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
- DSUB R4, R5, R7
- DAND R6, R1, R7
- OR R8, R1, R9
- XOR R10, R1, R11

- yap - R1 gets produced in the first instruction and used in every subsequent instruction

- 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 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
    * centralised 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 \(\rightarrow\) 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 \(\rightarrow\) 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 \(\rightarrow\) \(j\) gets incorrect old value
    - **WAW** – write after write
      - \(j\) writes before \(i\) writes \(\rightarrow\) breaks semantic order
      - not a problem in simple 5 stage MIPS pipe \(\text{YET}\)
    - **WAR** – write after read
      - \(j\) writes before \(i\) reads \(\rightarrow\) \(i\) gets incorrect too new value
        - no problem in simple 5 stage MIPS since writing occurs late in pipe and reads happen early in the pipe
    - **WAW & WAR** \(\rightarrow\) **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 Wen induced stalls
        - warmed pool helps until you run out
        - problem to register “freeze” is not an exact science
      - increase compiler complexity and time
        - usually compile time considered free
        - increasing compiler complexity and time
          - e.g. COSMOS

Instruction Scheduling

- Simple Dataflow Example

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

Simple Expression Trees
--- dependency digraph

<table>
<thead>
<tr>
<th>LW Rb, B</th>
<th>LW Rc, C</th>
<th>LW Re, E</th>
<th>LW Rf, F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Ra, Rb, Rc</td>
<td>Sub Rd, Re, Rf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW A, Ra</td>
<td>SW D, Rd</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Instruction Scheduling

- 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

<table>
<thead>
<tr>
<th>Situation</th>
<th>Sample Code</th>
<th>Action Required</th>
</tr>
</thead>
</table>
| FP operations | 0, 0, 0, 0 | Store results of FP operation to data bus 
| Operations requiring MEM | 0, 0, 1, 1 | Execute load instruction, load data to data bus 
| Operations not to be reordered | 0, 0, 1, 1 | Execute load instruction, load data to data bus 
| Operations to be reordered | 0, 0, 1, 1 | Execute load instruction, load data to data bus 
| Operations that might be reordered | 0, 0, 1, 1 | Execute load instruction, load data to data bus 
| Operations that might be reordered | 0, 0, 1, 1 | Execute load instruction, load data to data bus 
| Operations that might be reordered | 0, 0, 1, 1 | Execute load instruction, load data to data bus 

Page 3
Responsibility

- Compiler – HW or both
  - track dependencies
  - register centric viewpoint

- Overhead
  - forwarding \( \rightarrow \) compare \& mux \( \rightarrow \) 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

Register Source Compares

|------------------|------------------------|-----------------|-----------------------|-------------------------------|----------------------------|
| REGMEM REG REG ALU | 010 | REG REG ALU | 010 | Top ALU input | REG=REGALU x REG, REGALU x REG,
| REGMEM REG REG ALU | 010 | REG REG ALU | 010 | Bottom ALU input | REG=REGALU x REG, REGALU x REG,
| MEMREG REG REG ALU | 010 | REG REG ALU | 010 | Top ALU input | REG=REGALU x REG, REGALU x REG,
| MEMREG REG REG ALU | 010 | REG REG ALU | 010 | Bottom ALU input | REG=REGALU x REG, REGALU x REG,
| MEMMEM REG REG ALU | 010 | REG REG ALU | 010 | Top ALU input | REG=REGALU x REG, REGALU x REG,
| MEMMEM REG REG ALU | 010 | REG REG ALU | 010 | Bottom ALU input | REG=REGALU x REG, REGALU x REG,

ALU-Immediate Compares

|------------------|------------------------|-----------------|-----------------------|-------------------------------|----------------------------|
| EXMEM ALU Immediate | 010 | Top ALU input | 010 | REG=REGALU x REG,
| EXMEM ALU Immediate | 010 | Top ALU input | 010 | REG=REGALU x REG,
| MEMREG ALU Immediate | 010 | Top ALU input | 010 | REG=REGALU x REG,
| MEMREG ALU Immediate | 010 | Top ALU input | 010 | REG=REGALU x REG,
| MEMMEM ALU Immediate | 010 | Top ALU input | 010 | REG=REGALU x REG,
| MEMMEM ALU Immediate | 010 | Top ALU input | 010 | REG=REGALU x REG,
### Load Source Compares

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMWR Load</td>
<td>IDX</td>
<td>Reg Reg ALU</td>
<td>LJ, branch</td>
<td>Reg ALU input</td>
<td></td>
</tr>
<tr>
<td>MMMWR Load</td>
<td>IDX</td>
<td>Reg Reg ALU</td>
<td>Bottom ALU input</td>
<td>Reg (IWR,HWR, R16, R16)</td>
<td></td>
</tr>
</tbody>
</table>

### 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

![Diagram of 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
  - Basket bit - first branch prediction idea
  - pragmas
  - user 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 branches 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
      - 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

