Instruction-Level Parallelism (ILP)

• Pipelining
  – Limited form of ILP
  – Overlapping instructions, these instructions can be evaluated in parallel (to some degree)
  – Pipeline CPI = Idea pipeline CPI + Structural Stalls + RAW stalls = WAR stalls + WAW stalls + Control Stalls

• Focus now on the Control Stalls!
  – Methods to reduce the control stalls
  – Will use both hardware and software (i.e. compiler) methods

Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control placement</td>
<td>Control</td>
</tr>
<tr>
<td>Basic pipeline scheduling (we’re not sure if this exists)</td>
<td>RAW stalls</td>
</tr>
<tr>
<td>Dynamic scheduling with branch prediction</td>
<td>RAW stalls</td>
</tr>
<tr>
<td>Branch frequency was 15%</td>
<td></td>
</tr>
<tr>
<td>If evenly distributed, this means a branch every six or seven instructions</td>
<td></td>
</tr>
<tr>
<td>Means basic blocks are pretty small, not too much to exploit for parallelism</td>
<td></td>
</tr>
<tr>
<td>To get better performance we must exploit ILP across multiple basic blocks</td>
<td></td>
</tr>
<tr>
<td>Easiest target is the LOOP</td>
<td></td>
</tr>
</tbody>
</table>

ILP and Basic Blocks

• Basic Block
  – Straight line code without any branches except entry and exit
  – I.e. real code consists of multiple basic blocks connected via branches

• Branch frequency was 15%
  – If evenly distributed, this means a branch every six or seven instructions
  – Means basic blocks are pretty small, not too much to exploit for parallelism
  – To get better performance we must exploit ILP across multiple basic blocks
  – Easiest target is the LOOP
Loop Level Parallelism

- **Trivial Example**
  - for (i=1; i<=1000; i++)
    - $x[i] = x[i] + y[i]$;

- **Note**
  - No dependence between data values in iteration $i$ and iteration $i+1$, so each loop iteration is truly independent
  - We could execute all 1000 of these in parallel if we could!
  - Since independent, No data hazards $\rightarrow$ No stalls
  - **BUT**
    - There is a branch to implement the loop, ruining all this nice parallelism
    - Although note prediction is pretty easy here, but in general it may be more difficult

- **Vector Processor Model**
  - If we could execute four adds in parallel, we could have a loop up to 250 and simultaneously add $x[i]$, $x[i+1]$, $x[i+2]$, $x[i+3]$.

Assumptions - Latency

<table>
<thead>
<tr>
<th>Instruction Producing Value</th>
<th>Instruction Consuming Value</th>
<th>Interleaving Clock Cycles to Avoid Stalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP ALU Op</td>
<td>FP ALU Op</td>
<td>0</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 Load</td>
<td>Integer ALU Op</td>
<td>1</td>
</tr>
<tr>
<td>Integer ALU Op</td>
<td>Integer ALU Op</td>
<td>0</td>
</tr>
<tr>
<td>Branch Delay Miss</td>
<td>Something</td>
<td>1</td>
</tr>
</tbody>
</table>

Assume we need the following intervening cycles to avoid stalls. Different numbers from previous chapter to simplify things.

Loop with a Scalar

- **Adding a scalar value within a loop:**
  - for (i=1000; i>0; i--)
    - $x[i] = x[i] + s$;

- **Convert to DLX**
  - Loop:
    - LD F0, 0(R1) ; F0 gets $x[i]$
    - ADDD F4, F0, F2
    - SD 0(R1), F4
    - SUBI R1, R1, #8 ; doubleword
    - BNEZ R1, Loop ; Repeat

- **Let’s see how this loop executes without any special scheduling**

No Scheduling – Loop w/ Scalar

Loop:

- LD F0,0(R1) ; Cycle 1
- stall Cycle 2
- ADDD F4, F0, F2 Cycle 3
- stall Cycle 4
- stall Cycle 5
- SD 0(R1), F4 Cycle 6
- SUB R1, R1, #8 Cycle 7
- stall Cycle 8
- BNEZ R1, Loop Cycle 9
- stall Cycle 10

Total of 10 clocks per iteration!
For 1000 iterations, 10000 cycles
Optimized, Scheduled Version

Original:

```
LD F0, 0(R1) ; F0 gets x[i]
ADDI F4, F0, F2
SD 0(R1), F4
SUBI R1, R1, #8 ; doubleword
BNEZ R1, Loop ; Repeat
```

New:

```
LD F0, 0(R1) ; F0 gets x[i]
ADDI F4, F0, F2
SUBI R1, R1, #8
BNEZ R1, Loop ; Repeat
```

Loop Unrolling

- In the previous example, there were 3 cycles that did work in the loop (LD, SW, ADD) and 3 cycles that were just overhead to control the loop (SUB, BNEZ, stall)
- To get more instructions doing work relative to control instructions
  - We need to make our loop body larger
  - Can do this by replicating the loop body multiple times, and adjusting the loop termination code
    - Obviously can’t do it for the entire loop, if the loop iteration is high
    - Called loop unrolling
  - Tradeoff: Longer code, but higher ILP

Loop Unrolling Example

Four copies of the loop body per iteration

Loop:

```
Loop: LD F0, 0(R1)
ADDI F6, -8(R1), 8 ; Drop SUBI and BNEZ
ADDI F10, -16(R1), 8 ; Drop SUBI and BNEZ
ADDI F14, -24(R1), 8 ; Drop SUBI and BNEZ
```

Scheduling the Unrolled Loop

```
Loop: LD F0, 0(R1)
ADDI F6, -8(R1)
ADDI F10, -16(R1)
ADDI F14, -24(R1)
ADDI F16, -32(R1)
SUBI R1, #32
```

Without scheduling, a stall after every instruction, 28 cycles per iteration.

