions
  - Does the event prevent the current instruction from completing?
  - Exceptions occurring within instructions are usually synchronous, since the instruction triggers the exception.
  - Within is more difficult to implement than between since the former must be restarted.
- Resume vs. terminate
  - Can the event be handled (corrected) or must the program be terminated?
  - Restarting is harder (obviously), and is the more common case.



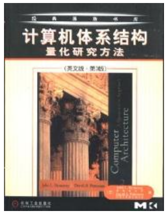
Types of Exceptions

Exception	Syn/Asynch	User request?	User maskable?	Within?	Resume?
I/O device	asynch	coerced	nonmaskable	between	resume
invoke OS	synch	user req.	nonmaskable	between	resume
tracing instr. execution	synch	user req.	user maskable	between	resume
breakpoint	synch	user req.	user maskable	between	resume
int overflow	synch	coerced	user maskable	within	resume
fp overflow	synch	coerced	user maskable	within	resume
page fault	synch	coerced	nonmaskable	within	resume
misaligned mem access	synch	coerced	user maskable	within	resume
mem-prot violation	synch	coerced	nonmaskable	within	term.
undef. instr	synch	coerced	nonmaskable	within	term.
hardware malf.	asynch	coerced	nonmaskable	within	term.



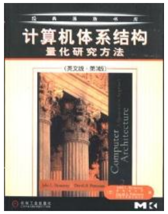
How to do when exception occurs ?

- Often, exception occurs while many instructions are in flight
  - **Ex**: a page fault on a load instruction will occur in stage 4 of the MIPS pipe
  - Pipeline must be safely shutdown when exception occurs and then **restarted at the offending instruction**



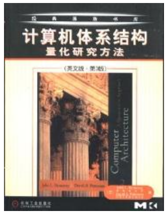
Stopping and Restarting Execution

- Force a trap instruction into the pipeline
- Until the trap is taken, turn off all writes for the faulting instruction and any instruction that issued after the faulting instruction
  - This prevents instructions from changing the state of the machine
- When the trap is taken, invoking the OS, the OS saves the PC of the offending instruction
- The OS fixes the exception (if possible) and then restarts the machine
  - Restarting usually means setting PC <-- offending instruction address
  - Replays instruction(s)



Precise Exceptions

- If the pipeline can be stopped so that the instructions issued before the faulting instruction complete and those after it can be restarted, then the pipeline is said to implement precise exceptions
  - All instructions before the faulting instruction complete
  - And instructions following the faulting instruction, including the faulting instruction, do not change the state of the machine.
- Under this model, restarting is easy:
  - Simply re-execute the original faulting instruction.
  - Or, if it is not a resumable instruction, start with the next instruction.

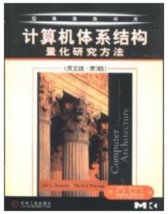


Imprecise Exceptions

- Difficult to do when some instructions take multiple cycles to complete
  - Some instructions may complete before an exception is detected
  - Example

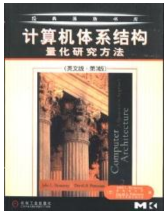
    Multiply r1, r2, r3 ; multiply takes 10 cycles

    Add r10,r11,r12 ; takes 5 cycles
  - Add will complete before multiply is done. If multiply overflows, then
  - an exception will be raised AFTER the add has updated the value in R10.
  - This is an imprecise exception.



Precise vs. Imprecise Exceptions

- Some machines implement both modes: imprecise and precise exceptions
  - Special software instructions to guarantee precise exceptions
  - Machine runs slower when one needs precise exceptions
  - In general, integer exceptions are precise, while FP exceptions may not be.



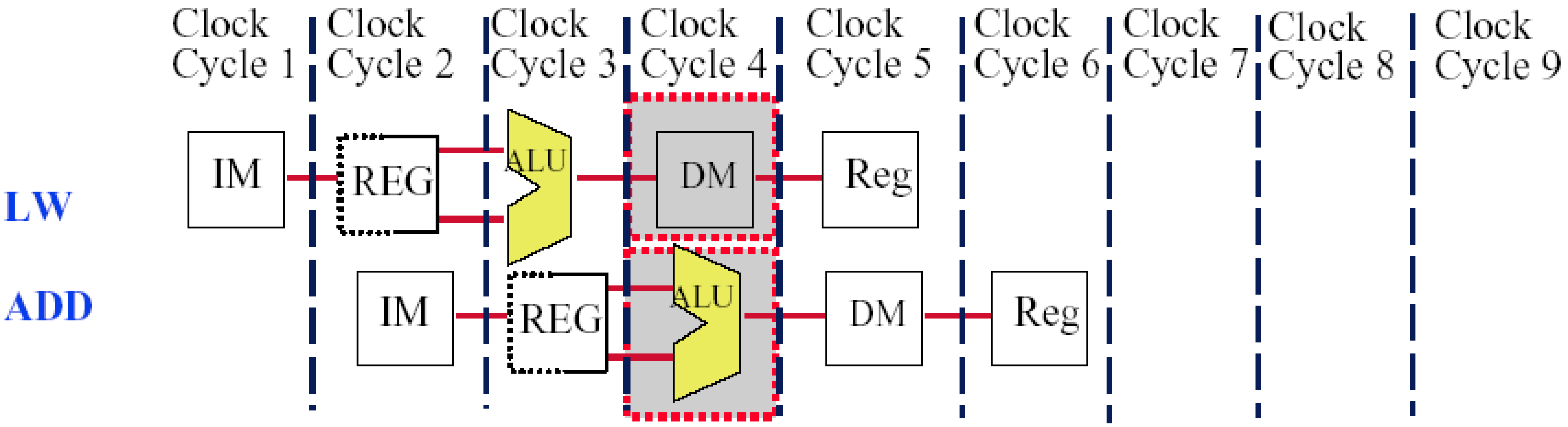
Exceptions and the MIPS Architecture

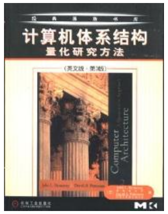
- Which stage can exceptions occur in?

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

Multiple Exceptions in one clock cycle

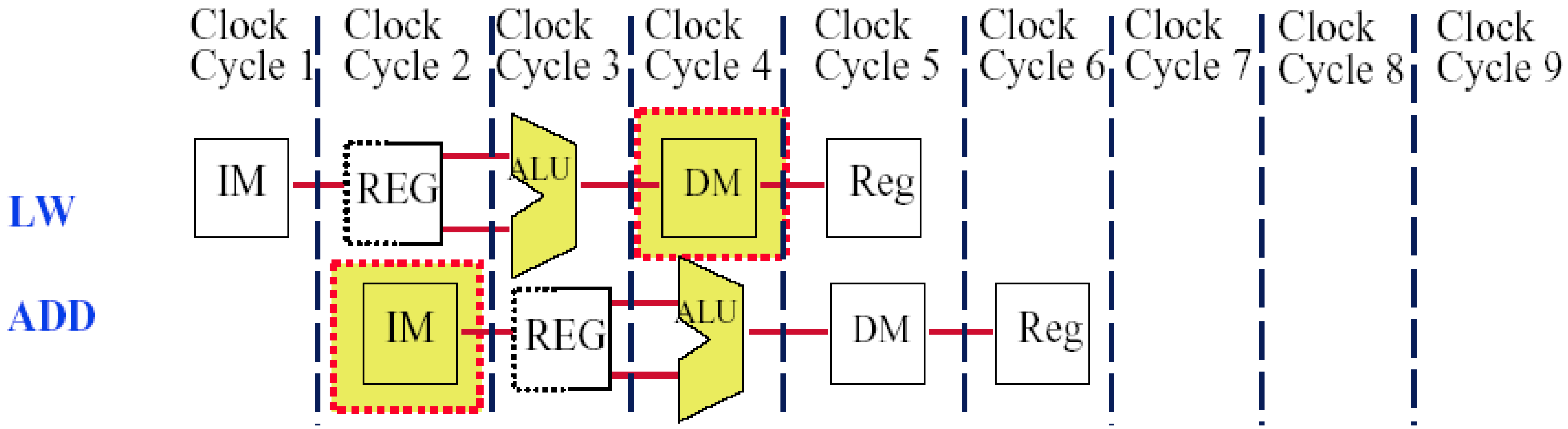
- In Clock Cycle 4, LW can have a data page fault while the ADD has an arithmetic exception
- Handled by servicing the page fault and then restarting the LW instruction
- The ADD's arithmetic exception will occur again because the ADD instruction is restarted after the exception is handled

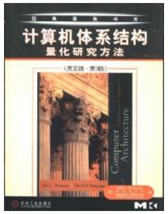




Multiple Exceptions out-of-order

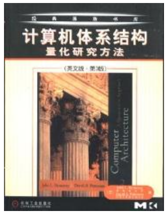
- ADD causes an exception in the instruction fetch stage while LW causes an exception in the memory access stage
- If we implement precise exceptions, LW exception must be handled first
- This is done by having hardware post exceptions by order of instruction





Exception ordering

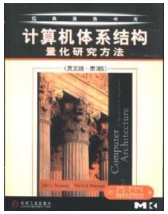
- When the instruction is about to exit the pipeline (MEM/WB), any pending exceptions for the instruction are examined.
- If an instruction generates multiple exceptions, the exception occurring in the earliest stage takes precedence.
- This is done by keeping an exception status vector for each instruction:
  - If an exception is posted, it is added to the vector and all writes that affect system state are disabled.



