Implementing a pipelined processor

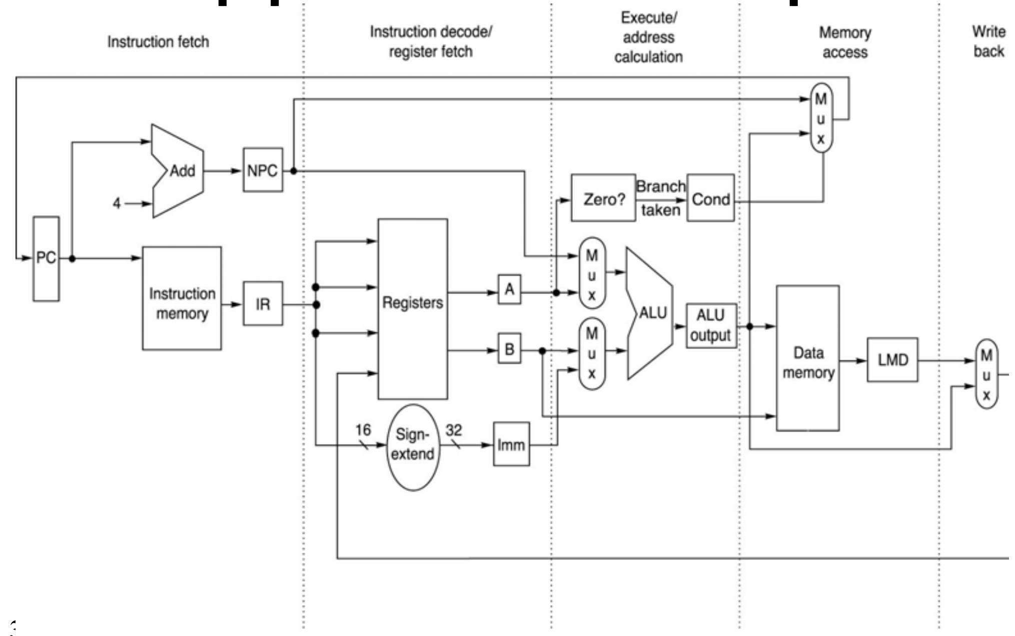
M. Sonza Reorda

Politecnico di Torino
Dipartimento di Automatica e Informatica

MIPS unpipelined version

- **Subset of MIPS supporting**
  - **Load-store word instructions**
  - **Branch equal zero instructions**
  - **Integer ALU instructions**
- **Every instruction can be executed in at most 5 clock cycles.**

Unpipelined MIPS data path

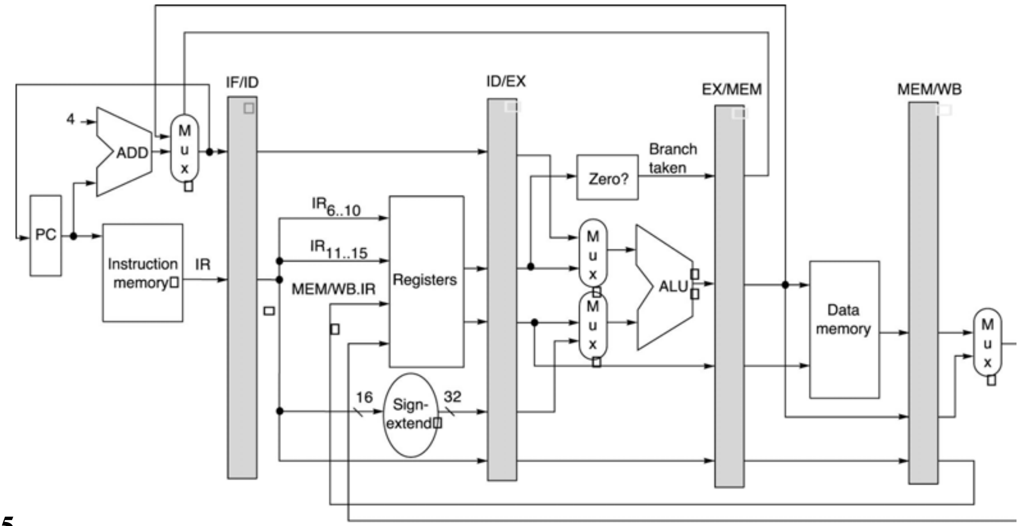


Pipelined MIPS version

A new instruction is started at each clock cycle.