28 * 250 = 7000 cycles total

Shuffle unrolled loop instead of concatenating

No stalls! Must incorporate previous tricks of filling the delay slot

14 instructions, 14 * 250 = 3500 cycles total

8(R1), F4
Stuff to Notice

• Eight unused registers
  – Could have unrolled 8 instead of 4 loop iterations without any register conflicts
  – What if we have a remainder? E.g., our loop was 1002, not evenly divisible by 4
  – Just compute the extra N mod L blocks independently before or after the unrolled loop
• Tricky to unroll the loop, even this simple one
  – 3 types of dependence: data, name, and control

Data Dependence

• Occurs when either
  – Instruction I produces a result used by instruction J
  – Instruction J is data dependent on instruction K, and instruction K is data dependent on instruction I
    - LD F0, 0(R1)
    - ADDD F4, F0, F2
    - SD 0(R1), F4
    - or
    - SUBI R1, R1, 8
    - BNEZ R1, Loop

• Introduces possible RAW hazards
  – Whether there is an actual hazard depends on the pipeline, forwarding mechanisms, split cycle, etc.

Name Dependence

• Occurs when
  – 2 instructions use the same register name without a data dependence
• If I precedes J
  – I is antidependent on J when J writes a register that I reads
    • a WAR hazard
    • ordering must be preserved to avoid the hazard
  – I is output dependent on J if both I and J write to the same register
    • WAW hazard
• E.g., occurs if we unroll a loop but don’t changing the register names
  – This can be solved by renaming registers via the compiler, or we can also do this dynamically

Name Dependence Example

Part of unrolled loop, always going to F0:

Loop:
- LD F0, 0(R1)
- ADDD F4, F0, F2
- SD 0(R1), F4
- LD F0, -8(R1)
- ADDD F4, F0, F2
- SD -8(R1), F4

Name dependencies force execution stalls to avoid potential WAW or WAR problems

Eliminated by renaming the registers as in earlier example
Control Dependences

- Occurs when
  - Conditional branch exists and we have instructions after the branch that may or may not be executed
- Constraints to maintain dependencies
  - Instructions controlled by the branch cannot be moved before the branch (e.g. put part of a “then” before the “if”)  
  - An instruction not controlled by the branch cannot be moved after the branch (e.g. put a statement before the “if” and put it into the “then”)
- Once in a while we can violate these constraints and get away with it…

Preserving Control Dependence

- Should preserve the exception behavior
  - Assume no delay branches
  - No data dependence in the following:
    - BEQZ R2, SomeLabel
    - LW R1,0(R2)
  - Safe to move this to the following?
    - LW R1,0(R2)
    - BEQZ R2, SomeLabel
  - If we ignore the control dependence, if the load instruction generates an exception (e.g. memory protection fault) we get a different behavior

Preserving Control Dependence

- Should preserve the data flow
  - Assume no delay branches
    - ADD R1, R2, R3
    - BEQZ R4, L
    - SUB R1, R5, R6
    - L: OR R7, R1, R8
  - R1 in the OR depends on if we took the branch or not
  - OR is also data dependent on the ADD, SUB
  - Must preserve the control dependence of the SUB on the branch to prevent an illegal change to the data flow

Some Examples

```c
for (i=1; i<=100; i++) {
    A[i+1]=A[i]+C[i]; // S1
    B[i+1]=B[i]+A[i+1]; // S2
}
```

Observations of dependencies:
- S1 uses S1 value produced in an earlier iteration
- S2 uses S2 value produced in an earlier iteration
- S2 uses an S1 value produced in the same iteration

Implications:
- Values dependent on the earlier iteration are called loop carried dependent; order must be preserved
- Non-loop carried dependences we can try and execute in parallel (but not in this case, due to other dependency)
One More Example

for (i=1; i<=100; i++) {
    A[i]=A[i]+B[i];  // S1
    B[i+1]=C[i]+D[i]; // S2
}

S1 uses previous value of S2, but despite the loop-carried dependence this is not circular, since neither statement depends on itself.

S2 doesn’t depend on S1

Implies S2 can be moved!


for (i=1; i<=99; i++) {
    B[i+1]=C[i]+D[i];
    A[i+1]=A[i+1]+B[i+1];
}

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

No more loop carried dependence so we can unroll the loop and expect good performance gains!

A variety of these transformations possible, but tricky.