Superscalar* Club Meeting #3

*we really mean: superscalar speculative out-of-order

James C. Hoe
Department of ECE
Carnegie Mellon University
References Used

☆ Yeager, MIPS R10K Superscalar Microprocessor, 1996.

Today’s Goal:  
Really get under the R10K paper
This should mean something to you

fetch

decode

rename tbl

dispatch

 RS

schedule

issue

regfile (inorder & lookahead)

ALU1

ALU2

LD/ST

ROB

release dependents

complete

writeback
Last Time: Metaflow DRIS
Metaflow Datapath

- branch pred.
- i-cache
- issue
- DRIS (Renaming + Reservation Stations + Reorder Buff.)
  - Retire
- Register File
- Scheduler

Lookahead state
Inorder state
Not Unreasonable if . . .

• Separate RS and address queue from DRIS/ROB
  – RS sized to expose ILP
    Can’t be large: CAM-intensive, critical timing loop
  – ROB sized to cover long latencies (cache miss)
    Modern ROB size much larger than RS size

• Use a map-table for rename, keeping in mind
  – cheaper but not exactly cheap
  – still need to see how to rewind a map-table (see branch rewind stack today)

End up looking like Pentium-Pro
Onto MIPS R10K
MIPS R10000 circa 1996

- 4-way superscalar
- 5 execution pipelines
  - 2 integer, FP add, FP mult, ld/st
- Micro-dataflow instruction scheduling
  - 16 int +16 FP instruction window
- Register renaming + memory renaming
  - 64 int registers for inorder and lookahead
- Speculative OOO
  - 32 instructions in-flight; 4 unresolved branches
- Precise Exception
Superscalar, Speculative, Out-of-order

External interface
System interface (64 bits)
Secondary cache ctrl. (128 bits)

Data cache refill and write-back
6-bit physical register numbers
6-bit data paths

FP register file
(64×64)
5 read
3 write

FP adder
Align
Add/N
Pack

FP multiplier
Mult
Sum/N
Pack

Div
Sqrt

Load Store

Load Store Integer register file
(64×64)
7 read
3 write

Integer ALU 1

Integer ALU 2

5-pipelined execution units

Instruction issue

Instruction fetch

Instruction cache refill

Instr. pre-decode

Instr. cache (32 Kbytes)

Instr. decode

Branch

Register map tables

Free register lists

Busy-bit tables

Register renaming

Active list (32 entries)

Instruction cache refill and write-back

[Fig 2, Yeager 1996, IEEE Micro]
Pipeline Stages

- 6 independent pipelines
- Floating-point latency = 2
- Load/store latency = 2
- Integer latency = 1
- Execution unit pipelines (5)
- Dynamic issue
- Read operands from register file
- Instruction fetch and decode pipeline fills queues
- 4 instructions in parallel
- Up to 4 branch instructions are predicted
- Fetching continues speculatively until prediction verified
- Write results in register file

[Fig 2, Yeager 1996, IEEE Micro]
Let’s Talk About

• Register renaming

• Instruction Scheduling

• Speculative Execution Rewind
On-the-fly HW Register Renaming

- Maintain mapping from ISA reg. names to physical registers
- When decoding an instruction that updates ‘r<sub>x</sub>’:
  - allocate unused physical register t<sub>y</sub> to hold inst result
  - set new mapping from ‘r<sub>x</sub>’ to t<sub>y</sub>
  - younger instructions using ‘r<sub>x</sub>’ as input finds t<sub>y</sub>
- De-allocate a physical register for reuse when it is never needed again?

