COMP4211 (Seminar) Intro to Instruction-Level Parallelism

05S1 Week 02

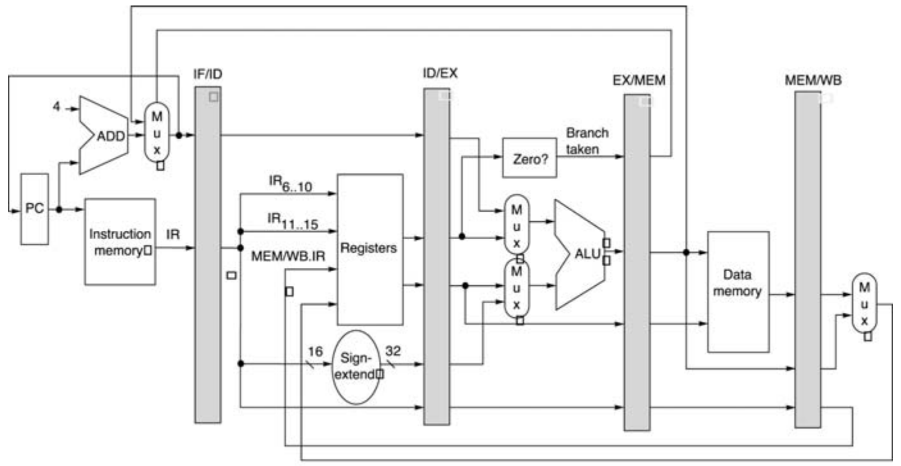
Oliver Diessel

Overview

- **Review pipelining from COMP3211**
- **Look at integrating FP hardware into the pipeline**
  - **Example: MIPS R4000**
- **Goal: increasing exploitation of ILP**
- **A little more on hazards…**

- **Refs:**
  - **Hennessy & Patterson, Appendix A; Chapter 3**

Five stage statically scheduled pipeline

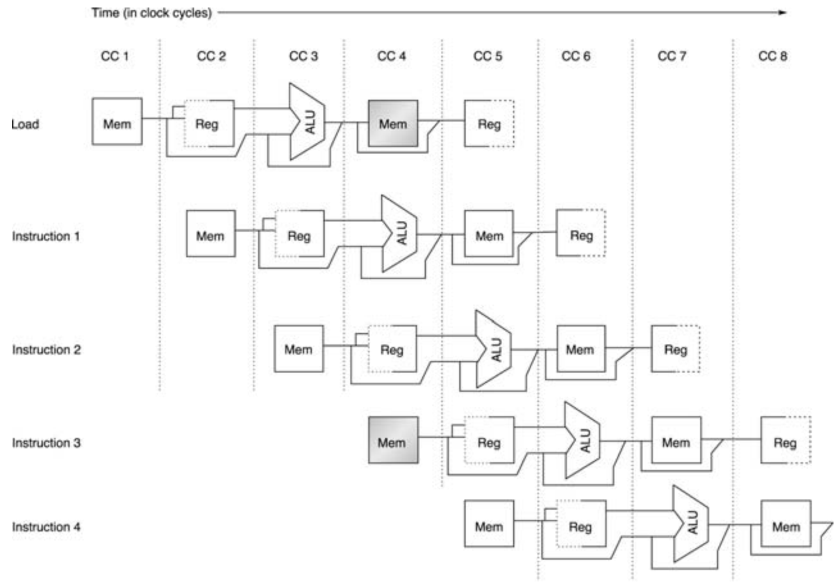


© 2003 Elsevier Science (USA). All rights reserved.

