Announcements

Guest lectures next week:
Monday, Prof. Marculescu, Power-Aware Microarchitecture
Wednesday, Prof. Nowatzyk, Multiprocessor Architecture

Last week: I/O & Interconnects

Final exam: Fri. December 14 5:30pm-8:30pm, PH 125C
Static Scheduling: Summary

Loop unrolling:
- large block to schedule
- reduces branch frequency
- expands code size
- have to handle “extra” iterations

Software pipelining:
- no dependences in loop body
- does not reduce branch frequency
- need start-up and finish-up blocks

Trace scheduling:
- works for non-loops
- more complex than unrolling and software pipelining
- does not seem to handle more general cases
**Ambiguous Dependences**

Software limitation
What compiler can't analyze => can't schedule
Ambiguous memory dependences
e.g.,
*ptr1 =
... = *ptr2 + 4
e.g.,
a[b[I]] =
..... = a[c[I]] -5
Registers don't cause this. Why?

---

**Software vs. Hardware**

Equivalent techniques
☑ differ in applicability

Hardware:
☑ high branch prediction accuracy
☑ has dynamic information on latencies (e.g., cache misses)
☑ transparent!
☑ works for generic code (any code)
  ✓ databases, desktop applications, numeric code, compilers
  ✗ limited reorder buffer size
  ✗ high cost/complexity
Software vs. Hardware

Software:
✓ can look at large amounts of code
✓ no hardware cost
✓ works for regular code
   ✓ engineering applications, numeric code, weather prediction
× low branch prediction accuracy
× does not have dynamic information on latencies

VLIW: Software ILP

Very long instruction word
Provide a long instruction word with one operation per FU
Instruction latencies are fixed
Compiler packs independent instructions into VLIW
   □ compiler schedules all hardware resources
Entire long word issues as a “unit”
Result: ILP ith simple hardware, simple control, fast clock
VLIW: Scheduling

Code scheduling in software only
Loop unrolling, software pipelining, trace scheduling

Architectural support
- deferred interrupts
  - enhances scheduling opportunity
- predicated execution (conditional moves)
  - less need for hardware prediction
- more registers
  - renaming not necessary?

VLIW: History

Floating point systems, array processors
- very successful in 70’s
- all latencies fixed
- fast, single-level memory

Multiflow
- Josh Fisher (now at HP)
- 1980’s minicomputer => failed

Cydrome
- Bob Rau (now at HP)
- 1980’s minicomputer => failed

Tera
- Burton Smith, 1980’s-1990’s supercomputer, keeps failing?
VLIW: History

Intel IA-64
- co-developed by HP (surprise, surprise!)
- will it succeed?
- latest news => compiler works well for regular applicatons!

VLIW Problems

Same as those with static scheduling
Latencies are not fixed!
Caches are here to stay
Need mechanisms to deal with latency dynamically
**Instruction Dispatch & Issue**

Places data may potentially reside:
1. registers
2. reservation stations
3. within functional units
4. reorder buffer

Places instructions issue from:
1. single shared queue
2. per-instruction type queue
3. reservation stations
Tomasulo’s Algorithm

Logical registers correspond to physical storage
Reservation stations are randomly accessed
No reorder buffer => imprecise interrupts

POWER1 (RS6000)’s Method

At dispatch, rename registers
Instruction issue queues
dynamic issue across (not within) queues

Conventional dataflow:
only between registers & FUs

Precise state maintained in piecemeal fashion
condition registers use history file
integer instrs complete in order (FP not allowed to pass int)
floating point exceptions are imprecise
More physical registers than logical registers
Renaming avoids WAW and WAR hazards
Turns instruction stream into “single assignment” form

Assume 8 logical, 16 physical registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register Map (before)</th>
<th>Register Map (after)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If f6, 34(r20)</td>
<td>Free list=p0,p2,p4,p6,p8</td>
<td>f6-&gt;p0</td>
</tr>
<tr>
<td>If f2, 45(r3)</td>
<td>f4-&gt;p4, f2-&gt;p6</td>
<td></td>
</tr>
<tr>
<td>mult f4, f2, f6</td>
<td>f2-&gt;p6</td>
<td></td>
</tr>
<tr>
<td>sub f12, f6, f4</td>
<td>f4-&gt;p8</td>
<td></td>
</tr>
<tr>
<td>div f14, f2, f6</td>
<td>f4-&gt;p8</td>
<td></td>
</tr>
</tbody>
</table>
**Reclaiming Registers in POWER 1**

Reclaiming registers (returning to free pool)
- Ignoring precise interrupts,
  - a register can be returned after last read is done
- Considering interrupts
  - a register can be returned after it is logically over-written

This can be done in many ways
  - e.g., via usage counter per register
  - later we will see a straightforward way

---

**Guri Sohi’s Method Illustrated**

Logical registers == physical registers
Reorder buffer: (1) commits state in order, (2) holds renamed registers
Reservation stations are randomly accessed
Registers read only at dispatch
Values come from: (1) reorder buffer, (2) functional units
MIPS R10000's Method

Uses logical registers ≠ physical registers (like POWER1)

Data movement only between regs and units

Reorder buffer used for control

○ register reservations
○ returning physical registers to free pool

Register reservation bits

○ apply to physical registers
○ monitored by instructions in RS (replaces tag compares)
MIPS R10000: Reclaiming Registers

Reclaiming physical registers
- reorder buffer contents:
  - PC, errors, result register, previous mapping
  - previous physical register mapping is copied to the ROB
  - when a new mapping is made
- when an instruction is committed from the reorder buffer
  - the old physical register is no longer needed
  - so can be reclaimed

Speculative Execution

Predict branches and speculatively execute instructions
1. Correct: great!
2. Incorrect:
   - Squash later speculative instructions
   - Exactly like precise interrupts

Reorder buffer can be used for squashing
- add “speculative bit” to reorder buffer
- do not commit any instruction if speculative bit is set
- correct => clear speculative bit
- incorrect => throw away speculative instrs in reorder buffer
- remove using “old mapping information” to restore map
Case Study: MIPS R10000

4-way superscalar - 1 memory, 2 ALU, 2 FP
register renaming (32 logical -> 64 physical)
reorder buffer
4-deep branch speculation
resume cache:
  ∙ keeps instructions on non-predicted path for fast recovery
2-way associative, 32 KB instruction and data caches