CSE 502 Graduate Computer Architecture
Lec 11 – More Instruction Level Parallelism Via Speculation

Larry Wittie
Computer Science, StonyBrook University

http://www.cs.sunysb.edu/~cse502 and ~lw

Slides adapted from David Patterson, UC-Berkeley cs252-s06
Review from Last Time #1

• Leverage Implicit Parallelism for Performance: Instruction Level Parallelism
• Loop unrolling by compiler to increase ILP
• Branch prediction to increase ILP
• Dynamic HW exploiting ILP
  – Works when can’t know dependence at compile time
  – Can hide L1 cache misses
  – Code for one machine runs well on another
Review from Last Time #2

- Reservations stations: *renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards
  - Allows loop unrolling in HW
- Not limited to basic blocks
  (integer units get ahead, beyond branches)
- Helps cache misses as well
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- 360/91 descendants are Pentium 4, Power 5, AMD Athlon/Opteron, …
Outline

• ILP
• Speculation
• Speculative Tomasulo Example
• Memory Aliases
• Exceptions
• VLIW
• Increasing instruction bandwidth
• Register Renaming vs. Reorder Buffer
• Value Prediction
Speculation For Greater ILP

• Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct
  – Speculation ⇒ fetch, issue, and execute instructions as if branch predictions were always correct
  – Dynamic scheduling ⇒ only fetches and issues instructions

• Essentially a data flow execution model: Operations execute as soon as their operands are available
Speculation For Greater ILP

• Three components of HW-based speculation:
  1. Dynamic branch prediction to choose which instructions to execute
  2. Speculation to allow execution of instructions before control dependences are resolved
     + Ability to undo effects of incorrectly speculated sequence
  3. Dynamic scheduling to deal with scheduling of combinations of basic blocks revealed at runtime
Adding Speculation to Tomasulo

• Must separate execution from allowing instruction to finish or “commit”
• This additional step is called instruction commit; it occurs whenever the branch prediction is confirmed for the branch immediately before a block of speculated instructions.
• When an instruction is no longer speculative, allow it to update the register file or memory.
• Requires an additional set of buffers to hold results of instructions that have finished execution but have not committed.
• This reorder buffer (ROB) is also used to pass results among instructions that may be speculated.
Reorder Buffer (ROB)

- In Tomasulo’s algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file.
- With speculation, the register file is not updated until the instruction commits. (i.e., when know for sure that the instruction should have executed)
- The ROB supplies operands in the interval between end of instruction execution and instruction commit.
  - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo’s algorithm.
  - ROB extends architectured registers as the reservation stations did.
Reorder Buffer Entry Fields

Each entry in the ROB contains four fields:

1. Instruction type
   - A branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, each of which has a register destination for writeback)

2. Destination
   - Register number (for loads and ALU operations) or memory address (for stores) - where the instruction result should be written

3. Value
   - Value of instruction result being held until the instruction commits

4. Ready
   - Indicates that instruction has completed execution, and the value is ready once the instruction commits
Reorder Buffer Operation

- Holds instructions in FIFO order, exactly as issued
- When instructions complete, results placed into ROB
  - Supplies operands to other instruction between execution complete & commit ⇒ more registers like RSs (reservation stations)
  - Tag results with ROB buffer number instead of reservation station number
- Instructions commit ⇒ values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions

Commit path from head of buffer
4 Steps of Speculative Tomasulo Algorithm
(steps added for speculation in blue)

1. Issue—get instruction from FP Op Queue
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called “dispatch”)

2. Execution—operate on operands (EX)
   Checks for RAW hazards; when both operands ready then execute; if not ready, watch Common Data Bus for result; when both in reservation station, execute (sometimes called “issue”)

3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. Commit—update register with reorder result
   When instr. at head of reorder buffer & result are present and the branch before it has been confirmed, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer above (executed after) the branch (sometimes called “graduation”)
Tomasulo With Reorder buffer:

