Basic Pipelining

Examples in daily life
Laundry, Waiting queue, ....

Instruction Execution

<table>
<thead>
<tr>
<th>Instruction fetch</th>
<th>Register fetch</th>
<th>Execution</th>
<th>Memory access</th>
<th>Write Register</th>
</tr>
</thead>
</table>

Pipelining

<table>
<thead>
<tr>
<th>Instruction fetch</th>
<th>Register fetch</th>
<th>Execution</th>
<th>Memory access</th>
<th>Write Register</th>
</tr>
</thead>
</table>

- Programs expect sequential execution
- Results be as if instructions were executed sequentially
- Depending on program constraints to determine the execution parallelism
  - data dependency
  - control dependency
Basic steps of Execution

- **Five Execution Steps**
  - Instruction Fetch
    \[ \text{MAR} \leftarrow \text{PC} \; ; \; \text{IR} \leftarrow M[\text{MAR}] \; ; \; \text{NPC} \leftarrow \text{PC}+4 \]
  - Instruction decode/register read
    \[ \text{A} \leftarrow \text{Rs1} \; ; \; \text{B} \leftarrow \text{Rs2} \]
  - Execution
    - **mem**: \[ \text{MAR} \leftarrow \text{A} +\text{IR}_{16..31} \; ; \; \text{MDR} \leftarrow \text{Rd} \]
    - **ALU**: \[ \text{ALUopt} \leftarrow \text{A} \, \text{op} \, (\text{B} \; \text{or} \; \text{IR}_{16..31}) \]
    - **Brch**: \[ \text{ALUopt} \leftarrow \text{PC} + \text{IR}_{16..31} \]
  - Memory access/branch completion
    - **mem**: \[ \text{MDR} \leftarrow M[\text{MAR}] \; \text{or} \; M[\text{MAR}] \leftarrow \text{MDR} \]
    - **Brch**: if (cond) then \[ \text{PC} \leftarrow \text{ALUopt} \]
  - Write back
    \[ \text{Rd} \leftarrow \text{ALUopt} \; \text{or} \; \text{MDR} \]

- **Five-stage Pipelining**

DLX datapath

- **clock cycles**:
  - branches take 4 cycles
  - all other require 5 cycles
- **several improvements are possible**
  - complete ALU earlier
  - merge two ALUs
  - multicycle implementation
**Principle of Pipelining**

- **Efficiency of Pipelining**
  
  \( T \): total execution time of an instruction
  
  An instruction requires \( n \) stages
  
  each stage takes
  
  - Without pipelining
    \[
    \text{latency} = T = \sum t_i
    \]
    \[
    \text{Throughput} = \frac{1}{T} = \frac{1}{\sum t_i}
    \]
  
  - With pipelining
    \[
    \text{latency} = n \times \max t_i \geq T
    \]
    \[
    \text{Throughput} = \frac{1}{\max t_i} \leq \frac{n}{\sum t_i}
    \]
  
  - Speedup
    \[
    \text{Speedup} = \frac{\text{old latency}}{\text{new latency}} \leq \frac{\sum t_i}{\max t_i} \leq n
    \]

---

**Efficiency of Pipelining**

\( \Delta > 0 \) extra delay between stage

- With an \( n \)-stage pipeline
  \[
  \text{latency} = n \times (\max t_i + \Delta) \geq T + n\Delta
  \]
  \[
  \text{Throughput} = \frac{1}{\Delta + \max t_i} < \frac{n}{\sum t_i}
  \]
  \[
  \text{Speedup} = \frac{\text{old latency}}{\text{new latency}} \leq \frac{\sum t_i}{\Delta + \max t_i} < n
  \]

- **Goal** => reduce clock cycle

- **Possible delay**
  
  - Latches
  
  - clock/data skew

- **No pipelining is useful if clock is less than sum of delay**
What have we learned from pipelining?

