CMSC 611: Advanced Computer Architecture

Instruction Level Parallelism

Exceptions in 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

- Multiple exceptions might occur since multiple instructions are executing
  - (LW followed by DIV might cause page fault and an arith. exceptions in same cycle)
- Exceptions can even occur out of order
  - IF page fault before preceeding MEM page fault

Pipeline exceptions must follow order of execution of faulting instructions not according to the time they occur

Stopping & Restarting Execution

- Some exceptions require restart of instruction
  - e.g. Page fault in MEM stage
- When exception occurs, pipeline control can:
  - Force a trap instruction into next IF stage
  - Until the trap is taken, turn off all writes for the faulting (and later) instructions
  - OS exception-handling routine saves faulting instruction PC

Stopping & Restarting Execution

- Precise exceptions
  - Instructions before the faulting one complete
  - Instructions after it restart
  - As if execution were serial
- Exception handling complex if faulting instruction can change state before exception occurs
- Precise exceptions simplifies OS
- Required for demand paging

Precise Exception Handling

- The MIPS Approach:
  - Hardware posts all exceptions caused by a given instruction in a status vector associated with the instruction
  - The exception status vector is carried along as the instruction goes down the pipeline
  - Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off
  - Upon entering the WB stage the exception status vector is checked and the exceptions, if any, will be handled according the time they occurred
  - Allowing an instruction to continue execution till the WB stage is not a problem since all write operations for that instruction will be disallowed

Stopping & Restarting Execution

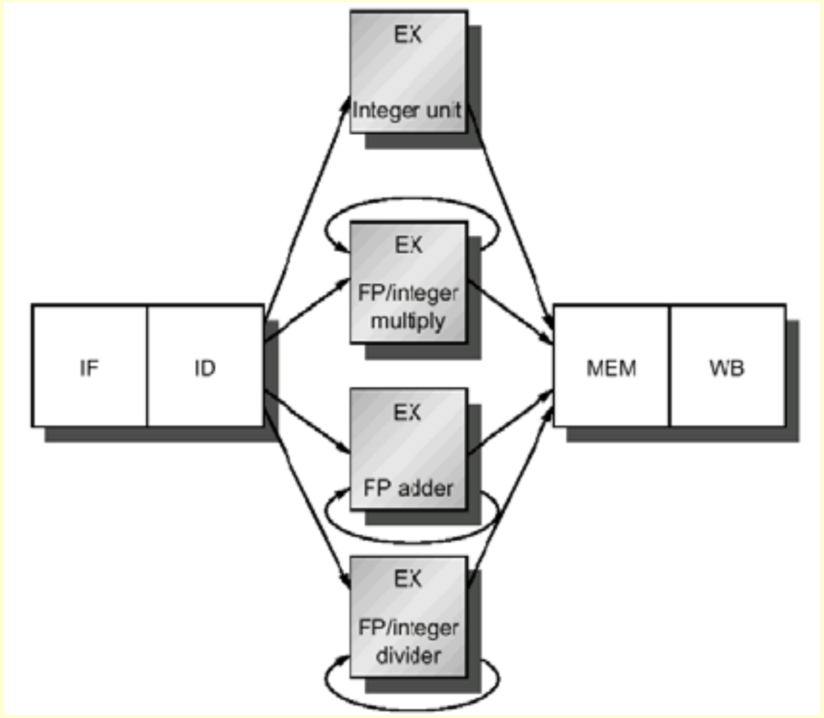
- Precise exceptions
  - Instructions before the faulting one complete
  - Instructions after it restart
  - As if execution were serial
- Exception handling complex if faulting instruction can change state before exception occurs
- Precise exceptions simplifies OS
- Required for demand paging