<table>
<thead>
<tr>
<th>Code</th>
<th>Total Wasted</th>
<th>Empty Slots</th>
<th>Cancelled</th>
</tr>
</thead>
<tbody>
<tr>
<td>compas</td>
<td>26%</td>
<td>12%</td>
<td>12%</td>
</tr>
<tr>
<td>sqrt</td>
<td>13%</td>
<td>14%</td>
<td>11%</td>
</tr>
<tr>
<td>reprex</td>
<td>23%</td>
<td>26%</td>
<td>10%</td>
</tr>
<tr>
<td>pr</td>
<td>36%</td>
<td>15%</td>
<td>35%</td>
</tr>
<tr>
<td>s</td>
<td>48%</td>
<td>16%</td>
<td>27%</td>
</tr>
<tr>
<td>dosbc</td>
<td>48%</td>
<td>16%</td>
<td>30%</td>
</tr>
<tr>
<td>nor</td>
<td>41%</td>
<td>27%</td>
<td>4%</td>
</tr>
<tr>
<td>hybrid</td>
<td>16%</td>
<td>6%</td>
<td>13%</td>
</tr>
<tr>
<td>dup</td>
<td>6%</td>
<td>1%</td>
<td>5%</td>
</tr>
<tr>
<td>sum</td>
<td>33%</td>
<td>7%</td>
<td>16%</td>
</tr>
</tbody>
</table>
CPI Effect

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

<table>
<thead>
<tr>
<th>Stalling Scheme</th>
<th>Branch Penalty</th>
<th>Effective CPI</th>
<th>Pipeline Speedup over Non-piped Version</th>
<th>Pipeline Speedup over Stall Pipe on Branch Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>1</td>
<td>1.47</td>
<td>1.5</td>
<td>1.6</td>
</tr>
<tr>
<td>Pred.Taken</td>
<td>1</td>
<td>1.14</td>
<td>1.4</td>
<td>1.25</td>
</tr>
<tr>
<td>Pred. Not Taken</td>
<td>1</td>
<td>1.08</td>
<td>1.5</td>
<td>1.35</td>
</tr>
<tr>
<td>Delayed Branch</td>
<td>0.5</td>
<td>1.67</td>
<td>1.6</td>
<td>1.31</td>
</tr>
</tbody>
</table>

5 Axis Exception Model

- Sync vs. Async
  - synch – associated with a particular instruction
    - handler replaces instruction and then retries or aborts
  - async – 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

<table>
<thead>
<tr>
<th>Exec. Type</th>
<th>Synchronization</th>
<th>Requested</th>
<th>Forced</th>
<th>maskable</th>
<th>within</th>
<th>resume-terminate</th>
</tr>
</thead>
<tbody>
<tr>
<td>3D Device Req</td>
<td>Sync</td>
<td>Requested</td>
<td>Non-maskable</td>
<td>within</td>
<td>Resume-terminate</td>
<td></td>
</tr>
<tr>
<td>invasive CS int.</td>
<td>Sync</td>
<td>User Requested</td>
<td>Non-maskable</td>
<td>within</td>
<td>Resume-terminate</td>
<td></td>
</tr>
<tr>
<td>Truncation</td>
<td>Sync</td>
<td>User Requested</td>
<td>Maskable</td>
<td>within</td>
<td>Resume-terminate</td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>Sync</td>
<td>Control</td>
<td>Maskable</td>
<td>within</td>
<td>Resume-terminate</td>
<td></td>
</tr>
<tr>
<td>Pop Fault</td>
<td>Sync</td>
<td>Control</td>
<td>Non-maskable</td>
<td>within</td>
<td>Resume-terminate</td>
<td></td>
</tr>
<tr>
<td>Unimplemented addr</td>
<td>Sync</td>
<td>Control</td>
<td>Maskable</td>
<td>within</td>
<td>Resume-terminate</td>
<td></td>
</tr>
<tr>
<td>Mem. prot violation</td>
<td>Sync</td>
<td>Control</td>
<td>Non-maskable</td>
<td>within</td>
<td>Resume-terminate</td>
<td></td>
</tr>
<tr>
<td>Undefined int</td>
<td>Sync</td>
<td>Control</td>
<td>Non-maskable</td>
<td>within</td>
<td>Terminate-??</td>
<td></td>
</tr>
<tr>
<td>All error</td>
<td>Sync</td>
<td>Control</td>
<td>Non-maskable</td>
<td>within</td>
<td>Terminate-??</td>
<td></td>
</tr>
<tr>
<td>Power Failure</td>
<td>Sync</td>
<td>Control</td>
<td>Non-maskable</td>
<td>within</td>
<td>Terminate-??</td>
<td></td>
</tr>
</tbody>
</table>

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
    - non-precise when your code works
- w/ exception type
- 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 multi/div/sqrt will take longer
  \[ \text{Note \# of stages are 1+ latency}_{EX} \]
  - IF Latency \( x \), then \( x+1 \)th inst. can use the result.

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>Initiation Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int/er ALU</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Loads</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP +/-</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP/Int Mul</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP/Int Div/SQRT</td>
<td>24</td>
<td>24 (why?)</td>
</tr>
</tbody>
</table>
### Increased Hazard Complexity

- **Out of order completion**
  - WAW becomes harder
  - 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
    - laminitariness 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