- Pipelining does not help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number of pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to fill pipeline and time to drain pipeline reduce speedup

Hazards of Pipelining

- Hazards
  - Situations that prevents the next instruction from execution
  - **Structural Hazards**
    - Resource conflicts
  - **Data Hazards**
    - Data dependence between instructions
  - **Control Hazards**
    - Due to the change of control stream

- Revisit Efficiency of Pipelining

\[
Ideal\ CPI = \frac{CPI_{unpipelined}}{n} \quad Speedup = \frac{\text{old time}}{\text{new time}} = \frac{\text{cycle}_{\text{unpip}} \times CPI_{\text{unpip}}}{\text{cycle}_{\text{pipe}} \times CPI_{\text{pipe}}}
\]

\[
= \frac{CPI_{\text{ideal}} \times n}{CPI_{\text{ideal}} + \frac{\text{stall cycles}}{\text{per inst}}}
\]

© by Tien-Fu Chen@CCU
**Structural Hazards**

Occurs when two instructions use the same resource

- **Solution 1: Stall**
  - Detects the hazard and stall execution
  - + low cost
  - - increase CPI (cause bubble)

- **Solution 2: Duplicate Resource**
  - add more hardware resource available
  - + Good performance
  - - increase cost
  - - may increase cycle time

- **Solution 3: Pipelined Resource**
  - make every use of resource in simple way
  - eg. at most once, at the same stage,
  - use exactly one cycle

---

**Data Hazards**

Occur when order of access in pipelining is different from sequential execution

- **Read-After-Write (RAW)**
  - data(true)-dependence, late write=>early read

- **Write-After-Read (WAR)**
  - anti-dependence: late read => early write

- **Write-After-Write (WAW)**
  - output-dependence
  - slow write => fast operation
Solutions for RAW

- **Stall/interlock**
  - detect RAW, then stall pipelining until the hazard is cleared
  - + low cost, simple solution
  - - Increase CPI - cause pipeline stall or bubble

- **Bypass/Forwarding/Short-circuiting**
  - detect hazard by hardware, then forward results to ALU input, instead of reading from registers
  - + reduce stalls
  - - extra hardware complexity

Solutions for RAW (cont)

- **Hardware requirements for bypassing**
  - Comparators between source and destinations
  - Multiplexors on inputs to ALU
  - Extra data path MDR from input to ALU
  - a set of result buffers to save

- **Delayed load - software solution**
  - A load requiring that the following instruction not use its results
  - delay slot (load delay):
    - the pipeline slot after a load
  - Compiler moves instructions to eliminate bubbles =>
    instruction scheduling
    pipelining scheduling
Situation of hazard detection

<table>
<thead>
<tr>
<th>Situation</th>
<th>Example code sequence</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>( \text{LW} \ R1, \ 45 \ (R2) ) ( \text{ADD} \ R5, R6, R7 ) ( \text{SUB} \ R8, R6, R7 ) ( \text{OR} \ R9, R6, R7 )</td>
<td>No hazard possible because no dependence exists on ( R1 ) in the immediately following three instructions.</td>
</tr>
<tr>
<td>Dependence requiring stall</td>
<td>( \text{LW} \ R1, \ 45 \ (R2) ) ( \text{ADD} \ R5, R1, R7 ) ( \text{SUB} \ R8, R6, R7 ) ( \text{OR} \ R9, R6, R7 )</td>
<td>Comparators detect the use of ( R1 ) in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX.</td>
</tr>
<tr>
<td>Dependence overcome by forwarding</td>
<td>( \text{LW} \ R1, \ 45 \ (R2) ) ( \text{ADD} \ R5, R6, R7 ) ( \text{SUB} \ R8, R1, R7 ) ( \text{OR} \ R9, R6, R7 )</td>
<td>Comparators detect use of ( R1 ) in SUB and forward result of load to ALU in time for SUB to begin EX.</td>
</tr>
<tr>
<td>Dependence with accesses in order</td>
<td>( \text{LW} \ R1, \ 45 \ (R2) ) ( \text{ADD} \ R5, R6, R7 ) ( \text{SUB} \ R8, R6, R7 ) ( \text{OR} \ R9, R1, R7 )</td>
<td>No action required because the read of ( 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>
</tbody>
</table>

