Lecture: Static ILP

- Topics: static ILP approaches, scheduling, loop unrolling, software pipelines
Problem 2

• Assume an unpipelined processor where it takes 5ns to go through the circuits and 0.1ns for the latch overhead. What is the throughput for 1-stage, 20-stage and 50-stage pipelines? Assume that the P.O.P and P.O.C in the unpipelined processor are separated by 2ns. Assume that half the instructions do not introduce a data hazard and half the instructions depend on their preceding instruction.

• 1-stage: 1 instr every 5.1ns
• 20-stage: first instr takes 0.35ns, the second takes 2.8ns
• 50-stage: first instr takes 0.2ns, the second takes 4ns
• Throughputs: 0.20 BIPS, 0.63 BIPS, and 0.48 BIPS
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
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 (?)
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

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

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

#### Source code

```plaintext
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
```

#### Assembly code

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

---

**10-cycle schedule**

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

---

**Chart:**

- 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?
Problem 1

Source code

\[
\text{for (i=1000; i>0; i--)} \\
\quad x[i] = y[i] * s;
\]

Assembly code

L.D F0, 0(R1) ; F0 = array element
MUL.D F4, F0, F2 ; multiply scalar
S.D F4, 0(R2) ; store result
DADDUI R1, R1,# -8 ; decrement address pointer
DADDUI R2, R2,#-8 ; decrement address pointer
BNE R1, R3, Loop ; branch if R1 != R3
NOP

• How many cycles do the default and optimized schedules take?
Problem 1

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

Source code

Loop:    L.D         F0, 0(R1)          ; F0 = array element
         MUL.D    F4, F0, F2        ; multiply scalar
         S.D         F4, 0(R2)          ; store result
         DADDUI  R1, R1,# -8      ; decrement address pointer
         DADDUI  R2, R2,#-8       ; decrement address pointer
         BNE        R1, R3, Loop    ; branch if R1 != R3
         NOP

Assembly code

- How many cycles do the default and optimized schedules take?

Unoptimized: LD 1s  MUL 4s  SD DA DA BNE 1s  -- 12 cycles

Optimized: LD DA MUL DA 2s BNE SD -- 8 cycles
Loop Unrolling

Loop:

- L.D F0, 0(R1)
- ADD.D F4, F0, F2
- S.D F4, 0(R1)
- L.D F6, -8(R1)
- ADD.D F8, F6, F2
- S.D F8, -8(R1)
- L.D F10, -16(R1)
- ADD.D F12, F10, F2
- S.D F12, -16(R1)
- L.D F14, -24(R1)
- ADD.D F16, F14, F2
- S.D F16, -24(R1)
- DADDUI R1, R1, #-32
- BNE R1, R2, Loop

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

Loop:  
L.D  F0, 0(R1)  
L.D  F6, -8(R1)  
L.D  F10, -16(R1)  
L.D  F14, -24(R1)  
ADD.D  F4, F0, F2  
ADD.D  F8, F6, F2  
ADD.D  F12, F10, F2  
ADD.D  F16, F14, F2  
S.D  F4, 0(R1)  
S.D  F8, -8(R1)  
DADDUI  R1, R1, # -32  
S.D  F12, 16(R1)  
BNE  R1,R2, Loop  
S.D  F16, 8(R1)  

• Execution time: 14 cycles or 3.5 cycles per original iteration
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 \((n \mod k)\) 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.
Problem 2

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

Source code

Loop: L.D F0, 0(R1) ; F0 = array element
      MUL.D F4, F0, F2 ; multiply scalar
      S.D F4, 0(R2) ; store result
      DADDUI R1, R1,#-8 ; decrement address pointer
      DADDUI R2, R2,#-8 ; decrement address pointer
      BNE R1, R3, Loop ; branch if R1 != R3
      NOP

Assembly code

• How many unrolls does it take to avoid stall cycles?
Problem 2

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

Loop:     L.D         F0, 0(R1)          ; F0 = array element
          MUL.D    F4, F0, F2        ; multiply scalar
          S.D         F4, 0(R2)          ; store result
          DADDUI  R1, R1,# -8      ; decrement address pointer
          DADDUI  R2, R2,#-8       ; decrement address pointer
          BNE        R1, R3, Loop    ; branch if R1 != R3
          NOP

• How many unrolls does it take to avoid stall cycles?

Degree 2: LD LD MUL MUL DA DA 1s SD BNE SD
Degree 3: LD LD LD MUL MUL MUL DA DA SD SD BNE SD
          – 12 cyc/3 iterations
## 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 5?
### Superscalar Pipelines

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

- Need unroll by degree 5 to eliminate stalls (fewer if we move DADDUI up)
- 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)
Problem 3

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

Source code

Loop:  L.D    F0, 0(R1)    ; F0 = array element
       MUL.D  F4, F0, F2    ; multiply scalar
       S.D    F4, 0(R2)     ; store result
       DADDUI R1, R1,# -8  ; decrement address pointer
       DADDUI R2, R2,#-8   ; decrement address pointer
       BNE    R1, R3, Loop  ; branch if R1 != R3
       NOP

Assembly code

• How many unrolls does it take to avoid stalls in the superscalar pipeline?
Problem 3

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

Source code

Loop:     L.D         F0, 0(R1)          ; F0 = array element
          MUL.D    F4, F0, F2        ; multiply scalar
          S.D         F4, 0(R2)          ; store result
          DADDUI  R1, R1,# -8      ; decrement address pointer
          DADDUI  R2, R2,#-8       ; decrement address pointer
          BNE        R1, R3, Loop    ; branch if R1 != R3
          NOP

Assembly code

LD -> any : 1 stall
FPMUL -> any: 5 stalls
FPMUL -> ST : 4 stalls
IntALU -> BR : 1 stall

• How many unrolls does it take to avoid stalls in the superscalar pipeline?
  7 unrolls. Could also make do with 5 if we moved up the DADDUIs.
Software Pipeline?!

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
Problem 4

```c
for (i=1000; i>0; i--)
    x[i] = y[i] * s;
```

### Source code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0, 0(R1)</td>
<td>F0 = array element</td>
</tr>
<tr>
<td>MUL.D F4, F0, F2</td>
<td>multiply scalar</td>
</tr>
<tr>
<td>S.D F4, 0(R2)</td>
<td>store result</td>
</tr>
<tr>
<td>DADDUI R1, R1,#-8</td>
<td>decrement address pointer</td>
</tr>
<tr>
<td>DADDUI R2, R2,#-8</td>
<td>decrement address pointer</td>
</tr>
<tr>
<td>BNE R1, R3, Loop</td>
<td>branch if R1 != R3</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
</tr>
</tbody>
</table>

### Assembly code

- LD -> any : 1 stall
- FPMUL -> any: 5 stalls
- FPMUL -> ST : 4 stalls
- IntALU -> BR : 1 stall

**Show the SW pipelined version of the code and does it cause stalls?**
Problem 4

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

Source code

Loop:  L.D  F0, 0(R1) ; F0 = array element
       MUL.D F4, F0, F2 ; multiply scalar
       S.D  F4, 0(R2) ; store result
       DADDUI R1, R1, # -8 ; decrement address pointer
       DADDUI R2, R2, # -8 ; decrement address pointer
       BNE  R1, R3, Loop ; branch if R1 != R3
       NOP

Assembly code

• Show the SW pipelined version of the code and does it cause stalls?

Loop:  S.D  F4, 0(R2)
       MUL  F4, F0, F2
       L.D  F0, 0(R1)
       DADDUI R2, R2, # -8
       BNE  R1, R3, Loop
       DADDUI R1, R1, # -8

There will be no stalls