Project Goal

- Design and implement an out-of-ordering superscalar SMIPSv2 processor

In-Order vs. Out-of-Order

...Out-of-Order (OoO) execution can increase IPC
Register Renaming

LW R2, 0(R1)
ADD R3, R6, R0
ADD R3, R2, R1
ADD R4, R3, R0
ADD R3, R6, R0

(R3 = R1 + R2)

Register renaming can solve this problem.

LW R2, 0(R1)
ADD R3, R2, R1
ADD R4, R3, R0
ADD R3, R6, R0

(R3 = R6 + R0)

Re-Ordering Buffer

<table>
<thead>
<tr>
<th>Inst</th>
<th>src1</th>
<th>src2</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
</tr>
<tr>
<td>ADD</td>
<td>P4</td>
<td>P5</td>
<td>P6</td>
</tr>
<tr>
<td>MUL</td>
<td>P7</td>
<td>P6</td>
<td>P6</td>
</tr>
<tr>
<td>AND</td>
<td>P5</td>
<td>P4</td>
<td>P7</td>
</tr>
</tbody>
</table>

ROB keeps information for OoO dispatch.

Superscalar Architecture

<table>
<thead>
<tr>
<th>Inst</th>
<th>src1</th>
<th>src2</th>
<th>dst</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
</tr>
<tr>
<td>ADD</td>
<td>P4</td>
<td>P5</td>
<td>P6</td>
</tr>
<tr>
<td>MUL</td>
<td>P7</td>
<td>P6</td>
<td>P6</td>
</tr>
<tr>
<td>AND</td>
<td>P5</td>
<td>P4</td>
<td>P7</td>
</tr>
</tbody>
</table>

Re-Ordering Buffer (ROB)

ADD P4
ADD P1
ADD P3
ADD P6
MUL P7
MUL P6
AND P5
AND P4

Mult
Adder
Outline
- Introduction
- Architectural Design
  - Main Tasks
  - Pipeline Stages
  - Microarchitectural Design
- Bluespec Implementation
- Results
- Conclusion

Main Tasks
- Branch
  - Resolve branches
  - Rollback on mis-predictions
- Commit
  - Finish ‘safe’ instructions

Main Tasks
- Insertion
  - Fetch instructions
  - Rename registers
  - Insert into ROB
- Dispatch
  - Send ‘ready’ instructions to execution units
- Update
  - Update ROB to show values are ready

Pipeline Stages
- ALU instructions
- Memory instructions
- Branches and Jumps
Microarchitectural Design

- Overall Design
  - Decode, Rename, Insert

- Status of entries in ROB
  - SUB updates value
  - ADD is dispatched to ALU
  - BNE is dispatched to branch unit
Microarchitectural Design

Status of entries in ROB

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
<th>F</th>
<th>Inst</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>BNE</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>SUB</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td>ADD</td>
<td>...</td>
</tr>
</tbody>
</table>

- BNE is resolved, mis-prediction

Outline

- Introduction
  - Architectural Design
  - Bluespec Implementation
    - Bluespec Rules and Methods
    - Rule Concurrency
    - Design Exploration
- Results
- Conclusion
Bluespec Rules in mkProc

- Insertion
  - discardFetch
  - decodeInsert
- Dispatch
  - dispatchALU
  - dispatchMem
  - memReq
- Branch
  - branchResolve
  - branchStep2Link
  - branchStep2
- Update
  - aluUpdate
  - memUpdate
  - memUpdateNOP
- Retire
  - retireInst

Actions in different stages should work at the same time. Why don't they?

Bluespec Methods in mkROB

- Insertion
  - Action insertEntry
  - Action insertMemEntry
- Dispatch
  - aluinFirst / Action aluinFirst
  - brinFirst / Action brinFirst
  - ActionValue memInFirstPop
  - ActionValue memInFirstPop
- Branch
  - ActionValue memReqPop

...High Method Concurrency in ROB First

Concurrency Analysis

- Initial Design
  - A huge register containing all ROB fields and entries
- Read-Write Pattern
  - Every entry is read and written by many methods
  - All action methods conflict
  - Compiling is slow

Rule Concurrency

- How to get high concurrency?
  - Every method write to RWire
    - Structural rule to handle all the cases
- Or Bluespec way!
  - Data structure separation
  - ROB Method ordering with EHR

MIT 6.375 Complex Digital Systems 2007 Spring
Data Structure Separation

- Separated Data Structure based on the number of reads and writes

<table>
<thead>
<tr>
<th>V</th>
<th>E</th>
<th>I</th>
<th>nsc 1</th>
<th>nsc 2</th>
<th>p</th>
<th>sid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Remained in Vector
  - global read
  - multiple writers

- Fitted in RegFile
  - limited read
  - single writer

- After this, compile time becomes reasonable

Lab 3 Example

- lab3 with normal FIFO:
  - wb < exe < pcgen: long path, higher IPC
  - pcgen < exe < wb: short path, lower IPC

- Why?
- FIFO as CF separator

Rule Ordering in mkProc

- Methodology of rule ordering propagation
  - Determine the top rule ordering
  - Change the method order of all leaf modules
  - Change the EHR index for state variables
  - Keep EHR index consistent within a rule

- Problem: longer critical path!