**FIGURE 3.17** Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions.

Forwarding and stall in DLX
Pipelining Scheduling

- Scheduling instruction at compile-time

- When cannot schedule the interlock, a NOP instruction is inserted

- For example, for A=B+C, D=E-F

<table>
<thead>
<tr>
<th>Unscheduled</th>
<th>Scheduled</th>
</tr>
</thead>
<tbody>
<tr>
<td>LW r1, B</td>
<td>LW r1, B</td>
</tr>
<tr>
<td>nop</td>
<td>LW r2, C</td>
</tr>
<tr>
<td>LW r2, C</td>
<td>LW r4, E</td>
</tr>
<tr>
<td>nop</td>
<td>ADD r3, r2, r1</td>
</tr>
<tr>
<td>ADD r3, r2, r1</td>
<td>LW r5, F</td>
</tr>
<tr>
<td>LW r1, E</td>
<td>SW r3, A</td>
</tr>
<tr>
<td>nop</td>
<td>SUB r6, r4, r5</td>
</tr>
<tr>
<td>LW r2, F</td>
<td>SW r6, D</td>
</tr>
<tr>
<td>nop</td>
<td>SUB r3, r2, r1</td>
</tr>
<tr>
<td>SW r3, D</td>
<td></td>
</tr>
</tbody>
</table>

- may increase register pressure

Control Hazards

- bubbles after branch

<table>
<thead>
<tr>
<th>i (br)</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>mem</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>i+1</td>
<td>IF</td>
<td>xxx</td>
<td>xxx</td>
<td>IF</td>
<td>ID</td>
</tr>
<tr>
<td>i+2</td>
<td>xxx</td>
<td>xxx</td>
<td>xxx</td>
<td>ID</td>
<td>EX</td>
</tr>
<tr>
<td>i+3</td>
<td>xxx</td>
<td>xxx</td>
<td>xxx</td>
<td>ID</td>
<td>mem</td>
</tr>
<tr>
<td>i+4</td>
<td>xxx</td>
<td>xxx</td>
<td>xxx</td>
<td>IF</td>
<td>ID</td>
</tr>
</tbody>
</table>

- Three cycle stall for each branch is significant, since may include 30% of branches.

- Original branch operations

  **EXE:**
  - ALUOut <- PC + IR16..31
  - cond <- A op 0

  **MEM:**
  - if (cond) PC <- ALUOut

- Separating operations of branch
  - find out the branch condition earlier
  - Computer target address earlier
Solutions for control hazards

- Move up control point to ID phase
  
  **ID:**
  
  A <- Rs1, B <- Rs2,
  TA <- PC + IR16..31
  if (A op 0) PC <- BTA

  **EXE:**

  **MEM:**
  - only one-cycle bubble
  - Extra cost (impact):
    - Additional PC adder required
    - cannot afford too complex condition check- how about EQ, NE, GT
    - may increase cycle time

- Delayed Branch
  - Execute next instruction regardless cond
  - Scheduling the branch-delay slot(s) at compile-time

- Branch Prediction

Pipelined DLX Datapath  
*Figure 3.22, page 163*
**Scheduling branch-delay slots**

- **Fill from before branch**
  - When: branch independent instruction
  - improve? always

- **Fill from target**
  - When: OK to execute target instruction
  - improve? branch taken
  - affect: may increase code size

- **Fill from fall through**
  - When: OK to execute following instruction
  - improve? branch not taken

- **When no instruction can be scheduled, no-op is filled**

- **Additional cost:**
  - multiple PCs for interrupt

---

**Delayed Branch (Cont.)**

- **Effectiveness of compiler on 1 slot**
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled

- **Difficulty**
  - restrictions on instructions that are scheduled into delay slots
  - Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)
  - ability to predict at compile time

- **Canceling/nullifying branch**
  - direction is included in branch
  - if correctly predicted, a delayed slot is normally executed
  - if predicted incorrectly, the branch slot is turned in to a no-op
  - eliminate the requirement of insertion
**Branch Prediction**
- Guess the direction of branch
- Guess the target of branch

- **Static - at compile time**
  - predict not-taken
  - predict taken
  - op-code as hints
  - use like/not-likely bit in instructions
  - delayed branches

- **Dynamic - at run time**
  - branch-prediction buffer
    - one-bit prediction
    - two-bit finite state prediction
  - branch folding
    - "fold" a nonbranch instruction and its following branch into a single instruction
    - => eliminate the branch
  - multiple prefetch and branch bypass