Values passed from one pipe stage to the next must be placed in registers (*pipeline registers*).

Pipelined MIPS data path

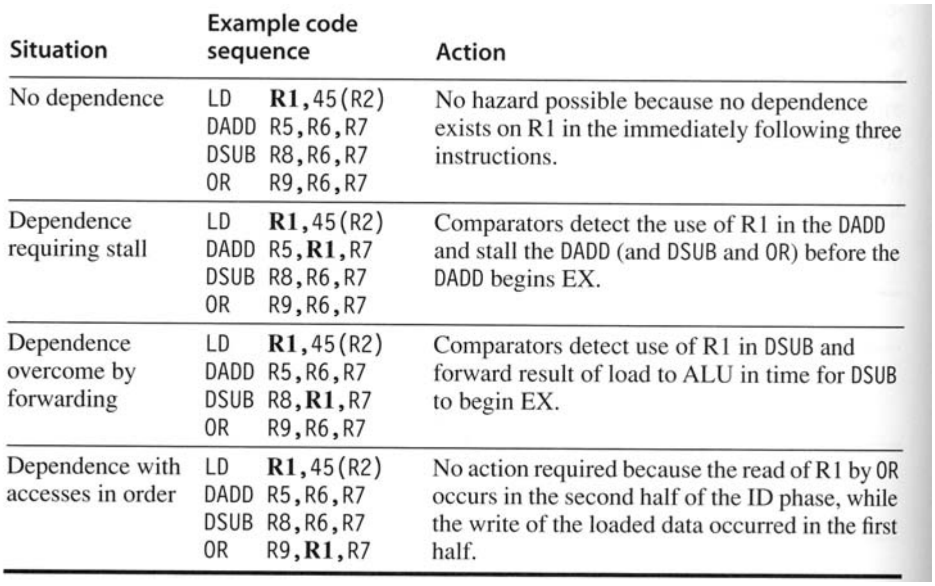


Implementing the Control

This requires that at each clock cycle:

- **All tests for detecting a possible data hazard concerning an instruction are performed when this is in the ID stage**
- **If a data hazard is detected, two actions can be alternatively taken:**
  - **the instruction is stalled before entering the EX stage (i.e., before being *issued*)**
  - **the appropriate forwarding is activated.**

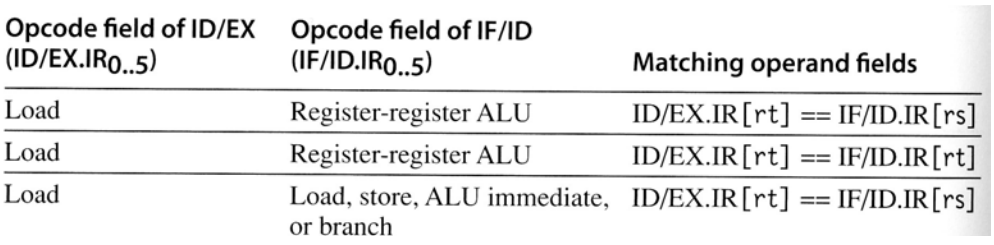
Load Interlock Detection



Situation	Example code sequence	Action
No dependence	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,R6,R7	No hazard possible because no dependence exists on R1 in the immediately following three instructions.
Dependence requiring stall	LD **R1**,45(R2) DADD R5,**R1**,R7 DSUB R8,R6,R7 OR R9,R6,R7	Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.
Dependence overcome by forwarding	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,**R1**,R7 OR R9,R6,R7	Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.
Dependence with accesses in order	LD **R1**,45(R2) DADD R5,R6,R7 DSUB R8,R6,R7 OR R9,**R1**,R7	No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Load Interlock Detection (cont'd)

The following checks have to be performed when a load instruction is in the EX stage and another instruction exploiting the loaded value is in the ID stage.



