Instruction Level Parallelism

- **Instruction-Level Parallelism (ILP):** overlap the execution of instructions to improve performance
- 2 approaches to exploit ILP:
  1. Rely on **hardware** to help discover and exploit the parallelism dynamically (e.g., Pentium 4, AMD Opteron, IBM Power), and
  2. Rely on **software** technology to find parallelism, statically at compile-time (e.g., Itanium 2)

---

Instruction-Level Parallelism (ILP)

- Basic Block (BB) ILP is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25%
  - \( \Rightarrow \) 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks
- Simplest: **loop-level parallelism** to exploit parallelism among iterations of a loop. E.g.,
  ```
  for (i=1; i<=1000; i=i+1)
  x[i] = x[i] + y[i];
  ```
Loop-Level Parallelism

- Exploit loop-level parallelism to parallelism by “unrolling loop” either by
  1. dynamic via branch prediction or
  2. static via loop unrolling by compiler
    - Another way is by using vectors
- Determining instruction dependence is critical to Loop Level Parallelism
- If 2 instructions are
  - parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
  - dependent, they are not parallel and must be executed in order, although they may often be partially overlapped

Data Dependence and Hazards

- Instr$_j$ is data dependent (or true dependence) on Instr$_i$ if:
  - Instr$_j$ tries to read operand before Instr$_i$ writes it
  - or Instr$_j$ is data dependent on Instr$_k$ which is dependent on Instr$_i$
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence $\Rightarrow$ data dependence in source code $\Rightarrow$ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, called a Read After Write (RAW) hazard
ILP and Data Dependencies, Hazards

- HW/SW must preserve program order: order instructions would execute in if executed sequentially as determined by original source program
  - Dependences are a property of programs
  - Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies
  1. indicates the possibility of a hazard
  2. determines order in which results must be calculated
  3. sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Name Dependence #1: Anti-dependence

- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
  - Instr$_j$ writes operand before Instr$_i$ reads it

```
I: sub r4, r1, r3
J: add r1, r2, r3
K: mul r6, r1, r7
```

Called an “anti-dependence” by compiler writers. This results from reuse of the name "r1"
- If anti-dependence caused a hazard in the pipeline, called a Write After Read (WAR) hazard
Name Dependence #2: Output dependence

- Instr\textsubscript{j} writes operand before Instr\textsubscript{i} writes it.

\[
\begin{align*}
I: & \text{ sub } r1, r4, r3 \\
J: & \text{ add } r1, r2, r3 \\
K: & \text{ mul } r6, r1, r7
\end{align*}
\]

- Called an “output dependence” by compiler writers
- This also results from the reuse of name “r1”
- If anti-dependence caused a hazard in the pipeline, called a Write After Write (WAW) hazard
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed
- so instructions do not conflict
- Register renaming resolves name dependence for regs
- Either by compiler or by HW

Register Renaming

- Example
  - I3 can not exec before I2 because I3 will overwrite R6
  - I5 can not go before I2 because I2, when it goes, will overwrite R2 with a stale value

Program code

\[
\begin{align*}
I1: & \text{ ADD } R1, R2, R3 \\
I2: & \text{ SUB } R2, R1, R6 \\
I3: & \text{ AND } R6, R11, R7 \\
I4: & \text{ OR } R8, R5, R2 \\
I5: & \text{ XOR } R2, R4, R11
\end{align*}
\]

<table>
<thead>
<tr>
<th>RAW</th>
<th>WAR</th>
<th>WAW</th>
</tr>
</thead>
</table>

Example:

- I3 can not exec before I2 because I3 will overwrite R6
- I5 can not go before I2 because I2, when it goes, will overwrite R2 with a stale value
Register Renaming

- Solution:
  Let's give I2 temporary name/location (e.g., S) for the value it produces.
- But I4 uses that value, so we must also change that to S...
- In fact, all uses of R2 from I2 to the next instruction that writes to R2 again must now be changed to S!
- We remove WAW deps in the same way: change R2 in I5 (and subsequent instrs) to T.

Program code

I1: ADD R1, R2, R3
I2: SUB R5, R1, R6
I3: AND R5, R11, R7
I4: OR  R8, R5, S
I5: XOR R2, R4, R11

Register Renaming

- Implementation
  - Space for S, T, U etc.
  - How do we know when to rename a register?
- Simple Solution
  - Do renaming for every instruction
  - Change the name of a register each time we decode an instruction that will write to it.

Program code

I1: ADD R1, R2, R3
I2: SUB S, R1, R6
I3: AND U, R11, R7
I4: OR  R8, R5, S
I5: XOR T, R4, R11
Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order.
  ```
  if p1 {
    S1;
  }
  if p2 {
    S2;
  }
  ```
  
- \( S_1 \) is control dependent on \( p_1 \), and \( S_2 \) is control dependent on \( p_2 \) but not on \( p_1 \).

