18-547 Lecture 16
*Instruction Level Parallelism*
Instructor: Prof. Falsafi

Electrical & Computer Engineering
Carnegie Mellon University

Slides developed by Profs. Hill, Wood, Sohi, and Smith of
University of Wisconsin — Madison

**Announcements**

Homework 4 is due
Status report this week
Pipeline Clocking

See references for more (reader)
Fast latches
Performing logic in latches
Latch overhead: clock skew
Optimal gate levels
**Fast Latches**

Latches are faster than flipflops

Often pipelines are built with 2-phase clocks
  - like master-slave flipflop with logic in between
We will consider a single phase
  - same basic principles apply for multi-phase clocks

![Basic latch equivalent to cross-coupled NANDs](image)

**Removing Overhead: Logic Built-In**

![Removing Overhead: Logic Built-In](image)
Clock Skew Overhead

Clock skew adds latch overhead
Skew = \( t_{\text{max}} - t_{\text{min}} \)
To compensate for skew
- clock must be slowed by skew amount

Optimal Number of Pipe Stages

Trade-off between
- number of gate delays per stage
  - the less gate delays, the faster the clock
  - the more parallelism
- number of (latch) overhead gate delays
  - the more stages, the more latches

Base pipeline assumes:
- 5 stages
- 16 gate delays per stage
- \( x \) be the number of stall cycles per instruction
- \( \Rightarrow \) there are 16 * \( x \) stall gate delays per instruction
- e.g., if half of instruction stall for 1 cycle
  - \( x = 0.5 \)
  - the number of gate delay stalls will be 16 * 0.5 = 8
Larger Number of Pipe Stages

Increase the number of pipe stages
- clock faster

Let n be the number pipe stages
- total number of clocks to execute an inst = 5 + x * n
- instructions/clock = 1/(1 + x * n/5)
- assume a latch overhead of o
- gate delays/clock = (16 * 5) / n + o = 80/n + o

Absolute throughput is:
- instruction/gate delay = (instructions/clock) * (clock/gate delays)
- = (1/(1 + x * n/5)) * (1/(80/n + o))

Effect of Latch (Cont.)

E.g.,
- Bhandarkar & Clark (Reader 1) => x = 0.8
- assume o = 1 gate delay

<table>
<thead>
<tr>
<th>n</th>
<th>IPC</th>
<th>Clock/g: Through</th>
<th>delays</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.56</td>
<td>0.059</td>
<td>0.033</td>
</tr>
<tr>
<td>10</td>
<td>0.38</td>
<td>0.11</td>
<td>0.042</td>
</tr>
<tr>
<td>20</td>
<td>0.33</td>
<td>0.166</td>
<td>0.040</td>
</tr>
</tbody>
</table>
**Pipeline CPI**

CPI = Ideal CPI + Pipeline stalls + Memory stalls  
Memory stalls = instruction stalls + data stalls  
Pipeline stalls = structural stalls + RAW stalls + WAR stalls + WAW stalls + control stalls

We dealt with memory stalls in chapter 5  
Lets deal with the rest of the CPI

---

**Reduce CPI**

How do we reduce Ideal CPI?  
- multiple instruction issue  
- superscalar/VLIW  
- can make Ideal CPI go to << 1!

How do we reduce RAW stalls?  
- instruction scheduling  
  - compiler dependence analysis  
  - loop unrolling, software pipelining, trace scheduling  
  - hardware (dynamic) scheduling

How do we reduce WAR/WAW stalls?  
- register (data) renaming

How do we reduce control stalls?  
- branch prediction
Instruction Level Parallelism

1. More parallelism
2. Sequential execution semantics
   - data dependences: RAW
   - name dependences: WAR and WAW
   - traps, exceptions
   - control dependences: branches and jumps
3. Must pay more attention to stalls!
   - more pressure on pipeline => more likely to stall

What We Will Do HERE

Dynamic Instruction Scheduling
   - scoreboardinig, Tomasulo’s algorithm
Compiler techniques
   - pipeline scheduling
   - loop unrolling
   - VLIW trace scheduling
Advanced superscalar processors
   - branch prediction, reorder buffer
Case studies
   - R10000, K5, Alpha 21264, P6, UltraSPARC III
In-Order Execution