Instruction Set Complications

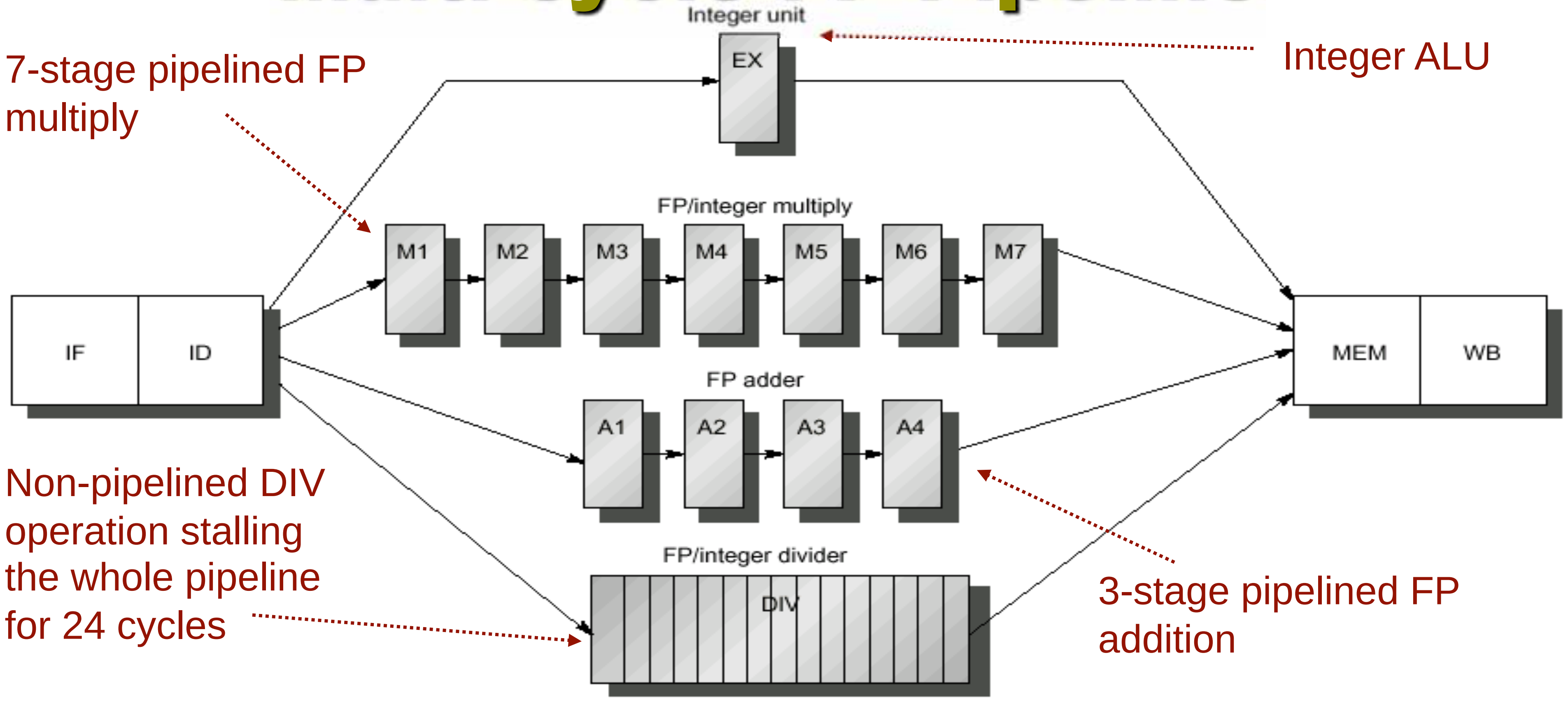
- Early-Write Instructions
  - MIPS only writes late in pipeline
  - Machines with multiple writes usually require capability to rollback the effect of an instruction
    - e.g. VAX auto-increment,
  - Instructions that update memory state during execution, e.g. string copy, may need to save & restore temporary registers
- Branching mechanisms
  - Complications from condition codes, predictive execution for exceptions prior to branch
- Variable, multi-cycle operations
  - Instruction can make multiple writes

Floating-Point Pipeline

- Impractical for FP ops to complete in one clock
  - (complex logic and/or very long clock cycle)
- More complex hazards
  - Structural
  - Data



Multi-cycle FP Pipeline


Integer unit
EX
Integer ALU
7-stage pipelined FP multiply
FP/integer multiply
M1
M2
M3
M4
M5
M6
M7
IF
ID
MEM
WB
FP adder
A1
A2
A3
A4
FP/integer divider
DIV
Non-pipelined DIV operation stalling the whole pipeline for 24 cycles
3-stage pipelined FP addition

MULTD	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
LD			IF	ID	*EX*	**MEM**	WB				
SD				IF	ID	EX	*MEM*	WB			

Example: *blue indicate where data is needed and red when result is available*

Multi-cycle FP: EX Phase

- Latency: cycles between instruction that produces result and instruction that uses it
  - Since most operations consume their operands at the beginning of the EX stage, latency is usually number of the stages of the EX an instruction uses
- Long latency increases the frequency of RAW hazards
- Initiation (Repeat) interval: cycles between issuing two operations of a given type