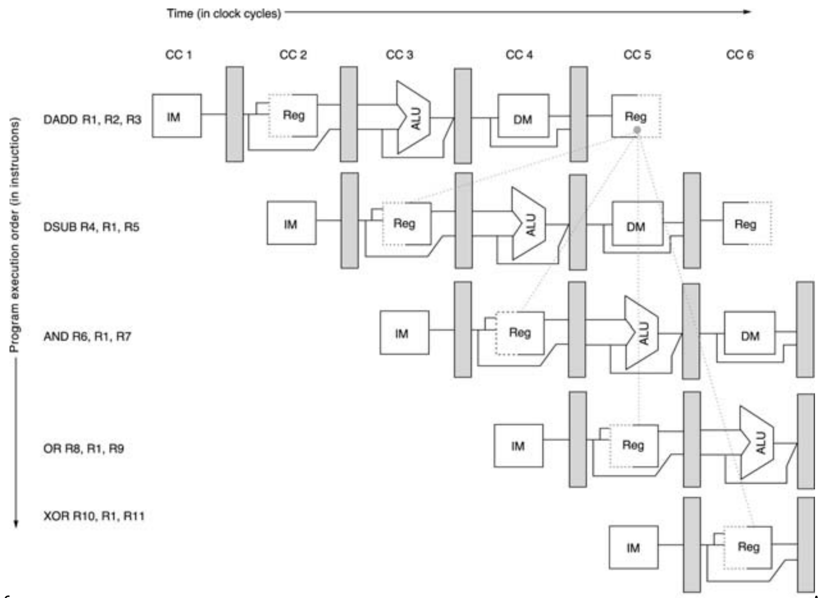
Pipeline characteristics

- **Parallelism**
- **1 instruction issued per cycle**
- **CPI Pipelined = Ideal CPI**
  + Pipeline stall cycles per instruction
- **Reduced performance due to hazards:**
  - **Structural**
    - **E.g. single memory – need to provide sufficient resources**
  - **Data**
    - **Use forwarding/stall**
  - **Control**
    - **Cope with hardware and software techniques**

