Anti-Inspiration

The material in this appendix, which was considered reasonably advanced for graduate students when this text first appeared in 1990, is now considered basic undergraduate material and can be found in processors costing less than $10!

It's true hard work never killed anybody, but I figure, why take the chance?
Ronald Reagan, US President

Hard work. Well, that's all right for people who don't know how to do anything else.
From The Devil and Daniel Webster
Dan Totheroh
Inspiration

Genius is one per cent inspiration, ninety-nine per cent perspiration.

Thomas Alva Edison (1847–1931), U.S. inventor.

Misery loves company.

Donald Freed, U.S. screenwriter

CS/CE USNews Ranking Course/Text
1/4 Carnegie Mellon University ECE 741/H&P
1/1 Massachusetts Institute of Technology 6.823/H&P
1/2 Stanford University EE282/H&P
1/3 University of California–Berkeley CSE502/H&P
5/5 University of Illinois–Urbana–Champaign CS333/H&P
Pipeline control

- We have 5 stages. What needs to be controlled in each stage?
  - Instruction Fetch and PC Increment
  - Instruction Decode / Register Fetch
  - Execution
  - Memory Stage
  - Write Back

- How would control be handled in a CS Dept?
  - One professor telling everyone what to do?
  - Or maybe a robotic “control” professor?
  - Should we use a finite state machine?
Pipeline Control

- Pass control signals along just like the data
Data Hazard Types

- Types of data hazards (Instr\textsubscript{i} followed by Instr\textsubscript{j})
  - Read After Write (RAW)
    - Instr\textsubscript{j} tries to read operand before Instr\textsubscript{i} writes it
  - Write After Read (WAR)
    - Instr\textsubscript{j} tries to write operand before Instr\textsubscript{i} reads it
  - Write After Write (WAW)
    - Instr\textsubscript{j} tries to write operand before Instr\textsubscript{i} writes it

- Control must detect data hazards and stall or forward data as needed
- **Example:** Load interlock \(\rightarrow\) RAW hazard
  - Implementation must compare source and dest registers
  - Four situations to be handled by control logic

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

- ID→EX is called instruction issue
- Detect data hazards prior to instruction issue
  - Stall prior to issue, identify forwarding
  - Simplifies hw since state update has not occurred

<table>
<thead>
<tr>
<th>Opcode field of ID/EX (ID/EX.IR₀..₅)</th>
<th>Opcode field of IF/ID (IF/ID.IR₀..₅)</th>
<th>Matching operand fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Register-register ALU</td>
<td>ID/EX.IR[rₜ] == IF/ID.IR[rs]</td>
</tr>
<tr>
<td>Load</td>
<td>Register-register ALU</td>
<td>ID/EX.IR[rₜ] == IF/ID.IR[rt]</td>
</tr>
<tr>
<td>Load</td>
<td>Load, store, ALU immediate, or branch</td>
<td>ID/EX.IR[rₜ] == IF/ID.IR[rs]</td>
</tr>
</tbody>
</table>

If hazard detected: set ID/EX to no-op and re-circulate IF/ID
## Implementing Forwarding in EX

<table>
<thead>
<tr>
<th>Pipeline register containing source instruction</th>
<th>Opcode of source instruction</th>
<th>Pipeline register containing destination instruction</th>
<th>Opcode of destination instruction</th>
<th>Destination of the forwarded result</th>
<th>Comparison (if equal then forward)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX/MEM</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR[rd] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>EX/MEM.IR[rd] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>EX/MEM.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR[rt] == ID/EX.IR[rs]</td>
</tr>
</tbody>
</table>
Example Implementation for MIPS Hazard Detection

- Fast pipeline registers
- Comparators and combinational logic
- Enlarged multiplexors for ALU inputs (to “by-pass” the output)
Interrupts and Exceptions

• Exceptions or interrupts (names sometimes used synonymously)
  - Events other than branches or jumps that change the normal flow of instruction execution
  - Unexpected event from within the processor
    • Ex: arithmetic overflow, system call, undefined instruction
  - Event that causes an unexpected change in control flow but comes from outside the processor
    • Ex: I/O device request
**How to implement exceptions?**

- **Nonpipelined implementation**
  - Handle undefined instruction and arithmetic overflow
    1. Must save address of offending instruction
    2. Transfer control to OS at some address
    3. OS determines if execution continues

- **OS needs to know**
  1. Reason for exception
    - Cause register (status reg holding field indicating reason for exception, single entry point for all exceptions - OS decodes reg to find cause)
    - Vectored interrupts (address to which control is transferred determined by cause of exception, OS knows reason by address)
  2. Instruction that caused exception (EPC)
Exception/Interrupt Categories

- Exception/Interrupt Requirements
  - Synchronous (internal) vs. asynchronous (external)
    - Synch - instruction triggers exception (assume same data and mem allocation)
      - Ex: page fault, system call, arithmetic overflow
    - Asynch can be handled after current instruction completes
      - Ex: I/O device request, h/w malfunction, power failure
  - User requested vs. coerced
    - User requested always handled after instr completes (system call)
    - Coerced caused by h/w event not under control of user (I/O request, page fault, overflow)
  - User maskable vs. nonmaskable
    - Determines whether h/w responds to exception
    - Maskable: underflow and overflow, misaligned memory accesses
    - Nonmaskable: h/w malfunction, system call, I/O request, memory protection
  - Within vs. between instructions
    - Within usu. Synchronous (overflow, page fault) - difficult to implement (restart)
    - Between: system call (synch), I/O request (asynch)
  - Resume vs. terminate
    - Does program execution continue or terminate?
Difficulties in pipelined implementation

• More difficult to implement interrupts within an instruction (vs. between)
  - Another program must
    • be invoked to save state of executing program
    • correct cause of exception
    • restore the state of the executing program
  - Restartable pipeline

• Easier to implement interrupts that terminate execution
  - No restart necessary
Safe Pipeline Control

• Handling exceptions with pipelining
  1. Force a trap instruction into the pipeline on next IF
  2. Until trap is taken:
    • Turn off all writes for faulting instruction and all succeeding instructions
    • Ensures no state changes from instructions following interrupt
  3. Handling routine saves PC of faulting instruction plus one PC for each branch delay slot
  4. Return from exception
    • Use special instructions to reload PCs and restart instruction stream
Precise Exception Support

- Do instructions prior to exception within pipeline complete prior to handling?
  - Necessary when lots of exceptions (e.g. demand paging in virtual memory)
  - Usually supported for integer pipelines since easier to implement
- Precise exceptions
  - supported by hardware
  - when pipeline can be stopped on interrupt, allowing all instructions prior to interrupt to complete and those after restarted from scratch
- Complications in precise exception handling performance
  - Floating point multiply taking many cycles
  - Other instruction overwrites original operands for multiply then interrupt occurs
  - Forcing precise mode would cause extreme slow down since operands must be held to restore state if interrupt occurs
  - Solution: allow for toggle in and out of precise mode