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

FP Pipeline Challenges

- Non-pipelined divide causes structural hazards
- Number of register writes required in a cycle can be larger than 1
- WAW hazards are possible
  - Instructions no longer reach WB in order
- WAR hazards are **NOT** possible
  - Register reads are still taking place during the ID stage
- Instructions can complete out of order
  - Complicates exceptions
- Longer latency makes RAW stalls more frequent

Instruction	**Clock cycle number**																
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
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	WB
SD 0(R2), F2					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

Example of RAW hazard caused by the long latency

Structural Hazard

Instruction	Clock cycle number										
	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	EX	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 F2, 0(R2)							IF	ID	EX	MEM	WB

- At cycle 10, MULTD, ADDD and LD instructions all in MEM
- At cycle 11, MULTD, ADDD and LD instructions all in WB
  - Additional write ports are not cost effective since they are rarely used
- Instead
  - Detect at ID and stall
  - Detect at MEM or WB and stall

WAW Data Hazards

Instruction	Clock cycle number										
	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	EX	MEM	WB					
…			IF	ID	EX	MEM	WB				
ADDD F2, F4, F6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
LD F2, 0(R2)						IF	ID	EX	MEM	WB	
….							IF	ID	EX	MEM	WB

- WAW hazards can be corrected by either:
  - Stalling the latter instruction at MEM until it is safe
  - Preventing the first instruction from overwriting the register
- Correcting at cycle 11 OK unless intervening RAW/use of F2
- WAW hazards can be detected at the ID stage
  - Convert 1st instruction to no-op
- WAW hazards are generally very rare, designers usually go with the simplest solution

Detecting Hazards

- Hazards among FP instructions & and combined FP and integer instructions
- Separate int & fp register files limits latter to FP load and store instructions
- Assuming all checks are to be performed in the ID phase:
  - Check for structural hazards:
    - Wait if the functional unit is busy (Divides in our case)
    - Make sure the register write port is available when needed
  - Check for a RAW data hazard
    - Requires knowledge of latency and initiation interval to decide when to forward and when to stall
  - Check for a WAW data hazard
    - Write completion has to be estimated at the ID stage to check with other instructions in the pipeline
- Data hazard detection and forwarding logic from values stored between the stages

Maintaining Precise Exceptions

- Pipelining FP instructions can cause out-of-order completion
- Exceptions also a problem:

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

  - No data hazards
  - What if DIVF exception occurs after ADDF writes F10?

Four FP Exception Solutions

1. Settle for imprecise exceptions
   - Some supercomputers still uses this approach
   - IEEE floating point standard requires precise exceptions
   - Some machine offer slow precise and fast imprecise exceptions
2. Buffer the results of all operations until previous instructions complete
   - Complex and expensive design (many comparators and large MUX)
   - History or future register file

Four FP Exception Solutions

3. Allow imprecise exceptions and get the handler to clean up any miss
   - Save PC + state about the interrupting instruction and all out-of-order completed instructions
   - The trap handler will consider the state modification caused by finished instructions and prepare machine to resume correctly
   - Issues: consider the following example