About Exceptions

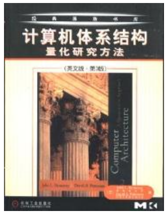
- One of the single messiest parts of designing a modern CPU
  - It isn't pretty, it's easy to get wrong
  - It's often not too elegant
  - It usually takes huge wads of special logic

- Further complicated by modern CPU mechanisms
  - Deep pipes
  - Superscalar --lots of instructions in flight in parallel
  - Out-of-order execution
    - √ time order of exceptions ≠ program order of the instructions on which the exceptions happened
  - Maintaining illusion of "sequential instruction execution" gets really complicated.



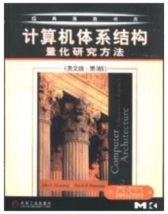
What Makes Pipelines Hard to Implement?

- Detecting and resolving hazards
  - OK. We have solved this problem.
- Exceptions and Interrupts
- Instruction Set complications
  - Very complex multicycle instructions are difficult to pipeline
  - Example:
  - stringMov from 0x1234, to 0x4000, 0x1000 bytes



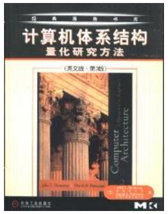
Instruction set complications-1

- An instruction is committed when it is guaranteed to complete.
  - On MIPS, all instructions are committed at the end of MEM.
  - Since no updates occur before instructions commit, precise interrupts are straightforward.
- In most RISC systems, each instruction writes only one result.
  - This means that the instruction can be cancelled any time before the instruction is committed, with no harm to the system state.



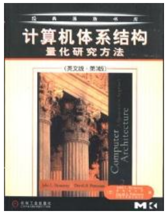
Instruction Set Complications-2

- This is not true for many CISC machines, i.e. VAX
  - On these machines, the system state may be modified well before the instruction or its predecessors are committed.
  - For example, if an instruction using autoincrement mode is aborted because of an exception, then the machine state may have been altered.
  - This leads to an imprecise exception making it difficult to restart the instruction.



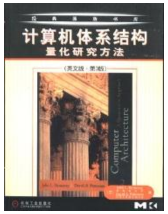
Instruction Set Complications-3

- The situation is worse for instructions that access and write memory in multiple places.
  - These instructions can generate multiple faults.
  - Therefore, it becomes difficult to know where to resume.
  - This is usually solved by using general purpose registers as work registers (that are saved and restored.)



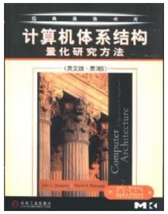
Instruction Set Complications-4

- Odd bits of state that may create additional pipeline hazards or may require extra hardware to save and restore.
  - Example: conditional codes
- Multicycle operation
  - The general solution used by more complex instruction set machines is to pipeline the microcode.
  - In 1990s, all companies moved to simpler ISA.

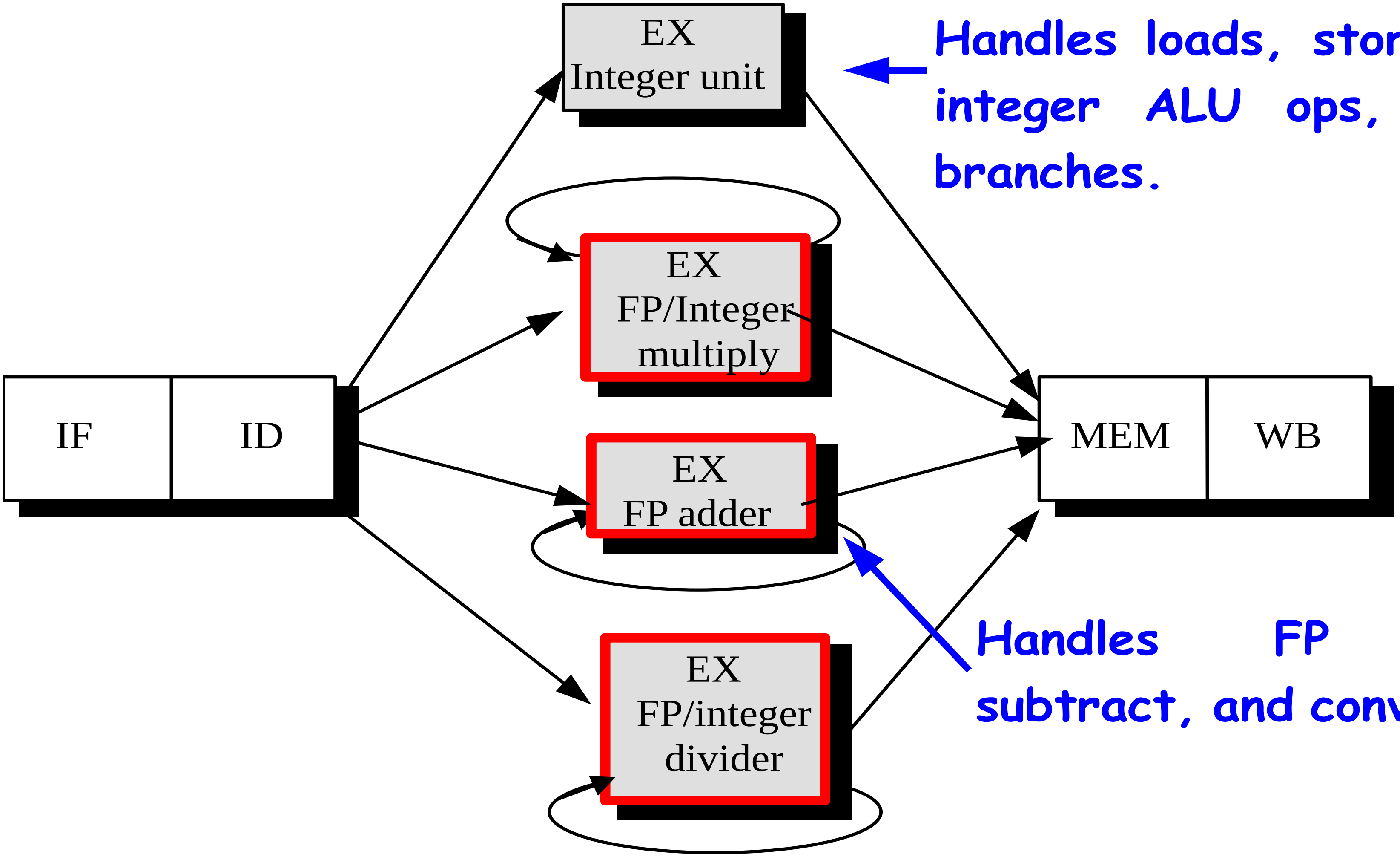


Extending the MIPS pipeline to handle MultiCycle Operations

- Alternative resolutions to handle floating-point operations
  - Complete operation in 1 or 2 clock cycles,
    - √ Which means using a slow clock,
    - √ or/and using enormous amounts of logic in FP units.
  - Allow for a longer latency for operations
    - √ The EX cycle may be repeated as many times as needed to complete the operation
    - √ There may be multiple FP units

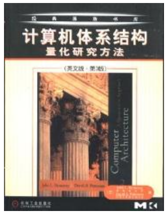


MIPS pipeline with FP units



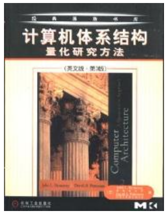
Handles loads, stores , integer ALU ops, and branches.

Handles FP add, subtract, and conversion



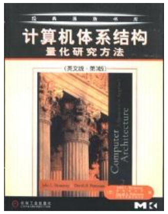
Pipelining some of the FP units

- Two terminologies
  - Latency----the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.
  - Initiation interval----the number of cycles that must elapse between instructions issue to the same unit.
    - √For full pipelined units, initiation interval is 1
    - √For unpipelined units, initiation interval is always the latency plus 1.

