Recall from Pipelining Review

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - **Ideal pipeline CPI**: Measure of the maximum performance attainable by the implementation
  - **Structural hazards**: Hardware cannot support this combination of instructions
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
  - **Control hazards**: Caused by delay between fetching of instructions and control flow decisions (branches and jumps)
    - e.g., in MIPS, j jump, jal call, jr return)

Instruction Level Parallelism

- **Instruction-Level Parallelism (ILP)**
  - Overlap the execution of instructions to run programs faster (“improve performance”)
  - Two approaches to exploit ILP:
    1. Rely on hardware to help discover and exploit the parallelism dynamically (e.g., Pentium 4, AMD Opteron, IBM Power)
    2. Rely on software technology to find parallelism, statically at compile-time (e.g., Itanium 2 (IA-64))
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%
  - => 4 to 7 instructions execute between a pair of branches
  - other problem: instructions in a BB likely depend on each other

To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks ("trace scheduling").

Loop-Level Parallelism

- Exploit loop-level parallelism by “unrolling loop” either by
  1. dynamic via branch prediction or
  2. static via loop unrolling by compiler
  (Another way is vectors, to be covered later)

- Determining dependences is critical!
- 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\textsubscript{J} is data dependent (aka true dependence) on Instr\textsubscript{I}:
  1. Instr\textsubscript{J} tries to read operand before Instr\textsubscript{I} writes it
  2. Instr\textsubscript{J} is data dependent on Instr\textsubscript{K} which is dependent on Instr\textsubscript{I}

- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence
  ⇒ data dependence in source code
  ⇒ effect of original data dependence must be preserved
- If data dependence causes a hazard in a pipeline, it is a Read After Write (RAW) hazard – “RAW is real”
ILP and Data Dependencies, Hazards

- HW/SW must preserve illusion (or program order)
  - Code must give the same results as if instructions were executed sequentially in the original order of the source program
  - Dependencies are a property of programs
- The presence of a dependence indicates the potential for a hazard, but the existence of an actual hazard and length of any stall are pipeline properties
- Importance of data dependencies
  1. Indicate the possibility of a hazard
  2. Determine the order in which results must be calculated
  3. Set upper bounds on how much parallelism can possibly be exploited to speedup a program
- 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, which may cause WAR and WAW hazards.
- Instr\textsubscript{J} writes operand before Instr\textsubscript{I} reads it
- Called an “anti-dependence” by compiler writers. This results from reuse of the name “r1”
- If anti-dependence caused a hazard in the pipeline, that’s a Write After Read (WAR) hazard

Name Dependence #2: Output dependence

- Instr\textsubscript{J} writes operand before Instr\textsubscript{I} writes it.
  - Called an “output dependence” by compiler writers
  - This also results from the reuse of name “r1”
  - If output-dependence caused a hazard in the pipeline, that’s a Write After Write (WAW) hazard
- Instructions involved in a name dependence can execute simultaneously, if we can make the instructions do not conflict.
  - Register renaming by hardware
  - Use different register names by compiler