^^^^^when is this exactly?
Rename: add rd, rs, rt

Assume new is ID of current instruction

RN1[new] = rs; RN2[new] = rt;
Locked1[new] = false; Locked2[new] = false;
ID1[new] = not_valid; ID2[new] = not_valid;
forall valid id // over all active DRIS entries
if ((RD[id] == rs) && Latest[id])
    ID1[new] = id;
    Locked1[new] = !Executed[id];
forall valid id
if ((RD[id] == rt) && Latest[id])
    ID2[new] = id;
    Locked2[new] = !Executed[id];
Elements of Register Renaming

• A pool of extra registers
  – Use as temporary, single-assignment registers in lookahead state (*eliminates WAW and WAR*)
  – logically separate from inorder committed state

• Allocation and mapping mechanism
  – given a source architectural reg name, where is its current definition (*value, location, ready?*)
  – given a dest architectural reg name, where to find an available new rename register
  – when to reclaim a rename register?
  – how to recover after misprediction or exception
ROB Rename Register Management

Need to copy from lookahead to inorder on commit

from [Gonzalez, et al., 2010]
“Free List” Physical Register File Management

No need to copy from lookahead to inorder on commit
For Example Intel P3 vs P4

Figure 5: Pentium® III vs. Pentium® 4 processor register allocation
[The Microarchitecture of the Pentium 4 Processor, Intel Technology Journal, 2001]
PReg Life Cycle, R10K

- at any moment, each preg index (ptag) must be in exactly one entry of map-table, valid free, or valid last
- # preg (and freelist size) can be decoupled from ROB
- Steps 1 and 2 need to be reversible
Easier Than You Think

• # physical register ($\text{preg}$) = 
  # arch reg (=32) 
  + # ROB entries (=32 in R10K)

• At any moment
  – 32 $\text{preg}$ hold committed inorder state
  – rest associated 1 per ROB entry---either in-use (lookahead dest) or not in-use (freelist)

• Freelist management can ride ROB’s coattail
Management Algorithm

- At rename/dispatch
  - rename \texttt{rd} to corresponding ROB/freelist \texttt{preg}
  - save in ROB \texttt{rd}'s previous \texttt{preg} mapping (\textit{read from map-table before updating})
  - if no dest or \texttt{rd}=r0, save unused new \texttt{preg} as \texttt{last}
- At commit \textit{do nothing}
  - current dest \texttt{preg} ⇒ inorder
  - write \texttt{last} mapping into freelist (\textit{deallocated})
- On rewind? On exception?

[Internal Use Draft] Superscalar Club Meeting #3, Slide 21, James C. Hoe, CMU/ECE/CALCM, ©2023 [Do not redistribute.]
Register Map-table

- To rename `rs`, look up `ptag` and `busy`
- R10K map-table needs 4-way x (3 read + 1 write port)!!

*Also need redirect across same-cycle renamed instructions*

- On rewind, map-table restored by Branch Rewind Stack
- On exception, map-table restored sequentially from `last`
Let’s Talk About

- Register renaming
- Instruction Scheduling
- Speculative Execution Rewind
Dataflow Execution Ordering

• Maintain a buffer of many pending instructions, a.k.a. reservation stations (RSs)
  – wait for functional unit to be free
  – wait for register RAW hazards to resolve (i.e., required input operands to be produced)
• Issue instructions for execution in dataflow order
  – select instructions in RS whose operands are available
  – give preference to older instructions (heuristical)
• A completing instruction frees pending, RAW-dependent instructions to execute

Sounds like good plan but exactly how?
Micro-Dataflow Scheduling

- The scheduler dispatches according to
  - availability of pending instructions’ operands
  - availability of the functional units
  - chronological order of the instructions

  Is oldest-first the “best” strategy?

