Exceptions

Increasing the pipeline hair factor

- ❑ Pipeline Benefit = Overlapped instructions
  - good: increased throughput (as we've seen)
  - sort of bad: hazards - but we've seen how to deal with them
  - bad: exceptions
- ❑ Exception oddities
  - multi-stage ==> multi-cyc le instructions
  - exceptions can happen anywhere
  - instruction order and exception order ma y be different
  - handling exceptions in instruction order is required
- ❑ So what' s the strategy?
  - depends on exception type

Exception Types

Terminology varies of course - another Tower of Babel

- I/O device request
- invoking an OS service from a user program
  - e .g. via an unimplemented instruction on a Mac
- tracing instruction execution
- breakpoint
- integer or FP arithmetic error suc h as overflow
- page fault
- misaligned memory access
- memory protection violation
- undefined instruction
- hardw are malfunction - like parity or ECC error
- power failure

Response Requirements - 5 axis model

- ❑ Sync hronous vs . Async hronous
  - sync hronous caused by a particular instruction
  - async hronous - external devices and HW failures
- ❑ User requested vs . Coerced
  - requested is predictable and can happen after the instruction
- ❑ User maskable vs . user non-maskable
  - e .g. arithmetic overflow on some mac hines is user maskable
- ❑ Within vs . Between instructions
  - within ==> sync hronous , key is that completion is prevented
  - some async hronous are also within
- ❑ Resume vs . Terminate program
  - implications for how muc h state must be preserved

Examples

Exc. Type	Synchronous - Asynch.	Requested - Coerced	mask - non-mask	within - between	resume - terminate
I/O Device Req.	Asynch	Coerced	Non-maskable	Between	Resume
Invoke OS svc.	Synch	User Requested	Non-maskable	Between	Resume
Trace/Bkpoint	Synch	User Requested	Maskable	Between	Resume
Arith. exception	Synch.	Coerced	Maskable	Within	Resume
Page Fault	Synch.	Coerced	Non-maskable	Within	Resume
Misaligned addr.	Synch	Coerced	Maskable	Within	Resume
Mem. prot. viola- tion	Synch	Coerced	Non-maskable	Within	Resume
Undefined Inst.	Synch.	Coerced - ???	Non-maskable	Within	Terminate
HW error	Asynch.	Coerced	Non-maskable	Within	Terminate
Power Failure	Asynch	Coerced	Non-maskable	Within	Terminate

Note some maskable - nonmask variance by machine masking is often done by the OS

Biggest Problem - Within & Resume

- ❑ For DLX - these tend to occur in the EX or MEM stage (e.g. late in the pipe)
- ❑ ==> Pipeline must be safely shut down
  - PC must be saved so restart point is known
  - if restart is a branch then it will need to be re-executed
    - which means the condition "code" state must not change
- ❑ Steps (in DLX)
  - force trap instruction into pipe on next IF
  - nuke all following instructions (0's to pipeline registers)
  - let all preceding instructions complete if they can
  - save the restart PC value - note 2 possibilities
    - faulting instruction or faulting instruction plus one (depends on handler)
  - call the OS to handle the exception

Making Things Harder for Performance

consider delayed branches

- ❑ A single restart PC won't do

Assume we have 2 branch delay slots

- branch is fine
- 1st delay slot causes a page fault
- 2nd instruction is nuked
- exception is handled and "default" restart is 1st delay slot
- 2nd slot instruction executed
- then next, etc. --- OOPS no branch
  - a side effect of the *effective* instruction reordering due to delayed branch
  - more complex reordering will occur shortly for performance improvements
- ❑ Hence must save delay slot size + 1 of PC's

Precise Interrupts

- ❑ Definition
  - instructions before fault complete
  - instructions after fault can be restarted from scratch
- ❑ Note assumption
  - faulting instruction doesn't change state
  - in some cases this can be relaxed in others it will be a requirement for precise handling to work
- ❑ e.g. Floating point exceptions
  - longer pipe - may have written result before fault is known
  - particular bummer if destination was also a source
  - hence must save the original operands as part of the argument stream passed to the exception handler

Precise and Non-Precise Modes

- ❑ Typical in today's high performance machines
  - e.g. Alpha 21164, MIPS R8000 and R10000, Power-3
  - precise mode is as much as 10x slower
  - biggest source of the problem is the FPU - and out of order completion
  - hence in precise mode - overlap (e.g. pipelining) is constrained
  - result is lots of "bubbles"
- ❑ Use precise when debugging
  - also a requirement in many systems - e.g. IEEE FP standard handlers and virtual memory support, also eases OS interface
  - not too difficult for the integer pipe anyway
- ❑ Use non-precise when you *think* your code works

Done laughing yet?

