Chapter 3 (CONT II)

Instructor: Josep Torrellas
CS433
Hardware-Based Speculation (Section 3.6)

• In multiple issue processors, stalls due to branches would be frequent:
  – You may need to execute 1 branch/cycle
  – Cannot start execution until branch is resolved
• Solution: Speculate the outcome of the branch and execute as if the guess was correct.
  – Note the extension over previous branch prediction: we now allow EX
  – This is called hardware speculation
  – Requires the ability to recover from mispredictions
• Hardware-based speculation combines:
  – Dynamic branch prediction
  – Speculation to allow EX
  – Aggressive dynamic scheduling across basic blocks (because now we aggressively execute past basic blocks)
Speculative Execution Based on Tomasulo

• Used in most processors: IBM, Intel, AMD…
• Key: separate the bypassing of results among instructions from full instruction completion
  – Allow a speculative instruction to pass its result to others
  – Not complete the instruction (not allow it to update any undoable state)
• When the instruction is no longer speculative: allow it to modify the reg file or memory → Instruction Commit Step
• Overall:
  – instructions execute ooo, but they commit in order
  – Prevent any irrevocable action until an instruction commits
Reorder Buffer (ROB)

- Buffer that holds the result of instructions that have completed but not committed
- Hardware buffer
- Also used to pass results among instructions
- Provides additional registers like the Reservation Stations in Tomasulo \( \rightarrow \) extend the register set
- Difference ROB-Tomasulo Res Stations:
  - Tom: Once the FU writes its result, subsequent instructions find the value in the register set
  - ROB: ROB supplies the results of completed instructions not yet committed
ROB

- Fields in an entry:
  - Instruction type: BR (no dest), SD (dest is mem), Other (dest is reg)
  - Destination field: destination register or destination mem address
  - Value: value until commit
  - Ready: has the instruction finished (and therefore the result is ready)

- We place the ROB instead of the Store Buffs
Figure 3.11
Functions

- **Res Stations:**
  - Place to buffer operations and operands from the time I issues until it can execute
  - RS track the ROB assigned for an instruction

- **ROB:**
  - Does the renaming
  - Every instruction has a position in the ROB from when it issues until it commits
  - We tag a result using the ROB entry number rather than the RS number
Overall Instruction Execution

• ISSUE or DISPATCH:
  – Get an instruction from instruction queue
  – Issue it if there is an empty RS and an empty ROB entry
  – Send ops to the RS if they are available in any reg or ROB entry
  – Send the number of the ROB entry to the RS, so that it can be used to tag the result of the execution when it is dumped in the CDB
  – If no RS or no ROB entry: stall instruction and successors

• EXECUTE:
  – Monitor for any of your inputs to be available in the CDB (this enforces RAW). If so, read it in the RS
  – When both ops ready and the FU free, execute /*note: some sources call this step “issue”. We will not use this word here */
  – May take several cycles
  – Loads require 2 steps (address generation and memory access); stores need only 1 step (address generation)
Overall Instruction Execution

• WRITE RESULT:
  – When EX complete, dump result on the bus, together with the ROB entry number. Mark RS as available.
  – The data is sent to other RS waiting for it, and to the correct ROB entry.
  – We assume that for Stores, it is at this time that the data is generated. So the value to be stored is also moved to the ROB entry.

• COMMIT or GRADUATION:
  – When an instruction reaches the head of the ROB and its result is present in the entry → update the Reg file and remove the ROB entry.
  – If the instruction is a Store: Do same except that memory is updated.
  – If the instruction is a branch with incorrect prediction: flush the ROB and restart execution at the correct successor.
  – If the instruction is a branch with correct prediction: as usual.
Example

• Example page 187 and Figure 3.12 (when MUL.D in W stage)

L.D F6,32(R2)
L.D F2,44(R3)
MUL.D F0,F2,F4
SUB.D F8,F2,F6
DIV.D F10,F0,F6
ADD.D F6,F8,F2

• No need to worry about timing

• Compare to non-speculative Tomasulo (fig 3.7): there, instructions complete ooo; here, they have to commit in order
Continuation of the Example

Major difference: with the ROB, dynamic execution can maintain a precise exception model. Example:

• If MULT suffers exception
• Wait until it reaches the head of the ROB
• service exception
• flush all pending instructions
Example of Branch Misprediction

