Group IV
Wei-Yin Chen
Myong Hyon Cho

RE-ORDER BUFFER FOR SUPERSCALAR SMIPSV2 PROCESSOR

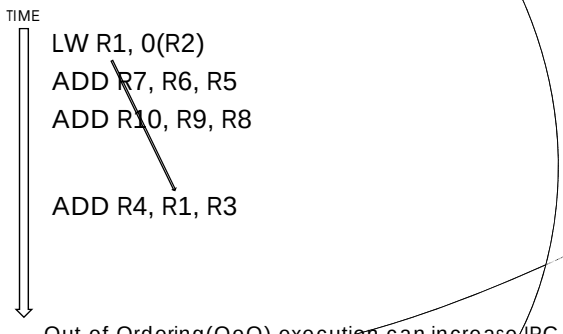
Outline

- Introduction
  - In-Order vs. Out-of-Order
  - Register Renaming
  - Re-Ordering Buffer
  - Superscalar Architecture
- Architectural Design
- Bluespec Implementation
- Results
- Conclusion

Project Goal

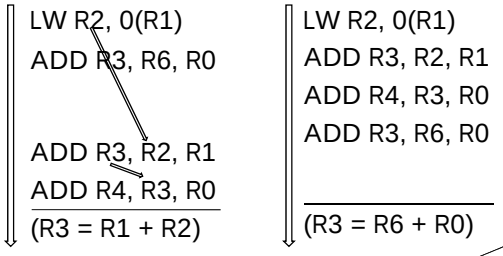
- Design and implement an out-of-ordering superscalar SMIPSv2 processor

In-Order vs. Out-of-Order



...Out-of-Ordering(OoO) execution can increase IPC

Register Renaming



Register renaming can solve this problem.

Re-Ordering Buffer

Inst	src1	src2	dst
ADD	P1	P2	P3
ADD	P4	P1	P6
MUL	P7	P6	P6
AND	P5	P4	P7

Re-Ordering Buffer (ROB)

ALU

ROB keeps information for OoO dispatch.

Register Renaming

ADDI R1, R0, 1024
LW R2, 0(R1)
ADD R3, R2, R1
ADD R4, R3, R0
ADD R5, R6, R0

R1	P1
R2	P2
R3	**P5** ~~P3~~
R4	P4

Rename Table

Superscalar Architecture

Inst	src1	src2	dst
ADD	P1	P2	P3
ADD	P4	P1	P6
MUL	P7	P6	P6
AND	P5	P4	P7

Re-Ordering Buffer (ROB)

Adder **Mult**

Outline

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

Main Tasks

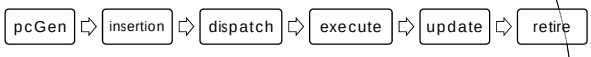
- Branch
  - Resolve branches
  - Rollback on mis-predictions
- Commit
  - Finish 'safe' instructions

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

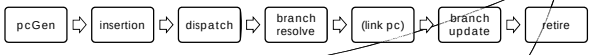
Pipeline Stages

- ALU instructions



- Memory instructions

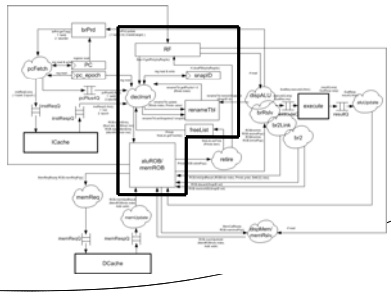
- Branches and Jumps



Microarchitectural Design

- Overall Design

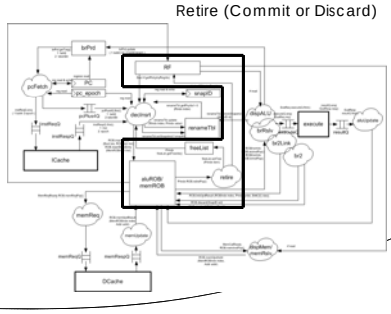
Decode, Rename, Insert



Microarchitectural Design

- Overall Design

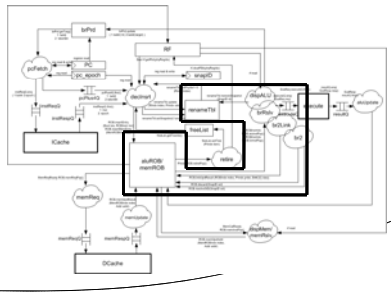
Retire (Commit or Discard)



Microarchitectural Design

- Overall Design

Branches



Microarchitectural Design

- Status of entries in ROB

V	E	F	inst	...
1	1		BNE	...
1	1	1	SUB	...
1	1		ADD	...

