Lecture: Static ILP

- Topics: compiler scheduling, loop unrolling, software pipelining (Sections C.5, 3.2)
Static vs Dynamic Scheduling

• Arguments against dynamic scheduling:
  ➢ requires complex structures to identify independent instructions (scoreboards, issue queue)
    ▪ high power consumption
    ▪ low clock speed
    ▪ high design and verification effort
  ➢ the compiler can “easily” compute instruction latencies and dependences – complex software is always preferred to complex hardware (?)
ILP

• Instruction-level parallelism: overlap among instructions: pipelining or multiple instruction execution

• What determines the degree of ILP?
  ➢ dependences: property of the program
  ➢ hazards: property of the pipeline
Loop Scheduling

• The compiler’s job is to minimize stalls

• Focus on loops: account for most cycles, relatively easy to analyze and optimize
Assumptions

• Load: 2-cycles  (1 cycle stall for consumer)
• FP ALU: 4-cycles (3 cycle stall for consumer; 2 cycle stall if the consumer is a store)
• One branch delay slot
• Int ALU: 1-cycle (no stall for consumer, 1 cycle stall if the consumer is a branch)
Loop Example

for (i=1000; i>0; i--)
    x[i] = x[i] + s;

Loop:
    L.D    F0, 0(R1) ; F0 = array element
    ADD.D  F4, F0, F2 ; add scalar
    S.D    F4, 0(R1) ; store result
    DADDUI R1, R1,# -8 ; decrement address pointer
    BNE    R1, R2, Loop ; branch if R1 != R2
    NOP

Source code

Assembly code
## Loop Example

For (i=1000; i>0; i--)

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

### Source code

```
for (i=1000; i>0; i--)
    x[i] = x[i] + s;
```

### Assembly code

```
Loop:   L.D        F0, 0(R1)  ; F0 = array element
    ADD.D     F4, F0, F2  ; add scalar
    S.D       F4, 0(R1)  ; store result
    DADDUI    R1, R1,# -8 ; decrement address pointer
    BNE       R1, R2, Loop ; branch if R1 != R2
    NOP
```

### 10-cycle schedule

- LD -> any : 1 stall
- FPALU -> any: 3 stalls
- FPALU -> ST : 2 stalls
- IntALU -> BR : 1 stall
Smart Schedule

- By re-ordering instructions, it takes 6 cycles per iteration instead of 10.
- We were able to violate an anti-dependence easily because an immediate was involved.
- Loop overhead (instrs that do book-keeping for the loop): 2
  Actual work (the ld, add.d, and s.d): 3 instrs
  Can we somehow get execution time to be 3 cycles per iteration?
### Loop Unrolling

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Instruction</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4, F0, F2</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F4, 0(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F6, -8(R1)</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F8, F6, F2</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F8, -8(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F10, -16(R1)</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F12, F10, F2</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F12, -16(R1)</td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>F14, -24(R1)</td>
<td></td>
</tr>
<tr>
<td>ADD.D</td>
<td>F16, F14, F2</td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>F16, -24(R1)</td>
<td></td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1, #32</td>
<td></td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2, Loop</td>
<td></td>
</tr>
</tbody>
</table>

- Loop overhead: 2 instrs; Work: 12 instrs
- How long will the above schedule take to complete?
Scheduled and Unrolled Loop

Loop:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D</td>
<td>F0</td>
<td>0(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F6</td>
<td>-8(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F10</td>
<td>-16(R1)</td>
</tr>
<tr>
<td>L.D</td>
<td>F14</td>
<td>-24(R1)</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F4</td>
<td>F0</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F8</td>
<td>F6</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F12</td>
<td>F10</td>
</tr>
<tr>
<td>ADD.D</td>
<td>F16</td>
<td>F14</td>
</tr>
<tr>
<td>S.D</td>
<td>F4</td>
<td>0(R1)</td>
</tr>
<tr>
<td>S.D</td>
<td>F8</td>
<td>-8(R1)</td>
</tr>
<tr>
<td>DADDUI</td>
<td>R1, R1</td>
<td># -32</td>
</tr>
<tr>
<td>S.D</td>
<td>F12</td>
<td>16(R1)</td>
</tr>
<tr>
<td>BNE</td>
<td>R1, R2</td>
<td>Loop</td>
</tr>
<tr>
<td>S.D</td>
<td>F16</td>
<td>8(R1)</td>
</tr>
</tbody>
</table>

