Lecture 18
Instruction Level Parallelism
Today’s lecture
• Pipelining and other forms of Instruction Level Parallelism (ILP)
• Building an efficient barrier
What makes a processor run faster?

• Registers and cache
• Vectorization (SSE)
• Instruction level parallelism
What is Instruction-level Parallelism?

• The potential to execute certain instructions in parallel, because they are independent
  \[ w = u / z; \]
  \[ a = b + c; \]

• Any technique for identifying and exploiting such opportunities
  ‣ Static: can be implemented by a compiler
  ‣ Dynamic: requires hardware support
How does ILP manifest itself?

- Basic blocks
  - Sequences of instruction that appear between branches
  - Usually no more than 5 or 6 instructions!
- Loops
  ```plaintext
  for ( i=0; i < N; i++)
    x[i] = x[i] + s;
  ```
- We can only realize ILP by finding sequences of independent instructions
- Dependent instructions must be separated by a sufficient amount of time, that is functional unit latency
Pipelining

• An implementation technique for overlapping instruction execution
• Like an automobile assembly line, we break the instruction’s execution into steps
• We can then overlap the steps so long as there are no dependences (*pipeline hazards*) between the steps
Pipelining

• Laundry analogy
  wash (30 min) + dry (40 min) + fold (20 min) = 90 min

6 PM  7  8  9

Time

30  40  40  40  40  20

• Sequential execution takes
  4 * 90 min = 6 hours

• Pipelined execution takes
  30 + 4*40 + 20 = 3.5 hours

• Bandwidth = loads/hour
• Pipelining helps bandwidth but not latency (90 min)

• Bandwidth limited by slowest pipeline stage
• Potential speedup = Number pipe stages
Multi-execution pipeline

- **MIPS R4000**

\[
\begin{align*}
\text{w} &= \text{u} / \text{v}; \\
\text{a} &= \text{b} + \text{c}; \\
\text{s} &= \text{q} \times \text{r}; \\
\text{i} &= \text{i} + \text{c}; \\
\text{x} &= \text{y} / \text{z}; \\
\text{t} &= \text{c} - \text{q}; \\
\text{a} &= \text{x} + \text{c};
\end{align*}
\]

David Culler

Scott B. Baden / CSE 160 / Wi '16
Sandy Bridge

Sandy Bridge

Instruction Fetch Unit

Branch Predictors

32KB L1 I-Cache (8 way)

16 Byte t

168 Predecode, Fetch Buffer

6 instructions

18+ Entry Instruction Queue

μcode Engine

Complex Decode

Simple Decode

Simple Decode

Simple Decode

1.5K μop Cache (8 way)

28 μop Decoder Queue

4 μops

4 μops

4 μops

4 μops

168 Entry Reorder Buffer (ROB)

144 Entry FP Physical Register File

160 Entry Physical Register File

54 Entry Unified Scheduler

Port 0

ALU

LEA

Shift

SIMD

MUL

Shift

Port 1

ALU

MUL

LEA

Port 5

alu

Shift

SIMD

MUL

Shift

Port 2

64-bit

AGU

Port 3

64-bit

AGU

Store Data

Port 4

64 Entry Load Buffer

128 bits

32KB L1 I-Cache (8 way)

256 bit

512 Entry

L2 TLB (4 way)

100 Entry

L1 DTLB (fully)

32KB L1 D-Cache (8 way)

256 bit

256KB L2 Cache (8 way)
Pipeline Hazards

- What if we have dependencies?
- A data dependence is one kind of pipeline hazard, a data hazard, that prevents the next instruction from executing when expected
- Structural hazards: physical resources aren’t available
- Control hazards: branches or other instructions that change the PC
- The simplest solution is to stall the pipeline, introducing bubble, but we can do better

```plaintext
x = y / z;
a = x + c;
t = c - q;
```
Limitations of ILP

- Consider this loop
  
  for ( i=0; i<N; i++)
    x[i] = x[i] + s;

L: LD       F0, 0(R1)       ; F0 is the vector element
ADDD F4, F0, F2          ; add the scalar
SD 0(R1), F4             ; store the result
SUBI R1, R1, #8          ; decrement by 8
                         ; to previous word
BNEZ R1, L               ; Branch if R1 ≠ 0
NOP                       ; Delayed branch slot
Delays

for ( i=0; i<N; i++)
    x[i] = x[i] + s;

<table>
<thead>
<tr>
<th>Unit</th>
<th>Latency</th>
<th>Init Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Memory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP Add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP Multiply</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP Div</td>
<td>24</td>
<td>24</td>
</tr>
</tbody>
</table>