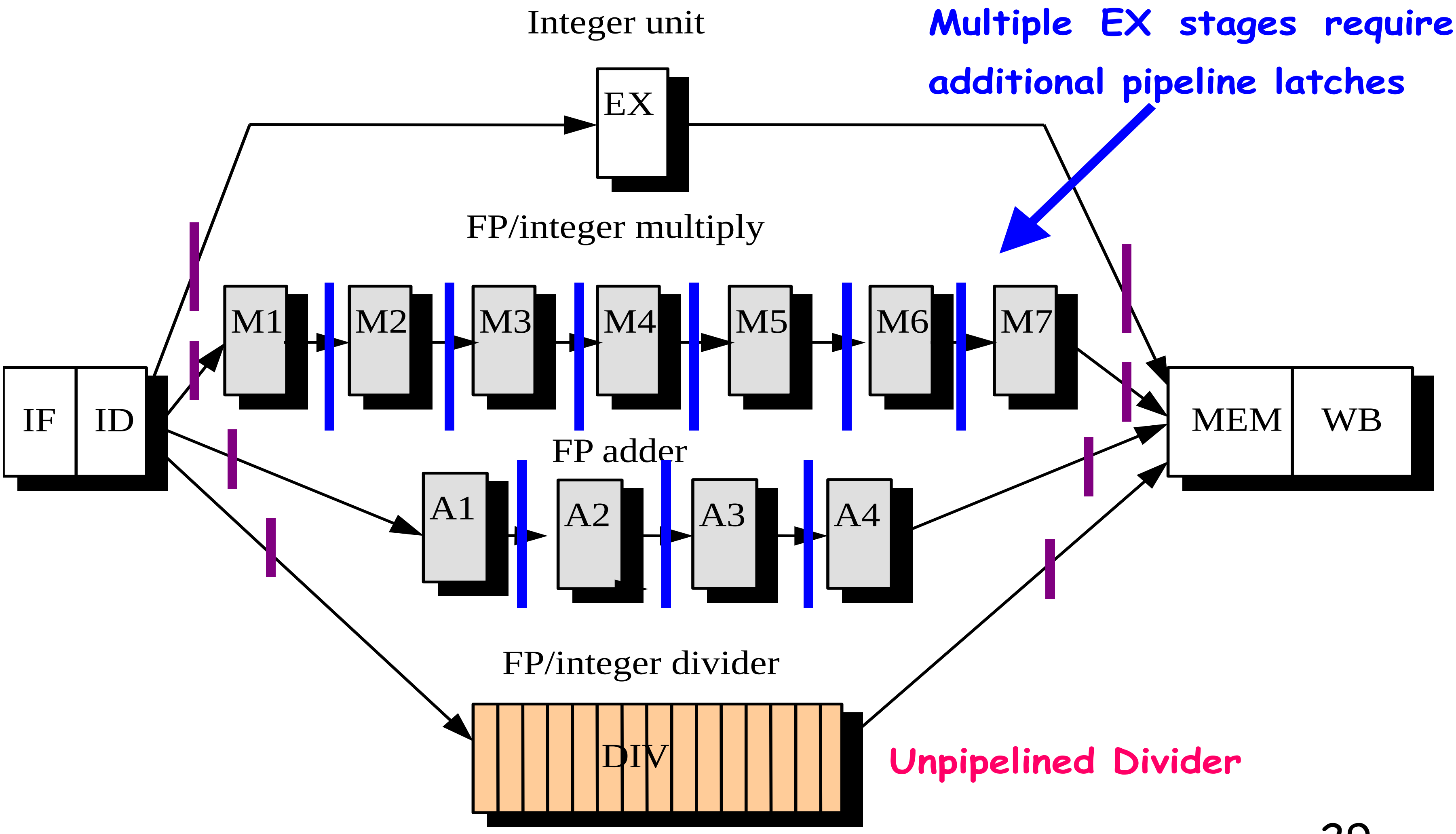


Latencies and initiation intervals for functional units

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

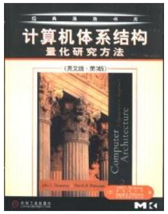


Pipeline supports multiple outstanding FP operations



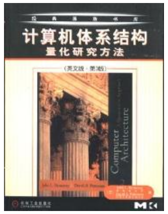
Multiple EX stages require additional pipeline latches

Unpipelined Divider



Specifications

- Memory bandwidth: double words/one cycle
- New pipeline latches are required:
  - M1/M2, M2/M3, M3/M4, M4/M5, M5/M6, M6/M7
  - A1/A2, A2/A3, A3/A4
- New connection registers are required:
  - ID/EX, ID/M1, ID/A1, ID/DIV
  - EX/MEM, M7/MEM, A4/MEM, DIV/MEM
- Because the divider unit is unpipelined, structural hazards can occur.
- Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1
- New data hazards: WAW is possible due to disorder WBs
- Due to longer latency of operations, stalls for RAW hazards will be more frequent.
- Problems with exceptions resulting from disorder completion

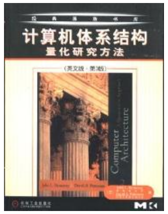


Issuing in order and completion out of order

Instruction	1	2	3	4	5	6	7	8	9	10	11		
MUL.D	IF	ID	***M1***	M2	M3	M4	M5	M6	**M7**	MEM	WB		
ADD. D		IF	ID	***A1***	A2	A3	**A4**	MEM	WB				
MUL.D			IF	ID	***M1***	M2	M3	M4	M5	M6	**M7**	MEM	WB
LD.D				IF	ID	***EX***	**MEM**	WB					
SD.D					IF	ID	***EX***	***MEM***	WB				

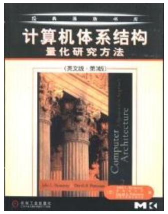
Structural Hazards for the FP register write port

Instruction	1	2	3	4	5	6	7	8	9	10	11
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
…		IF	ID	EXi	MEM	WB					
…			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
…						IF	ID	EX	MEM	WB	
LD F8, 0(R2)							IF	ID	EX	MEM	WB



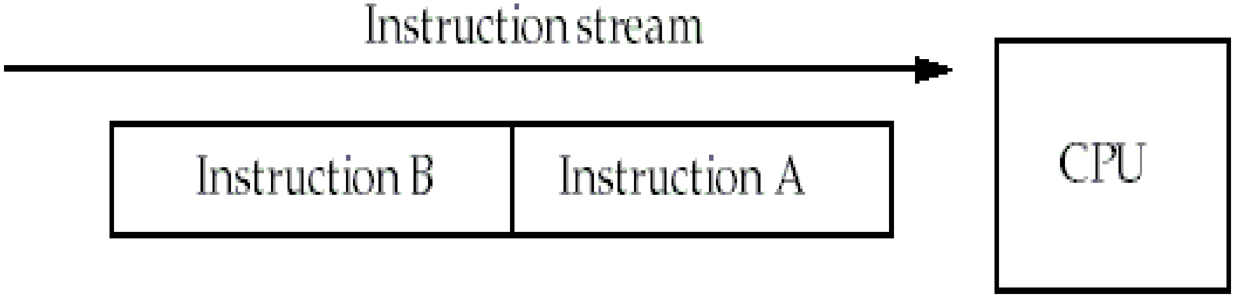
How to solve the write port conflict ?

- Increase the number of write ports
  - Unattractive at all !
  - No worthy since steady state usage is close to 1.
- Detect and insert stalls by serializing the writes
  - Track the use of the write port in the ID stage and to stall an instruction before it issues
    - √ Additional Hardware: a shift register+ write conflict logic
    - √ The shift register tracks when already-issued instructions will use the register file, and right shift 1 bit each clock.
    - √ The stalls might *aggravate* the data hazards
    - √ All interlock detection and stall insertion occurs in ID stage
  - To stall a conflicting instruction when it tries to enter the MEM or WB stage.
    - √ Easy to detect the conflict at this point
    - √ Complicates pipeline control since stalls can now occur in two places.

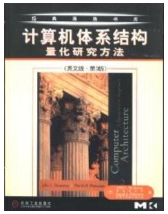


Types of data hazards

- Consider two instructions, A and B. A occurs before B.

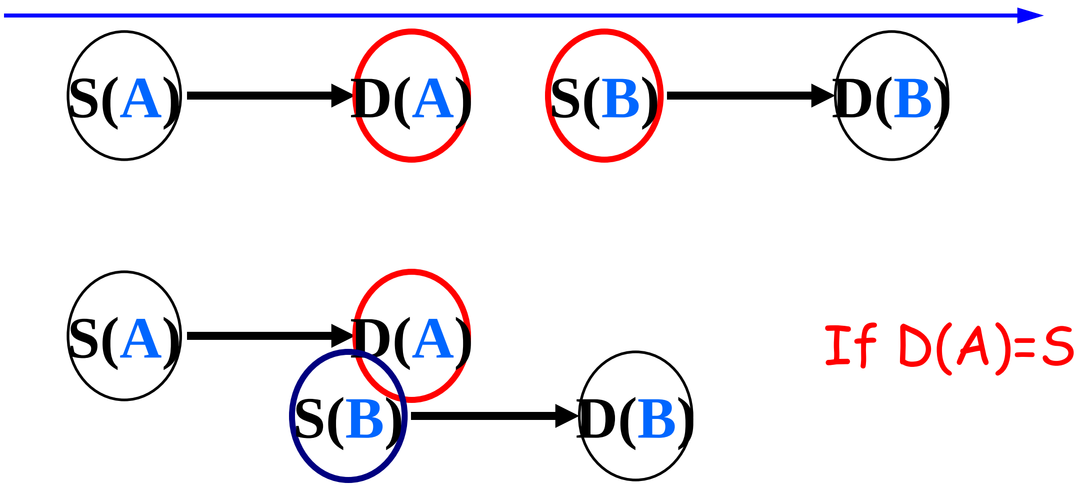


- RAW(Read after write) true dependence
  - Instruction A writes Rx，instruction B reads Rx
- WAW(Write after write) output dependence
  - Instruction A writes Rx，instruction B writes Rx
- WAR(Write after read) anti-denpendence
  - Instruction A reads Rx，instruction B writes Rx
- Hazards are named according to the ordering that MUST be preserved by the pipeline



RAW dependence

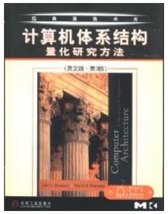
- B tries to read a register before A has written it and gets the old value.
- This is common, and forwarding helps to solve it.



