Lecture 13

Computer Architecture
(Spring 2020)

Instruction-Level Parallelism

Dr. Duo Liu (刘铎)
Office: Main Building 0626
Email: liuduo@cqu.edu.cn

Instruction-Level Parallelism

- ILP: overlap the execution of instructions
  - partially (through pipelining)
  - completely (through issuing on multiple functional units)
- Exploiting ILP
  - dynamically and hardware-intensive
    - mostly desktop and server markets (Pentium, MIPS, Alpha…)
  - static and software-intensive
    - embedded system markets (but also, Itanium...)
  - in practice these techniques are often combined
- Exploiting ILP and basic blocks
  - in a typical MIPS program, the average dynamic branch frequency is between 15% and 25%→average length of a basic block is between 4 and 7 instructions
    - necessary to exploit ILP across multiple basic blocks

Reducing CPI (or Increasing IPC)

$$CPI = CPI_{ideal} + stalls_{structural} + stalls_{data\ Hazard} + stalls_{control}$$

technique	reduces
forwarding/ bypassing	potential data-hazard stalls
delayed branches	control-hazard stalls
basic dynamic scheduling (scoreboarding)	data-hazard stalls from true dependecies
dynamic scheduling with register renaming	data-hazard, anti-dep. & output dep. stalls
dynamic branch prediction	control stalls
issuing multiple instruction per clock cycle	ideal CPI
speculation	data-hazard and control-hazard stalls
dynamic memory disambiguation	data-hazard stalls with memory
loop unrolling	control hazard stalls
basic compiler pipeline scheduling	data-hazard stalls
compiler dependency analysis	ideal CPI, data-hazard stalls
software pipelining & trace scheduling	ideal CPI, data-hazard stalls
compiler speculation	ideal CPI, data-hazard stalls

Dependences vs. Hazards

- If two instructions depend on each other
  - they must be executed in order
  - at most, they can be partially overlapped
- Dependences are a property of programs
  - a data dependency in the instruction sequence reflects a data dependency in the original source code
- Based on the given pipeline implementation a data dependency may result in an actual hazard being detected and a possible stall (which reduces or eliminates the instruction overlap)

e.g. moving the branch test for the MIPS from the EX to ID does reduce the branch delay to one, but it may cause a stall in presence of a data dependency

```
DADDIU  R1,  R8,  -8
BNE  R1,  R2,  Loop
```

Dependences vs. Hazards (cont.)

- Dependency ≡ hazard potential
  - occurrence of actual hazard and length of any stall is a property of the pipeline implementation
- Dependences determine the order in which results must be computed
- Dependences set an upper bound on the amount of instruction-level parallelism that can be exploited

- Strategies to overcome dependences
  - maintain them, but avoid hazards
  - eliminate them by transforming the code

Dynamic HW Scheduling vs. Static Pipeline Scheduling

- Dynamic Scheduling (HW)
  - tries to avoid stalls when dependences (which could generate hazards) are present
  - does not change the data flow
- Static Scheduling (SW)
  - the compiler tries to minimize stalls by separating dependent instructions so that they will not generate hazards
- The two techniques can be combined as statically scheduled code can run on a processor with a dynamically scheduled pipeline

Dynamic HW Scheduling: Motivations

```
DIV.D  F0, F2, F4
ADD.D  F6, F0, F8
SUB.D  F8, F8, F14
```

- SUB.D does not depend on the previous instructions but it can not execute because ADD.D is stalling due to RAW hazard

- With dynamic scheduling, the HW rearranges the instruction execution order to reduce the stalls while maintaining data flow and exception behavior
  - it simplifies compiling
    - code compiled for a given pipeline can be run efficiently on another
  - it handles cases where dependences are unknown at compile time (due to memory references)
  - it is the basis for hardware speculative execution

Out-of-Order Execution

- In-Order Execution: if an instruction is stalled in the pipeline, no later instructions can proceed.
  - Structural and Data hazards are checked at ID.
  - Even if there are later instructions that are independent and would not stall.
- Out-of-Order execution: Decode and Issue instructions in order, but execute the instructions as soon as their data operands are available.
  - It may result in out-of-order completion.
- To implement out-of-order execution, we must split the ID into two stages:
  - Issue—Decode instructions, check for structural hazards.
  - Read operands—Wait until no data hazards, then read operands
- In a dynamically scheduled pipeline:
  - All instructions pass through the issue stage in order (in-order issue);
  - However, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out-of-order.

Name Dependences and the New Hazards due to Out-of-Order Execution

- True data dependency: an instruction produces a value for a following instruction
  - may lead to a RAW hazard
- Name dependence: no real value must be transmitted between two instructions, but they both use the same register or memory location
  - anti-dependence
    - may lead to a WAR hazard
  - output dependence
    - may lead to a WAW hazard

Dynamic Scheduling & Data Hazards: The Scoreboard Approach

```
DIV.D   F0,  F2,  F4
ADD.D   F6,  F0,  F8
SUB.D   F8,  F8,  F14
MUL.D   F6,  F10, F9
```

- anti-dependency between ADD.D and SUB.D (may lead to WAR hazard)
- output dependency between MUL.D and ADD.D (may lead to WAW hazard)

- Goal: to maintain CPI≈1 with out-of-order execution by
  - issuing an instruction ASAP
  - taking care of issuing/execution, including all hazard detections
    1. checking for structural hazards
    2. waiting for absence of data hazards
- Scoreboard was introduced in 1963 with CDC 6600
  - 7 units for integer operations
  - 5 memory reference units
  - 4 FP units