L: LD
   \textit{stall} F0, 0(R1)
   ADDD \textit{stall} F4, F0, F2
   \textit{stall} SD \textit{stall} 0(R1), F4
   \textit{stall} SUBI \textit{stall} R1, R1, #8
   \textit{stall} BNEZ \textit{stall} R1, L

Scott B. Baden / CSE 160 / Wi '16
Instruction Reordering Reduces stalls

for ( i=0; i<N; i++)
    x[i] = x[i] + s;

L: LD        F0, 0(R1)
    stall
    ADDD    F4, F0, F2
    stall
    stall
    SD        0(R1), F4
    SUBI     R1, R1, #8
    stall
    BNEZ     R1, L
    stall

L: LD        F0, 0(R1)
    stall
    SUBI     R1, R1, #8
    ADDD    F4, F0, F2
    stall
    BNEZ     R1, L
    SD        0(R1), F4
Static scheduling limits performance

- We can’t always reorder, or even it were possible there is a limit to how far the compiler can look
- The ADDD instruction is stalled on the DIVide …
- …stalling further instruction issue, e.g. the SUBD

\[
\begin{align*}
\text{DIV} & \quad F0, \quad F2, \quad F4 \\
\text{ADDD} & \quad F10, \quad F0, \quad F8 \\
\text{MULD} & \quad F12, \quad F8, \quad F14
\end{align*}
\]

- But MULD doesn’t depend on ADDD or DIV
- If we have and adder and a multiplier unit, one will sit idle uselessly until the DIV finishes
Dynamic scheduling

- Idea: Modify the pipeline to enable instructions to execute as soon as their operands become available
- This is known as *out-of-order execution*
- Stalled instructions (MUL) can now proceed normally

\[
\begin{align*}
x &= y / z; \\
a &= x + c; \\
t &= c * q;
\end{align*}
\]

\[
\begin{align*}
x &= y / z; \\
t &= c * q; \\
a &= x + c;
\end{align*}
\]
Complications of dynamic scheduling

- Dynamically scheduled instructions also complete out of order
- Increased hardware complexity
  - Stations and execution units
  - Bookkeeping
  - Buffering
  - Slows down the processor

\[
\begin{align*}
  x &= y / z; \\
  a &= x + c; \\
  t &= c * q;
\end{align*}
\]
Dynamic scheduling splits the ID stage

- Issue sub-stage
  - Decode the instructions
  - Check for structural hazards
- Read operands sub-stage
  - Wait until there are no data hazards
  - Read operands
Consequences of a split ID stage

• We distinguish between the time when an instruction begins execution, and when it completes

• Previously, an instruction stalled in the ID stage, and this held up the entire pipeline

• Instructions can now be in a suspended state, neither stalling the pipeline, nor executing

• They are waiting on operands - need additional registers to store pending instructions that aren’t ready to execute
The three steps of instruction execution

• Issue
  ‣ Get instruction from fetch queue
  ‣ If there are available resources buffer the register operands at station
  ‣ If there is no such station, stall the instruction

• Execute
  ‣ Monitor internal interconnect for any need operands
  ‣ Intercept those operands and write to the station
  ‣ When both operands are present, execute the operation

• Write result
  ‣ Send to the result registers and any waiting units
Automatic, dynamic loop unrolling

L:       LD      F0, 0(R1)
       MULTD    F4, F0, F2
       SD       0(R1), F4
       SUBI     R1, R1, #8
       BNEZ     R1, L
Automatic, dynamic loop unrolling

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Iteration</th>
<th>Issue</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, 0(R1)</td>
<td>1</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Automatic, dynamic loop unrolling

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Iteration</th>
<th>Issue</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, 0(R1)</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MULTD F4, F0, F2</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Automatic, dynamic loop unrolling

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Iteration</th>
<th>Issue</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, 0(R1)</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MULTD F4, F0, F2</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SD 0(R1), F4</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
## Automatic, dynamic loop unrolling

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Iteration</th>
<th>Issue</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>MULTD</td>
<td>F4, F0, F2</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1), F4</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
<td>2</td>
<td>X</td>
</tr>
</tbody>
</table>
Automatic, dynamic loop unrolling

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Iteration</th>
<th>Issue</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>MULTD</td>
<td>F4, F0, F2</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>SD</td>
<td>0(R1), F4</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>LD</td>
<td>F0, 0(R1)</td>
<td>2</td>
<td>X</td>
</tr>
</tbody>
</table>
## Automatic, dynamic loop unrolling

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Iteration</th>
<th>Issue</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD F0, 0(R1)</td>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MULTD F4, F0, F2</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SD 0(R1), F4</td>
<td>1</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LD F0, 0(R1)</td>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MULTD F4, F0, F2</td>
<td>2</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SD 0(R1), F4</td>
<td>2</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Two schemes for dynamic scheduling

- **Scoreboard**
  - CDC 66000

- **Tomasulo’s algorithm**
  - IBM 360/91

- We’ll vary the number of functional units, their latency, and functional unit pipelining
What is a scoreboard?