Even for the DLX

Exceptions can happen anywhere

- ❑ IF
  - page fault, misaligned address, memory protection violation
- ❑ ID
  - undefined or illegal opcode
- ❑ EX
  - arithmetic exception
- ❑ MEM
  - page fault, misaligned address, memory protection violation
- ❑ WB
  - none

Hence - *within* any 1 clock cycle 4 exceptions could occur

Making the DLX Precise

- ❑ Exception order may not match pipeline order
- ❑ But we must take them in program order to be precise
  - currently program order = pipeline order for the DLX
- ❑ *Committing* an instruction
  - when instruction is guaranteed to complete - it commits
  - in the DLX this happens at the end of the MEM stage
- ❑ Prior to commit stage carry exception state
  - exception type, restart PC values pass through pipe
  - ***ALL*** destructive (memory or register writes) are deferred until commit point
- ❑ Easy for DLX hard for VAX (are you surprised?)
  - VAX HW must save *back-out* state - e.g. undo autoinc. etc.

Possible Design Decision Problems

since they complicate precise exception handling

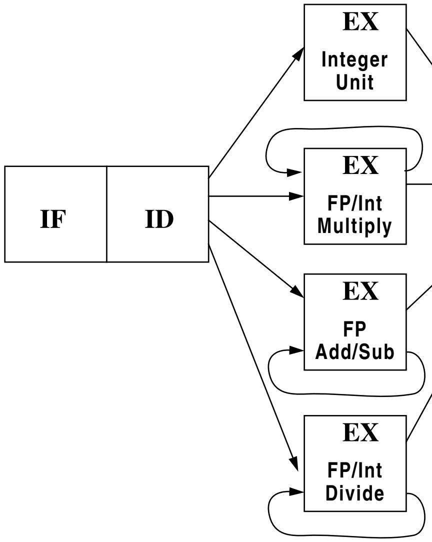
- ❑ early register changes
  - VAX: auto-increment and auto-decrement address modes
- ❑ iterative instructions
  - IBM 360 - block memory move
  - how much moved before the fault?
- ❑ use of registers as working storage
  - 80x86 string instructions
- ❑ condition codes - many machines use them
  - problem if they can be set in multiple stages
  - if set early then they must be restored on an exception
- ❑ multi-cycle instructions (all machines)

What about multi-cycle instructions?

Abundant options in the VAX ISA

- ❑ This can be fixed by a good ISA design
  - after all the VAX is ancient and we're smarter now
  - *buy this and there's this bridge in Brooklyn that you might be interested in buying!!!*
- ❑ Common modern multi-cycle causes
  - simple loads and stores that miss in their L0 caches
  - reasonable cost FPU latencies 5x + of their IU counterparts
  - co-processor or SFU instructions require temporal flexibility
- ❑ Result
  - stuck with the reality of multi-cycle instructions or stages
  - real *wrench in the works* for laminar pipeline design goals

An Expanded Multi-Cycle DLX Pipe



Initially consider:
using iterative EX units for complicated instructions

Unlikely the real FP-EX units will be iterative since throughput would truly suck!

Real method is different pipes - each having different latencies (depth)

Latency vs. Repeat Interval

- Latency - same=number of cycles to complete
  - *defined* to be the cycle distance *between* instructions using the result to the instruction that produces the result
- Repeat/Initiation interval
  - number of cycles that must elapse *between* issue of instructions of the same type

For the new - real FP capable DLX

usage penalty in reality

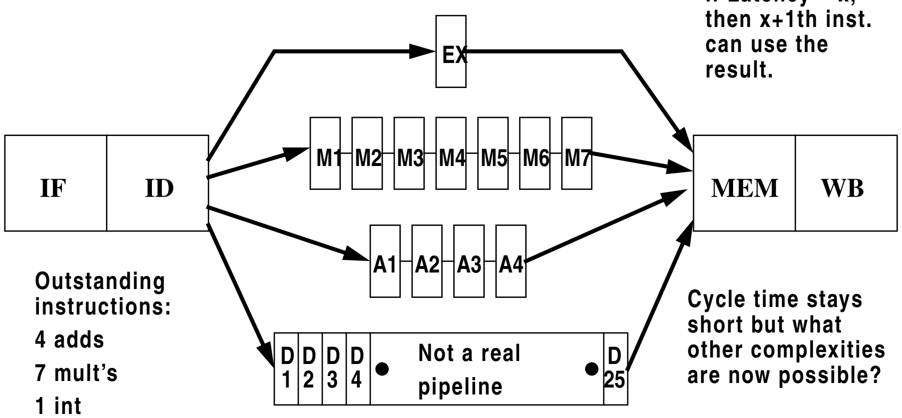
Functional Unit	Latency	Initiation Interval
Integer ALU	0	1
Data Memory (Integer and FP loads(1 less for store latency))	1	1
FP Add/SUB	3	1
FP & Integer Multiply	6	1
FP & Integer Divide + FP sqrt	24	24

Why?

