Lecture 7: Speculative Execution and Recovery using Reorder Buffer

Branch prediction and speculative execution, precise interrupt, reorder buffer, Tomasulo improvements
Control Dependencies

Every instruction is control dependent on some set of branches

```
if p1
  S1;
if p2
  S2;
```

*S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.*

*control dependencies must be preserved to preserve program order*
Performance Impact

If CPU stalls on branches, how much would CPI increase?

- Control dependence need not be preserved in the whole execution
  - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program

- Two properties critical to program correctness are data flow and exception behavior
Branch Prediction and Speculative Execution

- Speculation is to run instructions on prediction - predictions could be wrong.

- Branch prediction: crucial to performance, could be very accurate.

- Mis-prediction is less frequent event - but can we simply ignore?

Example:

```c
for (i=0; i<1000; i++)
    C[i] = A[i] + B[i];
```

- Branch prediction: predict the execution as accurate as possible (frequent cases)

- Speculative execution recovery: if prediction is wrong, roll the execution back.
Exception Behavior

Preserving exception behavior -- exceptions must be raised exactly as in sequential execution

- Same sequences
- No “extra” exceptions

Example:

DADDU R2, R3, R4
BEQZ R2, L1
LW R1, 0(R2)

L1:

Problem with moving LW before BEQZ?

Again, a dynamic execution must produces the same register/memory contents as a sequential execution, any time it is stopped
Precise Interrupts

- Tomasulo had:

  In-order issue, out-of-order execution, and out-of-order completion

- Need to “fix” the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.
Branch Prediction vs. Precise Interrupt

- Mis-prediction is “exception” on the branch inst

- Execution “branches out” on exceptions
  - Every instruction is “predicted” not to take the “branch” to interrupt handler

- Same technique for handling both issue: in-order completion or commit; change register/memory only in program order
  
  *How does it ensure the correctness?*
The Hardware: Reorder Buffer

- If inst write results in program order, reg/memory always get the correct values

- Reorder buffer (ROB) - reorder out-of-order inst to program order at the time of writing reg/memory (commit)

- If some inst goes wrong, handle it at the time of commit - just flush inst afterwards

- Inst cannot write reg/memory immediately after execution, so ROB also buffer the results

  *No such a place in Tomasulo original*
Reorder Buffer Details

- Holds branch valid and exception bits
  - Flush pipeline when any bit is set
  - How do the architectural states look like after the flushing?

- Holds dest, result and PC
  - Write results to dest at the time of commit
  - Which PC to hold?
  - A ready bit (not shown) indicates if the

- Supplies operands between execution complete and commit
ROB: Circular Buffer
Tag: ROB Index

Use ROB index as tag
  ■ Why not RS index any more?
  ■ Why is ROB index a valid choice?

◆ Register result status rename a register index to ROB index if the register is renamed
◆ Reservation stations now use ROB index for tracking dependence and for wakeup
◆ Again tag (now ROB index) and data are broadcast on CDB at writeback
◆ Inst may receive register values from (1) register, (2) data broadcasting, or (3) ROB
Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   - Condition: a free RS at the required FU
   - Actions: (1) decode the instruction; (2) allocate a RS and ROB entry; (3) do source register renaming; (4) do dest register renaming; (5) read register file; (6) dispatch the decoded and renamed instruction to the RS and ROB

2. Execution—operate on operands (EX)
   - Condition: At a given FU, at lease one instruction is ready
   - Action: select a ready instruction and send it to the FU

3. Write result—finish execution (WB)
   - Condition: At a given FU, some instruction finishes FU execution
   - Actions: (1) FU writes to CDB, broadcast to all RSs and to the ROB; (2) FU broadcast tag (ROB index) to all RS; (3) de-allocate the RS. Note: no register status update at this time
Speculative Tomasulo Algorithm

4. Commit—update register with reorder result
   - Condition: ROB is not empty and ROB head inst has finished execution
   - Actions if no mis-prediction/exception: (1) write result to register/memory, (2) update register status, (3) de-allocate the ROB entry
   - Actions if with mis-prediction/exception: flush the pipeline, e.g. (1) flush IFQ; (2) clear register status; (3) flush all RS and reset FU; (4) reset ROB
Speculative Execution Correctness

