 ILP: Concepts and Challenges

- ILP (Instruction Level Parallelism) – overlap execution of unrelated instructions
- Techniques that increase amount of parallelism exploited among instructions
  - reduce impact of data and control hazards
  - increase processor ability to exploit parallelism
- Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls
  - Reducing each of the terms of the right-hand side minimize CPI and thus increase instruction throughput

Basic Pipeline Scheduling: Example

- Simple loop: for(i=1; i<=1000; i++)
  x[i] = x[i] + s;

Assumptions:

<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>
<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>

R1 points to the last element in the array.

for simplicity, we assume that x[0] is at the address 0

Loop:

- L.D F0, O(R1); F0 = array el.
- ADD D F4, F0, F2; add scalar in F2
- S.D 0(R1), F4; store result
- SUBI R1, R1, #8; decrement pointer
- BNEI R1, Loop; branch
Executing FP Loop

1. Loop: LD F0, 0(R1)
2. Stall
3. ADDD F4, F0, F2
4. Stall
5. Stall
6. SD 0(R1), F4
7. SUBI R1, R1, #8
8. Stall
9. BNEZ R1, Loop
10. Stall

10 clocks per iteration (5 stalls)

=> Rewrite code to minimize stalls?

Revised FP loop to minimise stalls

1. Loop: LD F0, 0(R1)
2. SUBI R1, R1, #8
3. ADDD F4, F0, F2
4. Stall
5. BNEZ R1, Loop :delayed branch
6. SD 8(R1), F4 :altered and interch. SUBI

5 clocks per iteration (1 stall); but only 3 instructions do the actual work processing the array (LD, ADDD, SD)

Unroll loop 4 times to improve potential for instr. scheduling

Unrolled Loop

This loop will run 28 cc (14 stalls) per iteration; each LD has one stall, each ADDD 2, SUBI 1, BNEZ 1, plus 14 instruction issue cycles - or 28/4=7 for each element of the array (even slower than the scheduled version)!

=> Rewrite loop to minimize stalls

Where are the name dependencies?

1. Loop: LD F0, 0(R1)
2. ADDD F4, F0, F2
3. SD 0(R1), F4 ;drop DSUBUI & BNEZ
4. LD F0, -8(R1)
5. ADDD F4, F0, F2
6. SD -8(R1), F4 ;drop DSUBUI & BNEZ
7. LD F0, -16(R1)
8. ADDD F4, F0, F2
9. SD -16(R1), F4 ;drop DSUBUI & BNEZ
10. LD F0, -24(R1)
11. ADDD F4, F0, F2
12. SD -24(R1), F4
13. SUBUI R1, R1, #32 ;alter to 4*8
14. BNEZ R1, Loop
15. NOP

How can remove them?
Where are the name dependencies?

1 Loop: 
   L.D F0,0(R1)
   ADD.D F4,F0,F2
   S.D -8(R1),F4
   L.D F6,-8(R1)
   ADD.D F8,F6,F2
   S.D -8(R1),F8
   L.D F10,-16(R1)
   ADD.D F12,F10,F2
   S.D -16(R1),F12
   L.D F14,-24(R1)
   ADD.D F16,F14,F2
   S.D -24(R1),F16
   DSUBUI R1,R1, #32
   BNEZ R1,LOOP
   NOP

The Original "register renaming"

Unrolled Loop that Minimise Stalls

1 Loop:
   LD F0,0(R1)
   LD F6,-8(R1)
   LD F10,-16(R1)
   LD F14,-24(R1)
   ADDD F4,F0,F2
   ADDD F8,F6,F2
   ADDD F12,F10,F2
   ADDD F16,F14,F2
   SD 0(R1),F4
   SD -8(R1),F8
   SUBI R1,R1,#32
   SD 16(R1),F12
   BNEZ R1,Loop
   SD 8(R1),F4
   NOP

This loop will run 14 cycles (no stalls) per iteration; or 14/4=3.5 for each element!

Assumptions that make this possible:
- move LDs before SDs
- move SD after SUBI and BNEZ
- use different registers

When is it safe for compiler to do such changes?

