Pipeline Complexities

Today's topics:

Hazards & forwarding details

Distributed vs. Centralized control

Out of order completion issues

Exceptions

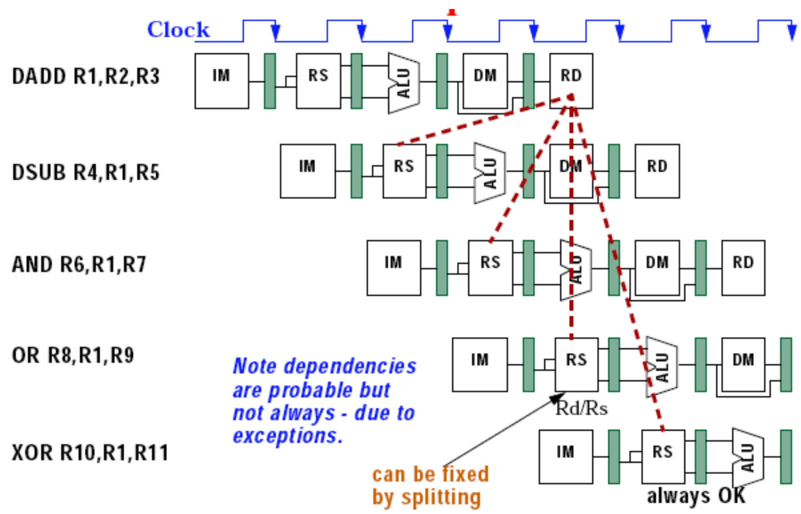
Pipeline Hazards

- **Types**
  - structural, data, and control
- **Pathological code snippet**

```
DADD    R1, R2, R3         R1 <-- R2 + R3
DSUB    R4, R5, R1
DAND    R6, R1, R7         yep - R1 gets produced
OR      R8, R1, R9         in the first instruction
XOR     R10, R1, R11       and used in every
                           subsequent instruction
                           maximal illustration
```

  - could be worse
    - » branches, long latency memory operations, exceptions
    - » stick w/ reg-reg as a start

Stage Resource View



Forward/Bypass

- **Simple concept – somewhat hairy w.r.t. control**
  - **key idea is track where and when value is valid**
    - » **ALU value is valid at the end of stage 3**
    - » **MEM value valid at end of stage 4**
      - **assumes L1 hit in 1 cycle**
        - – **reality is it could take ~3 cycles but assume 1 for now**
- **Control path responsibility**
  - **keep track of what is known when**
    - » **move data through mux paths to correct place to minimize stalls**
      - **select lines to appropriate mux at the right time**
  - **2 options**
    - » **centralized vs. distributed**
      - **central**
        - – **long wires but easier validation**
      - **distributed**
        - – **shorter wires, harder validation, state moves through pipeline**

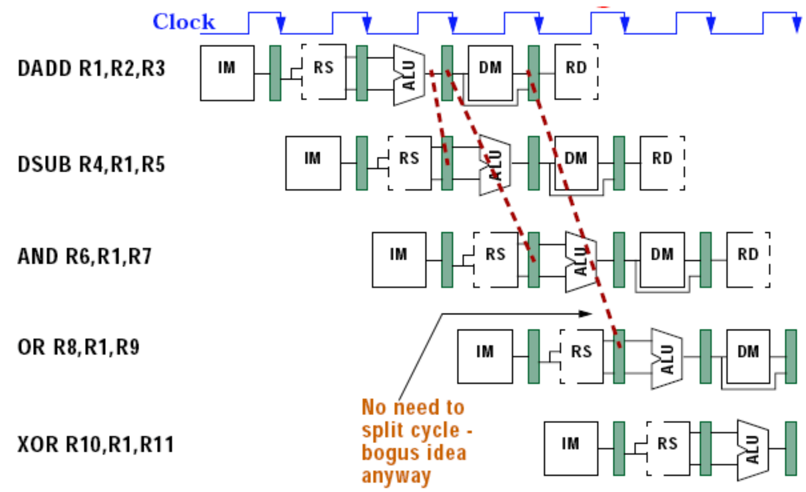
Distributed Control

- Tag and compare
  - add reg name tags and valid bits to pipeline registers
    - » value is associated with Rd before it is actually placed in WB stage
    - » valid bit set when value is produced
- 3 sources of Rs slot data
  - ALUout, ID/EX from the register, MEMout
  - clearly want the latest version → priority established
- Theory vs. reality
  - easy conceptually
  - but time marches on
    - » compare valid, and mux delays consume time
    - » increasingly difficult laminarity issue at increased frequencies
      - 3 GHz → 333 ps budget

