Lecture 5
Instruction Level Parallelism (3)

EEC 171 Parallel Architectures
John Owens
UC Davis
Credits

• © John Owens / UC Davis 2007–9.

VLIW Advantages

- Advantages
  - Simpler hardware (potentially less power hungry)
  - Potentially more scalable
    - Allow more instr’s per VLIW bundle and add more FUs
VLIW Disadvantages

- Programmer/compiler complexity and longer compilation times
  - Deep pipelines and long latencies can be confusing (making peak performance elusive)
- Lock step operation, i.e., on hazard all future issues stall until hazard is resolved (hence need for predication)
- Object (binary) code incompatibility
- Needs lots of program memory bandwidth
- Code bloat
  - Noops are a waste of program memory space
  - Loop unrolling to expose more ILP uses more program memory space
Review: Multiple-Issue Processor Styles

- **Dynamic multiple-issue processors** (aka *superscalar*)
  - Decisions on which instructions to execute simultaneously (in the range of 2 to 8 in 2005) are being made dynamically (at run time by the *hardware*).
  - E.g., IBM Power 2, Pentium 4, MIPS R10K, HP PA 8500 IBM.

- **Static multiple-issue processors** (aka *VLIW*)
  - Decisions on which instructions to execute simultaneously are being made statically (at compile time by the *compiler*).
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer).
    - 128 bit “bundles” containing 3 instructions each 41 bits + 5 bit template field (specifies which FU each instr needs).
    - Five functional units (IntALU, MMedia, DMem, FPALU, Branch).
    - Extensive support for speculation and predication.
<table>
<thead>
<tr>
<th></th>
<th>CISC</th>
<th>RISC</th>
<th>Superscalar</th>
<th>VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr size</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instr format</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory reference</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key Issues</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction flow</td>
<td><img src="#" alt="Flow Diagram" /></td>
<td><img src="#" alt="Flow Diagram" /></td>
<td><img src="#" alt="Flow Diagram" /></td>
<td><img src="#" alt="Flow Diagram" /></td>
</tr>
</tbody>
</table>
Today’s Outline

• How do we mitigate branches?
  • Branch prediction

• How do we avoid branches?
  • Predication
  • Speculation

• How do we build high-performance, predicated, speculative VLIW-ish hardware?
  • EPIC and Itanium
What is a basic block?

- “Experiments and experience indicated that only a factor of 2 to 3 speedup from parallelism was available within basic blocks. (A basic block of code has no jumps in except at the beginning and no jumps out except at the end.)”
  — “Very Long Instruction Word Architectures and the ELI-512”, Joseph A. Fisher
Branches Limit ILP

• Programs average about 5 instructions between branches

• Can’t issue instructions if you don’t know where the program is going

• Current processors issue 4–6 operations/cycle

• Conclusion: Must exploit parallelism across multiple basic blocks
### Branch Prediction Matters

- **Alpha 21264:**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Misprediction Rate</th>
<th>Performance Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td>16.5%</td>
<td>40%</td>
</tr>
<tr>
<td>compress</td>
<td>9%</td>
<td>30%</td>
</tr>
<tr>
<td>gcc</td>
<td>7%</td>
<td>20%</td>
</tr>
</tbody>
</table>

- (From Ranganathan and Jouppi, via Dan Connors)
Compiler: Static Prediction

- Predict at compile time whether branches will be taken before execution

- Schemes
  - Predict not-taken
  - Predict taken
    - Would be hard to squeeze into our pipeline
      - Can’t compute target until ID
Compiler: Static Prediction

- Predict at compile time whether branches will be taken before execution
- Schemes
  - Backwards taken, forwards not taken
    - How do we tell?
    - Why is this a good idea?
Compiler: Static Prediction

- Predict at compile time whether branches will be taken before execution

- Schemes
  - Predict taken
  - Backwards taken, forwards not taken (good performance for loops)

- No run-time adaptation: bad performance for data-dependent branches

- if (a == 0) b = 3; else b = 4;
Hardware-based Dynamic Branch Prediction

- Single level (Simple counters) – predict outcome based on past branch behavior
  - FSM (Finite State Machine)
- Global Branch Correlation – track relations between branches
  - GAs
  - Gshare
- Local Correlation – predict outcome based on past branch behavior
- Hybrid predictors (combination of local and global)
- Miscellaneous
  - Return Address Stack (RAS)
  - Indirect jump prediction
Mis-prediction Detections and Feedbacks