DLX Pipe

Note # of stages are 1+ latency$_{EX}$

If Latency = x, then x+1th inst. can use the result.



Outstanding instructions:

4 adds

7 mult's

1 int

1 divide/sqrt

Cycle time stays short but what other complexities are now possible?

New Hazard and Forwarding Problems

- Structural hazards increase
  - unpiped Divide causes big 24 cycle hog zone
  - number of register writes in a cycle goes up - 3 FPR write contributors now
- WAW hazards now possible since instructions now reach WB out of order
- WAR hazards are still no problem since read happens early - e.g. in the ID stage
- out of order completion complicates exception handling
- RAW hazard induced stalls will be more frequent due to longer latency instructions

Was it worth it? How would you determine this?

New Structural Hazard Sources

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	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 F8, 0(R2)							IF	ID	EX	MEM	WB

- Scan columns for common resource requirements
  - at cycle 9: 3 requirements for MEM
    - doesn't matter since MULTD and ADD don't do anything in this stage
  - at cycle 11: we have a problem however - ideas??

Dealing with Structural Hazards

Consider the single write port FPR's in previous example

- Option 1:
  - keep track of issued instructions and when they will write-back to the FPR's
  - stall instruction in ID if there is a collision
  - just takes a 1 bit - 25 stage shift register for all 3 pipes
  - 3 injection points and the same 3 test points
- Option 2:
  - stall instructions at MEM entry
    - may also want to give preference to longest latency
    - longest is most likely to cause RAW hazard delays anyway
  - problem is that the control path has to go all the way back to the front of the pipe!! Costly.

Consider RAW Hazard Stalls

long latency pipes cause frequency to go up

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
LD F4,0(R2)	IF	ID	EX	M	WB										
MULTD F0, F4, F6		IF	ID	St.	M1 F4	M2	M3	M4	M5	M6	M7 F0	M	WB		
ADDD F2, F0, F8			IF	St.	ID	St.	St.	St.	St.	St.	St.	A1	A2	A3	A4
SD F2, 0(R2)					IF	St.	St.	St.	St.	St.	St.	ID	EX	St.	St.

Note: Figure 3.46 in text is wrong

F2

finally on cycle 16 SD gets to enter Mem
EX doesn't need to stall since it's the EFA calculation which uses R2

New Hazard Sources

- Dependencies between GPR and FPR
  - due to MOVI2FP and MOVFP2I instructions
- Avoid all by adding ID stage issue checks
  - Structural
    - repeat interval check
    - make sure register write port is available when needed
  - RAW
    - list all pending destination registers
    - don't issue a source from a pending destination until it clears (value is available via forwarding logic)
  - WAW
    - use same list of pending destination registers
    - don't issue a new pending destination which matches an existing one until it has finished WB
    - can we do better than this? (hint: consider forwarding requirements)

Precise Exceptions and Long Pipes

- ❑ Consider

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

- ❑ Piece of cake - no dependencies
- ❑ Not!
  - both ADDF and SUBF will complete before DIVF
  - and what happens when DIVF causes an exception?
- ❑ Ideas?