- SUB updates value
- ADD is dispatched to ALU
- BNE is dispatched to branch unit

Microarchitectural Design

- Status of entries in ROB

V	E	F	inst	…
1	1	1	BNE	…
0	1	1	SUB	…
0	1		ADD	…

- BNE is resolved, mis-prediction

Microarchitectural Design

- Status of entries in ROB

V	E	F	inst	…
~~0~~	~~1~~	~~1~~	~~BNE~~	~~…~~
~~0~~	~~1~~	~~1~~	~~SUB~~	~~…~~
~~0~~	~~1~~	~~1~~	~~ADD~~	~~…~~

- ADD gets the result from ALU
- ADD is now retired (discarded)

Microarchitectural Design

- Status of entries in ROB

V	E	F	inst	…
~~0~~	~~1~~	~~1~~	~~BNE~~	~~…~~
~~0~~	~~1~~	~~1~~	~~SUB~~	~~…~~
0	1		ADD	…

- SUB is retired (discarded)
- ADD cannot be retired

Outline

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

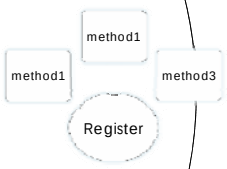
Bluespec Rules in mkProc

- 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?

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



Bluespec Methods in mkROB

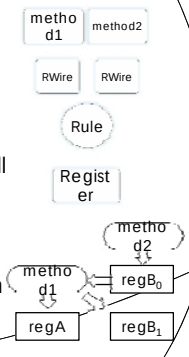
- Insertion
  - discardFetch
  - decodeInsert
- Dispatch
  - dispatchALU
  - dispatchMem
  - memReq
- Branch
  - branchStep2
- Update
  - aluUpdate
  - memUpdate
  - memUpdateNOP
- Retire
  - retireInst

- Insertion
  - `Action insertEntry`
  - `Action insertMemEntry`
- Dispatch
  - `aluInstFirst / Action aluInstPop`
  - `brInstFirst / Action brInstPop`
  - `ActionValue memInstPop`
  - `ActionValue memReqPop`
- Branch
- Update
  - `Action aluUpdResult`
  - `Action linkUpdResult`
  - `Action memUpdAddr`
  - `Bool getValidBit`
- Retire
  - `ActionValue retirePop`

...High Method Concurrency in ROB First

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



Data Structure Separation

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

V	E	F	IO	rsrc1	p	rsrc2	p	sid

rdst	lrdst	inst	taken	pc+4

Remained in Vector
- global read
- multiple writers

Fitted in RegFile
- limited read
- single writer

- After this, compile time becomes reasonable

Lab3 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 composible
- 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 bsv 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 lengthen the critical path

Design Exploration

- 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

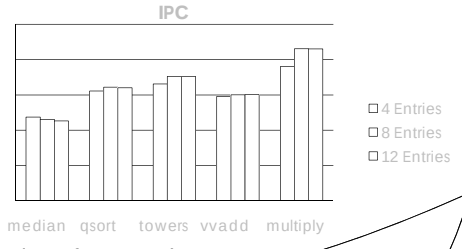
Design Exploration

- 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 SMIPSv2*
*** From the result of synthesis*

Design Exploration

- Different ROB Sizes



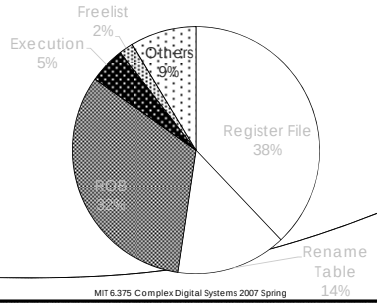
- The size of 8 was chosen.

Outline

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

Physical Numbers

- Area Analysis
  - Total Area 1.42 mm^2 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

- IPC Result

Outline

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

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

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.

Possible Follow-ups

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

Summary

- 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

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

Thank you