CMSC 411
Computer Systems Architecture
Lecture 7
Basic Pipelining (cont.) &
Instruction Level Parallelism

Alan Sussman
als@cs.umd.edu

Administrivia

- Questions about HW #1?
- HW #2, on pipelining, due next Thursday, Feb. 26
- Start reading Chapter 2 of H&P

A case study: MIPS R4000

- MIPS64 architecture, with deeper 8 stage pipeline
  - to get higher clock rates
  - extra stages come from memory accesses
  - techniques called superpipelining

MIPS R4000 pipeline stages

- IF – 1st half instruction fetch
  - PC selection and start instruction cache access

- IS – 2nd half instruction fetch
  - complete instruction cache access

- RF – instruction decode, register fetch, hazard checking, instruction cache hit detection

- EX – execution
  - includes effective address computation, ALU operation, branch target computation and condition evaluation
**MIPS R4000 pipeline (cont.)**

- **DF** – 1st half data fetch
  - 1st half of data cache access

- **DS** – 2nd half data fetch
  - complete data cache access

- **TC** – tag check
  - determine whether data cache access *hit*

- **WB** – write back for loads and ALU operations

---

**A 2 cycle load delay**

---

**Forwarding**

- Deeper pipeline increases number of levels of forwarding for ALU operations
  - 4 possible sources for an ALU bypass – EX/DF, DF/DS, DS/TC, TC/WB
**Floating point pipeline**

- 3 functional units
  - divider, multiplier, adder
- Double precision FP ops take from 2 (negate) up to 112 cycles (square root)
- Effectively 8 stages, combined in different orders for various FP operations
  - one copy of each stage, and some instructions use a stage zero or more times, and in different orders
- Overall, rather complicated ...
  - see H&P for more details

**R4000 pipeline performance**

- 4 major causes of pipeline stalls
  - load stalls – from using load result 1 or 2 cycles after load
  - branch stalls – 2 cycles on every taken branch, or empty branch delay slot
  - FP result stalls – RAW hazards for an FP operand
  - FP structural stalls – from conflicts for functional units in FP pipeline

**SPEC92 benchmarks**

Assuming a perfect cache – 5 integer and five FP programs

**Dynamically scheduled pipelines**

- We’ll cover this, and the scoreboard technique, in a bit
  - need some general background first
Pitfalls

• Unexpected hazards do occur ...
  – for example, when a branch is taken before a previous instruction finishes

• Extensive pipelining can slow a machine down, or lead to worse cost-performance
  – more complex hardware can cause a longer clock cycle, killing the benefits of more pipelining

Pitfalls (cont.)

• A poor compiler can make a good machine look bad
  – compiler writers need to understand the architecture in order to
    » optimize efficiently and
    » avoid hazards
  – better to eliminate useless instructions, than make them run faster

Outline

• ILP
• Compiler techniques to increase ILP
• Loop Unrolling
• Static Branch Prediction
• Dynamic Branch Prediction
• Overcoming Data Hazards with Dynamic Scheduling
• (Start) Tomasulo Algorithm
• Conclusion

INSTRUCTION-LEVEL PARALLELISM
**Recall from Pipelining**

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - **Ideal pipeline CPI**: measure of the maximum performance attainable by the implementation
  - **Structural hazards**: HW cannot support this combination of instructions
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
  - **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

**Instruction Level Parallelism**

- Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance
- 2 approaches to exploit ILP:
  1. Rely on hardware to help discover and exploit the parallelism dynamically (e.g., Pentium 4, AMD Opteron, IBM Power), and
  2. Rely on software technology to find parallelism, statically at compile-time (e.g., Itanium 2/IA-64)

**Instruction-Level Parallelism (ILP)**

- Basic Block (BB) ILP is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25%
    - => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- Need ILP **across** multiple basic blocks
- Simplest: **loop-level parallelism** to exploit parallelism among iterations of a loop. E.g.,

  ```
  for (i=1; i<=1000; i=i+1)
  x[i] = x[i] + y[i];
  ```

**Loop-Level Parallelism**

- Exploit loop-level parallelism by “unrolling loop” either by
  1. **dynamic** via branch prediction or
  2. **static** via loop unrolling by compiler
  (Another way is vectors, to be covered later)
- Determining dependences **critical**
- If 2 instructions are
  - **parallel**, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
  - **dependent**, they are not parallel and must be executed in order, although they may often be partially overlapped
Data Dependence and Hazards

- Instr$_j$ is **data dependent** (aka **true dependence**) on Instr$_i$
  1. Instr$_j$ tries to read operand before Instr$_i$ writes it
    
    ![Diagram showing I: add r1, r2, r3 and J: sub r4, r1, r3]
  2. or Instr$_j$ is data dependent on Instr$_k$ which is dependent on Instr$_i$
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence ⇒ data dependence in source code
  ⇒ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, that’s a **Read After Write (RAW) hazard**

ILP and Data Dependencies, Hazards

- HW/SW must preserve **illusion** of program order:
  order instructions would execute in if executed sequentially as determined by original source program
  - dependences are a property of programs
- Presence of dependence indicates **potential** for a hazard, but
  - actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies
  1) indicates the possibility of a hazard
  2) determines order in which results must be calculated
  3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Name Dependence #1: Anti-dependence

- **Name dependence:** when 2 instructions use same register or memory location, called a **name**, but no flow of data between the instructions associated with that name; 2 versions of name dependence
  - Instr$_j$ writes operand **before** Instr$_i$ reads it
    
    ![Diagram showing I: sub r1, r4, r3 and J: add r1, r2, r3]
  - Instr$_j$ writes operand **before** Instr$_i$ writes it
    
    ![Diagram showing I: sub r1, r4, r3 and J: add r1, r2, r3, K: mul r6, r1, r7]
  - Called an “anti-dependence” by compiler writers.
  - This results from reuse of the name “r1”
  - If anti-dependence caused a hazard in the pipeline, that’s a **Write After Read (WAR) hazard**

Name Dependence #2: Output dependence

- Instr$_j$ writes operand **before** Instr$_i$ writes it.
  
  ![Diagram showing I: sub r1, r4, r3 and J: add r1, r2, r3, K: mul r6, r1, r7]
- Called an “output dependence” by compiler writers
  This also results from the reuse of name “r1”
- If anti-dependence caused a hazard in the pipeline, that’s a **Write After Write (WAW) hazard**
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is **changed** so instructions do not conflict
  - Register renaming resolves name dependence for regs
  - Either by compiler or by HW