Lecture 7: Speculative Execution and Recovery

Branch prediction and speculative execution, precise interrupt, reorder buffer

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

Control Dependence Ignored

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: cannot be avoided, could be very accurate
- Mis-prediction is less frequent event – but can we ignore?

Example:

```
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 look like a sequential execution, any time when 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:

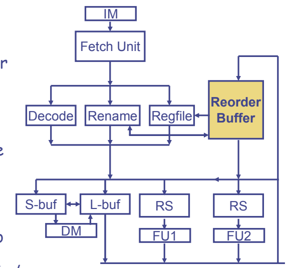
in-order completion or <u>commit</u>: change register/memory only in program order (sequential)

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

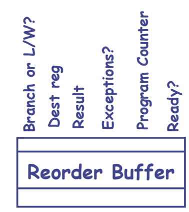


Four Steps of Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue
   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")
2. Execution—operate on operands (EX)
   When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")
3. Write result—finish execution (WB)
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
4. Commit—update register with reorder result
   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

Reorder Buffer Details

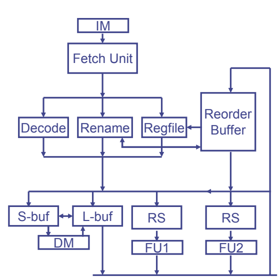
- Holds branch valid and exception bits
  - Flush pipeline when any bit is set
  - How do the architectural states look like after the flushing?
- Holds dest, result and PC
  - Write results to dest at the time of commit
  - Which PC to hold?
  - A ready bit (not shown) indicates if the
- Supplies operands between execution complete and commit



Speculative Execution Recovery

Flush the pipeline on mis-prediction

- MIPS 5-stage pipeline used flushing on taken branches
- Where is the flush signal from?
- When to flush?
- Which components are flushed?



Changes to Other Components

Use ROB index as tag

- Why not RS index any more?
- Why is ROB index a valid choice?

- Renaming table maps architecture registers 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 values from reg/mem, data broadcasting, or ROB

Code Example

```
Loop:    LD R2, 0(R1)
         DADDIU    R2, R2, #1
         SD R2, 0(R1)
         DADDIU R1, R1, #4
         BNE R2, R3, Loop
```

How would this code be executed?

Inst	Issue	Exec	Memory read	Write results	Commit
LD	1	2	3	4	5
…	…	…	…	…	…
…	…	…	…	…	…

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

Dynamic Scheduling: The Only Choice?

- Most high-performance processors today are dynamically scheduled superscalar processors
  - With deeper and n-way issue pipeline
- Other alternatives to exploit instruction-level parallelism
  - Statically scheduled superscalar
  - VLIW
- Mixed effort: EPIC – Explicit Parallel Instruction Computing
  - Example: Intel Itanium processors

Why is dynamic scheduling so popular today?

- Technology trends: increasing transistor budget, deeper pipeline, wide issue