Opcode field of ID/EX (ID/EX.$IR_{0..5}$)	Opcode field of IF/ID (IF/ID.$IR_{0..5}$)	Matching operand fields
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rs]
Load	Register-register ALU	ID/EX.IR[rt] == IF/ID.IR[rt]
Load	Load, store, ALU immediate, or branch	ID/EX.IR[rt] == IF/ID.IR[rs]

Introducing a stall

Introducing a stall in the EX stage can be done in the following way:

- forcing all 0s in the ID/EX pipeline register (corresponding to a `nop` instruction)
- forcing the IF/ID pipeline register to maintain the current value.

Forwarding Logic

Forwarding can be implemented

- from the ALU or data memory output
- to ALU inputs, data memory inputs, or the zero detection unit.

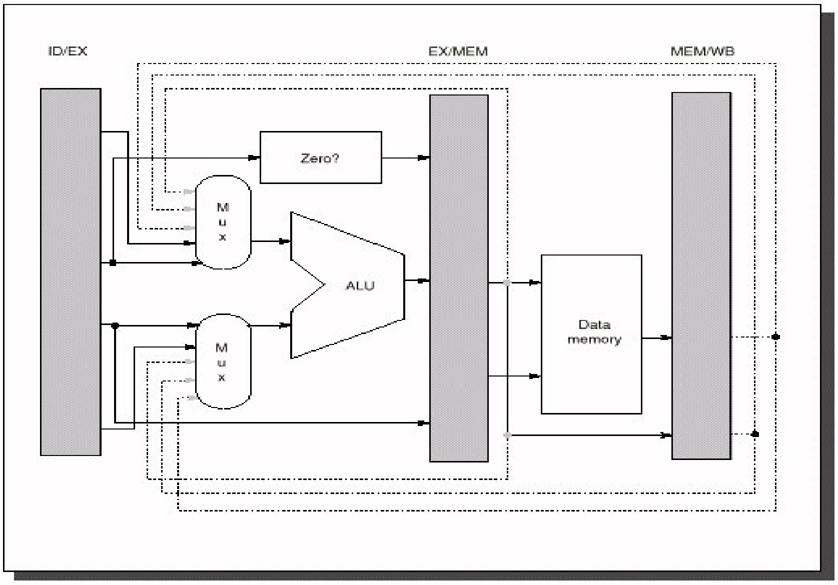
The forwarding logic must compare:

- the destination fields of the IR contained in the EX/MEM and MEM/WB registers *with*
- the source fields of the IR contained in the ID/EX and EX/MEM registers.

Forwarding to the ALU inputs

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR$_{16..20}$ = ID/EX.IR$_{6..10}$
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR$_{16..20}$ = ID/EX.IR$_{11..15}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR$_{16..20}$ = ID/EX.IR$_{6..10}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR$_{16..20}$ = ID/EX.IR$_{11..15}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	EX/MEM.IR$_{11..15}$ = ID/EX.IR$_{6..10}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	EX/MEM.IR$_{11..15}$ = ID/EX.IR$_{11..15}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR$_{11..15}$ = ID/EX.IR$_{6..10}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR$_{11..15}$ = ID/EX.IR$_{11..15}$
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	MEM/WB.IR$_{11..15}$= ID/EX.IR$_{6..10}$
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	MEM/WB.IR$_{11..15}$ = ID/EX.IR$_{11..15}$

