Arch: ILP

Alex Chi

Update: April 23, 2020

Contents

1 Approaches

- pipelining become universal, we may overlap execution of instructions
- software-based: compiler, static scheduling
- hardware-based: dynamic scheduling

2 What is ILP?

- basic block: a straight line code sequence with no branches
- parallelism with a basic block is limited
- must optimize across branches

2.1 Dependencies

- a=1, b=a (a depends on a)

2.2 Hazard

- pipeline has to be stopped to resolve hazards
- dependencies are cause of hazards

3 Dependencies

- data dependencies: true data dependencies, data flow between instructions
  - pipeline organization determines if a dependence results in an actual hazard
  - possibility of a hazard
  - defines order of calculation
  - upper bound on exploitable ILP

- name dependencies: two instruction use the same register
  - not true data dependency, but is a problem when reordering
  - anti-dependency: j writes to i reads (WAR)
  - output dependency: i and j write to same register (WAW)
  - use renaming techniques (e.g. Tomasulo)
- control dependencies
  - instruction control dependent on branch cannot be moved before the branch
  - instruction not control dependent cannot be moved after the branch

4 Data Hazards

- RAW
- WAW
- WAR

5 Basic Compiler Techniques

- energy efficient
- compiler schedules instructions on amount of ILP in program and latency in pipeline
- techniques: reorder, loop unrolling
- unroll by factor of 4 (reduce stalling)

6 Control Speculation

- execute instruction beyond branch, and flush them if branch is not taken
- mis-speculated
  - recovery
  - squash
- branch prediction
  - dynamic: gather runtime information and make choice
  - static: always make same choice on one instruction
- predict what?
  - single direction for unconditional jumps, calls, returns
  - binary direction for conditional branches
- target

    - uni-directional
    - taken / not taken
    - function pointer, indirect jump
- why branch is predictable?
    - for loops, for most of the time will be taken

6.1 FSM

- node is last prediction + state
- edge is actual branch

6.2 Dynamic Branch Predictor

- (simplest predictor) latest outcome in lower bit
    - issue: hash collision
    - 2-state FSM

6.3 2-bit saturating up/down counter

- strongly/weakly | taken/not taken

7 Overcome Data Hazards with Dynamic Scheduling

- ID divided into two parts
    - check for structural hazard
    - waiting for absence of data hazard
- in-order issue, out-of-order execution

7.1 Issues with OoO Execution

- RAW, WAR, WAW execution

7.2 Dynamic Scheduling

- pros

    - compiler doesn't need to know micro-architecture
    - handle cases where dependencies are unknown
- cons
    - increase in hardware complexity
    - complicates exceptions

7.3 Techniques

- scoreboard approach
- Tomasulo's approach

7.4 Scoreboard: a book-keeping technique

- ID stage: issue stage + read operands
- execute when no dependence and no hazard
- out-of-order commit
    - imprecise interrupt / exception
    - no forwarding (for now)
- key question: OoO completion for WAR, WAW?
    - WAR: stall write until registers have been read
    - WAW: stall issue until completion
    - multiple instruction running, book-keep dependencies

7.4.1 Stage

- decode, check structural hazard (ID1)
- read operands (wait data hazard, RAW) (ID2)
- execution: notify scoreboard
- write result (stall until no WAR)

7.4.2 Three Parts

- instruction status: tracking stage of instruction
- functional unit status: 9 fields
    - Busy, Op, Fi, Fj, Fk, Qj, Qk, Rj, Rk
- register result status: which FP write to register