Computer Architecture

A Quantitative Approach, Fifth Edition

Chapter 3

Instruction-Level Parallelism and Its Exploitation

Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”
- Beyond this, there are two main approaches:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMP processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to minimize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

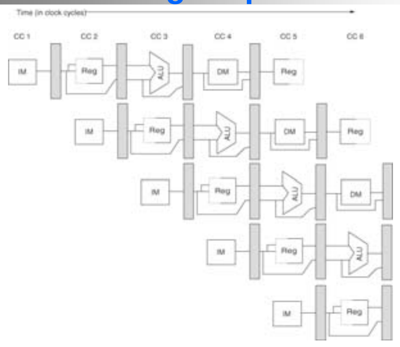
Data Dependence

- Loop-Level Parallelism
  - Unroll loop statically or dynamically
  - Use SIMD (vector processors and GPUs)
- Challenges:
  - Data dependency
    - Instruction *j* is data dependent on instruction *i* if
      - Instruction *i* produces a result that may be used by instruction *j*
      - Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall
- Data dependence conveys:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect

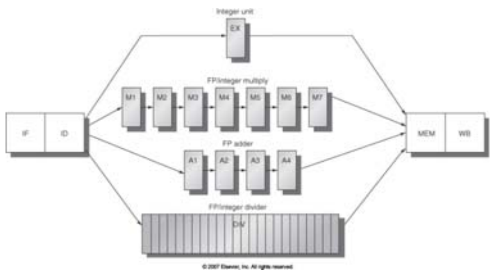
MIPS 5-Stage Pipeline



Hazards

- Structural hazards
- Data hazards
- Control hazards

Multiple Function Units



Data Dependence

- Loop: L.D F0,0(R1)
- ADD.D F4,F0,F2
- S.D F4,0(R1)
- DADDUI R1,R1,#-8
- BNE R1,R2,Loop

Name Dependence

- Two instructions use the same name but no flow of information
  - Not a true data dependence, *but is a problem when reordering instructions*
  - *Antidependence*: instruction j writes a register or memory location that instruction i reads
    - Initial ordering (i before j) must be preserved
  - *Output dependence*: instruction i and instruction j write the same register or memory location
    - Ordering must be preserved
- To resolve, use renaming techniques

Other Factors

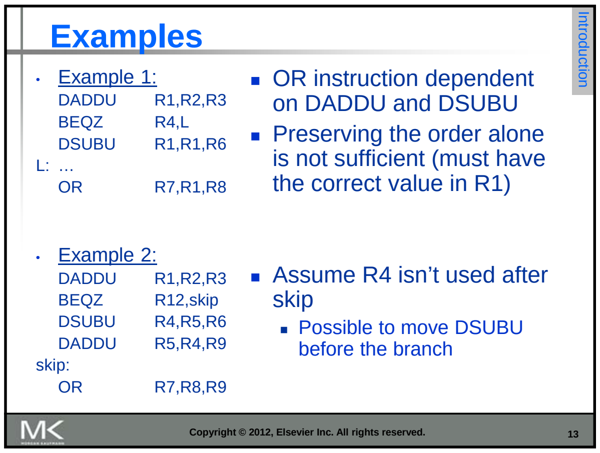
- Data Hazards
  - Read after write (RAW)
  - Write after write (WAW)
  - Write after read (WAR)
- Control Dependence
  - Ordering of instruction i with respect to a branch instruction
    - Instruction control dependent on a branch cannot be moved before the branch so that its execution is no longer controller by the branch
    - An instruction not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

Control Dependence

- Must preserve exception behavior.
- We should not change the exception behavior of the program.
- We often relax this to "reordering of instruction must not raise new exceptions"

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

- No data dependence prevents us from exchanging BEQZ and LW, but might result in memory protection exception



Examples

- Example 1:

```
    DADDU    R1,R2,R3
    BEQZ     R4,L
    DSUBU    R1,R1,R6
L:  …
    OR       R7,R1,R8
```

- OR instruction dependent on DADDU and DSUBU
- Preserving the order alone is not sufficient (must have the correct value in R1)

- Example 2:

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

- Assume R4 isn't used after skip
  - Possible to move DSUBU before the branch