     Instruction1: Long running, eventual exception

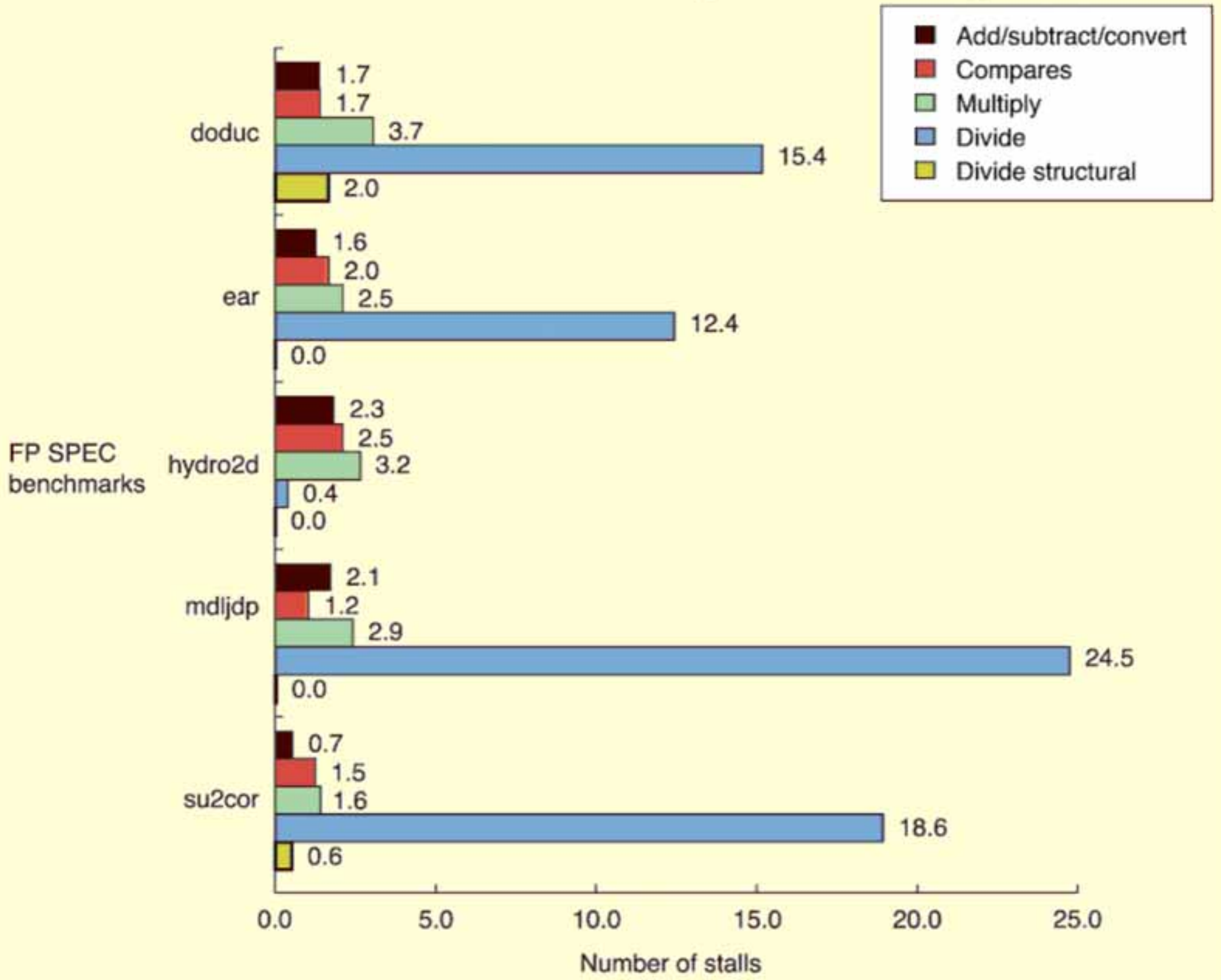
     Instructions 2 … (n-1) : Instructions that do not complete

     Instruction n : An instruction that is finished
   - The compiler can simplify the problem by grouping FP instructions so that the trap does not have to worry about unrelated instructions