Time

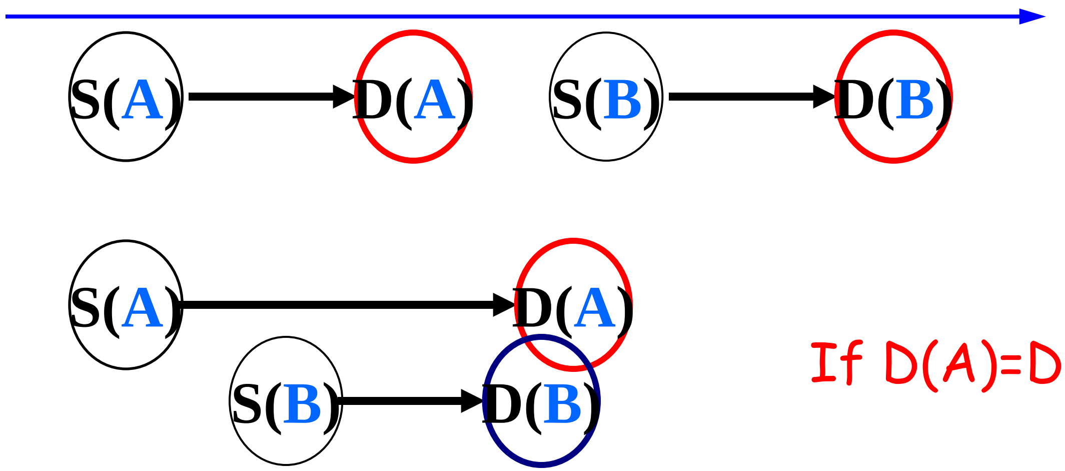
No hazard

If D(A)=S(B), hazard occur.



WAW dependence

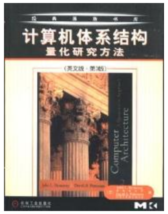
- B tries to write an operand before A has written it.
- After instruction B has executed, the value of the register should be B's result, but A's result is stored instead.
- This can only happen with pipelines that write values in more than one stage, or in variable-length pipelines (i.e. FP pipelines).



Time

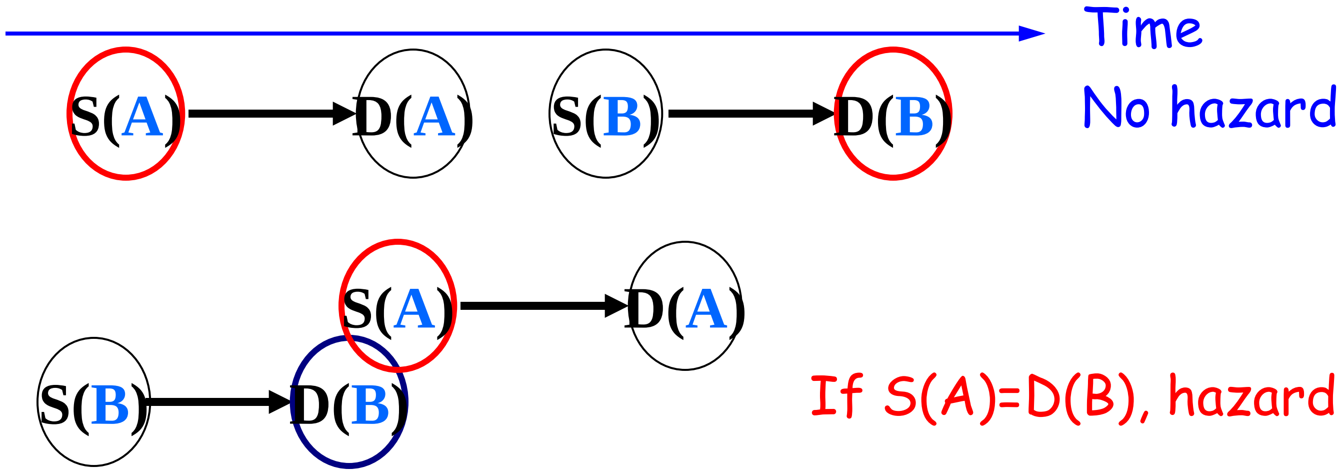
No hazard

If D(A)=D(B), hazard occur.

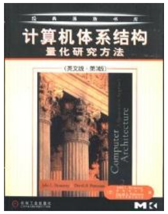


WAR dependence

- B tries to write a register before A has read it.
- In this case, A uses the new (incorrect) value.
- This type of hazard is rare because most pipelines read values early and write results late.
- However, it might happen for a CPU that had complex addressing modes. i.e. autoincrement.

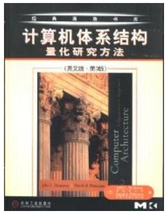


If S(A)=D(B), hazard occur.



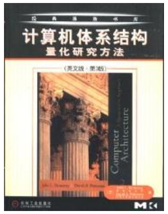
Stalls arising from RAW hazards

Instruction	1	2	3	4	5	6	7	8	9	10	11						
LD F4, 0(R2)	IF	ID	EX	MEM	WB												
MULTD F0, F4, F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADDD F2, F0, F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	
SD 0(R2) , F2					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM



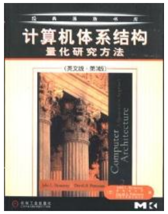
The WAW hazards

Instruction	1	2	3	4	5	6	7	8	9	10	11		
MULTD F0, F4, F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB		
…		IF	ID	EXi	MEM	WB							
…			IF	ID	EX	MEM	WB						
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	S	MEM	WB	
…					IF	ID	EX	MEM	WB				
LD F2, 0(R2)						IF	ID	EX	MEM	WB			
LD F8, 0(R2)							IF	ID	EX	S	S	MEM	WB



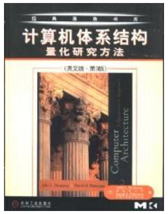
Solving the WAW hazard

- Stall an instruction that would "pass" another until after the earlier instruction reaches the MEM phase.
- Cancel the WB phase of the earlier instruction
- Both of these can be done in ID, i.e. when LD is about to issue.
- Since pure WAW hazards are not common, either method works.
- Pick the one that simplest to implement.
- The simplest solution for the MIPS pipeline is to hold the instruction in ID if it writes the same register as an instruction already issued.



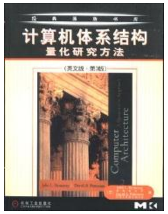
What other hazards are possible ?

- Hazards among FP instructions.
- Hazards between an FP instruction and an integer instruction.
  - Since two register files exist, only FP loads and stores and FP register moves to integer registers involve hazards.