FP Op Queue

Reorder Buffer

FP adders

FP multipliers

Reservation Stations

Registers

Dest

Dest

Dest from Memory

To Memory

LD F0,16(R2)

ADDD F10,F4,F0

DIVD F2,F10,F6

BNE F2,<...>

LD F4,0(R3)

ADDD F0,F4,F6

ST 0(R3),F4

LD F0,16(R2) N

ROB7

ROB6

ROB5

ROB4

ROB3

ROB2

ROB1

10+R2

3/10,15/11
Tomasulo With Reorder buffer:

FP Op Queue

Reorder Buffer

FP adders

FP multipliers

Reservation Stations

Registers

To Memory

from Memory

Dest Value Instruction Done?

ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1

LD F0,16(R2)
ADDD F10,F4,F0
DIVD F2,F10,F6
BNE F2,<…>
LD F4,0(R3)
ADDD F0,F4,F6
ST 0(R3),F4

Dest

F0

1 10+R2

3/10,15/11
Tomasulo With Reorder buffer:

Reorder Buffer

FP Op Queue

<table>
<thead>
<tr>
<th>Dest. Value</th>
<th>Instruction</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>DIVD F2,F10,F6</td>
<td>N</td>
</tr>
<tr>
<td>F10</td>
<td>ADDD F10,F4,F0</td>
<td>N</td>
</tr>
<tr>
<td>F0</td>
<td>LD F0,16(R2)</td>
<td>N</td>
</tr>
</tbody>
</table>

Registers

FP adders

FP multipliers

FP Op Queue

Reservation Stations

To Memory

from Memory

Dest from Memory

dest.

Value     Instruction

F2          | DIVD F2,F10,F6 |
F10         | ADDD F10,F4,F0 |
F0          | LD F0,16(R2)  |

Newest

Oldest

LD F0,16(R2)
ADDD F10,F4,F0
DIVD F2,F10,F6
BNE F2,<...>
LD F4,0(R3)
ADDD F0,F4,F6
ST 0(R3),F4
Tomasulo With Reorder buffer:

<table>
<thead>
<tr>
<th>Dest. Value</th>
<th>Instruction</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>F0</td>
<td>ADDD F0,F4,F6</td>
<td>N</td>
</tr>
<tr>
<td>F4</td>
<td>LD F4,0(R3)</td>
<td>N</td>
</tr>
<tr>
<td>--</td>
<td>BNE F2,&lt;&gt;</td>
<td>N</td>
</tr>
<tr>
<td>F2</td>
<td>DIVD F2,F10,F6</td>
<td>N</td>
</tr>
<tr>
<td>F10</td>
<td>ADDD F10,F4,F0</td>
<td>N</td>
</tr>
<tr>
<td>F0</td>
<td>LD F0,16(R2)</td>
<td>N</td>
</tr>
</tbody>
</table>

FP adders:
- ADDD R(F4), ROB1
- ADDD ROB5, R(F6)

FP multipliers:
- DIVD ROB2, R(F6)

Reservesation Stations:
- ADDD R(F4), ROB1
- DIVD ROB2, R(F6)
- ADDD ROB5, R(F6)

Registers:
- LD F0, 16(R2)
- ADDD F10, F4, F0
- DIVD F2, F10, F6
- BNE F2, <...
- LD F4, 0(R3)
- ADDD F0, F4, F6
- ST 0(R3), F4

FP adders:
- ADDD R(F4), ROB1
- ADDD ROB5, R(F6)

FP multipliers:
- DIVD ROB2, R(F6)

Reorder Buffer:
- ROB7
- ROB6
- ROB5
- ROB4
- ROB3
- ROB2
- ROB1

To Memory:
- Destination Value
- Instruction

From Memory:
- Destination
- Value

Instruction Queue:
- ADDD F0,F4,F6
- LD F4,0(R3)
- BNE F2,<...>
- DIVD F2,F10,F6
- ADDD F10,F4,F0
- LD F0,16(R2)
- BNE F2,<...>
- LD F4,0(R3)
- ADDD F0,F4,F6
- ST 0(R3),F4