Centralized Control

- Scoreboard state
  - knows where each instruction is in the pipe
    - » e.g. when value becomes valid
  - directly controls mux select lines
    - » at the right time
- Value not available
  - stall appropriate stages
- Distributed/Centralized hybrid
  - in reality some aspects of both employed
    - » depends on frequency and core complexity
      - lots of simple cores favor control
      - big & complex favor distributed

Result w/ Forwarding



Hazard Forms

- **Instruction i occurs before j**
  - **RAW – read after write**
    - » j reads before i writes → j gets incorrect *old* value
  - **WAW – write after write**
    - » j writes before i writes → breaks semantic order
      - not a problem in simple 5 stage MIPS pipe YET
  - **WAR – write after read**
    - » j writes before i reads → i gets incorrect *too new* value
      - no problem in simple 5 stage MIPS since writing occurs late in piple and reads happen early in the pipe
  - **WAW & WAR → Wax**
    - » terminology
  - **RAR – read after read**
    - » not a hazard since producer-consumer data dependency not violated
- **SW or HW fix – reorder instructions or NOP's**
  - **same idea different mechanism**

Additional Problems

- **Unknown memory latencies & speculation failure**
  - compiler can't predict this so HW mechanism required
- **Bubbles or Stalls will happen in the worst case**
  - distributed control
    - » pipeline stall blocks – predicate advance based on memory return or any speculative event
      - more complex for out of order memory returns (more later)
  - centralized control
    - » effectively the same
      - predicate "OK to advance" becomes an input to the FSM based control
- **Amdahl's Law – who are the main culprits?**
  - depends on the code but in general (no particular order)
    - » cache misses – high for TPC
    - » branch mispredicts – high for gcc
    - » small basic blocks with tight dependencies – high for eqntott

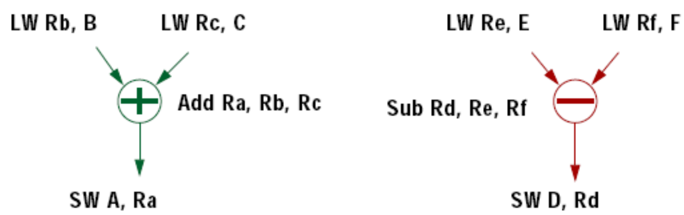
Instruction Scheduling

- **Key to exploiting ILP**
  - lots more next but preview now
- **Compiler**
  - IR is a partial order
    - » static dataflow
  - knows pipeline structure of target machine
  - reorder instructions to minimize stalls
    - » several downsides
      - compiler must be correct → conservative
      - increased register pressure to reduce Wax induced stalls
        - – renamed pool helps until your run out
          - – problem is register "liveness" is not an exact science
      - increases compiler complexity and time
        - – usually compile time considered free
          - – unless you're in compile almost once mode
            - – also called student mode
          - – or where compile made hard to shorten XEQ time
            - – e.g. COSMOS

Simple Dataflow Example

Consider the lowly
A = B + C
D = E - F

Simple Expression Trees
==> dependency digraph



Issue order choice proceeds via
simple cut set analysis
Optimal schedule requires look ahead and HAIR

Pipeline Control

- **After ID stage – finally know what's up**
  - **compiler may not give the HW the full picture**
    - » **typically doesn't – welcome to HW optimization limits**
  - **forward or stall?**
    - » **mitigated by predicates**
- **Situations:**

Situation	Sample Code	Action Required

No dependence	LW **R1**,45(R2)
	ADD R5, R6, R7
	SUB R8,R6,R7,
	OR R9,R6,R7
		No hazard since R1 doesn't show up in the next 3 instructions

Dependence requiring stall	LW **R1**,45(R2)
	ADD R5,**R1**,R7
	SUB R8,R6,R7,
	OR R9,R6,R7
		comparators must detect the use of R1 in ADD and stall since LW can only produce the value after cycle 4 and ADD needs it after cycle 2 - hence no way (note from LW's perspective ADD needs value after cycle 3 - since ADD got started one cycle later)