- **Detections:**
  - At the end of decoding
    - Target address known at decoding, and does not match
  - Flush fetch stage
  - At commit (most cases)
    - Wrong branch direction or target address does not match
  - Flush the whole pipeline

- **Feedbacks:**
  - Any time a mis-prediction is detected
  - At a branch’s commit
1-bit “Self Correlating” Predictor

- Let’s consider a simple model. Store a bit per branch: “last time, was the branch taken or not”.
- Consider a loop of 10 iterations before exit:
  - for (...)
    - for (i=0; i<10; i++)
      - a[i] = a[i] * 2.0;
- What’s the accuracy of this predictor?
Dynamic Branch Prediction

- Performance = $f(\text{accuracy}, \text{cost of misprediction})$

- Branch History Table: Lower bits of PC address index table of 1-bit values
  - Says whether or not branch taken last time
  - No address check

- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
  - End of loop case, when it exits instead of looping as before
  - First time through loop on next time through code, when it predicts exit instead of looping
Predictor for a Single Branch

General Form

1. Access PC

state

2. Predict Output T/NT

3. Feedback T/NT

1-bit prediction

Predict Taken 1

Predict Not Taken 0

Feedback

T

NT

Predict Not Taken
Dynamic Branch Prediction (Jim Smith, 1981)

- Solution: 2-bit scheme where change prediction only if get misprediction twice:
  - Red: stop, not taken
  - Green: go, taken
  - Adds hysteresis to decision making process
Simple ("2-bit") Branch History Table Entry

Prediction for next branch.
(1 = take, 0 = not take)
Initialize to 0.

Was last prediction correct?
(1 = yes, 0 = no)
Initialize to 1.

Flip bit if prediction is not correct and “last predict correct” bit is 0.

Set to 1 if prediction bit was correct.
Set to 0 if prediction bit was incorrect.
Set to 1 if prediction bit flips.
Branch prediction hardware

- Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)

- Note: must check for branch match now, since can’t use wrong branch address

![Branch prediction diagram]
Some Interesting Patterns

• Format: Not-taken (N) (0), Taken (T)(1)

• TTTTTTTTTTT
  • 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ... : Should give perfect prediction

• NTTTNNTTNNTT
  • 0 0 1 1 1 1 0 1 0 0 1 1 0 0 1 1 1 1 1 0 0 0 1 0 ... : Will mispredict 1/2 of the time

• N*N[TNTN]
  • 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 ... : Should alternate incorrectly

• N*T[TNTN]
  • 0 0 0 0 0 0 0 1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 ... : Should alternate incorrectly
Pentium 4 Branch Prediction

- Critical to Performance
  - 20 cycle penalty for misprediction

- Branch Target Buffer
  - 2048 entries
  - 12 bits of history
  - Adaptive algorithm
    - Can recognize repeated patterns, e.g., alternating taken–not taken

- Handling BTB misses
  - Detect in cycle 6
  - Predict taken for negative offset, not taken for positive (why?)
Branch Prediction Summary

- Consider for each branch prediction
  - Hardware cost
  - Prediction accuracy
  - Warm-up time
  - Correlation
  - Interference
  - Time to generate prediction
- Application behavior determines number of branches
  - More control intensive program...more opportunity to mispredict
- What if a compiler/architecture could eliminate branches?
Avoiding branches

• Consider the following code:

```java
// x is either 0 or 1
if (x == 0) {
    a = b;
} else if (x == 1) {
    a = c;
}
```

• How many instrs does this take on average (as is)?

• Write this code with no branches (4 instructions)
Conditional move

- Consider a “conditional move” instruction:
  \[\text{CMOVZ} \text{ dst, src, cond} \quad \text{# copies src to dst if cond} \neq 0\]

- MIPS, Alpha, PowerPC, SPARC, x86 (Pentium) have this

- // x is either 0 or 1
  if (x == 0) {
    a = b;
  } else if (x == 1) {
    a = c;
  }

- Write this code with no branches (2 instructions)
Predication

- Predication can be used to eliminate branches by making the execution of an instruction dependent on a “predicate”, e.g.,

  ```
  if (p) {statement 1 } else {statement 2 }
  ```

  would normally compile using two control-flow instructions. (Why?) With predication it would compile as

  ```
  (p) statement 1
  (~p) statement 2
  ```

- The use of `(condition)` indicates that the instruction is committed only if `condition` is true

- Predication can be used to speculate as well as to eliminate branches
Predication Main Idea

- Convert *control* dependence to *data* dependence
How do we support predication in hw?
Where to evaluate predicate

• Can we evaluate the predicate early in the pipeline (annul the predicated instruction), before it gets to the ALUs?

