Static Scheduling

• basic pipeline: single, in-order issue
• first extension: multiple issue (superscalar)
• second extension: scheduling instructions for more ILP
  • option #1: dynamic scheduling (by the hardware)
  • option #2: static scheduling (by the compiler)
Readings

H+P

- parts of chapter 2

Recent Research Paper

- EPIC/IA-64
VLIW: Very Long Instruction Word

• problem with superscalar implementation: complexity!
  – wide fetch+branch prediction (can partially fix w/ trace cache)
  – $N^2$ bypass (can partially fix with clustering)
  – $N^2$ dependence cross-check (stall+bypass logic)

One alternative: VLIW (very long instruction word)

• single-issue pipe, but unit is N-instruction group (VLIW)
  • instructions in VLIW are guaranteed (by compiler) to be independent
    + processor does not have to dependence-check within a VLIW
  • VLIW travels down pipe as a unit (“bundle”)
  • often “slotted” (i.e., 1st must be ALU, 2nd must be load, etc.)

| instruction 1 | instruction 2 | instruction 3 |
VLIW History

• started with microcode (“horizontal microcode”)
• academic projects
  • ELI-512 [Fisher, ‘85]
  • Illinois IMPACT [Hwu, ‘91]
• commercial machines
  • MultiFlow [Colwell+Fisher, ‘85] ⇒ failed
  • Cydrome [Rau, ‘85] ⇒ failed
  • EPIC (IA-64, Itanium) [Colwell,Fisher+Rau, ‘97] ⇒ failing/failed
  • Transmeta [Ditzel, ‘99]: translates x86 to VLIW ⇒ failed
  • many embedded controllers (TI, Motorola) are VLIW ⇒ success
Pure VLIW

• **pure VLIW**: no hardware dependence-checks at all
  • not even between VLIW groups
• compiler responsible for scheduling entire pipeline
  • including stall cycles
  • possible if you know structure of pipeline and latencies exactly
  
  – problem 1: pipe & latencies vary across implementations
    • recompile for new implementations (or risk missing a stall)?
    • TransMeta solved this problem by recompiling on-the-fly
  
  – problem 2: latencies are **NOT** fixed within implementation
    • don’t use caches? (forget it)
    • schedule assuming cache miss? (no point to having caches)

  not many VLIW purists left
A VLIW Compromise

compromise: EPIC (Explicitly Parallel Instruction Computing)

• less rigid than VLIW (not really VLIW at all)
• variable width instruction words
  • implemented as “bundles” with dependence bits
    + makes code compatible with different width machines
• assumes inter-bundle stall logic provided by hardware
  • makes code compatible with different pipeline depths, op latencies
    • enables stalls on cache misses (actually, out-of-order too)

+ exploits any information on parallelism compiler can give
+ compatible with multiple implementations of same arch
• e.g., IA-64: Itanium, Itanium2
ILP and Scheduling

no point to having an N-wide pipeline if, on average, many fewer than N independent instructions per cycle

• performance is important
• but utilization (actual/peak performance) is also
Code Example: SAXPY

- SAXPY (single-precision $A \times X + Y$)
  - linear algebra routine (used in solving systems of equations)
  - part of famous “Livermore Loops” kernel (early benchmark)

```c
for (I=0; I<N; I++)
    Z[I] = A*X[I] + Y[I]

ldf f0, X(r1)  // loop:
mul f4, f0, f2  // assume A in f2
ldf f6, Y(r1)  // X,Y,Z are constant addresses
addf f8, f6, f4
stf f8, Z(r1)
add r1, r1, #4  // assume I in r1
ble r1, r2, loop  // assume N*4 in r2
```
Default SAXPY Performance

- scalar (1-wide), pipelined processor (for illustration)
  - 5 cycle FP mult, 2 cycle FP add, both fully-pipelined
  - full bypassing, all branches predicted taken