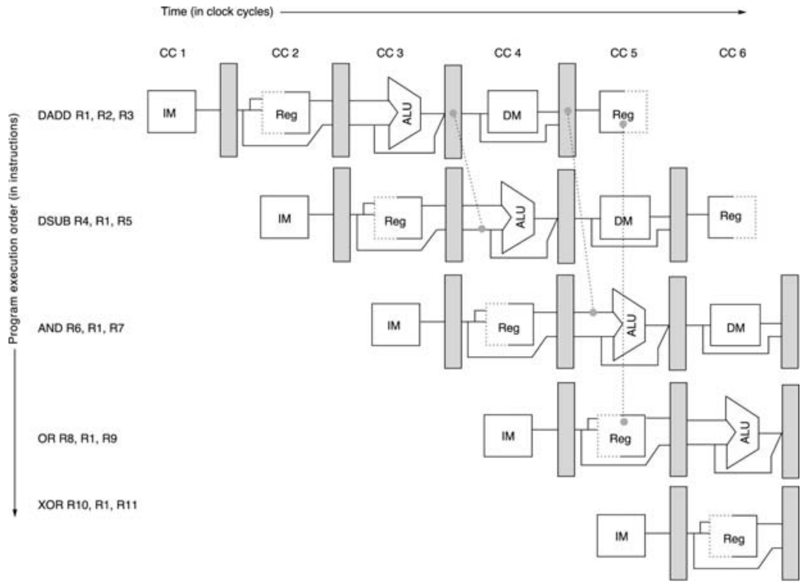
Structural hazard example



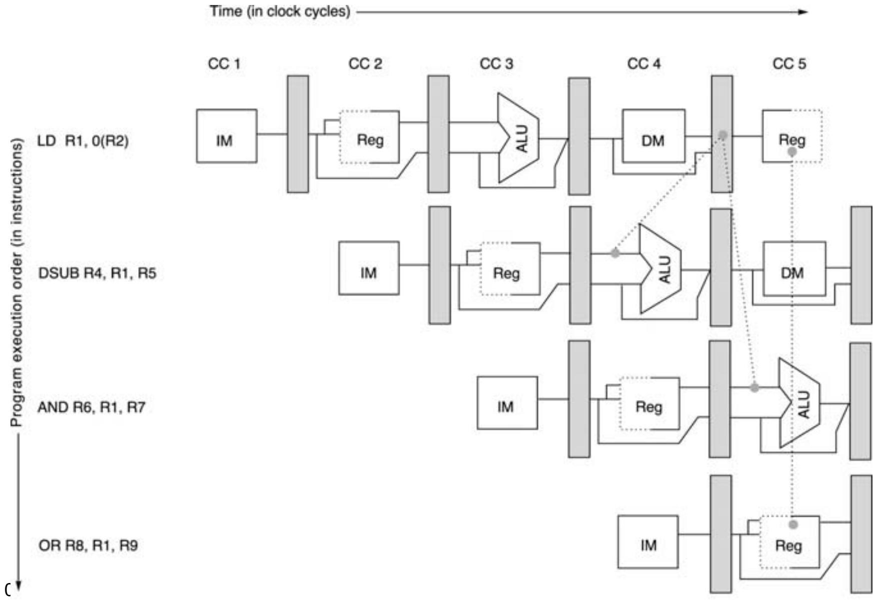
Data hazard examples



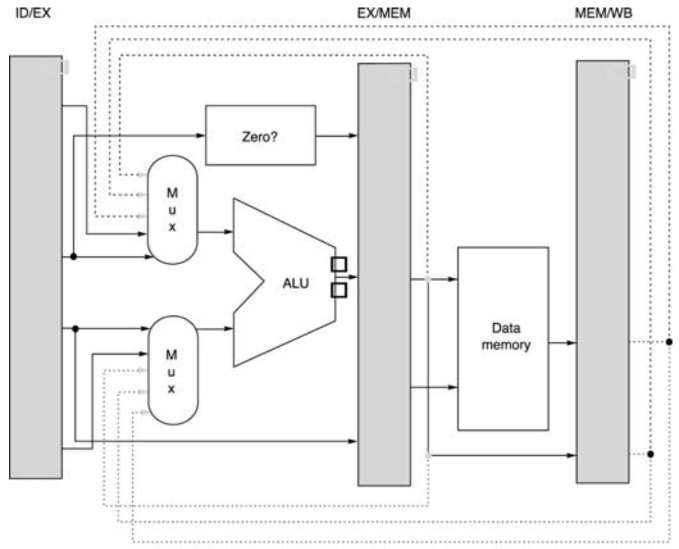
Data hazard remedy – forwarding



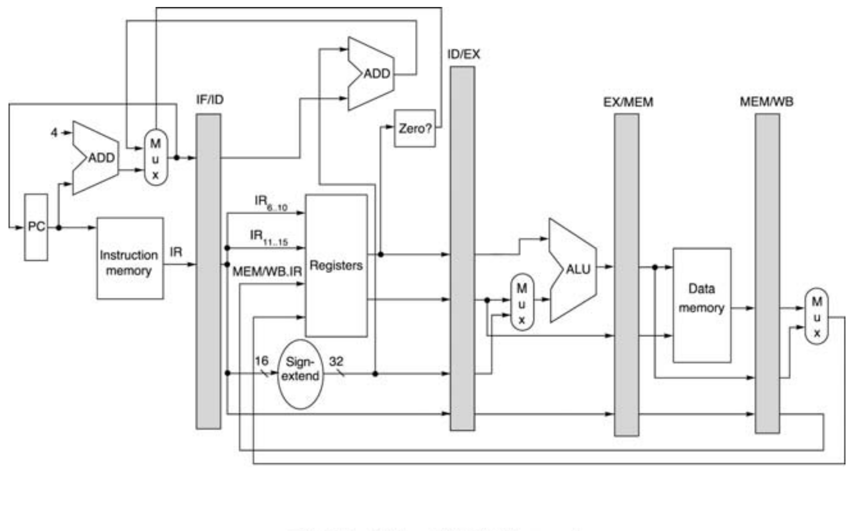
Data hazard needing stall



Forwarding hardware

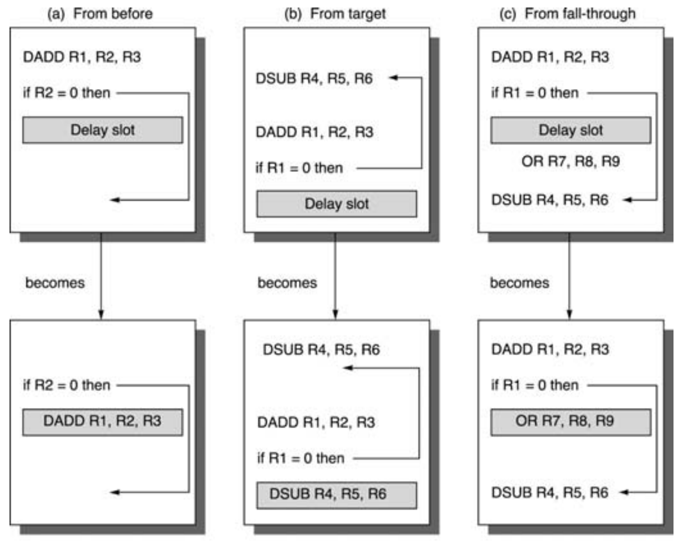


Control hazard – hardware amelioration



© 2003 Elsevier Science (USA). All rights reserved.