• Or should we evaluate the predicate (annul the predicated instruction) late in the pipeline, after the operation has been through the ALU?
Predicates Are Good

- Implementing short alternative control flows
- Eliminating unpredictable branches
- Reducing the overhead of global code scheduling
Predicates Are Bad

- Annulled predicated instructions still take resources
- If the predicate is evaluated late, it might cause a data hazard
- What about executing an operation across multiple branches?
- Possible speed penalty
Predication vs. Speculation

- Got a branch? Two possible paths of execution? Not sure what to do?
  - Predication: “Do ‘em both! We’ll figure it out at the end.”
    - Blackjack equivalent: Split
  - Speculation: “Make your best guess. We’ll figure it out at the end.”
    - Blackjack equivalent: Double down
Speculation

- Speculation is used to allow execution of future instructions that (may) depend on the speculated instruction

- Speculate on the outcome of a conditional branch (branch prediction)

- Compare to out-of-order machine with branch prediction

- Speculate that a store (for which we don’t yet know the address) that precedes a load does not refer to the same address, allowing the load to be scheduled before the store (load speculation)
Speculation

- Must have (hardware and/or software) mechanisms for
  - Checking to see if the guess was correct
  - Recovering from the effects of the instructions that were executed speculatively if the guess was incorrect
    - In a VLIW processor the compiler can insert additional instrs that check the accuracy of the speculation and can provide a fix-up routine to use when the speculation was incorrect
  - Ignore and/or buffer exceptions created by speculatively executed instructions until it is clear that they should really occur
Speculation to 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 to greater ILP

- 3 components of HW-based speculation:
  - Dynamic branch prediction to choose which instructions to execute
  - Speculation to allow execution of instructions before control dependences are resolved
    - + ability to undo effects of incorrectly speculated sequence
  - Dynamic scheduling to deal with scheduling of different combinations of basic blocks
Adding Speculation to HW

- Must separate execution from allowing instruction to finish or “commit”
- This additional step called instruction commit
- When an instruction is no longer speculative, allow it to update the register file or memory
- Requires 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
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
Problems with 1st Generation VLIW

- 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 on function units, but caches hard to predict
Problems with 1st Generation VLIW

- Binary code compatibility
  - Pure VLIW $\Rightarrow$ different numbers of functional units and unit latencies require different versions of the code
EPIC Goal

- Support compiler-based exploitation of ILP
- Predication
- Compiler-based parallelism detection
- Support for memory reference speculation
- ...
How EPIC extends VLIW

- Greater flexibility in indicating parallelism between instructions & within instruction formats
  - VLIW has a fixed instruction format
    - All ops within instr must be parallel
  - EPIC has more flexible instruction formats
  - EPIC indicates parallelism between neighboring instructions
- Extensive support for software speculation
Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- IA-64: instruction set architecture
- 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 $\Rightarrow$ binary compatibility over time)
- Predicated execution (select 1 out of 64 1-bit flags) $\Rightarrow$ 40% fewer mispredictions?
Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- Itanium™ was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800 MHz
  - 6-wide, 10-stage pipeline at 800 MHz on 0.18μ process
- Itanium 2™ is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666 MHz on 0.13μ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3
**EPIC Instruction Format**

- Major opcode (4 bits)
- Minor opcode
- Immediate operands (8–22 bits)
- Register result identifier(s) (6 or 7 bits)
- Register operand identifiers (7 bits)
- Qualifying predicates (6 bits)

A few instructions do not have a QP (nearly all do!)
Instruction Formats: Bundles

Template identifies types of instructions in bundle and delineates independent operations (through “stops”)

- Instruction types
  - M: Memory
  - I: Shifts and multimedia
  - A: ALU
  - B: Branch
  - F: Floating point
  - L+X: Long

- Template encodes types
  - MII, MLX, MMI, MFI, MMF, MI_I, M_MI
  - Branch: MIB, MMB, MFB, MBB, BBB
  - Template encodes parallelism
  - All come in two flavors: with and without stop at end
EPIC Rules