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| ldf f0,A(r1) | F | D | X | M | W |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| mulf f4,f0,f2 | F | D | d* | E* | E* | E* | E* | W |   |   |   |   |   |   |   |   |   |   |   |   |
| ldf f6,B(r1) | F | p* | D | X | M | W |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| addf f8,f6,f4 | F | D | d* | d* | d* | E+ | E+ | W |   |   |   |   |   |   |   |   |   |   |   |   |
| stf f8,C(r1) | F | p* | p* | p* | D | X | M | W |   |   |   |   |   |   |   |   |   |   |   |
| add r1,r1,#4 | F | D | X | M | W |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| ble r1,r2,loop | F | D | X | M | W |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

- single iteration (7 instructions) latency: 15 cycles
- performance: 7 instructions / 15 cycles ⇒ IPC = 0.47
- utilization: 0.47 actual IPC / 1 peak IPC ⇒ 47%
Performance and Utilization

• 2-wide pipeline (but still in-order)
  • same configuration, just two at a time

<table>
<thead>
<tr>
<th>Instruction</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldf f0,A(r1)</td>
<td>F</td>
<td>D</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mulf f4,f0,f2</td>
<td>F</td>
<td>D</td>
<td>d*</td>
<td>d*</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>E*</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldf f6,B(r1)</td>
<td>F</td>
<td>D</td>
<td>p*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>addf f8,f6,f4</td>
<td>F</td>
<td>p*</td>
<td>p*</td>
<td>D</td>
<td>d*</td>
<td>d*</td>
<td>d*</td>
<td>d*</td>
<td>E+</td>
<td>E+</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stf f8,C(r1)</td>
<td>F</td>
<td>p*</td>
<td>D</td>
<td>p*</td>
<td>p*</td>
<td>p*</td>
<td>p*</td>
<td>X</td>
<td>d*</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add r1,r1,#4</td>
<td>F</td>
<td>p*</td>
<td>p*</td>
<td>p*</td>
<td>D</td>
<td>p*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ble r1,r2,loop</td>
<td>F</td>
<td>p*</td>
<td>p*</td>
<td>p*</td>
<td>D</td>
<td>p*</td>
<td>d*</td>
<td>X</td>
<td>M</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• performance: still 15 cycles - not any better (why?)
• utilization: 0.47 actual IPC / 2 peak IPC ⇒ 24%!
• notice: more hazards → stalls (why?)
• notice: each stall more expensive (why?)
Scheduling and Issue

instruction scheduling: decide on instruction execution order

• important tool for improving utilization and performance
• related to instruction issue (when instructions execute)
  • pure VLIW: static scheduling with static issue
  • in-order superscalar, EPIC: static scheduling with dynamic issue
    • well, not completely dynamic...
• in-order pipeline relies on compiler to schedule well
Instruction Scheduling

- idea: independent instructions between slow ops and uses
  - otherwise pipeline sits idle waiting for RAW hazards to resolve
  - we have already seen dynamic pipeline scheduling

- to do this we need independent instructions

- scheduling scope: code region we are scheduling
  - the bigger the better (more independent instructions to choose from)
  - once scope is defined, schedule is pretty obvious
  - trick is making a large scope (schedule across branches???)

- compiler scheduling techniques (more about these later)
  - loop unrolling (for loops)
  - software pipelining (also for loops)
  - trace scheduling (for general control-flow)
Scheduling: Compiler or Hardware?

• compiler
  + large scheduling scope (full program), large “lookahead”
  + enables simple hardware with fast clock
    – low branch prediction accuracy (profiling? see next slide!)
    – no information on latencies like cache misses (profiling?)
    – pain to speculate and recover from mis-speculation (h/w support?)

• hardware
  + better branch prediction accuracy
  + dynamic information on latencies (cache misses) and dependences
  + easy to speculate & recover from mis-speculation
    – finite on-chip instruction buffering limits scheduling scope
    – more complicated hardware (more power? tougher to verify?)
    – slower clock
Aside: Profiling

profile: (statistical) information about program tendencies

- run program once with a test input and see how it behaves
- hope that other inputs lead to similar behaviors
- compiler can use this info for scheduling
- profiling can be a useful technique
  - must be used carefully - else, can harm performance
Loop Unrolling SAXPY

we want to separate dependent operations from one another

• but not enough flexibility within single iteration of loop

• longest chain of operations is 9 cycles
  • load result (1 cycle)
  • forward to multiply (5 cycles)
  • forward to add (2 cycles)
  • forward to store (1 cycle)
  • can’t hide 9 cycles of latency using 7 instructions
  • how about 9 cycles of latency twice in 14 instructions?

