Hazards

Situations that prevent starting the next logical instruction in the next clock cycle
1. Structural hazards
   - Required resource is busy (e.g., roommate studying)
2. Data hazard
   - Need to wait for previous instruction to complete its data read/write (e.g., pair of socks in different loads)
3. Control hazard
   - Deciding on control action depends on previous instruction (e.g., how much detergent based on how clean prior load turns out)

3. Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
- BEQ, BNE in MIPS pipeline
- Simple solution Option 1: Stall on every branch until have new PC value
  - Would add 2 bubbles/clock cycles for every Branch! (~20% of instructions executed)

Stall => 2 Bubbles/Clocks

Control Hazard: Branching

- Optimization #1:
  - Insert special branch comparator in Stage 2
  - As soon as instruction is decoded (Opcode identifies it as a branch), immediately make a decision and set the new value of the PC
  - Benefit: since branch is complete in Stage 2, only one unnecessary instruction is fetched, so only one no-op is needed
  - Side Note: means that branches are idle in Stages 3, 4 and 5

Question: What’s an efficient way to implement the equality comparison?
**Control Hazards: Branching**

- Option #3: Redefine branches
  - Old definition: if we take the branch, none of the instructions after the branch get executed by accident
  - New definition: whether or not we take the branch, the single instruction immediately following the branch gets executed (the branch-delay slot)
- Delayed Branch means we always execute inst after branch
- This optimization is used with MIPS

---

**Example: Nondelayed vs. Delayed Branch**

<table>
<thead>
<tr>
<th>Nondelayed Branch</th>
<th>Delayed Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{or} ) $8, ; 9, ; 10 )</td>
<td>( \text{add} ) $1, ; 2, ; 3 )</td>
</tr>
<tr>
<td>( \text{add} ) $1, ; 2, ; 3 )</td>
<td>( \text{sub} ) $4, ; 5, ; 6 )</td>
</tr>
<tr>
<td>( \text{sub} ) $4, ; 5, ; 6 )</td>
<td>( \text{beq} ) $1, ; 4, ; \text{Exit} )</td>
</tr>
<tr>
<td>( \text{beq} ) $1, ; 4, ; \text{Exit} )</td>
<td>( \text{or} ) $8, ; 9, ; 10 )</td>
</tr>
<tr>
<td>( \text{xor} ) $10, ; 1, ; 11 )</td>
<td>( \text{xor} ) $10, ; 1, ; 11 )</td>
</tr>
</tbody>
</table>

---

**Greater Instruction-Level Parallelism (ILP)**

- Deeper pipeline (5 => 10 => 15 stages)
  - Less work per stage \( \Rightarrow \) shorter clock cycle
- Multiple issue “superscalar”
  - Replicate pipeline stages \( \Rightarrow \) multiple pipelines
  - Start multiple instructions per clock cycle
  - CPI < 1, so use Instructions Per Cycle (IPC)
  - E.g., 4GHz 4-way multiple-issue
    - 16 BIPS, peak CPI = 0.25, peak IPC = 4
    - But dependencies reduce this in practice
Multiple Issue

• Static multiple issue
  – Compiler groups instructions to be issued together
  – Packages them into “issue slots”
  – Compiler detects and avoids hazards
• Dynamic multiple issue
  – CPU examines instruction stream and chooses instructions to issue each cycle
  – Compiler can help by reordering instructions
  – CPU resolves hazards using advanced techniques at runtime

Superscalar Laundry: Parallel per stage

Pipeline Depth and Issue Width

• Intel Processors over Time

<table>
<thead>
<tr>
<th>Processor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25 MHz</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>66 MHz</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200 MHz</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000 MHz</td>
<td>22</td>
<td>3</td>
<td>1</td>
<td>75W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3600 MHz</td>
<td>31</td>
<td>3</td>
<td>1</td>
<td>103W</td>
</tr>
</tbody>
</table>

Static Multiple Issue

• Compiler groups instructions into “issue packets”
  – Group of instructions that can be issued on a single cycle
  – Determined by pipeline resources required
• Think of an issue packet as a very long instruction
  – Specifies multiple concurrent operations

Scheduling Static Multiple Issue

• Compiler must remove some/all hazards
  – Reorder instructions into issue packets
  – No dependencies within a packet
  – Possibly some dependencies between packets
    • Varies between ISAs; compiler must know!
  – Pad issue packet with nop if necessary
MIPS with Static Dual Issue

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
  - ALU/branch, then load/store
  - Pad an unused instruction with nop

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction Type</th>
<th>Pipeline Stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>n ALU/branch</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>n = 4 Load/store</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>n = 8 ALU/branch</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
<tr>
<td>n = 12 Load/store</td>
<td>F ID EX MEM WB</td>
<td></td>
</tr>
</tbody>
</table>

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can’t use ALU result in load/store in same packet
    - add $s0, $s0, $s1
      - Load $s2, 0($s0)
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

Scheduling Example

- Schedule this for dual-issue MIPS

Loop: lw $t0, 0($s1)  # $t0-array element
addu $s0, $t0, $s2  # add scalar in $s2
sw $s0, 0($s1)     # store result
addi $s1, $s1, -4  # decrement pointer
bne $s1, $zero, Loop # branch $s1!=0

ALU/branch | Load/store | cycle |
---|---|---|
Loop: | | 1 |
  | | 2 |
  | | 3 |
  | | 4 |