Steps Compiler Performed to Unroll

- Determine that is OK to move the S.D after SUBUI and BNEZ, and find amount to adjust SD offset
- Determine that unrolling the loop would be useful by finding that the loop iterations were independent
- Rename registers to avoid name dependencies
- Eliminate extra test and branch instructions and adjust the loop termination and iteration code
- Determine loads and stores in unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent
  - requires analyzing memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield same result as the original code

Multiple Issue

- Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls
- Decrease Ideal pipeline CPI
- Multiple issue
  - Superscalar
    - Statically scheduled (compiler techniques)
    - Dynamically scheduled (Tomasulo’s alg.)
  - VLIW (Very Long Instruction Word)
    - parallelism is explicitly indicated by instruction
    - EPIC (explicitly parallel instruction computers)

Multiple Issue

- Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls
- Decrease Ideal pipeline CPI
- Multiple issue
  - Superscalar
    - Statically scheduled (compiler techniques)
    - Dynamically scheduled (Tomasulo’s alg.)
  - VLIW (Very Long Instruction Word)
    - parallelism is explicitly indicated by instruction
    - EPIC (explicitly parallel instruction computers)
### Superscalar MIPS

- Superscalar MIPS: 2 instructions, 1 FP & 1 anything else
  - Fetch 64-bits/clock cycle; int on left, FP on right
  - Can only issue 2nd instruction if 1st instruction issues
  - More ports for FP registers to do FP load & FP op in a pair

![Superscalar MIPS Diagram](Image)

#### Loop Unrolling in Superscalar

Unrolled 5 times to avoid delays

This loop will run 12 cycles (no stalls) per iteration - or 12/5=2.4 for each element of the array

<table>
<thead>
<tr>
<th>Integer Instr.</th>
<th>FP Instr.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: FP operations extend EX cycle

### Multiple Issue Processors

- Two variations
  - Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
    - IBM PowerPC, Sun UltraSparc, DEC Alpha, HP 8000
  - (Very) Long Instruction Words (V)LIW: fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates
    - Crusoe VLIW processor [www.transmeta.com](http://www.transmeta.com)
    - Intel Architecture-64 (IA-64) 64-bit address
    - Style: “Explicitly Parallel Instruction Computer (EPIC)”

- Anticipated success lead to use of Instructions Per Clock cycle (IPC) vs. CPI

### The VLIW Approach

- VLIWs use multiple independent functional units
- VLIWs package the multiple operations into one very long instruction
- Compiler is responsible to choose instructions to be issued simultaneously

![VLIW Approach Diagram](Image)
Unrolled 7 times to avoid delays
7 results in 9 clocks, or 1.3 clocks per element (1.8X)
Average: 2.5 ops per clock, 50% efficiency
Note: Need more registers in VLIW (15 vs. 6 in SS)

When Safe to Unroll Loop?
- Example: Where are data dependencies?
  \((A,B,C)\) distinct & nonoverlapping
  for \(i=0; i<100; \ i=\ i+1\) \{ 
    \(A[i+1] = A[i] + C[i]; \quad /* S1 */\)
    \(B[i+1] = B[i] + A[i+1]; \quad /* S2 */\)
  \}
- 1. S2 uses the value, \(A[i+1]\),
  computed by S1 in the same iteration
- 2. S1 uses a value computed by S1 in an earlier iteration,
  since iteration \(i\) computes \(A[i+1]\) which is
  read in iteration \(i+1\). The same is true of S2 for
  \(B[i]\) and \(B[i+1]\)
- This is a "loop-carried dependence": between iterations
- For our prior example, each iteration was distinct

Does a loop-carried dependence mean there is no parallelism???
- Consider:
  \[
  \text{for } (i=0; \ i<8; \ i=i+1) \{ 
  A = A + C[i]; \quad /* S1 */ 
  \}
  \]
- Could compute:
  "Cycle 1":
  \[
  \text{temp0} = C[0] + C[1]; \\
  \text{temp1} = C[2] + C[3]; \\
  \text{temp2} = C[4] + C[5]; \\
  \text{temp3} = C[6] + C[7]; \\
  \]
  "Cycle 2":
  \[
  \text{temp4} = \text{temp0} + \text{temp1}; \\
  \text{temp5} = \text{temp2} + \text{temp3}; \\
  \]
  "Cycle 3":
  \[
  A = \text{temp4} + \text{temp5}; \\
  \]
- Relies on associative nature of "+".
Another Example

- Loop carried dependences?
  