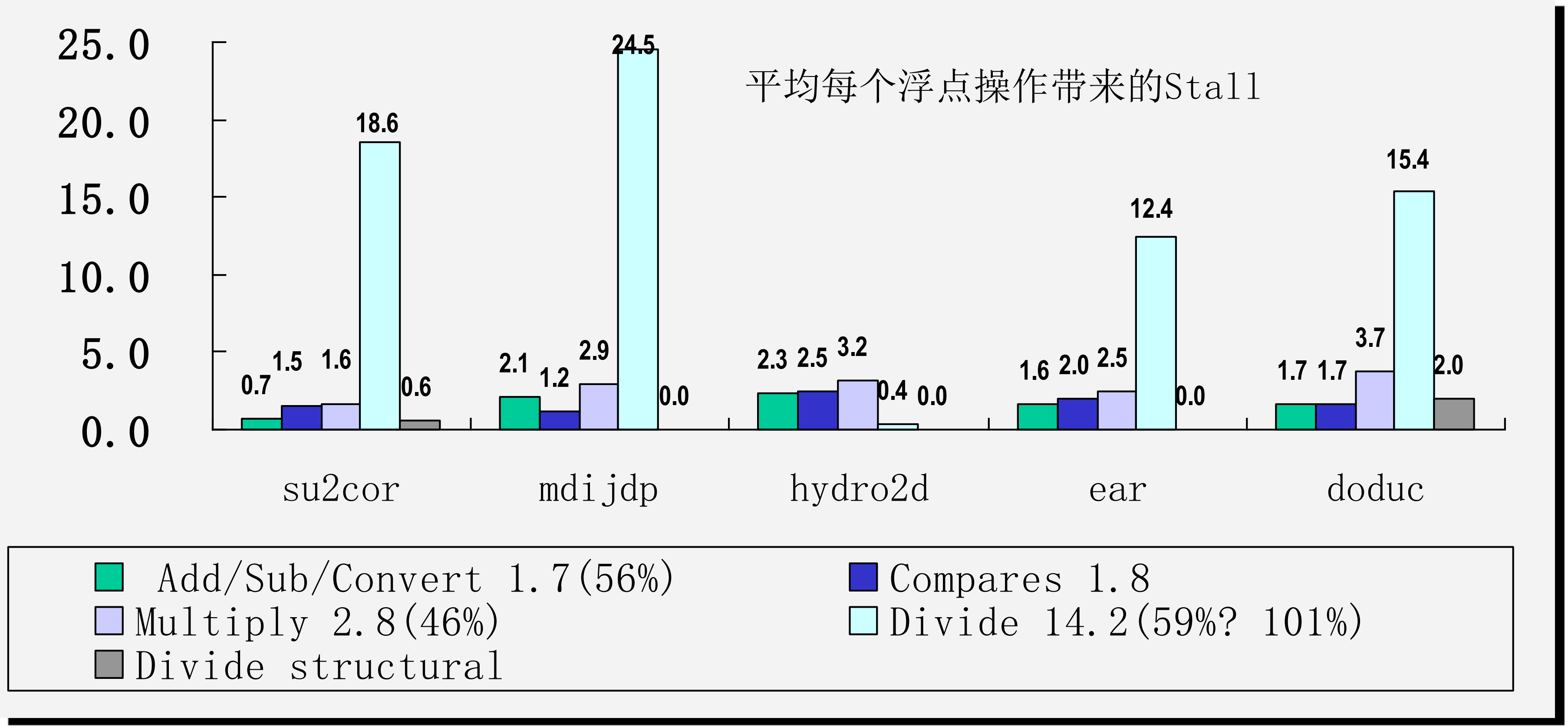


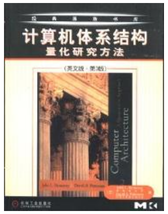
Checks are required in ID

- Check for structural hazards .
  - The divide unit and Register write port.
- Check for RAW hazards
  - The CPU simply stalls the instruction at ID stage until:
    - √ Its source registers are no longer listed as destinations in any of the execution pipeline registers (registers between stages of M and A) OR
    - √ Its source registers are no longer listed as the destination of a load in the EX/MEM register.
- Check for WAW hazards
  - Check instructions in A1, …, A4, Divide, or M1, …,M7 for the same destination register (check pipeline registers.)
  - Stall instruction in ID if necessary.

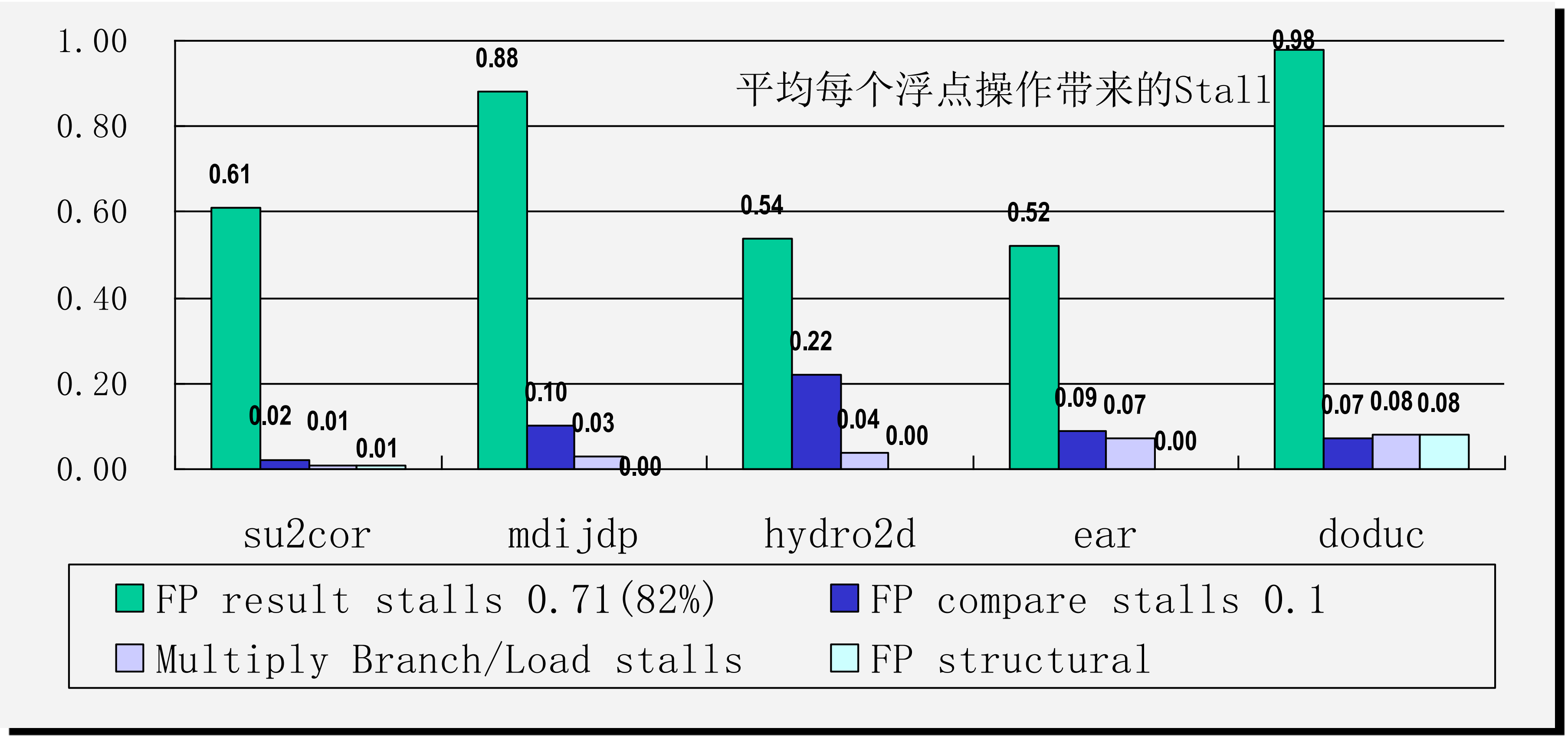


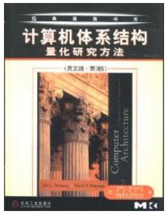
Performance of MIPS FP pipeline





Performance of MIPS FP pipeline



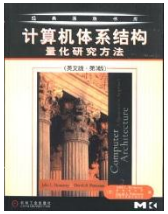


Maintaining precise Exception

- Exceptions are difficult because instructions may now finish out of order .
- Example

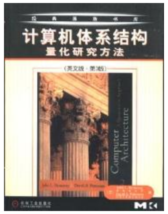
```
DIVF     F0, F2, F4
ADDF     F10, F10, F8
SUBF     F12, F12, F14
```

  - ADDF and SUBF are expected to complete before DIVF .----Out-of-order completion.
  - Suppose SUBF caused an arithmetic exception at a point where ADDF completed but DIVF has not.
  - The result is an imprecise exception . Fix here is to let pipeline drain.



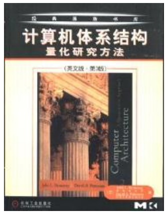
The worse case

- Worse, suppose DIVF had an exception after ADDF completed.
  - Since ADDF destroys one of its operands, we can not restore the state to what it was before the DIVF instruction, even with software !



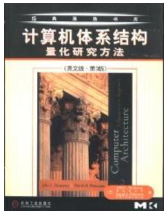
Handling exceptions -- first solution

- Ignore the problem (imprecise exceptions):
  - This may be fast and easy, but it's difficult to debug programs without precise exceptions.
  - Many modern CPUs, i.e. DEC Alpha 21064, IBM Power-1 and MIPS R800, provide a precise mode that allows only a single outstanding FP instruction at any time.
  - This mode is much slower than the imprecise mode, but it makes debugging possible



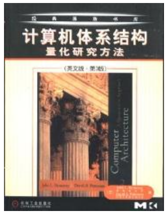
Handling exceptions -- Second solution

- Buffer the results and delay commitment
  - In this case, the CPU doesn't actually make any state (register or memory) changes until the instruction is guaranteed to finish.
  - This becomes difficult when the difference in running time among operations is large.
  - Lots of intermediate results have to be buffered (and forwarded, if necessary).



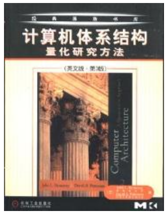
Variations of the second solution-1

- History file:
  - This technique saves the original values of the registers that have been changed recently.
  - If an exception occurs, the original values can be retrieved from this cache .
  - Note that the file has to have enough entries for one register modification per cycle for the longest possible instruction.
  - Similar to the solution used for the VAX for autoincrement and autodecrement addressing.