- $E(Sp, P)$ commits the same set of instructions as $E(S, P)$ executes.
- For any committed inst $i$ in $E(Sp, P)$, $i$ receives the outputs in $E(Sp,P)$ of its parents in $E(S,P)$.
- In $E(Sp, P)$ any register or memory word receives the output of a committed inst $j$, where $j$ is the last inst that writes to the register or memory word in $E(Sp, P)$.
Speculative Execution Correctness

For any committed inst \( i \) in \( E(Sp, P) \), \( i \) receives the outputs in \( E(Sp,P) \) of its parents in \( E(S,P) \)

Assume \( i \) has a source \( Rx \) produced by \( j \). Three possibilities at \( i \).rename:

1. \( Rx \) is not renamed
   \( \Rightarrow \) \( i \) receives \( j \)'s output from the register

2. \( Rx \) is renamed and \( j.WB \) has finished (or finishing)
   \( \Rightarrow \) \( i \) receives \( j \)'s output from ROB

3. \( Rx \) is renamed, and \( j.EXE \) has not finished
   \( \Rightarrow \) \( i \) will receive \( j \)'s value from CDB broadcasting

And \( i \)'s reading operands is not affected by later mis-speculated instructions
Code Example

Loop:

LW R2, 0(R1)
DADDIU R2, R2, #1
SW R2, 0(R1)
DADDIU R1, R1, #4
BNE R2, R3, Loop
LW R3, 0(R1)

... 

How would this code be executed? What if the BNE is incorrect predicted?
Tomasulo Summary

- Reservations stations: *implicit register renaming* to larger set of registers + buffering source operands
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards of Scoreboard
- Not limited to basic blocks when compared to static scheduling (integer units gets ahead, beyond branches)
- Today, helps cache misses as well
  - Don’t stall for L1 Data cache miss (insufficient ILP for L2 miss?)
  - Can support memory-level parallelism

- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation (discuss later)

- 360/91 descendants are Pentium III; PowerPC 604; MIPS R10000; HP-PA 8000; Alpha 21264
Tomasulo Complexity and Efficiency

Can dependent instructions be scheduled back-to-back?

Modern processors employ deep pipeline

=> Can the rename stage be finished in one fast cycle?
Review Tomasulo Inst Scheduling
Both in RS, no contention on CDB or FU

ADD \textbf{R2, R2, 45} \ # \ R2=>tag p, result = A  
SUB \textbf{R6, R2, R4} \ # \ R4 is ready, = B 

\textbf{Cycle 1:} \textbf{ADD} \ starts \ at \ FU, \ producing \ A  
\textbf{Cycle 2:} \textbf{ADD} \ broadcast \ p + A  
\textbf{SUB} \ matches \ on \ p \ and \ accepts \ A  
\textbf{Cycle 3:} \textbf{SUB} \ starts \ execution, \ FU \ calc \ A-B

\textbf{A is produced at cycle 1, but consumed at cycle 3 -- unavoidable?}
Review Data Forwarding

MIPS pipeline data forwarding:
FU/MEM => FU
Why not in Tomasulo?

REG/ROB

FU

ROB

bypass

But tag broadcasting has one cycle delay!!

When is it known that A will be ready?

Cycle 1: A is to be ready
Cycle 2: A and its tag are broadcast

If tag is broadcast one-cycle earlier ...

Cycle 2: forward A from FU output to FU input...
Revise Scheduling

RS1: ADD  R6, R2, R4
RS2: SUB  R10, R0, R6
RS3: ADD  R12, R10, R6

ADD(1) has been ready and selected
1. - ADD(1)'s tag is broadcast, and operands are sent to FU;
   - SUB is waken up and selected;
2. - SUB's tag is broadcast, operands are sent to FU;
   - forwarding logic replace 2nd FU operand with FU output;
   - ADD(2) is waken up and accepts FU output, and is selected
3. So on and so forth...

RS can be centralized or distributed

--- One cycle earlier

How to address CDB contention?
How to Handle Variable Latency?

One method: Use result shift register to track latency and control tag/data bus
Revised Pipeline Stages

- As efficient as MIPS pipeline (instruction throughput)
- With data forwarding and bypassing