---

**Comparison of branch scheme**

Given 14% of branch and 65% taken

\[
CPI_{\text{effective}} = 1 + CPI_{\text{branch}}
\]

\[
CPI_{\text{branch}} = \%\text{branch} \times (\%\text{taken} \times CPI_{\text{taken}} + \%\text{not taken} \times CPI_{\text{not taken}})
\]

\[
\text{Speedup over stall} = \frac{1 + \%\text{branch} \times \text{stall}}{1 + CPI_{\text{branch}}}
\]

<table>
<thead>
<tr>
<th>Scheduling scheme</th>
<th>Branch penalty</th>
<th>CPI</th>
<th>speedup v. unpipelined</th>
<th>speedup v. stall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stall pipeline</td>
<td>3</td>
<td>1.42</td>
<td>3.5</td>
<td>1.0</td>
</tr>
<tr>
<td>Predict taken</td>
<td>1</td>
<td>1.14</td>
<td>4.4</td>
<td>1.26</td>
</tr>
<tr>
<td>Predict not taken</td>
<td>1</td>
<td>1.09</td>
<td>4.5</td>
<td>1.29</td>
</tr>
<tr>
<td>Delayed branch</td>
<td>0.5</td>
<td>1.07</td>
<td>4.6</td>
<td>1.31</td>
</tr>
</tbody>
</table>

Conditional & Unconditional = 14%, 65% change PC
Interrupts

An event forcing the machine to abort instruction’s execution before its completion.

- **Examples**
  - Page fault, OS traps(calls)
  - Arithmetic overflow
  - Protection Violation
  - Breakpoint
  - I/O device requests

- **Why difficult in pipelining**
  - break in pipelining
  - should occur within instructions
  - must be restartable
    - linking return address
    - saving PSW (or CC codes)
    - correct state change

- **More complication on delayed branches**
  - instructions in pipelining may not be sequentially related
  - require (n+1) PC’s

Handling Interrupts

- **Precise interrupt**
  - a pipelining to handle interrupts follows sequential semantics.
    - instructions before faulting are completed
    - Effects of instructions after are squashed
    - Faulting instruction can be restartable

- **Precise interrupts are usually required**

- **Must handle simultaneous interrupts**
  - IF - memory problem (page fault, protection violation, misaligned memory access)
  - ID - illegal or privileged instructions
  - EX - arithmetic interrupts
  - MEM - memory problem
  - WB - none

- **What order should interrupts be handled**
  - in order - completely precise
    - handle in the order as it appears
Multi-cycle Operations

- Not all operations complete in one cycle
  - all in one cycle → slow clock
  - allow separate function units for pipelining
    - Integer unit
    - FP/integer multiply
    - FP adder
    - FP/integer divider
- Separate integer and FP registers
- all integer instructions operate on integer registers
- all FP instructions operate on FP registers

Extra work on instruction issuing

- check for structural hazards
- check for RAW hazards
- check for data forwarding
- check for overlapping instructions
  - contention for registers in WB
  - possible WAR and WAW hazards
  - difficult to provide precise interrupts

Dealing with overlapping

Contention in WB

- why? FP operations vary in exec time
- solution:
  - static priority, instruction stalls after issue

WAR hazards

- why?  DIVF  F0, F2, F4
  - SUBF  F4, F8, F10
- solutions:
  - always read register at the same time
  - should not occur in DLX

WAW hazards

- why?  DIVF  F0, F2, F4
  - SUBF  F0, F8, F10
- solutions
  - delay SUBF until DIVF enters MEM
  - stamp out DIVF results
Multicycle on interrupts

- DIVF: F0, F2, F4
- ADDF: F10, F8, F10
- SUBF: F12, F12, F14

- hard to maintain precise interrupts
  - out-of-order execution
    - instructions are completed in a different order from the order they are issued
  - Solutions:
    - ignore problems and assume imprecise interrupts - not acceptable
    - queue results of operations until preceding instructions are completed
      - has to store results
        - history table - roll back when interrupts
        - future table - keep newer values
    - software support
      - save information for trap handlers
      - software simulate unfinished instructions
    - allow issue only if preceding instructions are safe