Reorder Buffer Process:
1. Load F0, 16(R2)
2. ADDD F10, F4, F0
3. DIVD F2, F10, F6
4. BNE F2, <...>
5. LD F4, 0(R3)
6. ADDD F0, F4, F6
7. ST 0(R3), F4
Tomasulo With Reorder buffer:

**Reorder Buffer**

- LD  F0,16(R2)
- ADDD F10,F4,F0
- DIVD F2,F10,F6
- BNE  F2,<...>
- LD  F4,0(R3)
- ADDD F0,F4,F6
- ST  0(R3),F4

**FP Op Queue**

- ADDD R(F4),ROB1
- ADDD ROB5,R(F6)

**FP adders**

**FP multipliers**

**Reservation Stations**

**Dest**

- R(F4),ROB1
- ROB5,R(F6)
- ROB2,R(F6)

**Registers**

**To Memory**

- N
- N
- N
- N

**From Memory**

- 10+R2
- 0+R3

**Done?**

- ROB7
- ROB6
- ROB5
- ROB4
- ROB3
- ROB2
- ROB1

**FP adders**

**FP multipliers**

**Dest Value**

- Instruction
- Done?

- ROB5
- ST  0(R3),F4
- F0
- ADDD F0,F4,F6
- F4
- LD  F4,0(R3)
- --
- BNE  F2,<...>
- F2
- DIVD F2,F10,F6
- F10
- ADDD F10,F4,F0
- F0
- LD  F0,16(R2)

**Dest**

- 1
- 5

**Oldest**

**Newest**

3/10,15/11
Tomasulo With Reorder buffer:

Reorder Buffer

FP adders

FP multipliers

FP Op Queue

Registers

Reservation Stations

Dest Value | Instruction | Done?
---|---|---
M[10] | ST 0(R3),F4 | Y
F0 | ADDD F0,F4,F6 | N
F4 M[10] | LD F4,0(R3) | Y
-- | BNE F2,<...> | N
F2 | DIVD F2,F10,F6 | N
F10 | ADDD F10,F4,F0 | N
F0 | LD F0,16(R2) | N

LD F0,16(R2)
ADDD F10,F4,F0
DIVD F2,F10,F6
BNE F2,<...>
LD F4,0(R3)
ADDD F0,F4,F6
ST 0(R3),F4
Tomasulo With Reorder buffer:

### Reorder Buffer

<table>
<thead>
<tr>
<th>ROB7</th>
<th>ROB6</th>
<th>ROB5</th>
<th>ROB4</th>
<th>ROB3</th>
<th>ROB2</th>
<th>ROB1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>ROB2</td>
<td>ROB1</td>
<td>ROB2</td>
<td>ROB1</td>
<td>ROB2</td>
<td>ROB1</td>
<td>ROB2</td>
</tr>
<tr>
<td>ROB6</td>
<td>ROB5</td>
<td>ROB6</td>
<td>ROB5</td>
<td>ROB6</td>
<td>ROB5</td>
<td>ROB6</td>
</tr>
<tr>
<td>ROB7</td>
<td>ROB6</td>
<td>ROB7</td>
<td>ROB6</td>
<td>ROB7</td>
<td>ROB6</td>
<td>ROB7</td>
</tr>
</tbody>
</table>

### Registers

- LD   F0, 10 (R2)
- ADDD F10, F4, F0
- DIVD F2, F10, F6
- BNE  F2, <...>
- LD   F4, 0 (R3)
- ADDD F0, F4, F6
- ST   0 (R3), F4

### FP adders

- ADDD R(F4), ROB1

### FP multipliers

- DIVD ROB2, R(F6)

### From Memory

- 1 10+R2

### To Memory

- Dest Value
- Instruction
- Done?