Coherent EHR index

- Conflicting to sequentially composable
- Automation in future compiler
- Larger area
  - Multiple instances of combinational circuit for different EHR index
- Longer path
  - Directly stack multiple stages
Non-Coherent EHR Index

- Early read in rule condition
  Late write in body
  - Safe in general
  - Only influence performance
- Early read in body
  Late write in body
  - Not safe in general
  - Need domain knowledge

Non-Coherent EHR Index
Unsafe usage

- Domain Knowledge
  - Snapshot taken/restored cannot be of the same epoch
  - etc...
  - Non-Coherent EHR Index in rule body
    - Error prone optimization
    - Need huge effort to analyze the interaction

Non-Coherent EHR Index
Safe usage

- FIFO Example
- Coherent EHR index produces either BFIFO or LFIFO
  - BFIFO:
    - enq(...) if(!full[0]) {empty[0] <= False;...}
    - deq() if(!empty[1]) {full[1] <= False;...}
  - LFIFO:
    - enq(...) if(!full[1]) {empty[1] <= False;...}
    - deq() if(!empty[0]) {full[0] <= False;...}
- Combinational path between methods
- FIFO in bsr with EHR
  - enq(...) if(!full[0]) {empty[1] <= False;...}
  - deq() if(!empty[0]) {full[1] <= False;...}
  - Not Conflict Free

Result of Rule Concurrency

- Highest possible concurrency in systems with one-writing-port register file
- Similar critical path and area
  - Path 2% longer
  - Area 8% larger
**Design Exploration**

- **Adjusting Pipeline**
  - Merging execute stage with update stage
  - Shortens end-to-end dependency
  - Possibly lengthens the critical path

- **Different ROB Sizes**
  - The size of ROB determines how 'far' it can find executable instructions
  - If it is too small, the performance may suffer
  - If it is too large, the penalty for mis-prediction becomes too high

---

- **Adjusting Pipeline**
  - Simply implemented by using BFIFO
  - Result
    - High concurrency attained
    - IPC 5.8% higher than the most optimized version
    - The critical path is almost the same
    - 1% longer than the most optimized version

* For 5 benchmarks used for SMPSv2
** From the result of synthesis

- **Different ROB Sizes**
  - The size of 8 was chosen.
Outline

- Introduction
- Architectural Design
- Bluespec Implementation
- Results
  - Physical Numbers
  - Performance Results
- Conclusion

Physical Numbers

- Area Analysis
  - Total Area 1.42 mm² after place and route

Physical Numbers

- Critical Path
  - 4.44ns after synthesis
    - @1.88ns: A branch dispatched from ROB
    - @2.86ns: Source is read from register file
    - @4.24ns: PC register is updated
  - 9.40ns after place and route

Performance Results

- Case 1: LAB3 - low profile version
  - wbQ size 2, no bypassing register file
- Case 2: LAB3 - high profile version
  - wbQ size 8, bypassing register file, decoupled wbQ and mem
- Case 3: OoO Superscalar - non-concurrent version
- Case 4: OoO Superscalar - initial version
  - ROB size of 8, execution and update are separate
- Case 5: OoO Superscalar - merged stages
  - ROB size of 8, execution and update are merged
Performance Results

Analysis with LAB3 SMIPSv2

- LAB3 SMIPSv2 does not suffer much from data dependency.
- In order to exploit superscalar architecture, we need to fetch and commit multiple instructions at one cycle.
- Since more execution units can be added to the current design, the performance will excel LAB3 especially in the case with more complex instructions such as multiplications.

Outline

- Introduction
- Architectural Design
- Bluespec Implementation
- Results
- Conclusion
  - Summary
  - Possible Follow-ups

Summary

- Out-of-order execution
  - All ALU instructions and memory address calculations are out-of-order.
  - Branch resolutions and memory requests are in order.
  - Speculative execution:
    Even instructions after an unresolved branch can be executed out-of-order and possibly discarded properly in case of mis-predictions.
Summary

- Superscalar architecture
  - ALU execution, branch resolution, memory address calculation and sending memory request can be done simultaneously.

- Optimal rule concurrency
  - Achieved the highest rule concurrency with single write port register file and renaming table
  - IPC reaches 1 if no mispredictions
  - Even with memory operations if ROB is large enough to compensate memory latency

Possible Follow-ups

- Multiple instruction fetch and commitment
- More execution units
- Precise interrupt handling
- Complex ALU operations

Thank you

Thanks to Prof. Arvind and Prof. Asanovic
TA Myron and Ajay

And