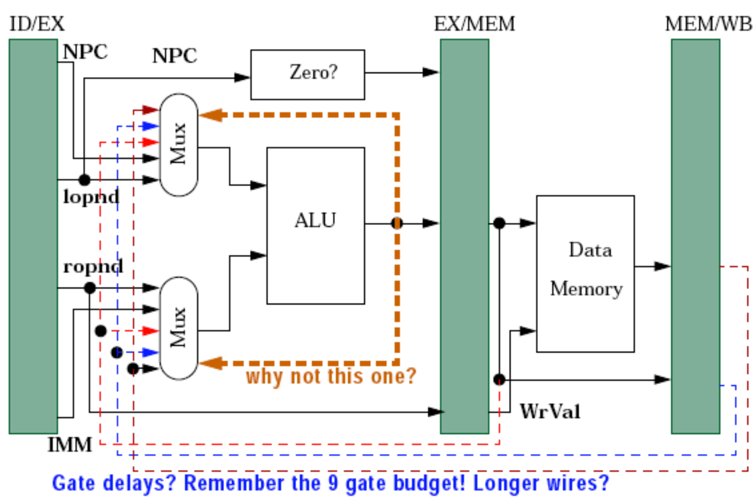
Dependence can be handled with forwarding	LW **R1**,45(R2)
	ADD R5, R6, R7
	SUB R8,**R1**,R7,
	OR R9,R6,R7
		comparators detect R1 use in SUB and setup forward of result to ALU from end of DM stage

Dependence but register file order splitting makes in a non-issue	LW **R1**,45(R2)
	ADD R5, R6, R7
	SUB R8,R6,R7,
	OR R9,**R1**,R7
		R1 will be written in the first half of WB
		R1 will be read in the second half of Idecode for OR
		Hence no problem - be happy in an older world

Responsibility

- Compiler – HW or both
  - track dependencies
    - » register centric viewpoint
- Overhead
  - forwarding → compare & mux → delay
  - mux rule
    - » increasing fan-in increases delay
      - decode select signals
      - fall-through delay of a wider mux
  - comparator rule
    - » quadratic w/ # of stages
- Delay
  - always a problem and must be carefully tracked
    - » more difficult when clock frequency increases
    - » designers need to stay within an F04 delay budget

EX Stage Mux Example



Register Source Compares

to support full forwarding in the MIPS

Source Pipe Reg.	Opcode of Source Inst.	Dest. Pipe Reg	Opcode of Dest. Inst.	Destination of forwarded result	Compare if EQ then forward
EX/MEM	Reg-Reg ALU	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	Rd(EX/MEM.IR$_{16..20}$) = Rs1(ID/EX.IR$_{6..10}$)
EX/MEM	Reg-Reg ALU	ID/EX	Reg-Reg ALU	Bottom ALU input	Rd(EX/MEM.IR$_{16..20}$) = Rs2(ID/EX.IR$_{11..15}$)
MEM/WB	Reg-Reg ALU	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	Rd(MEM/WB.IR$_{16..20}$) = Rs1(ID/EX.IR$_{6..10}$)
MEM/WB	Reg-Reg ALU	ID/EX	Reg-Reg ALU	Bottom ALU input	Rd(MEM/WB.IR$_{16..20}$) = Rs2(ID/EX.IR$_{6..10}$)

ALU-Immediate Compares

Source Pipe Reg.	Opcode of Source Inst.	Dest. Pipe Reg	Opcode of Dest. Inst.	Destination of forwarded result	Compare if EQ then forward
EX/MEM	ALU Immediate	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	Rd(EX/MEM.IR$_{11..15}$) = Rs1(ID/EX.IR$_{6..10}$)
EX/MEM	ALU Immediate	ID/EX	Reg-Reg ALU	Bottom ALU input	Rd(EX/MEM.IR$_{11..15}$) = Rs2(ID/EX.IR$_{11..15}$)
MEM/WB	ALU Immediate	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	Rd(MEM/WB.IR$_{11..15}$) = Rs1(ID/EX.IR$_{6..10}$)
MEM/WB	ALU Immediate	ID/EX	Reg-Reg ALU	Bottom ALU input	Rd(MEM/WB.IR$_{11..15}$) = Rs2(ID/EX.IR$_{11..15}$)

