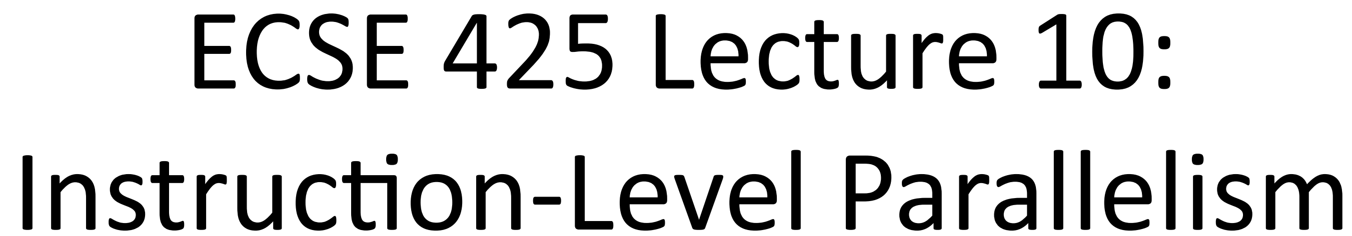


H&P Chapter 2

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer

Textbook figures © 2007 Elsevier Science

Last Time

- Exceptions
- Multi-cycle operations

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Today

- Chapter 2.1
- Introduction of ILP Techniques
- Dependencies and Hazards
  - True (data) dependence
  - Name dependence
  - Control dependence
- Carefully Ignoring Control Dependence

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Instruction-Level Parallelism

- Pipelining overlaps the execution of instructions
  - This overlap is *Instruction-Level Parallelism* (ILP)!
- We'll consider techniques to increase ILP
  - What limits ILP; how much we can expect to extract
  - How to best exploit the available ILP
- Two main techniques
  - Hardware (market winner: Intel Pentium series)
  - Software (special niche markets, Intel Itanium, DSPs)

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Pipeline CPI

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

- Ideal pipeline CPI
  - Maximum performance of the implementation
- Structural hazards
  - HW cannot support this combination of instructions
- Data hazards
  - Instruction consumes a result not yet produced
- Control hazards
  - Caused by time req'd for branch and jump resolution

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

ILP within Basic Blocks

- 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
- Instructions in BB likely to depend on each other
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks
- *What are the implications of this?*

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Safely Maximizing ILP

- *Program order*
  - Order of executed instructions would execute in, if
    - Executed sequentially, and
    - One at a time, as
    - Determined by original source program
- *HW/SW goal*
  - Exploit parallelism by preserving program order, *but*
  - Only do so where it affects the result of the program

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Major ILP Techniques

Technique	Reduces
Forwarding	Potential data hazard stalls
Delayed branches and simple branch scheduling	Control hazard stalls
Dynamic scheduling	Data hazard stalls
Branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data and control stalls
Dynamic memory disambiguation	Data hazard stalls involving memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis and software pipelining	Ideal CPI and data hazard stalls

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Loop-Level Parallelism (LLP)

- Exploit parallelism among iterations of a loop

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

- Vector execution is one way
  - Graphics, DSP, media apps.
  - Execute the same instructions on multiple data simultaneously
- If not vector, then either
  - dynamic exploitation via branch prediction or
  - static exploitation via loop unrolling

Turn LLP into ILP

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Parallel and Dependent Instructions

- Instructions are parallel if they can execute simultaneously, regardless of pipeline depth
- Dependent instructions
  - Are not parallel
  - Must be executed in order
  - But may still be partially overlapped
- There are three types of dependence
  - Data dependence (true data dependence)
  - Name dependence
  - Control dependence

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

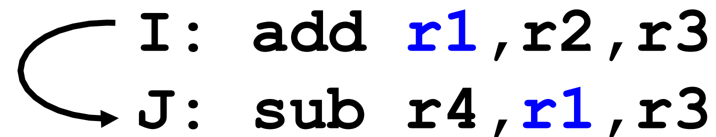
Dependence and Hazards

- Dependencies are a property of programs
- Dependency ⇒ potential for a hazard
  - Actual hazard, length of stall, are properties of pipeline organization
- Data dependencies
  - Indicate the possibility of a hazard
  - Determine the order for calculating results
  - Set an upper bound on available parallelism