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;
  }
  ```

- S1 is control dependent on proposition p1, and S2 is control dependent on p2, but not on p1.
Carefully Violate Control Dependencies

- Control dependence need NOT always be preserved
  - Can be violated by executing instructions that should not have been, if doing so does NOT affect program results
    - e.g., "speculative execution" HW throws away results of bad branch guesses.
  - Instead, 2 properties critical to program correctness are
    - Exception behavior and
    - Data flow

Exception Behavior Is Important

- 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, -1(R2)
    - L1:

      (This code example assumes branches are not delayed)

- What is the problem with moving LW before BEQZ?
  - e.g., Array overflow: what if R2=0, so -1+[R2] is out of program memory bounds?

Data Flow Of Values Must Be Preserved

- Data flow: actual flow of data values from instructions that produce results to those that consume them
  - branches make flow "dynamic" (since we know details only at runtime); must determine which instruction is the data supplier
  - Example:
    - DADDU R1, R2, R3
    - BEQZ R4, L
    - DSUBU R1, R5, R6
    - L1:
      - OR R7, R1, R8

  - OR depends on DADDU or DSUBU?
  - Compilers and HW must preserve data flow during execution

Outline

- ILP – Instruction Level Parallelism
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- (Start) Tomasulo Algorithm
- Conclusion
Software Techniques - Example

- This code adds 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 branches in these examples

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in 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>
<tr>
<td>Load double</td>
<td>Store double</td>
<td>0</td>
</tr>
<tr>
<td>Integer op</td>
<td>Integer op</td>
<td>0</td>
</tr>
</tbody>
</table>

FP Loop: Where are the Hazards?

- First, translate into MIPS code. To simplify, assume:
  - 8 is the lowest address,
  - F2 has \(s\)
  - R1 starts with address for \(x[1000]\)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction using result</th>
<th>Latency in cycle cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>;F0=vector element</td>
<td>0</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>;add scalar from F2</td>
<td>1</td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>;store result</td>
<td>1</td>
</tr>
<tr>
<td>DADDUI R1,R1,−8</td>
<td>;decrement pointer 8Bytes (DW)</td>
<td>1</td>
</tr>
<tr>
<td>BNEZ R1,Loop</td>
<td>;branch R1!=zero</td>
<td>1</td>
</tr>
</tbody>
</table>

Loop takes 7 clock cycles
- 3 for execution (L.D,ADD.D,S.D)
- 4 for loop overhead

How to make faster?

Revised FP Loop Minimizing Stalls

Swap DADDUI and S.D; change address offset of S.D

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Instruction using result</th>
<th>Latency in cycle cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>;F0=vector element</td>
<td>0</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>;add scalar from F2</td>
<td>1</td>
</tr>
<tr>
<td>S.D 0(R1,F4)</td>
<td>;store result</td>
<td>1</td>
</tr>
<tr>
<td>DADDUI R1,R1,−8</td>
<td>;decrement pointer 8Bytes (DW)</td>
<td>1</td>
</tr>
<tr>
<td>BNEZ R1,Loop</td>
<td>;branch R1!=zero</td>
<td>1</td>
</tr>
</tbody>
</table>

Loop takes 7 clock cycles
- 3 for execution (L.D,ADD.D,S.D)
- 4 for loop overhead

How to make faster?
Unroll Original Loop Four Times

- Straightforward way for time saving!!

Four loops take 27 clock cycles, or 6.75 per iteration!!
(Assumes R1 is a multiple of 4)

for (i=1000; i>0; i=i–1)

\[ x[i] = x[i] + s; \]

1 Loop:

\[ \text{L.D} \ F0,0(R1) \]

3 ADD.D \[ F4,F0,F2 \]

6 S.D \[ 0(R1),F4 \]

9 ADD.D \[ F8,F0,F2 \]

12 S.D \[ -8(R1),F8 \]

15 ADD.D \[ F12,F10,F2 \]

18 S.D \[ -16(R1),F12 \]

21 ADD.D \[ F16,F14,F2 \]

24 S.D \[ -24(R1),F16 \]

27 DADDUI \[ R1,R1,#-32 \]

[Alter to \( 4^*8 \)]

1 cycle stall

2 cycle stall

1 cycle stall

How to rewrite loop to minimize stalls?

Unrolled Loop with Minimal Stalls

- Four loops take 14 clock cycles, or 3.5 per loop

Loop Unrolling Detail - Strip Mining

- Do not usually know upper bound of loop

- Suppose it is \( n \), and we would like to unroll the loop to make \( k \) copies of the body

- Instead of a single unrolled loop, generate a pair of consecutive loops:
  - 1st executes \( (n \mod k) \) times and has a body that is the original loop (called ”strip mining” of a loop)
  - 2nd is the unrolled body surrounded by an outer loop that iterates \( \lfloor n/k \rfloor \) times

- For large values of \( n \), most of the execution time will be spent in the \( \lfloor n/k \rfloor \) unrolled loops

Five 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 if loop unrolling can be useful by finding that loop iterations are independent (except for loop maintenance code)
  2. Use different registers to avoid unnecessary constraints forced by using the same registers for different computations
  3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration increment/decrement code
  4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
    - Transformation requires analyzing memory addresses and finding that no pairs refer to the same address
  5. Schedule (reorder) the code, preserving any dependences needed to yield the same result as the original code
Three Limits to Loop Unrolling

1. Decrease in amount of overhead amortized with each extra unrolling – Amdahl’s Law
2. Growth in code size
   - For larger loops, size is a concern if it increases the instruction cache miss rate
3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
   - If not possible to allocate all live values to registers, code may lose some or all of the advantages of loop unrolling

"Software pipelining" is an older compiler technique to unroll loops systematically.

Loop unrolling reduces the impact of branches on pipelines; another way is branch prediction.

Compiler Software-Pipelining of $V=S*S$ Loop

- Software pipelining structure tolerates the long latencies of FP operations (l.s, mul.s, s.s are single precision (SP) FP. Load, Multiply, Store.)
- At start: $r1=addr[V(0)]; r2=addr[V(last)]+4; f0 = scalar SP FP multiplier.
- Instructions in iteration box are in reverse order, from different iterations. If have separate FP function boxes for L, M, S, can overlap S M L triples.
- Bg: marks prologue starting iterated code;
- Ep: marks epilogue to finish code.

I       TIME
E 1 2 3 4 5 6 7 8
S
R 2
M S
A 3
L M S
T 4
L M S
O 6
L M S
N

```
l.s f2,0(r1)          mul.s f4,f0,f2
s.s f4,0(r1)          mul.s f4,f0,f2
addi r1,r1,4         addi r1,r1,4
bne r1,r2,lp         bne r1,r2,lp
l.s f2,0(r1)          mul.s f4,f0,f2
l.s f2,0(r1)          mul.s f4,f0,f2
addi r1,r1,4         addi r1,r1,4
bne r1,r2,lp         bne r1,r2,lp
```

```
l.s f4,0(r1)          mul.s f4,f0,f2
s.s f4,0(r1)          mul.s f4,f0,f2
addi r1,r1,4         addi r1,r1,4
bne r1,r2,lp
```