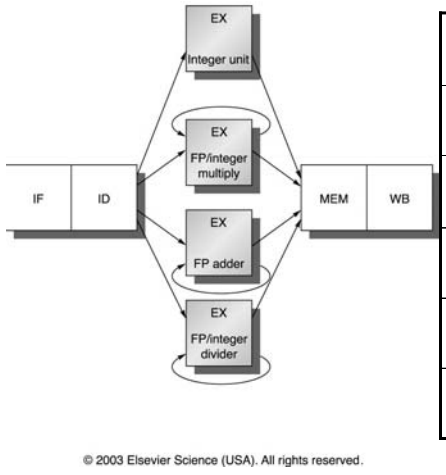
Control stalls – software amelioration



Notes on scheduling the delay slot

- **Scheduling an op that is above and independent of the branch into the delay slot, as in (a) is preferable**
- **If that is not possible, and we know the branch is usually taken, then as in (b)we can schedule from the target of the branch**
- **Otherwise, one of the fall-through instructions can be moved to the delay slot as in (c)**
- **In cases (b) and (c) it must not be the case that the moved instruction alters program correctness if the branch goes in the unexpected direction**

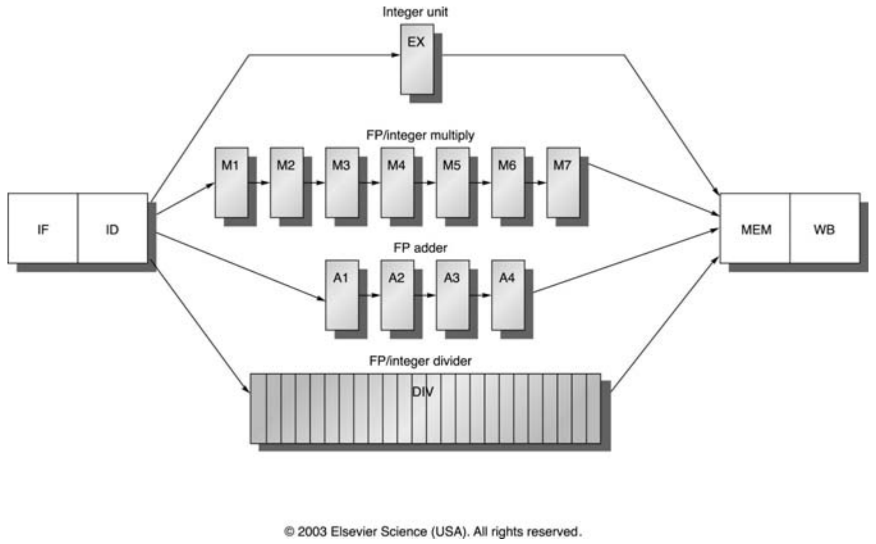
Extending MIPS to handle FP operations



© 2003 Elsevier Science (USA). All rights reserved.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Memory (Int & FP loads)	1	1
FP add	3	1
Mult (Int & FP)	6	1
Div (Int & FP)	24	25

Making the pipeline stages explicit



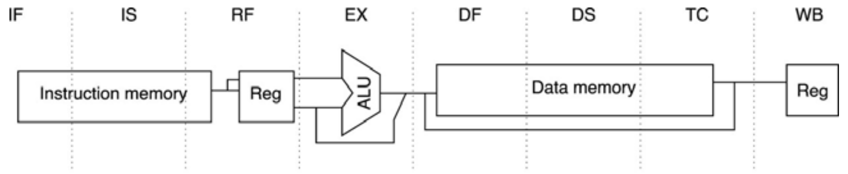
© 2003 Elsevier Science (USA). All rights reserved.