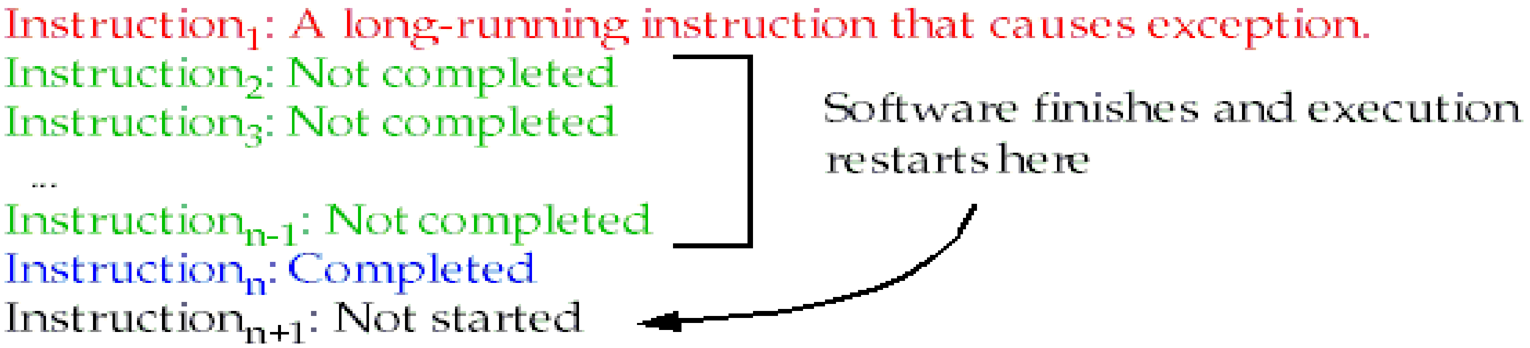
Variations of the second solution-2

- Future file:
  - – This method stores the newer values for registers.
  - – When all earlier instructions have completed, the main register file is updated from the future file.
  - – On an exception, the main register file has the precise values for the interrupted state.



Handling exceptions, third solution

- Keep enough information for the trap handler to create a precise sequence for the exception:
  - The instructions in the pipeline and the corresponding PCs must be saved.
  - After the exception, the software finishes any instructions that precede the latest instruction completed.



Instruction$_1$: A long-running instruction that causes exception.

Instruction$_2$: Not completed

Instruction$_3$: Not completed

...

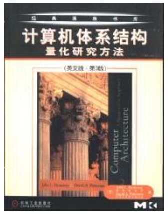
Instruction$_{n-1}$: Not completed

Instruction$_n$: Completed

Instruction$_{n+1}$: Not started

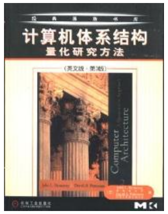
Software finishes and execution restarts here

- – Technique is used in the SPARC architecture.



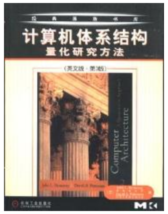
Handling exceptions, fourth solution

- Allow instruction issue only if it is known that all previous instructions will complete without causing an exception.
  - The floating point function units must determine if an exception is possible early in the EX stage, first couple clocks,
  - In order to prevent the following instructions from completing.
  - Sometimes it requires stalling the pipeline in order to maintain precise interrupts.
  - The R4000 and Pentium solution.



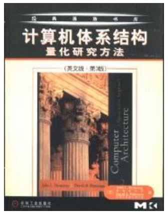
Guidelines for designing instruction sets for pipelining-1

- Avoid variable instruction lengths and running times whenever possible :
  - Variable length instructions complicate hazard detection and precise exception handling.
  - Sometimes it is worth it because of performance adv., i.e., caches .
  - Cause instruction running times to vary, when they miss.
  - Many times, the added complexity is delt with by freezing the pipeline.



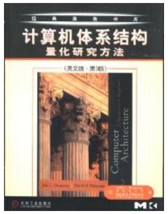
Guidelines for designing instruction sets for pipelining-2

- Avoid sophisticated addressing modes :
  - Addressing modes that update registers (post-autoincrement) complicates exceptions and hazard detection.
  - It also makes it harder to restart instructions.
  - Allowing addressing modes with multiple memory accesses also complicates pipelining.



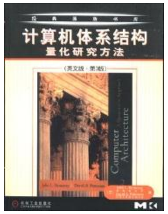
Guidelines for designing instruction sets for pipelining-3

- Don't allow self-modifying code
  - Since it is possible that the instruction being modified is already in the pipeline, the address being written must constantly be checked.
  - If it is found, then the pipeline must be flushed or the instruction updated !
  - Even if it's not in the pipeline, it could be in the instruction cache.



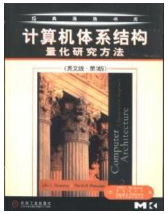
Guidelines for designing instruction sets for pipelining-4

- Avoid implicitly setting CCs in instructions
  - This makes it harder to avoid control hazards since it's impossible to determine if CCs are set on purpose or as a side effect.
  - For implementations that set the CC almost unconditionally :
  - Makes instruction reordering difficult since it is hard to find instructions that can be scheduled between the condition evaluation and the branch.



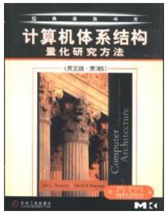
The MIPS R4000 pipeline

- IF—First half of instruction fetch. PC selection occurs. Cache access is initiated.
- IS—Second half of instruction fetch.
  - —This allows the cache access to take two cycles.
- RF—Decode and register fetch, hazard checking, I-cache hit detection.
- EX—Execution: address calculation, ALU Ops, branch target calculation and condition evaluation.
- DF/DS/TC
  - Data fetched from cache in the first two cycles.
  - The third cycle involves checking a tag check to determine if the cache access was a hit.
- WB—Write back result for loads and R-R operations.

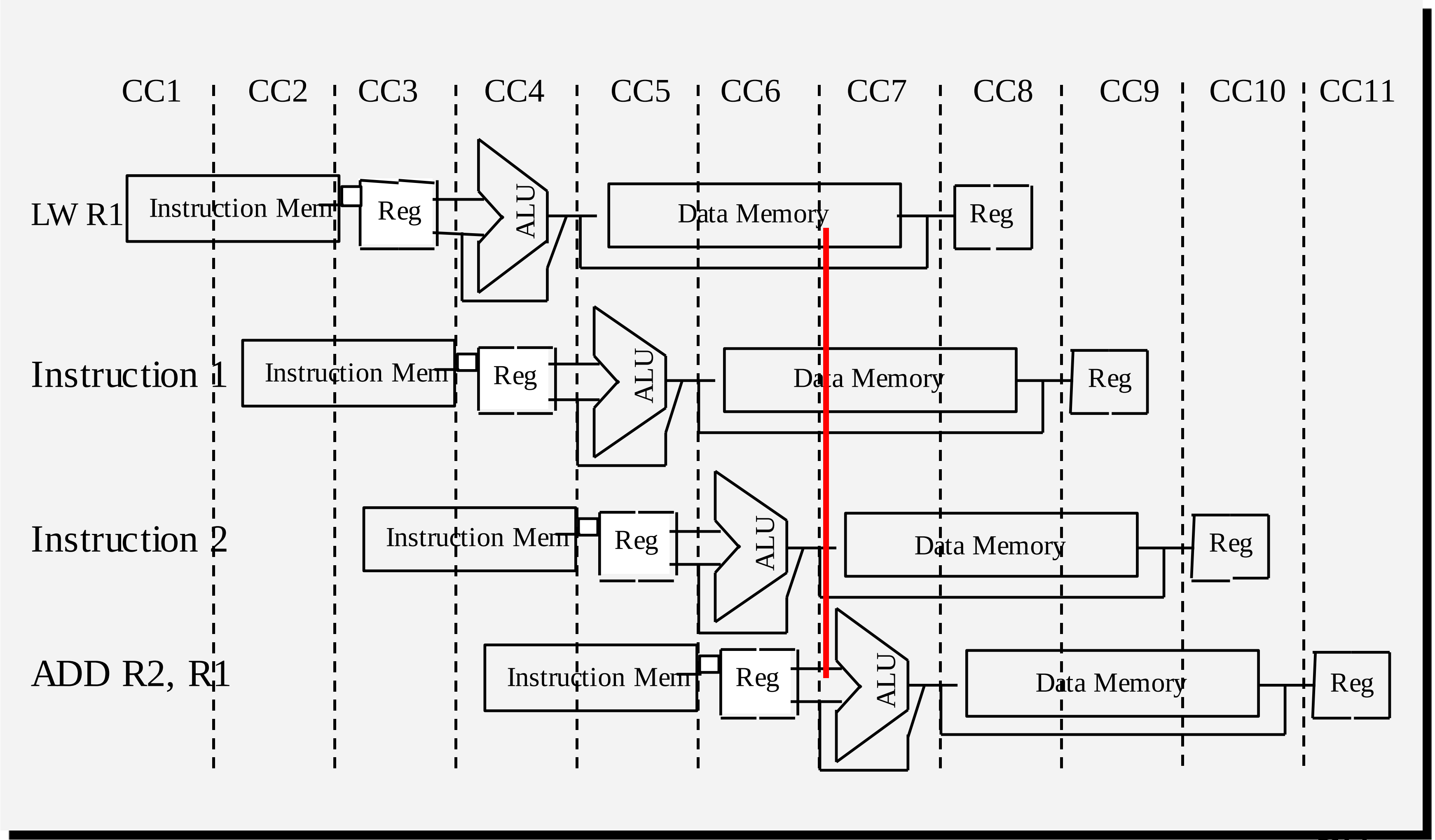


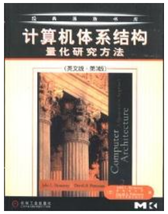
Possible stalls and delays

- Load delay: two cycles
  - The delay might seem to be three cycles, since the tag isn't checked until the end of the TC cycle.
  - However, if TC indicates a miss, the data must be fetched from main memory and the pipeline is backed up to get the real value.