Hardware changes to support forwarding to ALU inputs

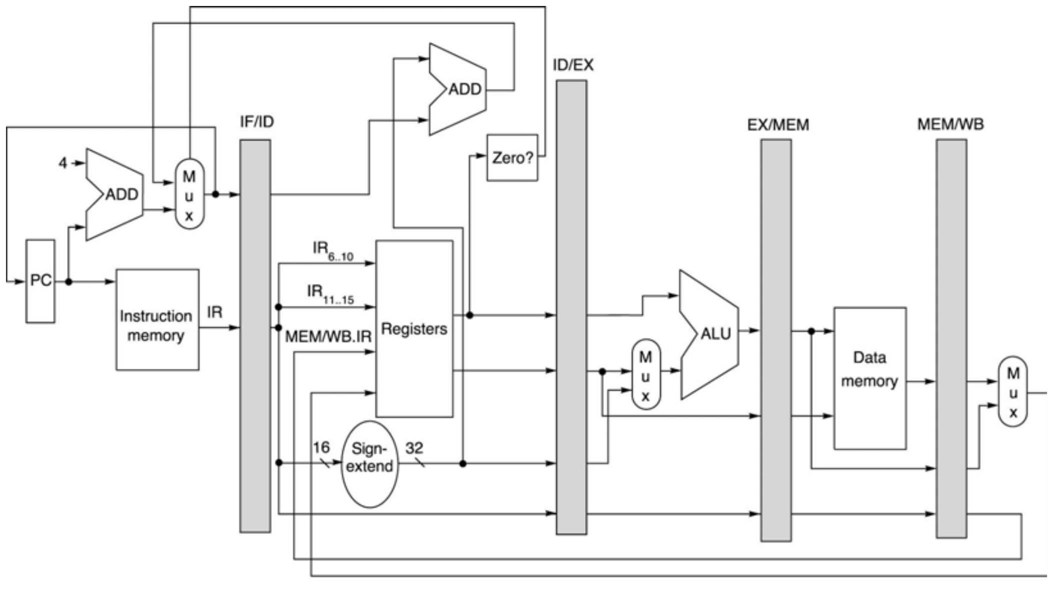


Dealing with branches

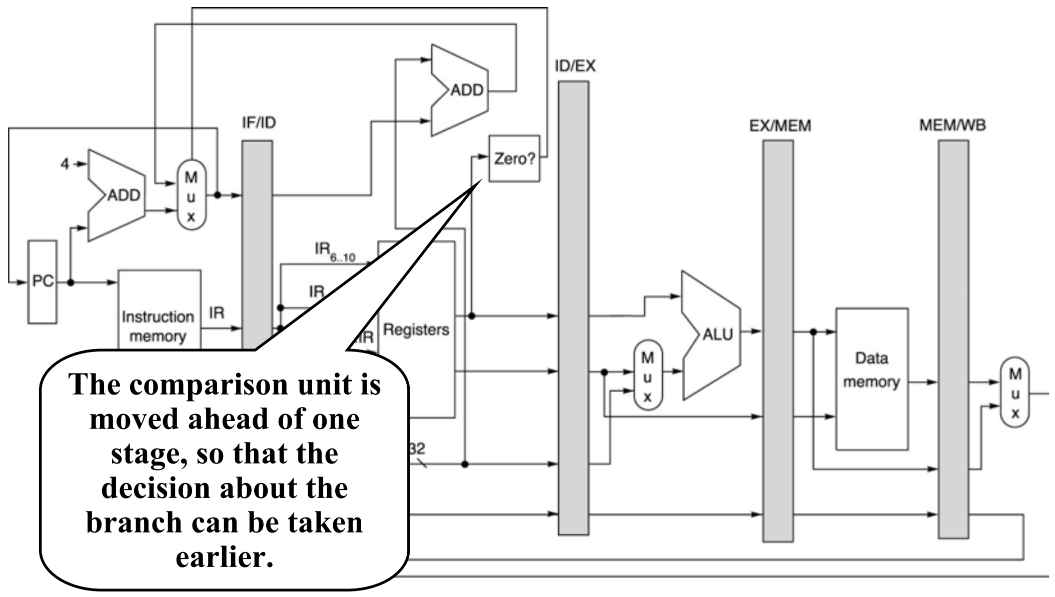
To reduce the performance degradation coming from control hazards, two steps must be done:

- **decide earlier whether the branch has to be taken or not**
- **compute earlier the new PC value.**