4 Precision Possibilities

1 & 2

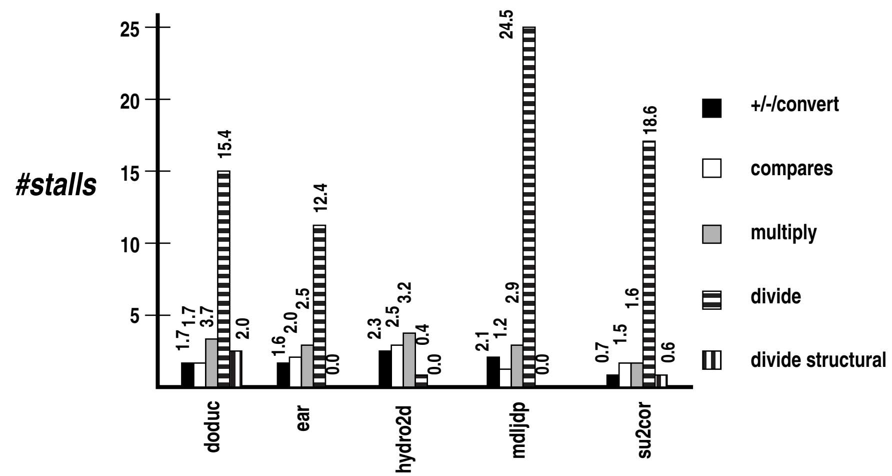
- ❑ Punt on precision
  - old supercomputer trick
  - not really viable today with IEEE standard exception handling and virtual memory
- ❑ Buffer something - 2 variants
  - Future File
    - buffer results at commit - post them in instruction order
  - History File
    - post ASAP
    - buffer original operands and roll-back to proper state if an exception occurs
  - bypassing is still required
    - so both get more expensive as pipelines get longer

Possibilities 3 & 4

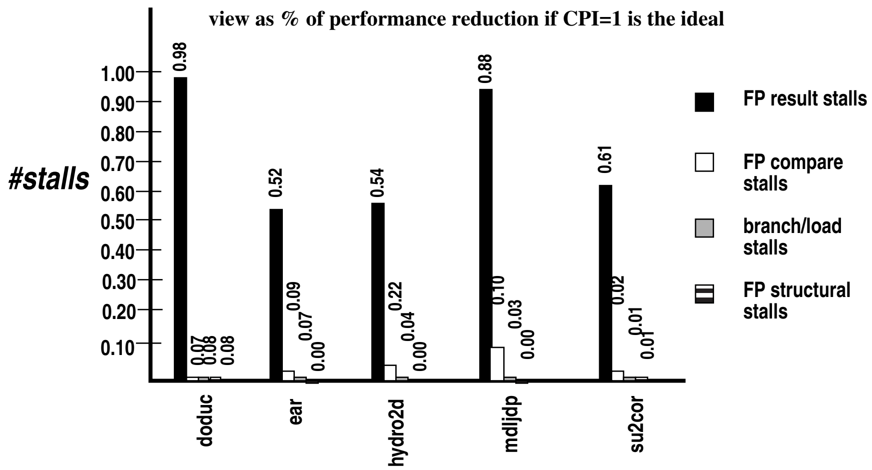
- ❑ Go imprecise with SW fixup
  - keep enough state around to do fixup
  - let SW emulate the instructions that are not yet finished but prior to the excepting instruction.
  - in general this is too hard
    - if the only uncompleted prior ones are FP
    - then the case is more tractable
- ❑ Stall issue until previous ones commit
  - improvement can be made by moving commit point forward in the pipe
    - realistically it can’t be moved very far forward
  - used by MIPS R2K to R4K and in the Pentium

DLX Pipeline Performance

stalls per FP operation



Total Stalls/Instruction



DLX FP Performance Averages

basis: SpecFP benchmarks

- Stalls per FP operation = approx. 50% of latency
  - mechanism to identify the chief culprit
    - 1.7 cycles for add/sub/convert or 56% of the 3 cycle latency
    - 2.8 for multiply or 40% of the 7 cycle latency
    - 14.2 for divide or 57% of the 25 cycle EX latency
- Total stalls
  - varies with application
    - Range from .65 for su2cor to 1.21 for doduc
    - Average over FP SPEC benchmarks is .87 per instruction
    - Note this is for all instructions executed - hence CPI = ideal CPI + .87
    - Since divides are rare (the compiler tries to avoid them if possible too) - the divide structural hazard component is small (doduc is the exception)
    - Major contributor is RAW result wait

ISA Design and Pipeline Complexity

Things you can do wrong

- Variable instruction lengths and CPI's
  - imbalance will cause stall frequency to increase
  - caches always do this but initial opinion was to freeze entire pipe
  - newer architectures playing with several non-freeze options
  - cache performance offsets the cost typically
- Sophisticated address modes
  - trashing registers during EFA calculation means state must be saved
  - as would be the case in auto-increment or auto-decrement modes

