Lecture Topics

- Today: Parallel Processors
  (P&H 6.1-6.10)
- Next: continued

Announcements

- Milestone #10 (due 4/20)
- Point summary later this week
Final Exam

- Wednesday, 5/3 (3:00-5:00 PM)
- Location to be determined
- 45 multiple choice questions
- 120 minutes, 27% of course grade
- Study suggestions on course website

Review: Pipelining

Goal: complete one instruction per cycle

Instruction 1
Instruction 2
Instruction 3
Instruction 4
Instruction 5
Review: Superpipelining

Goal: decrease cycle time by adding stages

Superpipelined Processors

- Pipeline stages in "standard" pipelined processor subdivided into more stages
  - each stage is simpler (fewer gates, levels)
  - clock frequency can be increased

- Benefit: more instructions can be in the pipeline (in flight) simultaneously, which increases the level of parallelism
MIPS R4000

Superpipelined Processors

- Drawback: with more instruction in flight, the cost of handling hazards increases
  - more potential data hazards
  - larger load penalty
  - larger branch penalty
- Requires significant increase in circuits for detection and forwarding
Limitations of Scalar Pipeline

- Maximum throughput for a scalar pipeline is one instruction per cycle
- Forcing all instructions to use same pipeline is inefficient (different types of instructions have different requirements)
- Stalls cause pipeline bubbles (which raises CPI)

Next: Superscalar

Goal: complete multiple instructions per cycle by fetching more than one
Superscalar Processors

- Can execute multiple instructions independently and concurrently
- Must have multiple "functional units" to perform the work (normally pipelined)
- Extension: allow instructions to be executed in a different order than in the original program
• Initial approach: parallel pipelines

• Width determines potential speedup

• Interconnection logic more complex, additional ports needed for register file and caches

Example: Intel Processors

Intel i486
• 5 stages
• scalar

Intel Pentium
• derived from i486
• last three stages independent
- Better: diversified pipelines
- Customize each pipeline to type of instruction
- Longer latencies in some pipelines

Example: Motorola 88110
Dynamic Pipelines

- Initial approach: if N pipelines, issue up to N *consecutive* instructions at same time
- Inefficient – rare to find much parallelism in consecutive instructions
- Better: allow instructions to be executed in a different order than the original order in the program

- Use dispatch buffer to hold instructions until resources ready
- Use reorder buffer to put "results" back into same order as original program would have used
Resources

- Instructions may have to wait until a particular "functional unit" is available

- Instructions may have to wait until all of its operands are available
  - registers
  - memory

- Focus on operands in registers (similar processing for operands in memory)

Data Dependencies

- Consider the MIPS code sequence:
  
  \[
  \begin{align*}
  \text{add} & \quad \text{\$12, \$2, \$8} \\
  \text{sub} & \quad \text{\$6, \$12, \$9}
  \end{align*}
  \]

- The SUB instruction depends on the result of the ADD instruction
RAR Sequences

- **Read After Read:**
  
  - `xor   $12, $2, $18`
  - `or    $6, $18, $9`

  - R18 read by both instructions; no change to the contents of R18, so order of execution is irrelevant

RAW Sequences

- **Read After Write:**
  
  - `add   $12, $2, $8`
  - `sub   $6, $12, $9`

  - The ADD instruction must update R12 before SUB reads R12

  - Called *true dependency*
WAR Sequences

- Write After Read:
  
  \[
  \text{nor} \quad \$4, \$2, \$16 \\
  \text{and} \quad \$16, \$12, \$9
  \]

  - The NOR instruction must read R16 before the AND instruction updates R16
  - Called *anti dependency*

WAW Sequences

- Write After Write:
  
  \[
  \text{nor} \quad \$14, \$2, \$16 \\
  \text{xor} \quad \$14, \$12, \$9
  \]

  - The NOR instruction must update R16 before the XOR instruction updates R16
  - Called *output dependency*
Handling Data Dependencies

- RAR – not really a dependency
- RAW – forwarding and/or stalling
- WAR – register renaming
- WAR – register renaming

Forwarding

<table>
<thead>
<tr>
<th>Cycle 1 (i + 1)</th>
<th>Cycle 2 (i + 2)</th>
<th>Cycle 3 (i + 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>i + 1: ( \leftarrow R_1 )</td>
<td>i + 2: ( \leftarrow R_1 )</td>
<td>i + 3: ( \leftarrow R_1 )</td>
</tr>
<tr>
<td>i: ( R_1 ) ( \leftarrow )</td>
<td>i + 1: ( \leftarrow R_1 )</td>
<td>i + 2: ( \leftarrow R_1 )</td>
</tr>
<tr>
<td>i: ( R_1 ) ( \leftarrow )</td>
<td>i + 1: ( \leftarrow )</td>
<td></td>
</tr>
<tr>
<td>i: ( R_1 ) ( \leftarrow )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(i \( \rightarrow i + 1 \)) Forwarding via path a

(ii \( \rightarrow i + 2 \)) Forwarding via path b

(i \( \rightarrow i + 3 \)) i writes \( R_1 \) before \( i + 3 \) reads \( R_1 \)
Idealized Superscalar Processor

- Buffers (*reservation stations*) for each functional unit – instructions must wait until operands are available
- Instruction *completed* when it updates machine state
- Instruction *retired* when it updates memory state
Control Flow

- Control transfer instructions are about 20% of a typical program
- Stalls generated to handle control hazards can have a significant impact on CPI
- Branch prediction can be used to lessen the impact of control hazards

- Branch penalty of 3 cycles
- If 4-way superscalar, cost is really 12 instructions
Example:
PowerPC 604

CDC 6600 (1963)