<table>
<thead>
<tr>
<th>Template</th>
<th>Slot 0</th>
<th>Slot 1</th>
<th>Slot 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
<td>I</td>
<td>I</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>L</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>M</td>
<td>L</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>M</td>
<td>M</td>
<td>I</td>
</tr>
<tr>
<td>7</td>
<td>M</td>
<td>M</td>
<td>I</td>
</tr>
<tr>
<td>8</td>
<td>M</td>
<td>M</td>
<td>I</td>
</tr>
<tr>
<td>9</td>
<td>M</td>
<td>F</td>
<td>I</td>
</tr>
<tr>
<td>10</td>
<td>M</td>
<td>M</td>
<td>F</td>
</tr>
<tr>
<td>11</td>
<td>M</td>
<td>M</td>
<td>F</td>
</tr>
<tr>
<td>12</td>
<td>M</td>
<td>F</td>
<td>I</td>
</tr>
<tr>
<td>13</td>
<td>M</td>
<td>F</td>
<td>I</td>
</tr>
<tr>
<td>14</td>
<td>M</td>
<td>M</td>
<td>F</td>
</tr>
<tr>
<td>15</td>
<td>M</td>
<td>M</td>
<td>F</td>
</tr>
<tr>
<td>16</td>
<td>M</td>
<td>I</td>
<td>B</td>
</tr>
<tr>
<td>17</td>
<td>M</td>
<td>I</td>
<td>B</td>
</tr>
<tr>
<td>18</td>
<td>M</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>19</td>
<td>M</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>20</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>21</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>22</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>23</td>
<td>B</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>24</td>
<td>M</td>
<td>M</td>
<td>B</td>
</tr>
<tr>
<td>25</td>
<td>M</td>
<td>M</td>
<td>B</td>
</tr>
<tr>
<td>26</td>
<td>M</td>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>27</td>
<td>M</td>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>28</td>
<td>M</td>
<td>F</td>
<td>B</td>
</tr>
<tr>
<td>29</td>
<td>M</td>
<td>F</td>
<td>B</td>
</tr>
</tbody>
</table>

**Figure G.6** The 24 possible template values (8 possible values are reserved) and the instruction slots and stops for each format. Stops are indicated by heavy lines and may appear within and/or at the end of the bundle. For example, template 9 specifies that the instruction slots are M, M, and I (in that order) and that the only stop is between this bundle and the next. Template 11 has the same type of instruction slots but also includes a stop after the first slot. The L + X format is used when slot 1 is L and slot 2 is X.
Speculation Support

- Control speculation (we’ve seen this)
- Memory reference speculation
  - Loads moved above stores have a different opcode, ld.a (advanced load)
  - Why?
  - Advanced loads put their addresses in a table (ALAT)
  - Stores check the ALAT when storing
- Exceptions
  - Poison bits set on speculative ops that cause exceptions
  - Poison bits propagate, fault on non-speculative instructions
  - Storing a poison bit == bad
Evaluating Itanium

• “The EPIC approach is based on the application of massive resources. These resources include more load-store, computational, and branch units, as well as larger, lower-latency caches than would be required for a superscalar processor. Thus, IA-64 gambles that, in the future, power will not be the critical limitation, and that massive resources, along with the machinery to exploit them, will not penalize performance with their adverse effect on clock speed, path length, or CPI factors.” —M. Hopkins, 2000
Itanium 2 (2005)

- 1.6 GHz, 4x performance of Itanium
- 592M xtors, 423 mm², 130 W
  - P4 Extreme is 125M xtors, 122 mm²
- 3 level memory hierarchy on chip
- 11 functional units
  - 2 I-units, 4 M-units (2 ld, 2 st), 3 B-units, 2 F-units
- Fetches and issues 2 bundles (6 instrs) per cycle
Itanium 2 Pipeline: 8 Stages

- Front-end (IPG, Rotate)
  - Prefetch 32B/clock (2 bundles)
  - Branch predictor: “multilevel adaptive predictor”
- Instruction delivery (EXP, REN)
  - Distributes up to 6 instrs to 11 functional units
  - Renames registers
- Operand delivery (REG)
  - Accesses RF, bypasses, scoreboards, checks predicate
  - Stalls within one instr bundle do not cause entire bundle to stall
- Execution (EXE, DET, WRB)
  - Executes instrs, exceptions, retires instrs, writeback
Itanium Features

- Dynamic branch prediction
- Register renaming
- Scoreboarding
- Many stages before execute
  - Looks very complex!
- Why?
  - Dynamic techniques help (e.g. branch prediction)
  - Dynamic scheduling necessary for cache misses