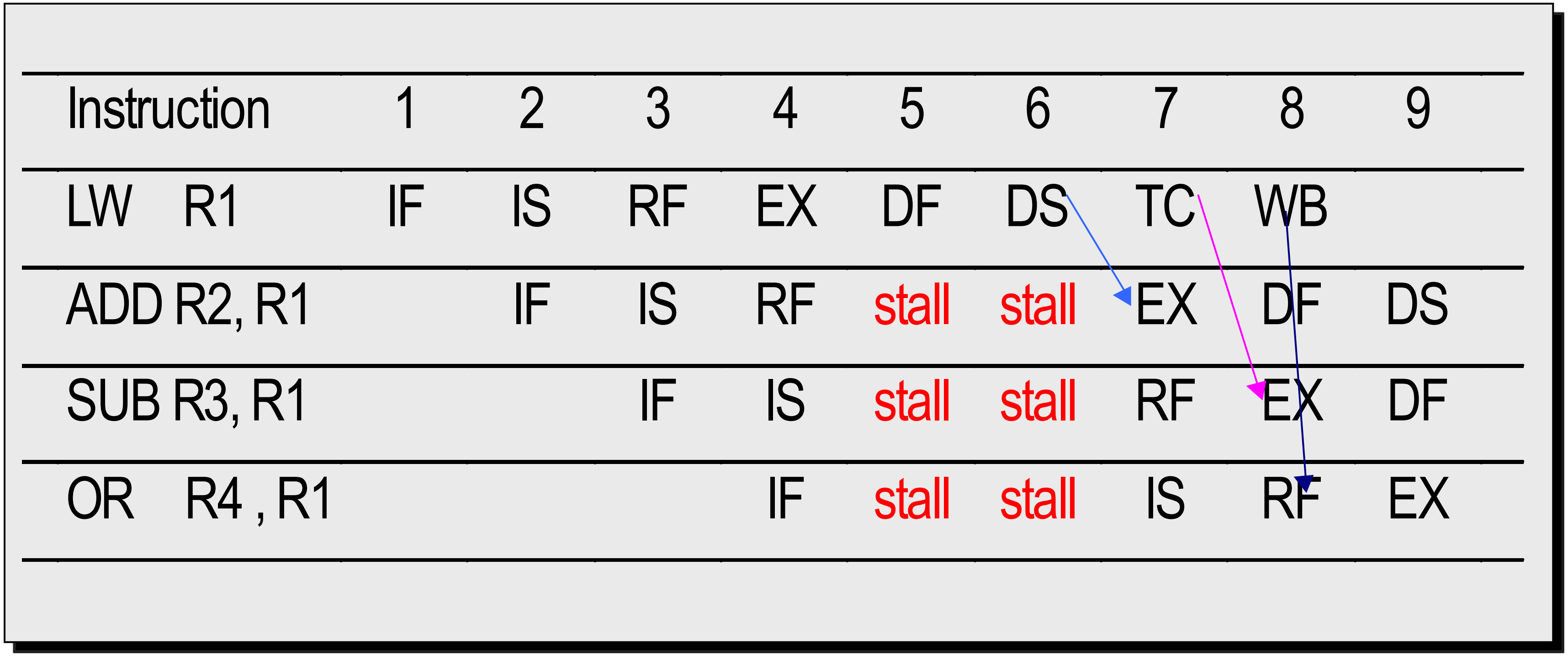


Load stalls

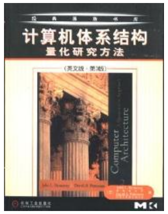

CC1
CC2
CC3
CC4
CC5
CC6
CC7
CC8
CC9
CC10
CC11
LW R1
Instruction 1
Instruction 2
ADD R2, R1
Instruction Mem
Reg
ALU
Data Memory
Reg



Example: load stalls

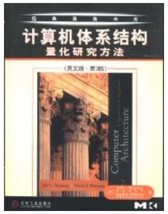


Instruction	1	2	3	4	5	6	7	8	9
LW R1	IF	IS	RF	EX	DF	DS	TC	WB	
ADD R2, R1		IF	IS	RF	stall	stall	EX	DF	DS
SUB R3, R1			IF	IS	stall	stall	RF	EX	DF
OR R4 , R1				IF	stall	stall	IS	RF	EX

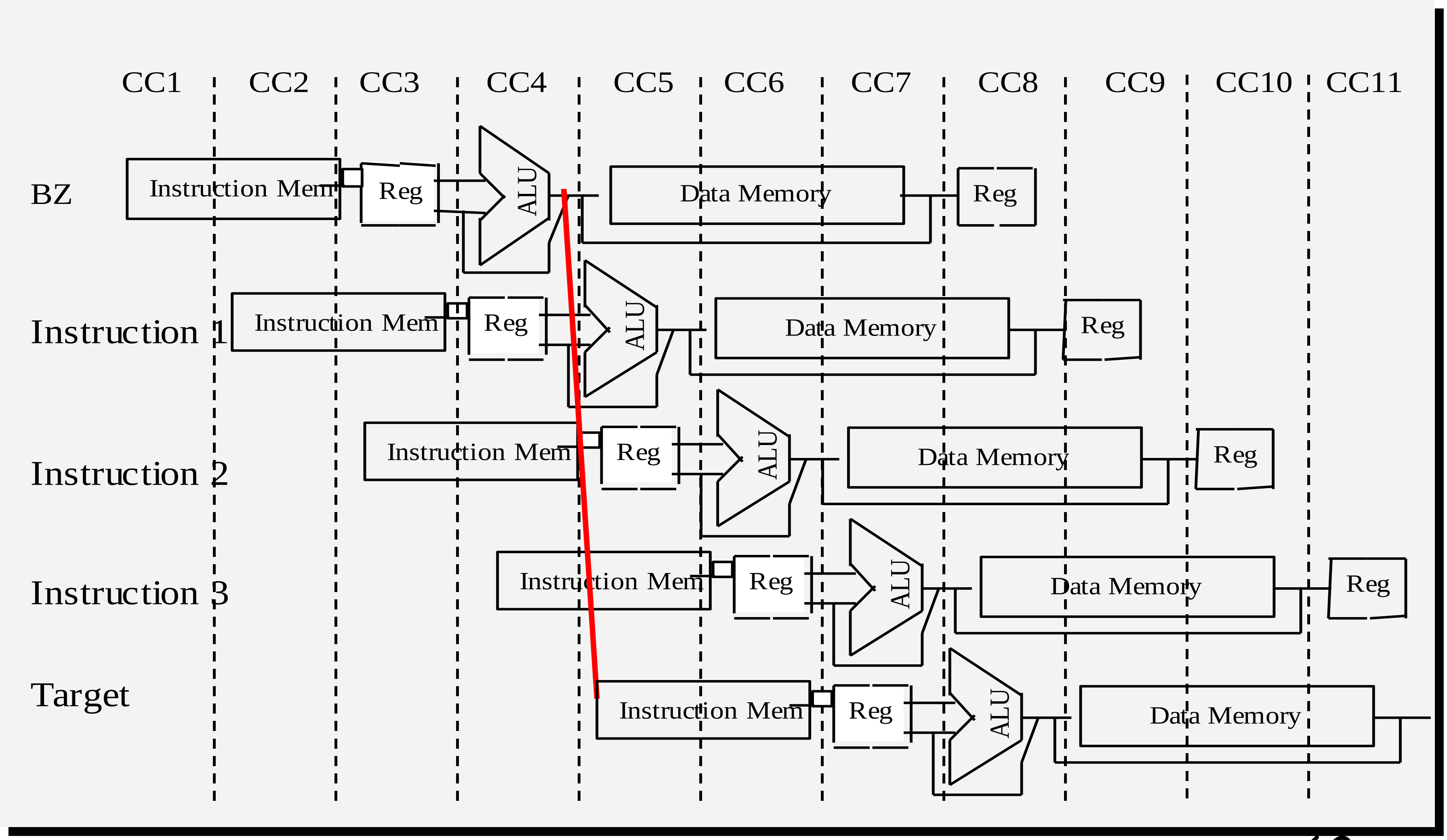


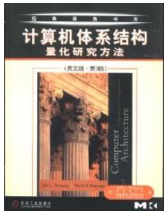
Branch delay: three cycles

- Branch delay: three cycles (including one branch delay slot)
  - The branch is resolved during EX, giving a 3 cycle delay.
  - The first cycle may be a regular branch delay slot (instruction always executed) or a branch-likely slot (instruction cancelled if branch not taken).
  - MIPS uses a predict-not-taken method presumably because it requires the least hardware.