- Example page 189 and Figure 3.13
- No need to worry about timing
- After a branch is mispredicted, recover by
  - clear the ROB for all instructions that follow the branch
  - Let branch and the previous instructions to commit
  - fetch at the correct branch successor
- Exceptions
  - Record the exception in the ROB entry
  - If the excepting instruction is in the mispredicted path, simply squash the instruction and flush the exception.
  - When the excepting instruction reaches the head ROB: process it
Handling Stores

• Tomasulo’s algorithm: stores update caches/memory as soon as they complete the WB

• Speculative processor: stores only update caches/memory when they reach the head of the ROB
  – Therefore: memory not updated by a speculative instruction

• Note: Stores in speculative processor can go through the WRITE RESULTS stage even if the value to be stored is not ready; the value is only needed to COMMIT.
  – When the value is ready, it is stored in the ROB.
Dependences Through Memory

- **WAW**: Eliminated because the updating of memory occurs in order, when the WR is at the head of ROB
  - Therefore, no earlier LD or ST can be pending

- **WAR**: same as WAW

- **RAW**: Maintained with two restrictions:
  - LD cannot be sent to memory if there is a previous ST in the ROB with the same address
  - LD computes its effective address in order relative to all earlier ST
  - Note: some speculative machines actually allow the bypass of a value from a previous ST to a successor LD without the LD having to go to memory
Multiple Issue Processors

- If all techniques described are successful → CPI = 1
- To reduce CPI below 1 ⇒ issue multiple inst per cycle
  - superscalar processors (a)
  - VLIW or EPIC processors (b)

(a) : issue varying # of inst/clock
  may be statically scheduled (in-order execution)
  dynamically scheduled based on Tomasulo (ooo execution)

(b): issue fixed #instr/clock (usually as a large instr)
  statically scheduled

See Fig 3.15
Superscalars

• For example, in a cycle, issue 0-8 instructions
  – usually instructions are indep
  – have to satisfy some constraints e.g. no more 1 mem ref
  – if an instruction does not meet this issue criteria, do not issue it; do not issue following instr.

• VLIW: Compiler has responsibility of creating the package of instructions to simultaneously issue.

If it cannot find enough, put NOOPS

eg. Superscalar: issue an integer + fpfp instruction per cycle
Multiple Issue with Dynamic Scheduling and Speculation

- We use Tomasulo
- For example: issue 2 instructions per cycle. These instructions may even have dependences with each other
- In general, for n-issue width: done with a combination of two approaches:
  - Run the issue step at higher speed, so that e.g. two instructions can be issued in 1 cycle
  - Build wider logic, so that multiple instructions can be issued at the same time and all dependences are checked.
Example 2-issue

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

• Assume separate FUs for: effective address calc, ALU ops, branch condition eval
• Assume 2 inst of any type can commit/cycle
Example 2-issue

• Figure 3.19: no speculation
  – note: here we assume that LD/ST cannot complete effective address calculation until previous branch resolved. Could save 1 cycle by allowing address calculation but not mem access

• Figure 3.20: speculation (up to 2 inst can commit/cycle)
  – since branch is key performance limitation, speculation helps!
Considerations for Spec Machines

1. ROB vs Register Renaming:
   • Using ROB
     – architectural registers contained in combination of register set, reservation stations, and ROB
     – if we do not issue instr for a while: as instructions commit, reg values will appear in the register file
   • Explicit set of physical regs + register renaming:
     – Some ROB entries did not need destination register
     – Physical regs hold both the architecturally visible regs and temp values
     – Extended regs play role of both res stations + ROB
     – WAW and WAR hazards eliminated by renaming the destination register
Considerations for Spec Machines

2. How much to speculate:
   - Adv of speculation: execute past branches
   - Dsv: may execute a lot of useless instructions
   - Consequently: execute under speculative mode only low-cost events:
     - yes: misses in L1
     - no: TLB misses. In this case, wait until the instruction causing the event becomes non-speculative
Considerations for Spec Machines

3. Speculating through multiple branches:
   • Some programs have a high frequency of branches or branch clustering --> speculate across multiple branches
   • Complicates the process of speculation recovery
   • It is hard to speculate on more than one branch per cycle