• A technique that allows instructions to execute out of order...
  ‣ So long as there are sufficient resources and
  ‣ No data dependencies

• The goal of scoreboard
  ‣ Maintain an execution rate of one instruction per clock cycle
Scoreboarding Strategy

• Hardware data structures keep track of
  ‣ When instructions complete
  ‣ Which instructions depend on the results
  ‣ When it’s safe to write a reg.

• Deals with data hazards
  ‣ WAR (Write after read)
  ‣ RAW (Read after write)
  ‣ WAW (Write after write)

\[ a = b \times c \]
\[ x = y - b \]
\[ q = a / y \]
\[ y = x + b \]
Where do RAW hazards appear?

A. (2)
B. (3)
C. (4)
D. (2) and (3)
E. (3) and (4)

1. \( a = b \times c \)
2. \( x = y - b \)
3. \( q = a \div y \)
4. \( y = x + b \)
Where do WAR hazards appear?

A. (2)
B. (3)
C. (4)
D. (2) and (3)
E. (2), (3) and (4)

(1) \( a = b \times c \)
(2) \( x = y - b \)
(3) \( q = a / y \)
(4) \( y = x + b \)
Hazards

- **WAR** (Write after read)
- **RAW** (Read after write)
- **WAW** (Write after write)

\[
a = b \times c \\
x = y - b \\
q = a / y \\
y = x + b
\]
Scoreboard controls the pipeline

Instruction Fetch → Instruction Issue → Pre-execution buffers → Read operands → Execution unit 1 → Write results

Instruction Decode → Pre-issue buffer → Read operands → Execution unit 2

Scoreboard / Control Unit

Pre-execution buffers

Post-execution buffers

WAW

WAR

RAW

Mike Frank
What are the requirements?

• Responsibility for instruction issue and execution, including hazard detection
• Multiple instructions must be in the EX stage simultaneously …
• … either through pipelining or multiple functional units
• Our processor (DLX) has:
  2 multipliers, 1 divider, 1 integer unit
  (memory, branch, integer arithmetic)
How does it work?

- As each instruction passes through the scoreboard, construct a description of the data dependencies (Issue)
- Scoreboard determines when the instruction can read operands and begin execution
- If the instruction can’t begin execution, the scoreboard keeps a record, and it listens for one the instruction can execute
- Also controls when an instruction may write its result
- All hazard detection is centralized
Multiple execution pipelines in DLX with scoreboard

- A centralized bookkeeping table: tracks instructions + registers they depend or modify
- Status of result registers (who is going to write to a given register)
- Status of the functional units

<table>
<thead>
<tr>
<th>Unit</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>Memory</td>
<td>1</td>
</tr>
<tr>
<td>FP Add</td>
<td>2</td>
</tr>
<tr>
<td>FP Multiply</td>
<td>10</td>
</tr>
<tr>
<td>FP Div</td>
<td>40</td>
</tr>
</tbody>
</table>

Scott B. Baden / CSE 160 / Wi '16
Sandy Bridge

Sandy Bridge

144 Entry L1 TLB (4 way)

32KB L1 I-Cache (8 way)

168 Predecode, Fetch Buffer

6 instructions

18+ Entry Instruction Queue

μop Decoder Queue

168 Entry Reorder Buffer (ROB)

144 Entry FP Physical Register File

54 Entry Unified Scheduler

Port 0

ALU Shift

SIMD MUL Shift

256-bit FMUL, Blend

Port 1

SIMD MUL

256-bit FADD

Port 2

ALU LEA Shift

SIMD LEA MUL

256-bit Shuffle Blend

Port 3

ALU LEA

SIMD MUL, Branch

64-bit AGU

Port 4

SIMD ALU Shuffle

256-bit Shuffle

64-bit AGU

Store Data

512 Entry L2 TLB (4 way)

100 Entry L1 DTLB (fully)

32KB L1 D-Cache (8 way)

256-bit

256KB L2 Cache (8 way)
Loop Unrolling

- Common loop optimization strategy
- Duplicate the body of the loop

```
for (int i=0; i < n ; i++)
    z[i] = x[i] + y[i];
```

- Plays well with instruction level parallelism
- Register utilization, instruction scheduling
- Not always advantageous. Why?

```
for (int i=0; i < n ; i+=4){
    z[i+0] = x[i+0] + y[i+0];
    z[i+1] = x[i+1] + y[i+1];
    z[i+2] = x[i+2] + y[i+2];
    z[i+3] = x[i+3] + y[i+3];
}
```