<table>
<thead>
<tr>
<th>ROB2</th>
<th>ROB1</th>
<th>ROB2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD F0, F4, F6</td>
<td>M[10]</td>
<td>ST 0 (R3), F4</td>
</tr>
<tr>
<td>M[10]</td>
<td>LD F4, 0 (R3)</td>
<td></td>
</tr>
<tr>
<td>BNE F2, &lt;...&gt;</td>
<td>DIVD F2, F10, F6</td>
<td></td>
</tr>
<tr>
<td>ADDD F10, F4, F0</td>
<td>LD F0, 10 (R3)</td>
<td></td>
</tr>
<tr>
<td>LD F0, 10 (R2)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### FP Op Queue

<table>
<thead>
<tr>
<th>Dest</th>
<th>FP adders</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>ADDD R(F4), ROB1</td>
</tr>
</tbody>
</table>

### Reservation Stations

- Newest
- Oldest
Tomasulo With Reorder buffer:

<table>
<thead>
<tr>
<th>Dest. Value</th>
<th>Instruction</th>
<th>Done?</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>M[10]</td>
<td>Y</td>
</tr>
<tr>
<td>F0</td>
<td>ADDD F0,F4,F6</td>
<td>Ex</td>
</tr>
<tr>
<td>F4</td>
<td>LD F4,0(R3)</td>
<td>Y</td>
</tr>
<tr>
<td>--</td>
<td>BNE F2,&lt;...&gt;</td>
<td>N</td>
</tr>
<tr>
<td>F2</td>
<td>DIVD F2,F10,F6</td>
<td>N</td>
</tr>
<tr>
<td>F10</td>
<td>ADDD F10,F4,F0</td>
<td>N</td>
</tr>
<tr>
<td>F0</td>
<td>LD F0,10(R2)</td>
<td>N</td>
</tr>
</tbody>
</table>

FP adders
FP multipliers
Reservation Stations
Reorder Buffer
Registers
To Memory
from Memory

What about memory hazards???

FP Op Queue

Oldest
Newest

Tomasulo With Reorder buffer:
Avoiding Memory Hazards

• WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending

• RAW hazards through memory are avoided by two restrictions:
  1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the Addr. field of the load, and
  2. maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.

• these restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data
Exceptions and Interrupts

• IBM 360/91 invented “imprecise interrupts”
  – “Computer stopped at this PC; error likely near this address”
  – Not so popular with programmers
  – Also, what about Virtual Memory? (Not in IBM 360)

• Technique for both precise interrupts/exceptions and speculation: in-order completion and in-order commit
  – If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
  – Exactly the same must be done for precise exceptions

• Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB
  – If a speculated instruction raises an exception, the exception is recorded in the ROB
  – There are reorder buffers in all new processors to provide precise interrupts, which help programmers find errors faster.
Multi-Issue - Getting CPI Below 1

- CPI ≥ 1 if issue only 1 instruction every clock cycle
- Multiple-issue processors come in 3 flavors:
  1. statically-scheduled superscalar processors,
  2. dynamically-scheduled superscalar processors, and
  3. VLIW (very long instruction word) processors (static sched.)
- The 2 types of superscalar processors issue varying numbers of instructions per clock
  - use in-order execution if they are statically scheduled, or
  - out-of-order execution if they are dynamically scheduled
- VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium)
VLIW: Very Large Instruction Word

• Each “instruction” has explicit coding for multiple operations
  – In IA-64, grouping called a “packet”
  – In Transmeta, grouping called a “molecule” (with “atoms” as ops)
  – Moderate LIW also used in Cray/Tera MTA-2

• Tradeoff instruction space for simple decoding
  – The long instruction word has room for many operations
  – By definition, all the operations the compiler puts in one long instruction word are independent => can execute in parallel
  – E.g., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
    » 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 bits wide
  – Need compiling techniques to schedule across several branches (called “trace scheduling”)

Thrice Unrolled Loop that Eliminates Stalls for Scalar Pipeline Computers

1 Loop: L.D F0,0(R1)  
2 L.D F6,−8(R1)  
3 L.D F10,−16(R1)  
4 ADD.D F4,F0,F2  
5 ADD.D F8,F6,F2  
6 ADD.D F12,F10,F2  
7 S.D 0(R1),F4  
8 S.D −8(R1),F8  
9 DSUBUI R1,R1,#24  
10 BNEZ R1,LOOP  
11 S.D 8(R1),F12 ; 8−24 = −16

