Lecture: Static vs. Dynamic ILP

- Topics: software pipelines, predication, branch prediction intro
- Midterm date switched to Oct 6 (HW 5 after fall break)
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
Recall Superscalar Pipeline Example

<table>
<thead>
<tr>
<th>Integer pipeline</th>
<th>FP pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop:</td>
<td></td>
</tr>
<tr>
<td>L.D F0,0(R1)</td>
<td>ADD.D F4,F0,F2</td>
</tr>
<tr>
<td>L.D F6,-8(R1)</td>
<td>ADD.D F8,F6,F2</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>ADD.D F12,F10,F2</td>
</tr>
<tr>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F16,F14,F2</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>ADD.D F20,F18,F2</td>
</tr>
<tr>
<td>S.D F4,0(R1)</td>
<td></td>
</tr>
<tr>
<td>S.D F8,-8(R1)</td>
<td></td>
</tr>
<tr>
<td>S.D F12,-16(R1)</td>
<td></td>
</tr>
<tr>
<td>DADDUI R1,R1,# -40</td>
<td></td>
</tr>
<tr>
<td>S.D F16,16(R1)</td>
<td></td>
</tr>
<tr>
<td>BNE R1,R2,Loop</td>
<td></td>
</tr>
<tr>
<td>S.D 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 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?
Problem 4

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

Source code

Assembly 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

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?

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
Predication

- A branch within a loop can be problematic to schedule

- Control dependences are a problem because of the need to re-fetch on a mispredict

- For short loop bodies, control dependences can be converted to data dependences by using predicated/conditional instructions
Predicated or Conditional Instructions

if (R1 == 0)
  R2 = R2 + R4
else
  R6 = R3 + R5
  R4 = R2 + R3

R7 = !R1
R8 = R2
R2 = R2 + R4  (predicated on R7)
R6 = R3 + R5  (predicated on R1)
R4 = R8 + R3  (predicated on R1)
Predicated or Conditional Instructions

• The instruction has an additional operand that determines whether the instr completes or gets converted into a no-op

• Example: lwc R1, 0(R2), R3  (load-word-conditional) will load the word at address (R2) into R1 if R3 is non-zero; if R3 is zero, the instruction becomes a no-op

• Replaces a control dependence with a data dependence (branches disappear); may need register copies for the condition or for values used by both directions

if (R1 == 0)
    R2 = R2 + R4
else
    R6 = R3 + R5
R4 = R2 + R3

R7 = !R1 ; R8 = R2 ;
R2 = R2 + R4  (predicated on R7)
R6 = R3 + R5  (predicated on R1)
R4 = R8 + R3  (predicated on R1)
Problem 1

• Use predication to remove control hazards in this code

```plaintext
if (R1 == 0)
    R2 = R5 + R4
    R3 = R2 + R4
else
    R6 = R3 + R2
```
Problem 1

• Use predication to remove control hazards in this code

```plaintext
if (R1 == 0)
    R2 = R5 + R4
    R3 = R2 + R4
else
    R6 = R3 + R2
    R7 = !R1 ;
    R6 = R3 + R2  (predicated on R1)
    R2 = R5 + R4  (predicated on R7)
    R3 = R2 + R4  (predicated on R7)
```
Complications

• Each instruction has one more input operand – more register ports/bypassing

• If the branch condition is not known, the instruction stalls (remember, these are in-order processors)

• Some implementations allow the instruction to continue without the branch condition and squash/complete later in the pipeline – wasted work

• Increases register pressure, activity on functional units

• Does not help if the br-condition takes a while to evaluate
Support for Speculation

• When re-ordering instructions, we need hardware support
  ➢ to ensure that an exception is raised at the correct point
  ➢ to ensure that we do not violate memory dependences
Detecting Exceptions

- Some exceptions require that the program be terminated (memory protection violation), while other exceptions require execution to resume (page faults).

- For a speculative instruction, in the latter case, servicing the exception only implies potential performance loss.

- In the former case, you want to defer servicing the exception until you are sure the instruction is not speculative.

- Note that a speculative instruction needs a special opcode to indicate that it is speculative.
Static vs. Dynamic

• To get high performance with a compiler-based approach, we need support for predication, tables to analyze dependences, etc. Plus, scheduling goes haywire if there are cache misses.

• Difficult to achieve the highest performance with a purely static (compiler-based) approach – it continues to have value for highly simple in-order processors

• But for highest performance, dynamic/hardware approaches are most effective, and the compiler can help such processors as well
Amdahl’s Law

• Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power

• Amdahl’s Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play

• Example: a web server spends 40% of time in the CPU and 60% of time doing I/O – a new processor that is ten times faster results in a 36% reduction in execution time (speedup of 1.56) – Amdahl’s Law states that maximum execution time reduction is 40% (max speedup of 1.66)
Principle of Locality

• Most programs are predictable in terms of instructions executed and data accessed

• The 90-10 Rule: a program spends 90% of its execution time in only 10% of the code

• Temporal locality: a program will shortly re-visit $X$

• Spatial locality: a program will shortly visit $X+1$
Pipeline without Branch Predictor

In the 5-stage pipeline, a branch completes in two cycles →
If the branch went the wrong way, one incorrect instr is fetched →
One stall cycle per incorrect branch
Pipeline with Branch Predictor

In the 5-stage pipeline, a branch completes in two cycles →
If the branch went the wrong way, one incorrect instr is fetched →
One stall cycle per incorrect branch
1-Bit Bimodal Prediction

• For each branch, keep track of what happened last time and use that outcome as the prediction

• What are prediction accuracies for branches 1 and 2 below:

```c
while (1) {
    for (i=0;i<10;i++) {                     branch-1
        ...
    }
    for (j=0;j<20;j++) {                     branch-2
        ...
    }
}
```
Bimodal 1-Bit Predictor

Branch PC

10 bits

Table of 1K entries

Each entry is a bit

The table keeps track of what the branch did last time