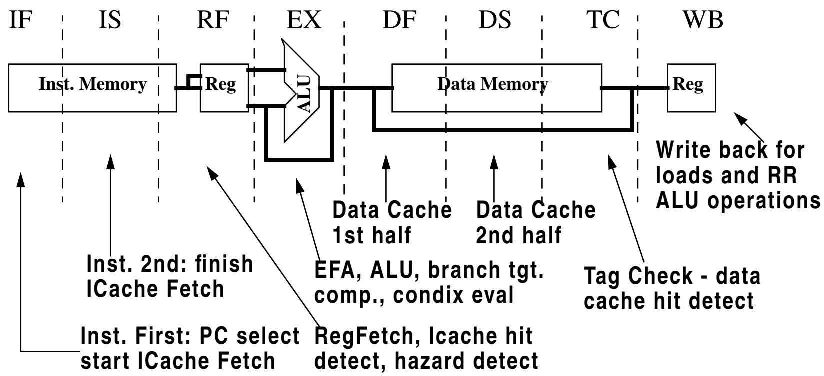
More Things You Can Do Wrong

- Permit self-modifying code (ala the 80x86)
  - what if instruction in the pipe is overwritten
  - then restarting becomes a bit of a bummer
  - old value must be saved
  - 80x86 takes extra undecoded instruction all the way to commit
- Non-uniform Implicitly set condition codes
  - Ugh
  - newer machines use uniform stage set
  - plus explicit bit in the instruction to enable set-CC

The MIPS R4000 Pipeline

❑ Real 64-bit machine

- ran between 100MHz and 200MHz
- deeper pipelining (sometimes called *superpipelining*)
- instruction set very similar to the DLX



Things to notice

❑ Load delays are 2 cycles

- 2 instructions/bubbles must be between a load and a dependent instruction
- value available at end of DS ==> use before tag check
  - Whoa seems odd since you have to *take it back* on a miss
  - but miss is rare (less than 10%) so common case is a hit
  - and dependent instructions will be behind so stalling them is easy
  - they can't do a destructive operation until after you know that it was a hit

Also Notice

❑ 3 cycle branch delay

- 3 instructions between branch and target instruction
- target is only really known after the EX stage
- 1st delay slot happens no matter what
- slots 2 and 3?
  - filled if predict-not-taken
  - NOP's if predict-taken

R4000 FP Pipeline

8 types of stages
3 functional units: FP-DIV, FP-MUL, FP-ADD
Adder used on final step of MUL and DIV

Stage	Functional Unit	Description
A	FP Add	Add Mantissa
D	FP Divide	Divide stage
E	FP Multiplier	Exception test
M	FP Multiplier	First Multiply stage
N	FP Multiplier	Second Multiply stage
R	FP Adder	Rounding
S	FP Adder	Operand Shift
U	all 3 functional units	Unpack FP numbers

FP Operations in the R4000

FP Instruction	Latency	Initiation Interval	Pipe stage sequences
Add, Sub	**4**	**3**	**U, S+A, A+R, R+S**
Multiply	**8**	**4**	**U, E+M, M, M, M, N, N+A, R**
Divide	**36**	**35**	**U, A, R, D^{28}, D+A, D+R, D+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**

Can you figure out why this is the case?
Sort of a start on the homework!

R4000 Performance

Benchmark	Pipe CPI	Load Stalls	Branch Stalls	FP Res. Stalls	FP Struc. Stalls
compress	1.20	0.14	0.06	0.00	0.00
eqntott	1.88	0.27	0.61	0.00	0.00
espresso	1.42	0.07	0.35	0.00	0.00
gcc	1.56	0.13	0.43	0.00	0.00
li	1.64	0.18	0.46	0.00	0.00
INTEGER AVERAGE	**1.54**	**0.16**	**0.39**	**0.00**	**0.00**
doduc	2.84	0.01	0.22	1.39	0.22
mkljdp2	2.66	0.01	0.31	1.20	0.15
ear	2.17	0.00	0.46	0.59	0.12
hydro2d	2.53	0.00	0.62	0.75	0.17
su2cor	2.18	0.02	0.07	0.84	0.26
FP AVERAGE	**2.48**	**0.01**	**0.33**	**0.95**	**0.18**
OVERALL AVERAGE	**2.00**	**0.10**	**0.36**	**0.46**	**0.09**

Some things to observe

- unexpected (e.g. branch) may cause problems
  - if compiler was supposed to generate NOP's then OOPS
  - HW should detect hazards anyway
- Pipelining cannot be done arbitrarily
  - laminarity becomes a problem
  - stalls tend to increase
  - exception and branch penalties go up
  - there is a diminishing return point
- performance of unoptimized code is misleading
  - especially true of pipelined machines
  - instruction scheduling removes most of the stalls

Easy huh? On to ILP and multiple issue/pipelines