Modified pipeline

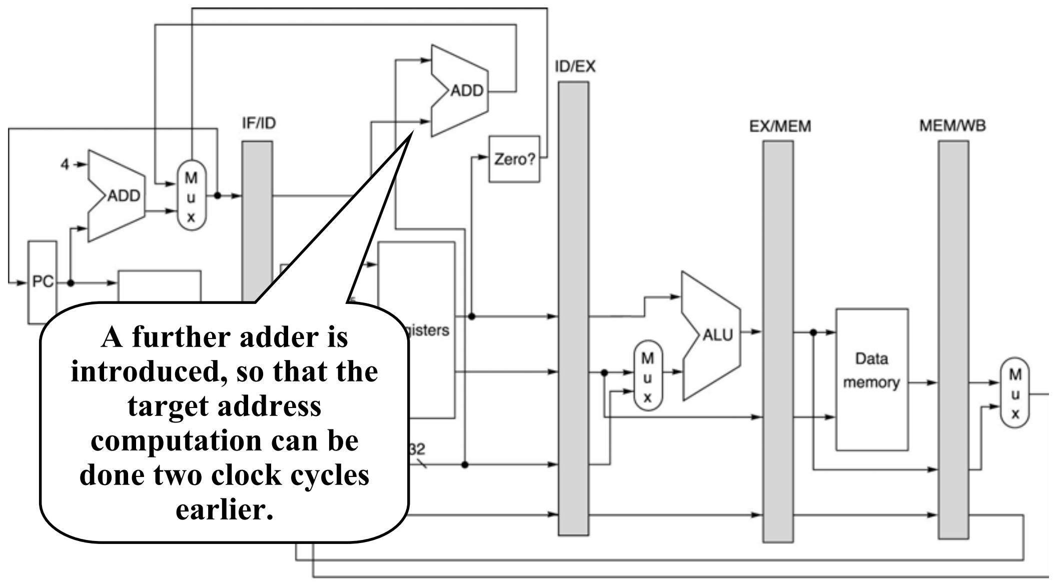


Modified pipeline



The comparison unit is moved ahead of one stage, so that the decision about the branch can be taken earlier.

Modified pipeline

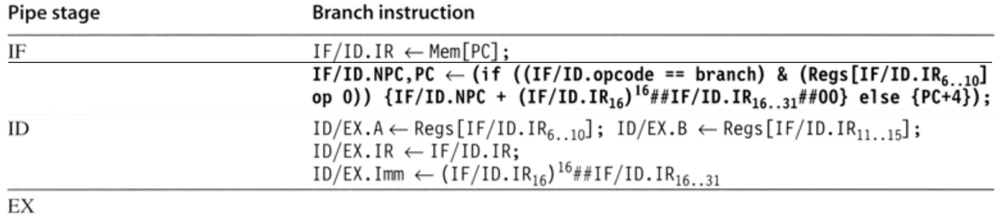


A further adder is introduced, so that the target address computation can be done two clock cycles earlier.

Modified pipeline

With the above modifications, the penalty for a branch is reduced to a single clock cycle.

Modified pipeline operations



Pipe stage	Branch instruction
IF	IF/ID.IR ← Mem[PC];
	IF/ID.NPC,PC ← (if ((IF/ID.opcode == branch) & (Regs[IF/ID.IR$_{6..10}$] op 0)) {IF/ID.NPC + (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$##00} else {PC+4});

ID	ID/EX.A ← Regs[IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs[IF/ID.IR$_{11..15}$];
	ID/EX.IR ← IF/ID.IR;
	ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$

EX	
MEM	
WB	

EXCEPTIONS

Exceptional events (exceptions, interrupts, or faults) are more complex to handle in pipelined architectures because several instructions are being executed at a time.