Branch Delays: 3 stalls

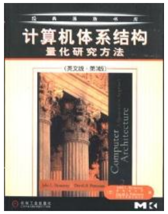




Pipeline status for branch latency

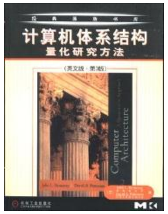
Instruction	1	2	3	4	5	6	7	8	9
Branch Ins.	IF	IS	RF	EX	DF	DS	TC	WB	
Delayed slot		IF	IS	RF	EX	DF	DS	TC	WB
Stall			stall	stall	stall	stall	stall	stall	stall
Stall				stall	stall	stall	stall	stall	stall
Branch target					IF	IS	RF	EX	DF

Instruction	1	2	3	4	5	6	7	8	9
Branch Ins.	IF	IS	RF	EX	DF	DS	TC	WB	
Delayed slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch ins +2			IF	IS	RF	EX	DF	DS	TC
Branch ins +3				IF	IS	RF	EX	DF	DS



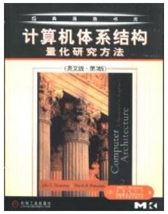
The FP 8-stage operational pipeline

Stage	Functional unit	Description
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP Multiplier	Exception test stage
M	FP Multiplier	First stage of multiplier
N	FP Multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers



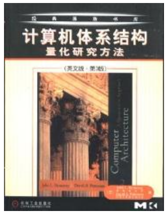
Latency and initiation intervals

FP instruction	Latency	Initiation interval	Pipe 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^{27},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
FP compare	3	2	U, A, R



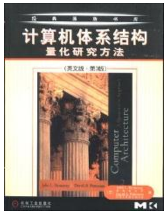
Structural hazards-1

Operation	Issue /stall	0	1	2	3	4	5	6	7	8	9
Multiply	Issue	U	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					U	S+A	A+R	R+S		
	Stall						U	S+A	A+R	R+S	
	Issue							U	S+A	A+R	R+S
	Issue								U	S+A	A+R



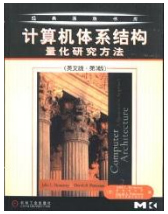
Structural hazards-2

Operation	Issue /stall	0	1	2	3	4	5	6	7	8	9
Add	Issue	U	S+A	A+R	R+S						
Multiply	Issue		U	M	M	M	M	N	N+A	R	
	Issue			U	M	M	M	M	N	N+A	R



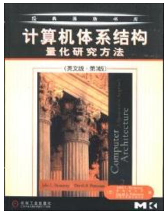
Structural hazards-3

Operation	Issue /stall	25	26	27	28	29	30	31	32	33	34	35
Divide	Issue in cycle 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				U	S+A	A+R	R+S				
	Stall					U	S+A	A+R	R+S			
	Stall						U	S+A	A+R	R+S		
	Stall							U	S+A	A+R	R+S	
	Stall								U	S+A	A+R	R+S
	Stall									U	S+A	A+R
	Issue										U	S+A
	Issue											U



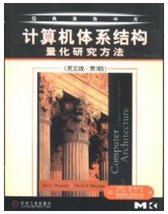
Structural hazards-4

Operation	Issue	0	1	2	3	4	5	6	7	8	9
Add	Issue	U	S+A	A+R	R+S						
Divide	stall		U	A	R	D	D	D	D	D	D
	Issue			U	A	R	D	D	D	D	D



Effects and Benefits of longer pipeline

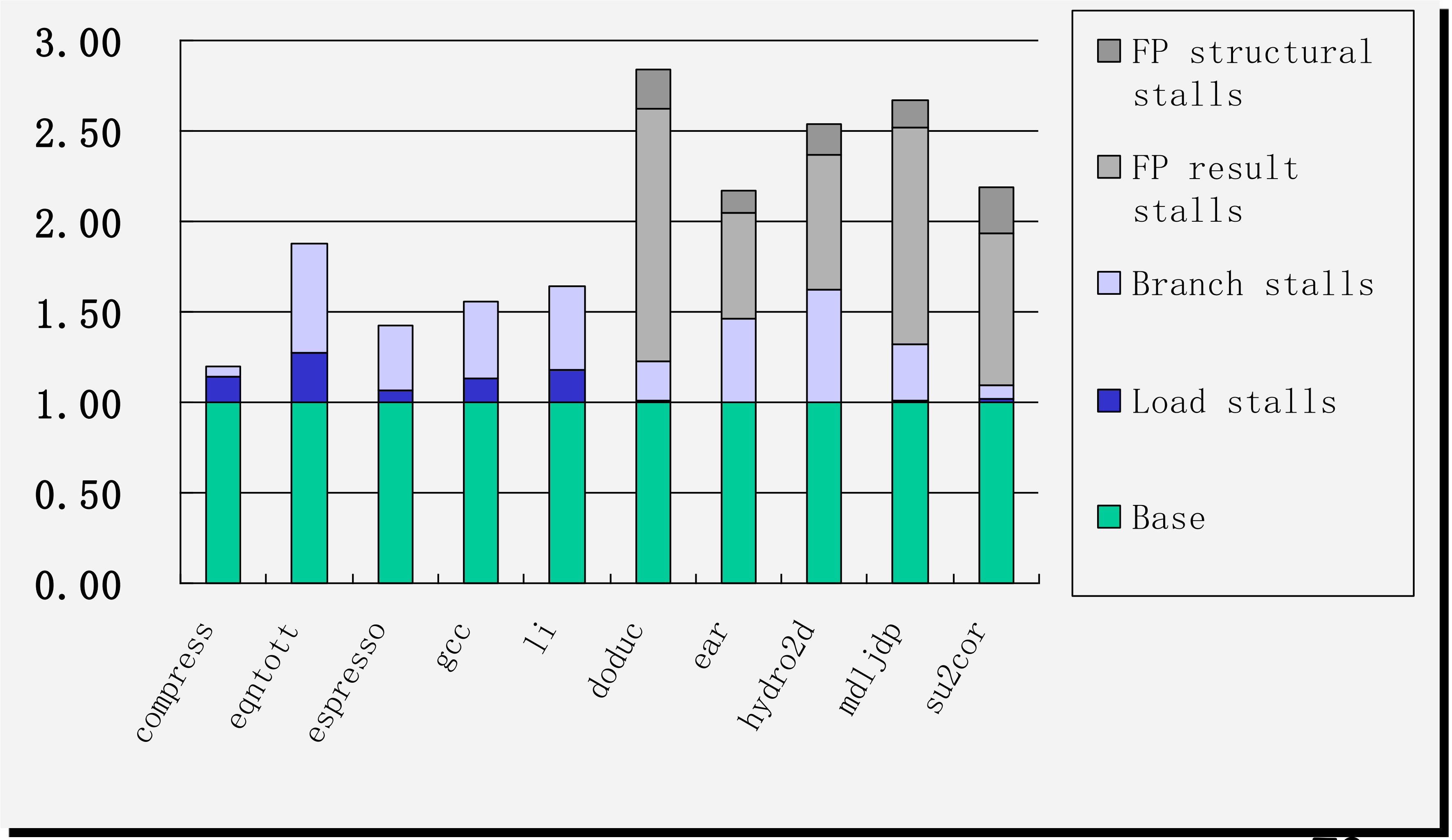
- Effects of longer pipeline:
  - In addition to the longer (and possibly more frequent) stalls just mentioned, the longer pipeline requires additional forwarding hardware.
  - It also requires more complex hazard detection to find dependencies in the additional stages.
- Benefits of longer pipeline
  - The major benefit to a longer pipeline is that each stage may be shorter.
  - This means that the clock cycle can be shorter, allowing more instructions to be issued in a fixed time.
  - Of course, the added stalls might eat up this benefit, but the hope is that at least some speedup will be left.



Performance issues (integer only)

- The ideal CPI for the pipelined CPU is 1.
- The biggest contributor to stalls is branch stalls.
- Load stalls contribute very little.
  - This is probably because the compiler can usually reorganize code to avoid stalling on loads.
- Since load latency is two cycles, though, the job is harder than it might be on processors with a single-cycle latency.

Performance loss measurements


compress
eqntott
espresso
gcc
li
doduc
ear
hydro2d
mdljdp
su2cor
FP structural stalls
FP result stalls
Branch stalls
Load stalls
Base