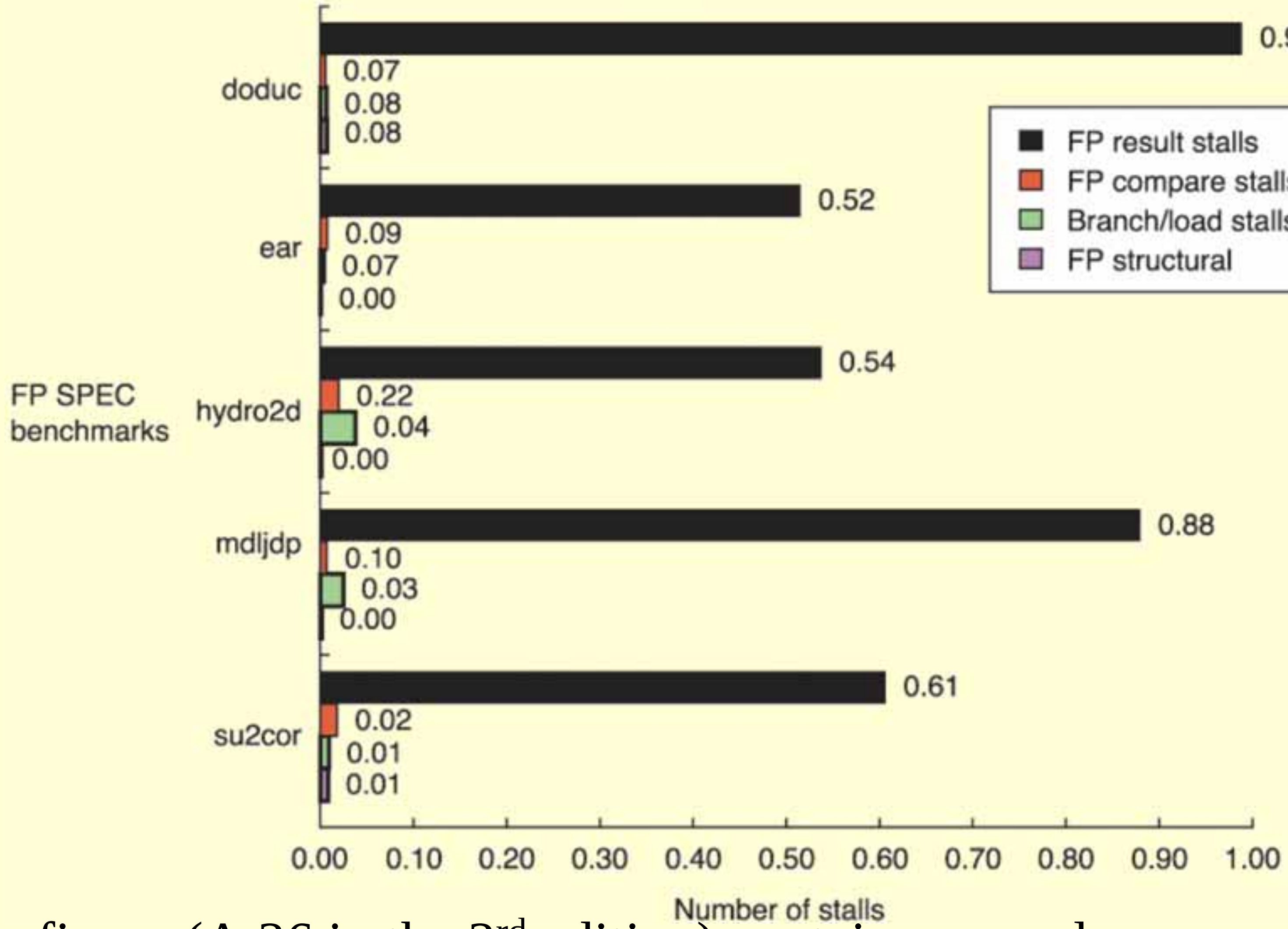
Four FP Exception Solutions

4. Allow instruction issue to continue only if previous instruction are guaranteed to cause no exceptions:
   - Mainly applied in the execution phase
   - Used on MIPS R4000 and Intel Pentium

Stalls/Instruction, FP Pipeline



More FP Pipeline Performance



This figure (A.36 in the 3rd edition) contains several.
Only take-home: result stalls are most common by far

Instruction Level Parallelism (ILP)

- Overlap the execution of unrelated instructions
- Both instruction pipelining and ILP enhance instruction throughput not the execution time of the individual instruction
- Potential of IPL within a basic block is very limited
  - in “gcc” 17% of instructions are control transfer meaning on average 5 instructions per branch

Loops: Simple & Common

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

- Techniques like loop unrolling convert loop-level parallelism into instruction-level parallelism
  - statically by the compiler
  - dynamically by hardware
- Loop-level parallelism can also be exploited using vector processing
- IPL feasibility is mainly hindered by data and control dependence among the basic blocks
- Level of parallelism is limited by instruction latencies

Major Assumptions

- Basic MIPS integer pipeline
- Branches with one delay cycle
- Functional units are fully pipelined or replicated (as many times as the pipeline depth)
  - An operation of any type can be issued on every clock cycle and there are no structural hazard

Instruction producing result	Instruction using results	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store Double	2
Load Double	FP ALU op	1
Load Double	Store Double	0

Motivating Example

```
for(i=1000; i>0; i=i-1)
   x[i] = x[i] + s;
```

```
Loop: LD    F0,x(R1)   ;F0=x[i]
      ADDD  F4,F0,F2   ;add F2(=s)
      SD    x(R1),F4   ;store result
      SUBI  R1,R1,8    ;i=i-1
      BNEZ  R1,Loop    ;loop to 0
```

Standard Pipeline execution

```
Loop: LD    F0,x(R1)
      stall
      ADDD  F4,F0,F2
      stall
      stall
      SD    x(R1),F4
      SUBI  R1,R1,8
      stall
      BNEZ  R1,Loop
      stall
```

Smart compiler

```
Loop: LD    F0,x(R1)
      SUBI  R1,R1,8
      ADDD  F4,F0,F2
      stall       ;F4
      BNEZ  R1,Loop
      SD    x+8(R1),F4
```

Sophisticated compiler optimization reduced execution time from 10 cycles to only 6 cycles