Exception sources

Possible causes of exceptions are:

- **I/O device request**
- **Operating system call by a user program**
- **Tracing instruction execution**
- **Breakpoint (programmer-requested interrupt)**
- **Integer arithmetic overflow or underflow**
- **FP arithmetic anomaly**
- **Page fault**
- **Misaligned memory accesses**
- **Memory-protection violation**
- **Undefined instruction**
- **Hardware malfunction**
- **Power failure.**

Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. user nonmaskable
- Within vs. between instructions
- Resume vs. terminate.

Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskab
- Within vs. l
- Resume vs

An exception is synchronous if it can be triggered always at the same position in the code. Asynchronous exceptions are normally generated by external devices.

Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskabl vs. user nonmaskable
- Within vs. b
- Resume

Coerced exceptions are out of the control of the user program.
User requested exceptions are similar to procedures.

Exception Classification

- Synchronous vs. asynchronous
- User requested vs. coerced
- User maskable vs. user nonmaskable
- Within vs between instru
- Resume vs. te

The user can normally force the hardware not to answer to exception requests.

Exceptio

Exceptions can occur either within or between instructions. In the former case they are generated by the instruction itself.

- Synchronous vs
- User requested
- User maskable vs
- Within vs. between instructions
- Resume vs. terminate.

Exception Cla

After activating an exception, the program can either terminate, or execute something and then resume.

- Synchronous vs. asynchr
- User requested vs. co
- User maskable vs. no
- Within vs. between instructions
- Resume vs. terminate.

Restartable machines

Restartable machines are able to handle an exception, save the state, and restart without affecting the execution of the program.

Nearly all processors nowadays are restartable machines.

Stopping the execution

When an exception occurs, the pipeline must execute the following steps:

- force a trap instruction into the pipeline on the next IF stage
- until the trap is taken, turn off all writes for the faulting instruction and for all the following instructions in the pipeline
- when the exception-handling procedure receives control, it immediately saves the PC of the faulting instruction.

Restarting the execution

After the exception has been handled, special instructions return the machine from the exception by reloading the PC and restarting the instruction stream.

Precise exceptions

A processor has *precise exceptions* if the pipeline can be stopped so that

- the instructions just before the faulting instruction are completed, and
- the instructions following the faulting instruction can be restarted from scratch.

Restarting after an exception may be really hard if exceptions are not handled precisely.

Precise exception handling is a must for most architectures, at least for integer instructions.

Cost of precise exceptions

Some processors (Alpha 21064, MIPS R800) implement precise exceptions in debug mode, only.

This mode is about 10 times slower than usual normal mode.

MIPS: Possible sources of Exceptions

Pipeline stage	*Cause of exception*
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

Contemporary exceptions

Example

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

A data page fault exception can occur in the MEM stage of the `LD` instruction, and an arithmetic exception in the EX stage of the `DADD` instruction.

The first exception is processed and, if its cause is removed, the second exception only occurs.

Exception order

There are cases in which two exceptions can occur in the opposite order of the instructions they relate to.

Example

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

An exception caused by an instruction page fault in the second instruction occurs before an exception caused by a data page fault in the first instruction.

Exception order (cont'd)

A possible solution is the following:

- a status flag is associated to each instruction in the pipeline
- if an instruction causes an exception, the status flag is set
- if the status flag is set, the instruction can not perform any write operation
- when an instruction reaches the last stage, and its status flag is set, an exception is triggered.

Instruction set complications

When an instruction is guaranteed to complete it is called *committed*.

Some machines have instructions that change the state before they are committed (e.g., those using autoincrement addressing modes).

If one of these instructions is aborted because of an exception, it leaves the machine state altered. Implementing precise exceptions is thus very difficult.

A possible solution is based on allowing to roll-back all the state changes made by an instruction before it is committed.

Instruction set complications (cont'd)

Instructions implicitly updating condition codes create complications:

- they can cause data hazards
- they require to be saved and restored in the case of an exception
- they make more difficult the task of the compiler for filling possible delay slot between the instruction writing the condition codes and the branch one.

Instruction set complications (cont'd)

Complex instructions are difficult to implement in a pipeline. Forcing them to have the same length can hardly be achieved.

Sometimes these problems are solved by pipelining the microinstructions implementing each instruction.