- Find instructions such that

\[
\text{valid}[id] \land \neg \text{Locked1}[id] \land \neg \text{Locked2}[id] \land \neg \text{Dispatched}[id] \land \neg \text{Executed}[id] \land \neg \text{notBusy}(\text{fxnUnit}[id])
\]

Think about the circuits & multiply for superscalar
R10K Integer Queue

- Like Tomasulo’s Reservation Stations but without operand value
  - operands represented by renamed ptag and busy status
  - an instruction issues when operands ready (either in regfile or can be forwarded in time)
- Keep in mind, busy is cleared when dependent-on instruction selected for issue not when it completes
Basic Integer Timing (Best Case)

- (stage 2) Request
- (stage 3) Issue
- (stage 4) Execute

1. Request
2. Issue
3. Operands
4. Execute

- **Combinationally**
  - I₁’s new data dep. info latched
  - I₁ requests issue
  - I₁ granted issue
  - I₁ fetches operands (RF or forwarding)
  - I₁ announces data dep. resolution (to I₂)

- Dataflow resolution
  - I₁’s result forwarded to I₂
  - I₁’s result written to RF
  - I₁’s result latched
  - I₂’s result latched
Integer Queue Entry

- ptag from LD
- ptag from ALU2

How many CAM ports do you count?

priority scheduler 1-per-cyc

request grant

x16

fetch operand (RF or forward)

issue/op, stage 3

execute, stage 4
Scheduling Loop Critical Path

How many CAM ports do you count?

priority scheduler
1-per-cyc

request
grant

execute, stage 4

issue/op, stage 3

fetch operand (RF or forward)

ptag from LD

ptag from ALU2


Op from ALU2

Op from LD

CarnegieMellon

[Internal Use Draft] Superscalar Club Meeting #3, Slide 29, James C. Hoe, CMU/ECE/CALCM, ©2023 [Do not redistribute.]
Forwarding

- Integer reg file
- Integer queue
- ptag

Stage 3:
- ALU1 result
- ALU2 result
- Load result

Stage 4:
- ALU

Flow of data from stage 3 to stage 4.
Load Data Path

[Fig 10, Yeager 1996, IEEE Micro]
Best Case Load Timing

Request

Issue

Operands

Addr Calc

D-cache/TLB

Execute

L₀ requests issue
L₀ granted issue if tag and data array free next cycle

L₀’s result forwarded to I₁

L₀’s address presented to cache & TLB

Dataflow resolution 1 cyc later

L₀’s result

hit data

L₀’s address presented to cache & TLB
Load Timing if Miss or Other Issues

- **Request**: 
  - L₀

- **Issue**: 
  - L₀
  - I₁

- **Operands**: 
  - L₀
  - L₀

- **Addr Calc**: 
  - D-cache/TLB

- **Execute**: 
  - L₀
  - I₁

- **Dependence resolution 1 cyc later**
- **Miss signal**
- **Canceled and lose 1 issue cyc since too late to find an alternate**
- **L₀’s address presented to cache & TLB**
- **L₀ requests issue**
- **L₀ granted issue if tag and data array free next cycle**
- **L₀ will rerun later**
Address Queue

- From IntQ
- From IntQ
- To IntQs

Stage 3: Issue

Stage 4: Addr calculation

Stage 5: Cache lookup

Stage 6: Writeback
“Tentative” Release and Cancellation

IntQ Dest

IntQ Dest

Load Dest (tentative)

OpA

set reset

Rdy

cancel

issue
decision

set 1-cyc cancellation window if released by load

set reset
tentative

cache miss
Let’s Talk About

• Register renaming

• Instruction Scheduling

• Speculative Execution Rewind
Control Flow Speculation

• Leading Speculation
  – follow through multiple branch predictions
  – track speculative instructions as lookahead
  – preserve µarch state at branch dispatch for rewind
Mis-speculation Recovery

• When a branch is evaluated, if prediction confirmed, nothing more to do (except to deallocate no-longer-needed recovery state)
• Else use recovery state (deallocate after use)
  – clear wrongpath instructions and their effects
  – restart down “correct” path
Rewinding Tomasulo with ROB

- Inorder RF state never needs undo’ing √
- Lookahead RF state tied to ROB √
- Restoring architectural state view (i.e. map-table)
  - at decode, record in ROB, the logical dest and the overwritten previous mapping
  - on rewind, walk-back ROB one entry at a time to restore register map-table

What happens if previous mapping is to an already retired ROB entry? How do we know that?

from [Gonzalez, et al., 2010]
Rewind cannot wait for the head/oldest

- Set tail ptr to after mispredicted branch to restart . . .
- What about Latest?