• loop unrolling: schedule 2 loop iterations together
Unrolling Part 1: Fuse Iterations

- combine two (in general, N) iterations of loop
  - fuse loop control (induction increment + backward branch)
  - adjust implicit uses of internal induction variables (r1 in example)

```assembly
ldf f0,X(r1)  
mulf f4,f0,f2  
ldf f6,Y(r1)  
addf f8,f6,f4  
stf f8,Z(r1)  
add r1,r1,#4  
ble r1,r2,loop
```

```assembly
ldf f0,X(r1)  
mulf f4,f0,f2  
ldf f6,Y(r1)  
addf f8,f6,f4  
stf f0,Z(r1)  
add r1,r1,#4  
ble r1,r2,loop
```

```assembly
ldf f0,X(r1)  
mulf f4,f0,f2  
ldf f6,Y(r1)  
addf f8,f6,f4  
stf f8,Z(r1)  
add r1,r1,#4  
ble r1,r2,loop
```

```assembly
ldf f0,X+4(r1)  
mulf f4,f0,f2  
ldf f6,Y+4(r1)  
addf f8,f6,f4  
stf f8,Z+4(r1)  
add r1,r1,#8  
ble r1,r2,loop
```
Unrolling Part 2: Pipeline Schedule

- pipeline schedule to reduce RAW stalls
  - have seen this already (as done dynamically by hardware)

\[
\begin{align*}
\text{ldf } f0, X(r1) \\
\text{mulf } f4, f0, f2 \\
\text{ldf } f6, Y(r1) \\
\text{addf } f8, f6, f4 \\
\text{stf } f8, Z(r1) \\
\text{ldf } f0, X+4(r1) \\
\text{mulf } f4, f0, f2 \\
\text{ldf } f6, Y+4(r1) \\
\text{addf } f8, f6, f4 \\
\text{stf } f8, Z+4(r1) \\
\text{add } r1, r1, #8 \\
\text{ble } r1, r2, \text{loop}
\end{align*}
\]
Unrolling Part 3: Rename Registers

- pipeline scheduling caused WAR hazards
  - so we rename registers to solve this problem (similar to w/hardware)

\begin{verbatim}
ldf f0,X(r1)
ldf f0,X+4(r1)
mulf f4,f0,f2
mulf f4,f0,f2
ldf f6,Y(r1)
ldf f6,Y+4(r1)
addf f8,f6,f4
addf f8,f6,f4
stf f8,Z(r1)
stf f8,Z+4(r1)
add r1,r1,#8
ble r1,r2,loop
\end{verbatim}

\begin{verbatim}
ldf f0,X(r1)
ldf f10,X+4(r1)
mulf f4,f0,f2
mulf f14,f10,f2
ldf f6,Y(r1)
ldf f16,Y+4(r1)
addf f8,f6,f4
addf f18,f16,f14
stf f8,Z(r1)
stf f18,Z+4(r1)
add r1,r1,#8
ble r1,r2,loop
\end{verbatim}
Unrolled SAXPY Performance

- 2 iterations (12 instructions) → 17 cycles (fewer stalls)
  - before unrolling, it took 15 cycles for 1 iteration!
Shortcomings of Loop Unrolling

- code growth
- poor scheduling along “seams” of unrolled copies
- doesn’t handle inter-iteration dependences (recurrences)

```c
for(I=0;I<N;I++)
    X[I] = A*X[I-1]; // each iteration depends on prior
```

Unroll:

```
ldf f2,X-4(r1)
mulf f4,f2,f0
stf f4,X(r1)
add r1,r1,#4
ble r1,r2,loop
```

```
ldf f12,X-4(r1)
mulf f14,f12,f0
stf f14,X(r1)
```

```
add r1,r1,#8
ble r1,r2,loop
```

1 dependence chain → can’t schedule