Load Source Compares

Source Pipe Reg.	Opcode of Source Inst.	Dest. Pipe Reg	Opcode of Dest. Inst.	Destination of forwarded result	Compare if EQ then forward
MEM/WB	Load	ID/EX	Reg-Reg ALU ALU-imm, LD, ST, branch	Top ALU input	Rd(MEM/WB.$IR_{11..15}$) = Rs1(ID/EX.$IR_{6..10}$)
MEM/WB	Load	ID/EX	Reg-Reg ALU	Bottom ALU input	Rd(MEM/WB.$IR_{11..15}$) = Rs2(ID/EX.$IR_{11..15}$)

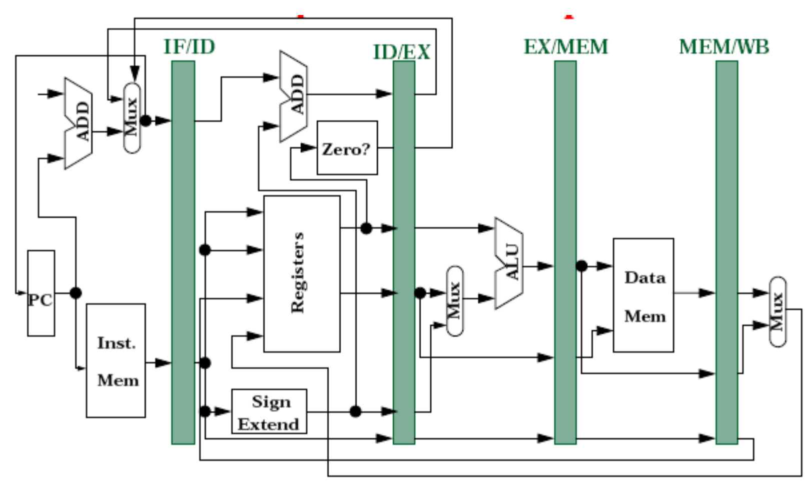
Control Hazards

- More evil than data hazards
  - since forwarding doesn't help
- Need 3 things – 2 happen late in the pipeline
  - branch target
    - » PC+4 if branch not taken or address (computed or immediate)
  - condition true?
    - » output of zero unit in MIPS
    - » condition code, … in other architectures
  - decode stage recognizes a branch or jump
- Result
  - IF of wrong instruction has already started
- Simple MIPS pipeline has 3 cycle branch delay penalty
  - effective address not known until EX
  - condition set in MEM (stage 4)
    - » 3 branch delay slots

Branch Delay Reduction

- **Hardware**
  - **compute address and zero detect earlier in the pipeline**
    - » **additional ALU**
      - **BTA (branch taken address) can be computed in ID**
      - **PC+4 already in ID**
      - **move zero detect earlier in the pipe – e.g. ID**
    - » **result: 1 branch delay slot**
- **Depends on proper ISA choice**
  - **MIPS: BEQZ, BNEZ**
    - » **allows the condition to be a simple zero detect**
    - » **which can be determined in the ID stage**

Improved MIPS Pipeline



See any potential problem?

Taken vs. Not Taken

- **Probability of a branch**
  - 11-17% branch, 2-8% jumps for MIPS, & 8086
    - » note this is for single issue
- **What did the source code look like**
  - if-then-else – 50% chance of going either way?
  - loops – branch is the common case >90%
  - bit or flag test
    - » usually to check for an error condition – taken rarely
- **Other possibilities for the HW**
  - backward vs. forward branch
    - » loops are backward
  - what happened last time
    - » Baskett bit – first branch prediction idea
  - pragma's
    - » user or compiler hints about program dynamics

Control Hazard Avoidance

- **Easy but slow**
  - freeze pipe until you know for sure
    - » same as NOP insertion
  - negates the whole idea of pipelining
- **Use some form of branch prediction**
  - details next week
  - prediction will always fail sometime
    - » must prevent destructive change until outcome is known
      - destructive = write to register or memory
- **2 options**
  - wait until you know
    - » write value temporarily held until commit
      - see any issues with forwarding?
  - just write and then back up if you have to
    - » what's the problem here?

