• Final exam
  • Tuesday, May 11, 2-4:30pm
  • NH 110

• Project 5 (4) due Apr 30
  • 0% late penalty until May 4 2pm
  • 100% penalty afterward
Instruction scheduling

• Note: none of this applies to the project
  • including branch delay slots
• SPIM ignores instruction latency issues
Instruction-level parallelism

- Modern processors can execute several adjacent instructions simultaneously
  - pipelined machines
  - very long instruction word machines (VLIW)
  - superscalar machines
  - dynamic scheduling/out-of-order machines
- ILP limited by execution constraints
  - data dependence constraints
  - resource constraints ("hazards")
  - control hazards
Execution constraints

- Data dependence constraints
  - if instruction A computes a value that is read by instruction B, then B cannot execute before A completes

- lw $t2, -28($fp)
- add $t1, $t2, $t3

- Resource hazards
  - limited number of functional units
  - limited instruction issue
  - limited register set
Instruction scheduling

• Purpose is to order instructions to maximize ILP
  • keep all resources busy every cycle
  • eliminate data dependences and resource hazards if necessary

• NP-complete (and bad in practice)
  • need heuristics
Instruction scheduling

• Open area of research
• Most optimizing compilers perform good local IS, simple global IS
• Biggest opportunities are scheduling code for loops
Should the compiler do IS?

• Many modern processors perform dynamic reordering of instructions
  • called **out-of-order execution**
  • use additional registers and register renaming to eliminate dependences
  • more complex hardware, longer cycle times, more power
  • some newer multicore processors don’t do OOOE
von Neumann model

• Instruction starts only after predecessor completes

- instr 1 - instr 2

- time

• Not very efficient
  • “von Neumann bottleneck”
  • “memory wall”
Pipelines

- Processors today use pipelines to allow overlap of instructions
- Divide execution of an instruction into stages; each stage performed by separate part of the processor
  - Pentium 4: 20-stage pipeline
  - multicore: pipelines getting shorter (less logic, less power)
- Each stage completes its operation in one cycle
  - shorter than the von Neumann cycle
- Instruction still takes the same amount of time to execute

<table>
<thead>
<tr>
<th>instr</th>
<th>F</th>
<th>D</th>
<th>E</th>
<th>M</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>F:</td>
<td>Fetch instruction from cache or memory.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D:</td>
<td>Decode instruction.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E:</td>
<td>Execute. ALU operation or address calculation.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M:</td>
<td>Memory access.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W:</td>
<td>Write back result into register.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thursday, April 22, 2010
Pipelines

Overlap the stages:

- **Instr 1**: Fetch instruction from cache or memory.
- **Instr 2**: Decode instruction.
- **Instr 3**: Execute. ALU operation or address calculation.
- **Instr 4**: Memory access.
- **Instr 5**: Write back result into register.
- **Instr 6**: Conditional branch address available only after Execute.
- **Instr 7**: Unconditional jump address available only after Decode.

---

Draining the pipeline

Steady state

Filling the pipeline

Time
Pipeline hazards

- Structural hazards
  - 2 instructions need same resource at same time
- Data hazards
  - Instruction needs results of previous instruction
  - Solved by forwarding or stalling
  - Cache miss?
- Control hazards
  - Jump and branch address not known until later in pipeline
  - Solved by delay slot or branch prediction
Control hazards

- Unconditional jump address available only after Decode
- Conditional branch address available only after Execute
Control hazards

• One option: stall the pipeline (hardware solution)

• Another option, insert a no-op (software)
Control hazards

- Another option: have branch take effect **after** the delay slots
- Execute an instruction after the branch but before the branch takes effect
Delay slot

- Instruction after a branch (delay slot) *always* executed when the branch is executed, regardless of branch result
Branch prediction

- Processors speculatively execute at conditional branches
  - if correct, great!
  - if not, flush the pipeline before write back
Data hazards

\[
\begin{align*}
& r_1 = r_2 + r_3 \\
& r_4 = r_1 + r_1
\end{align*}
\]

\[
\begin{array}{cccccc}
F & D & E & M & W \\
\end{array}
\]

\[
\begin{align*}
& r_2 + r_3 \text{ available here} \\
& [r_2] \text{ available here}
\end{align*}
\]
Dependences