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Data Dependence

- Instr *J* is data dependent on instr *I* if
  - *J* tries to read an operand before *I* writes it, or

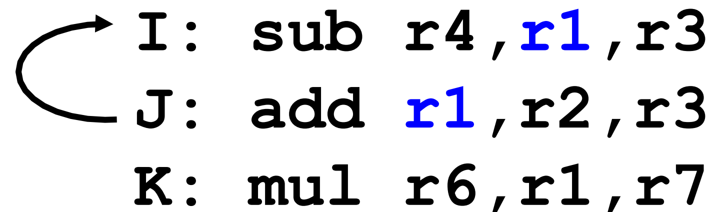


  - *J* is data dependent on instr *K* which is dependent on *I*
- "True Dependence" (compiler term)
  - Can cause Read After Write (RAW) hazards

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Name Dependence #1: Anti-dependence

- *Name dependence*
  - Two instructions use same register or memory location (name)
  - No actual flow of data between the instructions
- *Anti-dependence*
  - *J* writes an operand before *I* reads it



- Can cause Write After Read (WAR) hazards

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Name Dependence #2: Output dependence

- *J* writes an operand before *I* writes it

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Can cause Write After Write (WAW) hazards
- In the case of naming dependencies: change the name, remove the dependence!
  - *Register renaming* for register naming dependencies
  - Compiler (static) or by HW (dynamic)
- Detecting naming dependencies is harder for memory addresses

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Control Dependence

- Every instruction (except in the very first basic block) is control dependent on some set of branches
- In general, these control dependencies must be preserved to preserve program order

```
if p1 {
   S1;
}
if p2 {
   S2;
}
```

  - S1 is control dependent on p1
  - S2 is control dependent on p2 but not on p1

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Control Dependence Ignored

- Control dependence need not be preserved
  - as long as program correctness is preserved
- E.g., executing instructions
  - that should not be executed, or
  - in and order that differs from program order, and
  - thereby violating control dependencies
- Two properties critical to program correctness
  - Exception behavior
  - Data flow

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Preserving Exception Behavior

- Any changes in instruction execution order must not change how exceptions are raised in program
  - Alternatively, no new exceptions (more relaxed)
- Example:

```
        DADDU     R2,R3,R4
        BEQZ      R2,L1
        LW        R1,0(R2)
L1:
```

- Can we move LW before BEQZ?
  - No data dependence, only control dependence
  - Possibility of memory protection exception in the LW
    - handled by speculation (to come later)

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Preserving Data Flow

- *Data flow*: **movement of values** among instructions that produce results and those that consume them
- Involves both control and data dependences
  - branches make flow dynamic,
  - need to determine which instruction supplies data
- Example 1:

```
      DADDU    R1,R2,R3
      BEQZ     R4,L
      DSUBU    R1,R5,R6
L:    …
      OR       R7,R1,R8   ⟵ cannot move to before branch
```

- Does OR depend on DADDU or DSUBU? Both!
  - Determined using *data flow* analysis (compiler technique)
  - Preserving data dependence alone is not enough, in this case
  - Must preserve data flow (both data and control dependence)!

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Safely Violating Control Dependencies

- Sometimes, control dependencies can be violated without affecting data flow
- Example 2:

```
      DADDU   R1,R2,R3
      BEQZ    R12,skip
      DSUBU   R4,R5,R6
      DADDU   R5,R4,R9
skip: OR      R7,R8,R9
```

- If R4 is unused after skip, we can move DSUBU to before BEQZ
  - This is called (software) speculation, and is dependent on
  - *liveness* analysis (compiler technique)

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Summary

- ILP is small within basic blocks
  - We need techniques that safely expose ILP across BBs
- Dependence and Hazards
  - Data dependence: true dependence!
  - Name dependence: can be removed
  - Control dependence: can (sometimes) be ignored
- Change code, but preserve program correctness
  - Don't need to preserve program order, just
  - Exception behavior, and
  - Data flow

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science

Next Time

- Basic Compiler Techniques for Exposing ILP
  - Chapter 2.2
- On Friday, Branch prediction
  - Chapter 2.3

© 2011 Patterson, Gross, Hayward, Arbel, Vu, Meyer; © 2007 Elsevier Science