Other Options

- **Delayed Branch**
  - fill delay slots
    - » since these happen before the branch happens anyway
      - view the branch as being delayed
    - » compiler schedules instructions or generates NOPs
- **Additional tactic**
  - nullify delay slots (text term is “cancel”)
  - if prediction correct then things just move along
  - if not
    - » all or some of the delay slots get nullified
    - » consider 5 stage MIPS
      - destructive ops happen in stage 4 & 5
      - if branch resolves in stage 2 or 3
        - – plenty of time to cancel the 1 or 2 delay slots prior to their arrival at 4 & 5
  - common practice in HP’s PA architectures
    - » which morphed eventually into the IA64 (Itanium)
      - although predication in IA64 is at the lunatic fringe

MIPS Wasted Delay Slots

- **Represents 2 – 17% of total instructions**
  - so actual impact to IPC/CPI is 2-17% of the waste
    - » numbers vary radically w/ branch penalty, speculation level, pipeline, and branch predictor

Code	Total Wasted	Empty Slots	Cancelled
compress	30%	18%	12%
eqntott	35%	24%	11%
espresso	34%	30%	4%
gcc	26%	15%	11%
li	46%	19%	27%
doduc	40%	34%	6%
ear	41%	37%	4%
hydro2d	16%	1%	15%
mdljdp	8%	1%	7%
su2cor	17%	7%	10%

SPECint

SPECfp

CPI Effect

- **With ideal CPI=1 and stalls = freq x penalty**
  - **note penalty = 3 is bad but =1 or less is about the same**

$$\text{Pipeline Speedup} = \frac{\text{Pipeline Depth}}{1 + \text{Branch Frequency} \times \text{Branch Penalty}}$$

Scheduling Scheme	Branch Penalty	Effective CPI	Pipeline Speedup over Non-piped Version	Pipeline Speedup over Stall Pipe on Branch Scheme
Stall pipeline	3	1.42	3.5	1.0
Predict Taken	1	1.14	4.4	1.26
Predict Not Taken	1	1.09	4.5	1.29
Delayed Branch	0.5	1.07	4.6	1.31

5 Axis Exception Model

- **Sync vs. Async**
  - **synch – associated with a particular instruction**
    - » **handler replaces instruction and then retries or aborts**
  - **asynch – instruction independent (e.g. OS timeout)**
    - » **flush pipe and then handle**
    - » **power fail: may not have time to do a complete flush**
- **Code requested vs. coerced**
  - **req'd is predictable and can happen after instruction**
- **Maskable or not**
  - **arith. overflow: the code can care or not**
- **Within vs. between instructions**
  - **similar to sync/async w/ small difference**
- **Resume vs. terminate program**
  - **handle and resume, OR**
  - **fatal exception just terminates (e.g. segment error)**

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

Within & Resume: Biggest Problem

- Shut down pipe safely
  - e.g. complete instructions before exception
    - » PC of restart point must be saved
      - common PC and PC+4 saved
      - retry resume or resume on next instruction
- MIPS strategy
  - set PC to start of handler and fetch
  - nuke/nullify instructions after excepting instruction
  - handle exception and then resume at the right spot
- OOPS – delayed branches (assume 2 delay slots)
  - 1st delay slot generates a page fault
  - 2nd instruction nuked and is restart point
  - then next and no branch happens
  - → save delay slot size +1 of PC's
    - » plus keep state of whether branch was taken or not

Exceptions

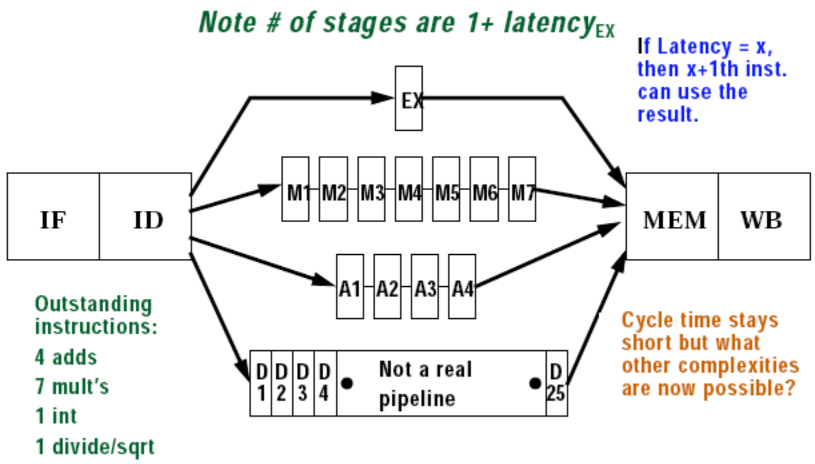
- **The ultimate pain**
  - semantic model guarantee
    - » instructions happen in order
  - problem
    - » instruction order and exception order happen out of order
- **Precise exception model**
  - rule
    - » all instructions before excepting instruction complete
    - » instructions after don't happen
    - » excepting instruction handling varies
      - w/ exception type
  - reality
    - » conservatism is slow
    - » typical is settable precise or not control
      - precise when debugging
      - non-precise when your code works
        - – done laughing yet?