FP pipeline hazards

- Structural – only divide unit
- RAW – easiest to stall at ID stage if a source is not yet available
- WAW – stall at ID if necessary

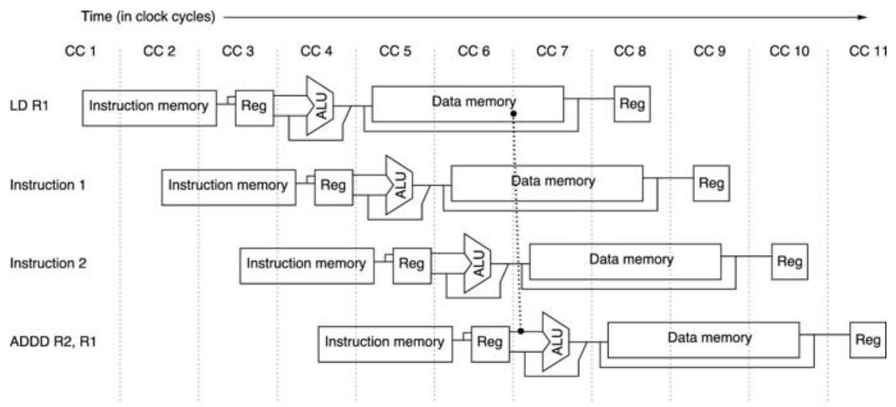
Example: MIPS R4000 pipeline

- 8 stage pipeline
- Extend IF & MEM stages to account for cache overheads
  - Split into “First” and “Second” stages followed by a “Tag Check” for misses
  - The IF tag check is done in the RF stage



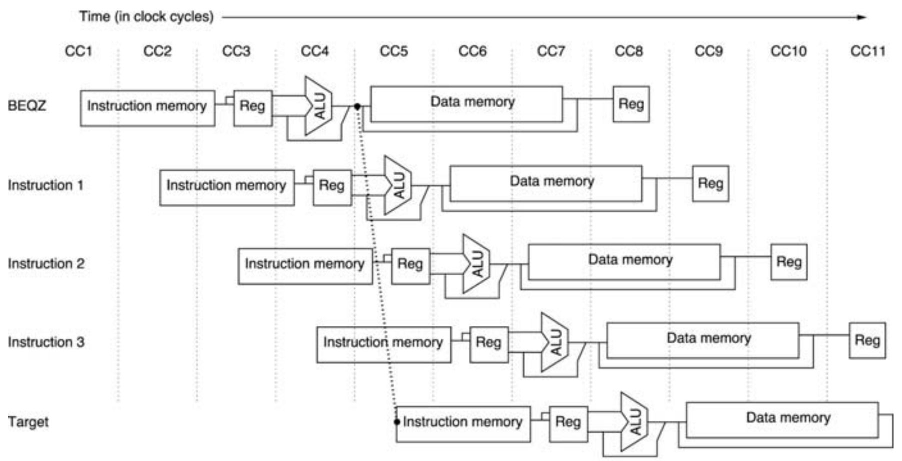
© 2003 Elsevier Science (USA). All rights reserved.

R4000: 2 cycle LoaD delay



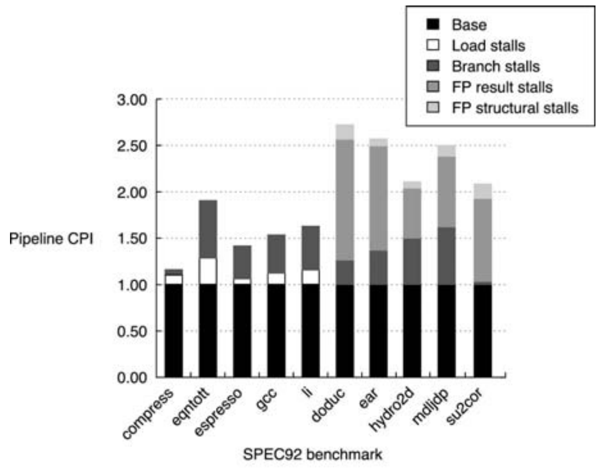
© 2003 Elsevier Science (USA). All rights reserved.

R4000: 3 cycle BRanch delay



© 2003 Elsevier Science (USA). All rights reserved.

R4000: SPEC92 performance



© 2003 Elsevier Science (USA). All rights reserved.