  ```
  for (i=1; i<100; i=i+1) {
    A[i] = A[i] + B[i];  /* S1 */
    B[i+1] = C[i] + D[i]; /* S2 */
  }
  ```

- To overlap iteration execution:
  ```
  for (i=1; i<100; i=i+1) {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
  }
  ```

  ```
  B[101] = C[100] + D[100];
  ```

Another possibility: Software Pipelining

- Observation: if iterations from loops are independent, then can get more ILP by taking instructions from different iterations.

- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop (~ Tomasulo in SW).

Software Pipelining Example

- Before: Unrolled 3 times
  ```
  1 LD   F0,0(R1)
  2 ADD  F4,F0,F2
  3 SD   0(R1),F4
  4 LD   F6,-8(R1)
  5 ADD  F8,F6,F2
  6 SD   -8(R1),F8
  7 LD   F10,-16(R1)
  8 ADD  F12,F10,F2
  9 SD   -16(R1),F12
  10 SUBU R1,R1,R24
  11 BNEZ R1,LOOP
  ```

- After: Software Pipelined
  ```
  1 LD   F0,0(R1)
  2 ADD  F4,F0,F2
  3 ADD  F8,F4,F2
  4 ADD  F12,F8,F2
  5 BNEZ R1,LOOP
  ```

Things to Remember

- Pipeline CPI = Ideal pipeline CPI + Structural stalls + RAW stalls + WAR stalls + WAW stalls + Control stalls

- Loop unrolling to minimise stalls

- Multiple issue to minimise CPI
  - Superscalar processors
  - VLIW architectures

Symbolic Loop Unrolling

- Maximize result-use distance
- Less code space than unrolling
- Fill & drain pipe only once per loop
- Once per each unrolled iteration in loop unrolling
**Statically Scheduled Superscalar**

- E.g., four-issue static superscalar
  - 4 instructions make one issue packet
  - Fetch examines each instruction in the packet in the program order
    - instruction cannot be issued
    - will cause a structural or data hazard
      - either due to an instruction earlier in the issue packet or due to an instruction already in execution
    - can issue from 0 to 4 instruction per clock cycle

**Multiple Issue with Dynamic Scheduling**

**Loop:**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>1st Issue</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td></td>
<td>Wait for ADD.D</td>
</tr>
<tr>
<td>S.D (R1), F4</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td>Wait for ALU</td>
</tr>
<tr>
<td>DADDIU R1,R1,-#8</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
<td>8</td>
<td>11</td>
<td></td>
<td></td>
<td>Wait for DADDU</td>
</tr>
<tr>
<td>L.D F0,0(R1)</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>Wait for ALU</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>10</td>
<td>13</td>
<td></td>
<td></td>
<td>Wait for DADDU</td>
</tr>
<tr>
<td>S.D (R1), F4</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>Wait for ALU</td>
</tr>
<tr>
<td>DADDIU R1,R1,-#8</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>Wait for DADDU</td>
</tr>
<tr>
<td>L.D F0,0(R1)</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td>Wait for ALU</td>
</tr>
<tr>
<td>ADD.D F4,F0,F2</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td>Wait for DADDU</td>
</tr>
<tr>
<td>S.D (R1), F4</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td>Wait for ALU</td>
</tr>
</tbody>
</table>

Assumptions:

- One FP and one integer operation can be issued;
- Resources: ALU (int + effective address), a separate pipelined FP for each operation type, branch prediction hardware, 1 CDB
- 2 cc for loads, 3 cc for FP Add
- Branches single issue, branch prediction is perfect
Multiple Issue with Dynamic Scheduling:

**Resource Usage**

<table>
<thead>
<tr>
<th>Inst.</th>
<th>Issue</th>
<th>Exe.</th>
<th>Mem.</th>
<th>Write to CDB</th>
<th>CDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>2</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>3</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>4</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>5</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>6</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>7</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>8</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>9</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>10</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>11</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>12</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>13</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>14</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>15</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>16</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>17</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
<tr>
<td>18</td>
<td>S.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
<td>1/ADD.D</td>
</tr>
</tbody>
</table>

Multiple Issue with Dynamic Scheduling:

- **DADDIU** waits for ALU used by S.D
  - Add one ALU dedicated to effective address calculation
  - Use 2 CDBs
- Draw table for the dual-issue version of Tomasulo’s pipeline