Exceptions by Stage: Simple MIPS

- **IF**
  - page fault, TLB fault, misaligned address, mem protection violation
- **ID**
  - undefined or illegal opcode
- **EX**
  - arithmetic exception – overflow, underflow, NaN, …
- **Mem**
  - page fault, TLB fault, misaligned address, mem protection violation
- **WB**
  - none
- **Result – on any cycle 4 exceptions could occur**
  - rule – handle first one in program order

EX stage isn't just a single pipeline

- **Int +/- can be 1 cycle**
  - **all FP instructions and any mult/div/sqrt will take longer**



Latency & Repeat Interval

- **Latency – number of cycles to generate value**
  - **w/ forwarding defined in your text to be the number of intervening instructions**
  - **hence 0 means next instruction can consume the result**
- **Repeat/Initiation interval**
  - **how often can you issue another one of this type of instruction**
    - » **defined in cycles – 1 means next cycle**
- **Example**

XU	Latency	Initiation Interval
Inteter ALU	0	1
Loads	1	1
FP +/-	3	1
FP/Int Mult	6	1
FP/Int Div/SQRT	24	24 (why?)

Increased Hazard Complexity

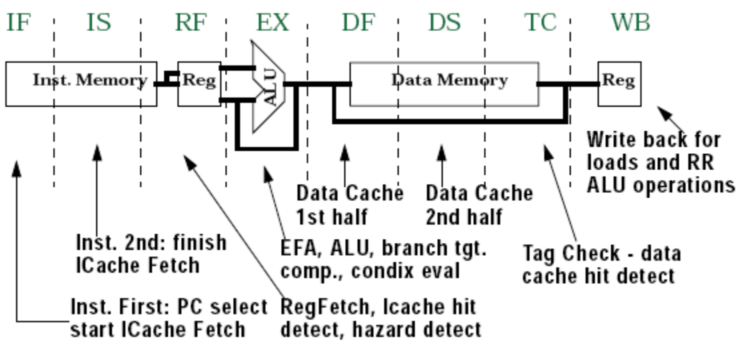
- **Out of order completion**
  - **WAW becomes harder**
    - » since earlier write may complete later than later write
  - **RAW hazards become more frequent**
    - » due to increased latency of some instructions
    - » increases the load use delay separation
  - **WAR not a problem**
    - » since reads happen in ID and not influenced by expanded EX pipeline issues
- **Exceptions**
  - **oh crap!**
    - » lots more possible on any given cycle
      - need to keep track of program order
  - **reorder buffer**
    - » more later
- **Is complexity worth it – calculate and find out**

Things you can do wrong

- **Sophisticated address modes**
  - **trashing registers during EFA calculation → state save required → more registers or higher register pressure**
    - » fill and spill to memory is expensive in time
    - » e.g. auto-increment or decrement
- **Permit self-modifying code (ala 80x86)**
  - **overwrite an instruction in the pipeline**
    - » exception and restart a different instruction
    - » oops
- **Implicitly set condition codes**
  - **later instruction sets code**
  - **earlier but finishes later instruction sets code**
  - **branch comes along and uses the stale condition**
  - **fix?**

MIPS R4000 Pipeline

- **Real 64-bit machine**
  - ran between 100-200 MHz
    - » still used in embedded world
  - deeper pipe but very similar ISA



Take Home Wisdom

- **Pipelining**
  - simple concept – arbitrarily hard to get right in reality
- **Things will get even harder**
  - superscalar – multiple issue per cycle
  - deeper pipelines to increase frequency
    - » laminarity and stall probability problems increase
  - compiler instruction scheduling gets trickier
    - » can the hardware make up some of the slack?
      - yes but it's complicated
- **Late 80's**
  - improved performance ran out of gas
  - multiple issue saves the 90's (ILP)
  - multiple cores saves the next decade – TBD?
    - » TLP affects the programmer
    - » pipelining and ILP didn't
      - for the most part