Scheduling Example

- Schedule this for dual-issue MIPS

Loop: lw $t0, 0($s1)  # $t0-array element
addu $s0, $t0, $s2  # add scalar in $s2
sw $s0, 0($s1)     # store result
addi $s1, $s1, -4  # decrement pointer
bne $s1, $zero, Loop # branch $s1!=0

ALU/branch | Load/store | cycle |
---|---|---|
Loop: | | 1 |
  | | 2 |
  | | 3 |
  | | 4 |

Scheduling Example

- Schedule this for dual-issue MIPS

Loop: lw $t0, 0($s1)  # $t0-array element
addu $s0, $t0, $s2  # add scalar in $s2
sw $s0, 0($s1)     # store result
addi $s1, $s1, -4  # decrement pointer
bne $s1, $zero, Loop # branch $s1!=0

ALU/branch | Load/store | cycle |
---|---|---|
Loop: | | 1 |
  | | 2 |
  | | 3 |
  | | 4 |
Scheduling Example

- Schedule this for dual-issue MIPS

```
Loop: lw $t0, 0($s1) # $t0-array element
   addu $t0, $t0, $s2 # add scalar in $s2
   sw $t0, 0($s1) # store result
   add $s1, $s1, -4 # decrement pointer
   bne $s1, $zero, Loop # branch $s1!=0
```

IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

<table>
<thead>
<tr>
<th>ALU/Branch</th>
<th>Load/Store</th>
<th>cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop:</td>
<td>lw $t0, 0($s1)</td>
<td>1</td>
</tr>
<tr>
<td>add $s1, $s1, -4</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>bne $s1, $zero, Loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Loop Unrolling

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

```
Loop: addi $s1, $s1, -16
   lw $t0, 20($s2)        # $t0-array element
   addu $t1, $t0, $t2
   subu $s4, $s4, $t3
   slti $t5, $s4, 20      # $s4 < 20?
   jrc $t5, $zero, Loop   # branch $t5!=0
```

IPC = 4/8 = 0.5

```
Loop: addi $s1, $s1, -16
   lw $t0, 20($s2)        # $t0-array element
   addu $t1, $t0, $t2
   subu $s4, $s4, $t3
   slti $t5, $s4, 20      # $s4 < 20?
   jrc $t5, $zero, Loop   # branch $t5!=0
```

Dynamic Multiple Issue

- “Superscalar” processors
- CPU decides whether to issue 0, 1, 2, … each cycle
  - Avoiding structural and data hazards
- Avoids the need for compiler scheduling
  - Though it may still help
  - Code semantics ensured by the CPU

Dynamic Pipeline Scheduling

- Allow the CPU to execute instructions out of order to avoid stalls
  - But commit result to registers in order
- Example

```
lw $t0, 20($s2)
   addu $t1, $t0, $t2
   subu $s4, $s4, $t3
   slti $t5, $s4, 20
```

- Can start subu while addu is waiting for lw

Why Do Dynamic Scheduling?

- Why not just let the compiler schedule code?
- Not all stalls are predictable
  - e.g., cache misses
- Can’t always schedule around branches
  - Branch outcome is dynamically determined
- Different implementations of an ISA have different latencies and hazards
Speculation

• “Guess” what to do with an instruction
  – Start operation as soon as possible
  – Check whether guess was right
    • If so, complete the operation
    • If not, roll-back and do the right thing
• Common to static and dynamic multiple issue
• Examples
  – Speculate on branch outcome (Branch Prediction)
    • Roll back if path taken is different
  – Speculate on load
    • Roll back if location is updated

Out-of-Order Laundry: Don’t Wait

• A depends on D; rest continue; need more resources to allow out-of-order

Out Of Order Intel

• All use OOO since 2001

<table>
<thead>
<tr>
<th>Microprocessor</th>
<th>Year</th>
<th>Clock Rate</th>
<th>Pipeline Stages</th>
<th>Issue width</th>
<th>Out-of-order Speculation</th>
<th>Cores</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>i486</td>
<td>1989</td>
<td>25MHz</td>
<td>5</td>
<td>1</td>
<td>No</td>
<td>1</td>
<td>5W</td>
</tr>
<tr>
<td>Pentium</td>
<td>1993</td>
<td>60MHz</td>
<td>5</td>
<td>2</td>
<td>No</td>
<td>1</td>
<td>10W</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>1997</td>
<td>200MHz</td>
<td>10</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>29W</td>
</tr>
<tr>
<td>P4 Willamette</td>
<td>2001</td>
<td>2000MHz</td>
<td>22</td>
<td>3</td>
<td>Yes</td>
<td>1</td>
<td>73W</td>
</tr>
<tr>
<td>P4 Prescott</td>
<td>2004</td>
<td>3000MHz</td>
<td>21</td>
<td>3</td>
<td>Yes</td>
<td>2</td>
<td>102W</td>
</tr>
</tbody>
</table>

Does Multiple Issue Work?

• Yes, but not as much as we’d like
• Programs have real dependencies that limit ILP
• Some dependencies are hard to eliminate
  – e.g., pointer aliasing
• Some parallelism is hard to expose
  – Limited window size during instruction issue
• Memory delays and limited bandwidth
  – Hard to keep pipelines full
• Speculation can help if done well

“And in Conclusion..”

• Pipelining is an important form of ILP
• Challenge is (are?) hazards
  – Forwarding helps w/many data hazards
  – Delayed branch helps with control hazard in 5 stage pipeline
  – Load delay slot / interlock necessary
• More aggressive performance:
  – Longer pipelines
  – Superscalar
  – Out-of-order execution
  – Speculation