Control Dependence Ignored

- Control dependence need not be preserved
  - Willing to execute instructions that should not have been executed, thereby violating the control dependencies, if can do so without affecting correctness of the program

- Instead, 2 properties critical to program correctness are
  1. exception behavior
  2. data flow
Exception Behavior

- Preserving exception behavior
  ⇒ any changes in instruction execution order must not change how exceptions are raised in program
  ⇒ no new exceptions
- Example:
  
  ```
  DADDU R2, R3, R4
  BEQZ R2, L1
  LW R1, 0(R2)
  
  L1:
  ```
  
- (Assume branches not delayed)
- Problem with moving **LW** before **BEQZ**?

Data Flow

- **Data flow**: actual flow of data values among instructions that produce results and those that consume them
  - Branches make flow dynamic, determine which instruction is supplier of data
- Example:
  
  ```
  DADDU R1, R2, R3
  BEQZ R4, L
  DSUBU R1, R5, R6
  
  L: ...
  OR R7, R1, R8
  ```
  - **OR depends on** **DADDU** or **DSUBU**?
  - Must preserve data flow on execution
Software Techniques - Example

- This code, add a scalar to a vector:
  
  ```
  for (i=1000; i>0; i=i–1)
    x[i] = x[i] + s;
  ```

- Assume following latencies for all examples
  - Ignore delayed branch in these examples

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in cycles</th>
<th>stalls between in cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

FP Loop: Where are the Hazards?

- First translate into MIPS code:
  To simplify, assume 8 is lowest address

  ```
  Loop:
  L.D F0, 0(R1) ; F0=vector element
  ADD.D F4,F0,F2 ; add scalar from F2
  S.D 0(R1),F4 ; store result
  DADDUI R1,R1,-8 ; decrement pointer 8B
  BNEZ R1,Loop ; branch R1!=zero
  ```
FP Loop Showing Stalls

1 Loop: L.D F0,0(R1) ;F0=vector element
2 stall
3 ADD.D F4,F0,F2 ;add scalar in F2
4 stall
5 stall
6 S.D 0(R1),F4 ;store result
7 DADDUI R1,R1,-8;decrement pointer 8B (DW)
8 stall ;assumes can't forward to branch
9 BNEZ R1,Loop ;branch R1!=zero

Revised FP Loop Minimizing Stalls

1 Loop: L.D F0,0(R1)
2 DADDUI R1,R1,-8
3 ADD.D F4,F0,F2
4 stall
5 stall
6 S.D 8(R1),F4 ;altered offset when
7 BNEZ R1,Loop ;move DSUBUI

Swap DADDUI and S.D by changing address of S.D

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU op</td>
<td>Another FP ALU op</td>
<td>3</td>
</tr>
<tr>
<td>FP ALU op</td>
<td>Store double</td>
<td>2</td>
</tr>
<tr>
<td>Load double</td>
<td>FP ALU op</td>
<td>1</td>
</tr>
</tbody>
</table>

9 clock cycles: Rewrite code to minimize stalls?

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead
How can we make this faster?
Unroll Loop Four Times (straightforward way)

Rewrite loop to minimize stalls?

1 Loop: L.D F0,0(R1)  
2 ADD.D F4,F0,F2  
3 S.D 0(R1),F4 ;drop DSUBUI & BNEZ  
4 L.D F6,-8(R1)  
5 ADD.D F8,F6,F2  
6 S.D F6,-8(R1)  
7 ADD.D F8,F6,F2  
8 S.D F8,-16(R1)  
9 ADD.D F10,F8,F2  
10 S.D F10,-16(R1)  
11 ADD.D F12,F10,F2  
12 S.D F12,-16(R1)  
13 ADD.D F14,F12,F2  
14 S.D F14,-24(R1)  
15 DADDUI R1,R1,#-32 ;alter to 4*8  
16 S.D F16,F14,F2  
17 BNEZ R1,LOOP

27 clock cycles, or 6.75 per iteration (Assumes R1 is multiple of 4)

Unrolled Loop That Minimizes Stalls

1 Loop: L.D F0,0(R1)  
2 L.D F6,-8(R1)  
3 L.D F10,-16(R1)  
4 L.D F14,-24(R1)  
5 ADD.D F4,F0,F2  
6 ADD.D F8,F6,F2  
7 ADD.D F10,F8,F2  
8 ADD.D F12,F10,F2  
9 ADD.D F14,F12,F2  
10 S.D 0(R1),F4  
11 S.D -8(R1),F8  
12 DSUBUI R1,R1,#32  
13 S.D F16,F14,F2  
14 BNEZ R1,LOOP

14 clock cycles, or 3.5 per iteration
5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:
  1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
  2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
  3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
  4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
  5. Schedule the code, preserving any dependences needed to yield the same result as the original code

3 Limits to Loop Unrolling

- Growth in code size
  - For larger loops, concern it increases the instruction cache miss rate
- Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
  - If not be possible to allocate all live values to registers, may lose some or all of its advantage
- Loop unrolling reduces branch impact on pipeline; another way is branch prediction (next time)