Code fragment

\[
\begin{align*}
\text{divf} & \quad f_0, f_2, f_4 \\
\text{addf} & \quad f_{10}, f_0, f_8 \\
\text{multf} & \quad f_7, f_8, f_{14}
\end{align*}
\]

Problem

- addf stalls due to RAW hazard
- multf stalls because addf stalls

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
\text{divf} & \text{IF} & \text{ID} & \text{E/} & \text{E/} & \text{E/} & \text{E/} & \text{MEM WB} \\
\text{addf} & \text{IF} & \text{ID} & -- & -- & -- & \text{E+} & \text{MEM WB} \\
\text{multf} & \text{IF} & -- & -- & -- & \text{ID} & \text{EX} & \text{MEM WB}
\end{array}
\]

In-order execution limits performance!

Dynamic vs. Static Scheduling

Solutions

- dynamic vs. static scheduling

Static scheduling (software)

- compiler reorganizes instructions
- simpler hardware
- can use more power algorithms
- Itanium, Cruso, lots of DSP chips

Dynamic scheduling (hardware)

- handles dependences unknown at compile time
- hardware reorganizes instructions!
- more complex hardware but code more portable
- “the real stuff”
Register Hazards

Reordering must not violate RAW hazards
(true dependences)

WAR and WAW may inhibit reordering
(false dependences)

WAR/WAW hazards

\[
\begin{align*}
\text{addf } & f0, f2, f4 \\
\text{addf } & f0, f0, f8 \\
\text{divf } & f12, f0, f10 \\
\text{multf } & f8, f8, f14 \\
\text{multf } & f10, f6, f14
\end{align*}
\]

Instruction may have WAW or WAR with itself, when?

Dynamic Scheduling: Scoreboard

Centralized control scheme

- controls all instruction issue
- detects all hazards

Implemented in CDC6000 in 1964

- CDC 6000 has 18 separate functional units (not pipelined)
- 4 FP: 2 multiply, 1 add, 1 divide
- 7 memory units: 5 loads, 2 stores
- 7 integer units, add, shift, logical, etc

Dynamic DLX:

- 2 FP mult, 1 FP add, 1 FP divide, 1 integer
Dynamic DLX Pipeline

Ignore IF and MEM for simplicity

4-stage execution

- issue - check struct/WAW hazards, stall all issue till clear
- read ops - check RAW, wait till operands ready, read regs
- execute - execute operation, notify scoreboard when done
- write - check for WAR, stall write until clear

Completing instr can’t write dest if an earlier instr has’nt read dest
Scoreboarding

3 data structures
   - instruction status
   - functional unit status
   - register result status

Instruction status
   - which execution stage is the instruction in

Functional unit status
   - busy -- indicate whether functional unit is busy
   - op -- operation
   - F_i -- destination register
   - F_j, F_k -- source registers containing src1 and src2
   - Q_j, Q_k -- functional units producing src1 and src2
   - R_j, R_k -- ready bits for F_j, F_k

Register result status
   - indicates which functional unit (if any) will write the register
Scoreboarding Example

\[
\begin{align*}
&\text{If } f6, 34(r2) \\
&\text{If } f2, 45(r3) \\
&\text{mult} f0, f2,f4 \\
&\text{sub} f8, f6, f2 \\
&\text{div} f10, f0, f6 \\
&\text{add} f6, f8, f2
\end{align*}
\]
Results

Speedup
- 1.7 for FORTRAN programs
- 2.5 for hand-coded assembly (what about modern compilers?)

Hardware
- scoreboard equal to one FU (main cost was busses)

Limitations
- no bypassing
- data dependences handled through registers
- WAW cause stalls (limits dynamic unrolling of loops)

Dynamic Scheduling

\[ \text{DO } I = 1, N \]
\[ C[I] = A[I] + s \times B[I] \]

Assembly
- \text{if } f0, A(r1)
- \text{if } f2, B(r1)
- \text{mult } f4, f2, f2  \quad \# s \text{ in } f4
- \text{addf } f0, f2, f2
- \text{stf } f2, [C+r1]

What would scoreboard do?
Can we do better?