11 clock cycles, or 3.67 per iteration
**Loop Unrolling in VLIW**

Unrolled 7 times to avoid stall delays from ADD.D to S.D
7 results in 9 clocks, or 1.3 clocks per iteration (2.8X: 1.3 vs 3.67)
Average: 2.5 ops per clock (23 ops in 45 slots), 51% efficiency

Note: 8, not -48, after DSUBUI R1,R1,#56 - which may be out of place. See next slide.

Note: We needed more registers in VLIW (used 15 pairs vs. 6 in SuperScalar)

### Memory references

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>S.D -8(R1),F8</td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>S.D -24(R1),F16</td>
<td></td>
<td></td>
<td>DSUBUI R1,R1,#56</td>
<td>8</td>
</tr>
<tr>
<td>S.D -32(R1),F20</td>
<td>S.D -40(R1),F24</td>
<td></td>
<td></td>
<td>BNEZ R1,LOOP</td>
<td>9</td>
</tr>
<tr>
<td>S.D 8(R1),F28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

L.D to ADD.D: +1 Cycle
ADD.D to S.D: +2 Cycles
Loop Unrolling in VLIW

L.D to ADD.D: +1 Cycle
ADD.D to S.D: +2 Cycles

<table>
<thead>
<tr>
<th>Memory reference 1</th>
<th>Memory reference 2</th>
<th>FP operation 1</th>
<th>FP op. 2</th>
<th>Int. op/branch</th>
<th>Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F0,0(R1)</td>
<td>L.D F6,-8(R1)</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>L.D F10,-16(R1)</td>
<td>L.D F14,-24(R1)</td>
<td>ADD.D F4,F0,F2</td>
<td>ADD.D F8,F6,F2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>L.D F18,-32(R1)</td>
<td>L.D F22,-40(R1)</td>
<td>ADD.D F12,F10,F2</td>
<td>ADD.D F16,F14,F2</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>L.D F26,-48(R1)</td>
<td></td>
<td>ADD.D F20,F18,F2</td>
<td>ADD.D F24,F22,F2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>S.D 0(R1),F4</td>
<td>S.D -8(R1),F8</td>
<td>ADD.D F28,F26,F2</td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>S.D -16(R1),F12</td>
<td>S.D -24(R1),F16</td>
<td></td>
<td>DSUBUI R1,R1,#56</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>S.D 24(R1),F20</td>
<td>S.D 16(R1),F24</td>
<td></td>
<td>BNEZ R1,LOOP</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>S.D 8(R1),F28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>L.D to ADD.D: +1 Cycle</td>
<td></td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

Having 7 results in 9 clocks, or 1.3 clocks per iteration is 2.8 times faster (1.3 vs 3.67).
Having 23 operations issued in 45 slots means 51% efficiency in filling issue slots.

The earlier slide with 11 cycles of one pipeline for 3 loop iterations shows no stall between the integer ops DSUBUI and BNEZ, but does show a 1-cycle branch delay slot after the BNEZ. This slide puts the two integer ops a cycle earlier. S.D 8 ... fills the slot.

Note: 24 not -32, 16 not -40, and 8 not -48 after the early DSUBUI R1,R1,#56.