- flow dependence  W->R
- anti dependence  R->W
- output dependence  W->W
- input dependence  R->R
Examples

S1) a=0;
S2) b=a;
S3) c=a+d+e;
S4) d=b;
S5) b=5+e;

Not generally defined
Register renaming

• Often, dependences are not “real” – renaming variables/registers can eliminate them
  • output dependence (WW)
    • A and B write to the same variable
  • anti dependence (RW)
    • A reads from a variable that B overwrites

• Can change order of A and B if variables renamed
  • Can sometimes be done in hardware
Register renaming

- Can perform register renaming after register allocation
  - constrained by available registers
  - constrained by live on entry/exit

- Instead, do scheduling before register allocation
Phase ordering problem

• Do scheduling before register allocation
  • eliminates data dependences

• Do scheduling after register allocation
  • to avoid hazards introduced by register allocation

• Many compilers do both
Scheduling approaches

- List scheduling
- Trace scheduling
- Software pipelining
List scheduling

• Idea:
  • Build DAG of dependences
  • Do a **topological sort** of dependence DAG
  • Choose instructions to satisfy sort order without causing a stall
  • Use heuristics when needed
• while there are nodes to schedule
  • Choose node whose predecessors in graph have been scheduled and where enough time has elapsed since predecessors are scheduled
  • Schedule the node
• “Enough time has elapsed”
  • if n has predecessor \( p_i \) scheduled at cycle \( S(p_i) \)
  • and \( p_i \) takes time \( T(p_i) \)
  • and latency of \( p_i \rightarrow n \) is \( L(p_i) \)

• then, cannot schedule n before
  • \( \max_i (S(p_i) + T(p_i) + L(p_i)) \)
• if n scheduled earlier => stall
Heuristics

• If multiple nodes can be scheduled, need to pick one
  • pick node with longest path to a leaf in the dependence DAG
• pick node that can go to a less busy pipeline
• pick node with most immediate successors
  • (gives scheduler more freedom to choose next instruction)
Other issues

- Branch delay scheduling done before basic block scheduling
  - schedule some instruction from same BB in the branch delay slot

- Assume all memory references hit in the cache
  - (otherwise delays may be much larger)

- Different hardware have different pipeline characteristics
  - even different versions of the h/w
  - => should automatically generate scheduler from constraints
Trace scheduling

- Find likely sequence of basic blocks (trace)
- => long sequence of straight-line code
- use profile information collected at run time

- Do list scheduling on the sequence
  - Be careful not to reorder branches
  - If moving code across conditional branch, insert compensation code
Software pipelining

• Schedule instructions across several loop iterations

• Idea:
  • unroll the loop, schedule, then reroll
Software pipelining

```c
for (i = 1; i <= n; i++) {
    a[i+1] = a[i] + 1;
    b[i] = a[i+1] / 2;
    c[i] = b[i] + 3;
    d[i] = c[i];
}
```

```c
c[1] = b[1] + 3;
d[1] = c[1];
```

```c
d[2] = c[2];
```

```c
d[3] = c[3];
```

```c
d[4] = c[4];
```
Software pipelining

\begin{align*}
a[2] &= a[1] + 1; \\
b[1] &= a[2] / 2; \\
c[1] &= b[1] + 3; \\
d[1] &= c[1]; \\
d[2] &= c[2]; \\
d[3] &= c[3]; \\
d[4] &= c[4]; \\
\end{align*}

Prelude
\begin{align*}
a[2] &= a[1] + 1; \\
b[1] &= a[2] / 2; \\
c[1] &= b[1] + 3; \\
\end{align*}

Postlude
\begin{align*}
&\text{for (i = 1; i <= n-3; i++)} \{ \\
&\quad d[i] = c[i]; \\
&\quad c[i+1] = b[i+1] + 3; \\
&\quad b[i+2] = a[i+3] / 2; \\
&\quad a[i+4] = a[i+3] + 1; \\
&\}
&d[n-2] = c[n-2]; \\
c[n-1] = b[n-1] + 3; \\
d[n-1] = c[n-1]; \\
b[n] = a[n+1] / 2; \\
c[n] = b[n] + 3; \\
d[n] = c[n];
\end{align*}
Software pipelining

• In steady state, no (or few) dependences in a single loop iteration
• Hardware can run most instructions in each iteration in parallel
• Effect is similar to infinite loop unrolling