Branch Stack (more next wk)

from 0002
Restoring State on Rewind

- ROB held state—easy
  - rewind by decrementing tail pointer; head pointer never affected by rewind
  - R10K freelist synchronized with ROB
- Recover overwritten state—expensive
  - take full snapshot (e.g. map table) at branch time
  - constant time restore on mispredict
- Delete non-ROB wrongpath state—messy
  - locate anything younger than rewind point
  - many sites to check (e.g., issue queue, FU pipeline)
  - too expensive to decide by comparing tags
Branch Rewind Stack (BRS)

- Not a stack; not a monolithic structure
  - allocate a slot when a branch dispatch
  - deallocate when branch resolve (right or wrong)
  - deallocate when branch (on wrongpath) killed
- A BRS slot snapshots at branch dispatch
  - tail (but not head) pointer of ROB
  - head (but not tail) pointer of freelist (if decoupled)
  - complete map-table *(Map-table cannot be vanilla multiported SRAM)*

*R10K dispatch stops after 1st branch in a cycle*

- On misprediction, restore from snapshot
Rewinding Out-of-Order Entities

- A bitmask indicates currently allocated BRS slots
  - each set-bit corresponds to an unresolved branch
  - a speculative, out-of-order entity picks up bitmask
    value at time of its creation—*need to be removed if
    any of the indicated branches mispredicts*

- Examples of speculative out-of-order entities
  - instructions in RS or anywhere else not ROB
  - a BRS slot itself

- A resolved branch broadcasts its BRS position
  - ignored by older entities, bit not set in their mask
  - caught by younger entities, bit is set in their mask
Mis-speculation Recovery

- Allocate BRS entry on decoding a branch; **highlighted bit** indicate allocated to that branch
- Inst pickup current BRS mask of unresolved branch
  - non-zero mask means speculative
  - 1 bit indicates dependence on unresolved branch
- If --**1**-- branch mispredicts, abort insts w. mask **xx1x**
Mis-speculation Recovery

- Level 3 branch resolved
  - no longer occupy a stack entry
  - subsequent insts only depend on level 1 and 2
- If 1--- branch resolves to be correct
  - reset 1--- in all insts (including branches)
  - free corresponding BRS entry for reuse
Mis-speculation Recovery

- Level 1 branch resolved
  - no longer occupy a stack entry
  - following inst segment no longer speculative
- If -1-- branch mispredicts
  - abort all insts with mask x1xx
  - free corresponding BRS entry for reuse
Mis-speculation Recovery

- No unresolved branches
- No BRS in use
- No speculative instructions
Mis-speculation Recovery

- On next branch prediction, allocate any free BRS entry to continue speculatively.
Reset and Abort

- On misprediction, any out-of-order entity with branch mask intersecting with the abort mask is eliminated.
- On correct prediction, the corresponding bit (reset mask) is cleared in all branch masks in system.

*Either way, corresponding BRS slot freed for reuse*
What about Exception

• Different from branch misprediction rewind
  – could occur at any instruction (not just branches)
  – doesn’t happen very frequently
  – only handled as the oldest instruction in ROB

• Easy to clear younger instructions once exception is oldest in ROB:
  – rewind ROB tail pointer (and freelist head pointer)
  – zap all out-of-order structures and state

• No backup map-table at all exception points
  – R10K reads back sequentially from last of ROB
  – Intel P4 maintains a retirement map-table
Before Next Time

• Memory Dataflow
  – Section 6.4 of Gonzalez, et al.
  – “Address Queue”, R10K, p34.
• https://github.com/jhoecmu/ooo-beta
  – download, build, and do walk-thru in README.md
  – run in a debugger to step through a few instructions’ worth of operations
  – drill into a specific structure, e.g., map table