Note: We need more registers in VLIW (we used 15 pairs vs. 6 in SuperScalar)
Problems with 1st Generation VLIW

• Increase in code size
  – generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  – whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding

• Operated in lock-step; no hazard detection HW
  – a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  – Compiler might predict function unit stalls, but cache stalls are hard to predict

• Binary code incompatibility
  – Pure VLIW => different numbers of functional units and unit latencies require different versions of the code
Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- **IA-64**: instruction set architecture – 64 bits per integer
- 128 64-bit integer regs + 128 82-bit floating point regs
  - Not separate register files per functional unit as in old VLIW
- Hardware checks dependencies (interlocks => binary compatibility over time)
- **Predicated execution** of some instructions avoids many branches (select 1 out of 64 1-bit flags) “if f12, add op” => 40% fewer mispredictions?
- **Itanium™** was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process
- **Itanium 2™** is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 µ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3
Increasing Instruction Fetch Bandwidth

- Predicts next instruction address, sends it out before decoding instruction
- PC of branch sent to BTB
- When match is found, Predicted PC is returned
- If a branch is predicted taken, instruction fetch continues at Predicted PC

Branch Target Buffer (BTB)

- PC of instruction to fetch
- Look up
- Number of entries in branch-target buffer
- Branch predicted taken or untaken
- Yes: then instruction is branch and predicted PC should be used as the next PC
- No: instruction is not predicted to be branch; proceed normally

© 2003 Elsevier Science (USA). All rights reserved.
Inst. Fetch BW: Return Address Predictor

- Small buffer of return addresses acts as a stack
- Caches most recent return addresses
- Call $\Rightarrow$ Push a return address on stack
- Return $\Rightarrow$ Pop an address off stack & predict as new PC

![Graph showing misprediction frequency vs. return address buffer entries for different programs (go, m88ksim, cc1, compress, xlis, jpeg, perl, vortex).]
More Instruction Fetch Bandwidth

- **Integrated branch prediction**: Branch predictor is part of instruction fetch unit and is constantly predicting branches.

- **Instruction prefetch**: Instruction fetch units prefetch to deliver multiple instructions per clock, integrating it with branch prediction.

- **Instruction memory access and buffering**: Fetching multiple instructions per cycle:
  - May require accessing multiple cache blocks (prefetch to hide cost of crossing cache blocks).
  - Provides buffering, acting as on-demand unit to provide instructions to issue stage as needed and in quantity needed.
Speculation: Register Renaming vs. ROB

• Alternative to ROB is a larger physical set of registers combined with register renaming
  – Extended registers replace function of both ROB and reservation stations

• Instruction issue maps names of architectural registers to physical register numbers in extended register set
  – On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards)
  – Speculation recovery easy because a physical register holding an instruction destination does not become the architecturally visible register until the instruction commits

• Most Out-of-Order processors today use extended registers with renaming
Value Prediction – just so-so (not used)

- Attempts to predict value produced by instruction
  - E.g., Loads a value that changes infrequently
- Value prediction is useful only if it significantly increases ILP
  - Focus of research has been on loads; so-so results, no processor uses value prediction
- Related topic is address aliasing prediction
  - RAW for load and store or WAW for 2 stores
- Address alias prediction is both more stable and simpler since need not actually predict the address values, only whether such values conflict
  - Has been used by a few processors
(Mis) Speculation on Pentium 4

% of micro-ops not used (Instr. becomes micro-ops as fetched)

- Integer: 43%, 45%, 39%
- Floating Point: 24%, 24%, 20%, 3%, 1%, 1%, 0%
Perspective

- Early interest in multiple-issue because wanted to improve performance without affecting uniprocessor programming model
- Taking advantage of ILP is conceptually simple, but design problems are amazingly complex in practice
- Conservative in ideas, just faster clock and bigger chip
- Processors of last 5 years (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the first dynamically scheduled, multiple-issue processors announced in 1995
  - Clocks 10 to 20X faster, caches 4 to 8X bigger, 2 to 4X as many renaming registers, and 2X as many load-store units ⇒ performance 8 to 16X
- Peak v. delivered performance gap increasing
In Conclusion …

• Interrupts and exceptions either interrupt the current instruction or happen between instructions
  – Possibly large quantities of state must be saved before interrupting

• Machines with *precise exceptions* provide one single point in the program as PC to restart execution
  – All instructions before that point have completed
  – No instructions after or including that point have completed

• Hardware techniques exist for precise exceptions even in the face of out-of-order execution!
  – Important enabling factor for out-of-order execution