- Execution time: 14 cycles or 3.5 cycles per original iteration

LD -> any : 1 stall
FPALU -> any: 3 stalls
FPALU -> ST : 2 stalls
IntALU -> BR : 1 stall
Loop Unrolling

- Increases program size
- Requires more registers

- To unroll an n-iteration loop by degree k, we will need \((n/k)\) iterations of the larger loop, followed by \((n \text{ mod } k)\) iterations of the original loop
Automating Loop Unrolling

• Determine the dependences across iterations: in the example, we knew that loads and stores in different iterations did not conflict and could be re-ordered

• Determine if unrolling will help – possible only if iterations are independent

• Determine address offsets for different loads/stores

• Dependency analysis to schedule code without introducing hazards; eliminate name dependences by using additional registers
## Superscalar Pipelines

<table>
<thead>
<tr>
<th>Integer pipeline</th>
<th>FP pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Handles L.D, S.D, ADDUI, BNE</td>
<td>Handles ADD.D</td>
</tr>
</tbody>
</table>

- What is the schedule with an unroll degree of 4?
## Superscalar Pipelines

<table>
<thead>
<tr>
<th>Loop:</th>
<th>Integer pipeline</th>
<th>FP pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>L.D F0,0(R1)</td>
<td>ADD.D F4,F0,F2</td>
</tr>
<tr>
<td></td>
<td>L.D F6,-8(R1)</td>
<td>ADD.D F8,F6,F2</td>
</tr>
<tr>
<td></td>
<td>L.D F10,-16(R1)</td>
<td>ADD.D F12,F10,F2</td>
</tr>
<tr>
<td></td>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F16,F14,F2</td>
</tr>
<tr>
<td></td>
<td>L.D F18,-32(R1)</td>
<td>ADD.D F20,F18,F2</td>
</tr>
<tr>
<td></td>
<td>S.D F4,0(R1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S.D F8,-8(R1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S.D F12,-16(R1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DADDUI R1,R1,# -40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S.D F16,16(R1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BNE R1,R2,Loop</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S.D F20,8(R1)</td>
<td></td>
</tr>
</tbody>
</table>

- Need unroll by degree 5 to eliminate stalls
- The compiler may specify instructions that can be issued as one packet
- The compiler may specify a fixed number of instructions in each packet: Very Large Instruction Word (VLIW)
Loop:  L.D  F0, 0(R1)
ADD.D  F4, F0, F2
S.D    F4, 0(R1)
DADDUI R1, R1,# -8
BNE    R1, R2, Loop
Software Pipeline

Original iter 1

Original iter 2

Original iter 3

Original iter 4

New iter 1

New iter 2

New iter 3

New iter 4
Software Pipelining

Loop:  
    L.D      F0, 0(R1)
    ADD.D    F4, F0, F2
    S.D      F4, 0(R1)
    DADDUI   R1, R1,# -8
    BNE      R1, R2, Loop

Loop:  
    S.D      F4, 16(R1)
    ADD.D    F4, F0, F2
    L.D      F0, 0(R1)
    DADDUI   R1, R1,# -8
    BNE      R1, R2, Loop

- Advantages: achieves nearly the same effect as loop unrolling, but without the code expansion – an unrolled loop may have inefficiencies at the start and end of each iteration, while a sw-pipelined loop is almost always in steady state – a sw-pipelined loop can also be unrolled to reduce loop overhead

- Disadvantages: does not reduce loop overhead, may require more registers
Title

- Bullet