Instruction-level parallelism

- **Pipelining commonly used since 1985 to overlap the execution & improve performance – since instructions evaluated in parallel, known as *instruction-level parallelism (ILP)***
- **Here we look at extending pipelining ideas by increasing the amount of parallelism exploited among instructions**
- **Start by looking at limitation imposed by data & control hazards, then look at increasing the ability of the processor to exploit parallelism**

Two main approaches

- Two largely separable approaches to exploiting ILP:
  - Dynamic techniques (HP, Ch. 3) depend upon hardware to locate parallelism
  - Static techniques (HP, Ch. 4) rely much more on software
- Practical implementations typically involve a mix or some crossover of these approaches
- Dynamic, hardware-intensive approaches dominate the desktop and server markets; examples include Pentium, Power PC, and Alpha
- Static, compiler-intensive approaches have seen broader adoption in the embedded market, except, for example, IA-64 and Itanium

Questions this raises:

- What are the features of programs & processors that limit the amount of parallelism that can be exploited among instructions?
- How are programs mapped to hardware?
- Will a program property limit performance? If so, when?

Recall from Pipelining Review

- 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)
- In order to increase *instructions/cycle (IPC)* we need to pay increasing attention to dealing with stalls

Ideas to Reduce Stalls

	Technique	Reduces
Chapter 3	Dynamic scheduling	Data hazard stalls
Chapter 3	Dynamic branch prediction	Control stalls
Chapter 3	Issuing multiple instructions per cycle	Ideal CPI
Chapter 3	Speculation	Data and control stalls
Chapter 3	Dynamic memory disambiguation	Data hazard stalls involving memory
Chapter 4	Loop unrolling	Control hazard stalls
Chapter 4	Basic compiler pipeline scheduling	Data hazard stalls
Chapter 4	Compiler dependence analysis	Ideal CPI and data hazard stalls
Chapter 4	Software pipelining and trace scheduling	Ideal CPI and data hazard stalls
Chapter 4	Compiler speculation	Ideal CPI, data and control stalls

First limits on exploiting ILP

- The amount of parallelism available within a *basic block* – a straight-line code sequence with no branches in or out except to the entry and from the exit – is quite small
- Typical dynamic branch frequency is often between 15% and 25% – between 4 and 7 instructions execute between branch pairs – these instructions are likely to depend upon each other, and thus the overlap we can exploit within a basic block is typically less than the average block size
- To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks

Loop-level parallelism

- *Loop-level parallelism* increases the amount of parallelism available among iterations of a loop
- In the code:

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

  every iteration can overlap with any other, although within a loop iteration there is little or no opportunity for overlap
- We will examine techniques for unrolling loops to convert the loop-level parallelism to ILP
- Another approach to exploiting loop-level parallelism is to use vector instructions. While processors that exploit ILP have almost totally replaced vector processors, vector instruction sets may see a renaissance for use in graphics, DSP, and multimedia

Data dependences and hazards

- In order to exploit ILP we must determine which instructions can be executed in parallel
- *Parallel* instructions can execute simultaneously without causing stalls assuming there are no structural hazards (sufficient resources)
- *Dependent* instructions are not parallel and must be executed in order, although they may often be partially overlapped
- Three types of dependences exist:
  - *Data* dependences
  - *Name* dependences
  - *Control* dependences

Data dependence

- Instruction *j* is data dependent on instruction *i* if:
  - *i* produces a result that may be used by *j*, or
  - *j* is data dependent on instruction *k,* and *k* is data dependent on instruction *i.*
- Example

```
Loop:    L.D    F0,0(R1)     ;F0 = array element
         ADD.D  F4,F0,F2     ;add scalar in F2
         S.D    F4,0(R1)     ;store result
         ADDUI  R1,R1,#-8    ;dec pointer 8 bytes
         BNE    R1,R2,LOOP   ;branch R1!=R2
```

  has data dependences on consecutive pairs of instructions

Dependencies vs Hazards

- Processors with pipeline interlocks will detect a hazard and stall if such instructions are scheduled simultaneously
- Compilers for processors without interlocks that rely on compiler scheduling cannot schedule dependent instructions to allow complete overlap
- Dependences are properties of *programs* – whether a dependence results in an actual hazard being detected and whether that hazard causes a stall is a property of the *pipeline organization*
- A dependence
  1. Indicates the possibility of a hazard;
  2. Determines the order in which results must be calculated; and
  3. Sets an upper bound on how much parallelism can possibly be exploited

Maximise ILP by reducing hazards

- Since data dependences limit the amount of ILP we can exploit, we focus on how to overcome those limitations
- A dependence can be overcome by
  – Maintaining the dependence but avoiding a hazard, and
  – Eliminating the dependence by transforming the code.
- Here we primarily consider hardware techniques for scheduling the code dynamically as it is executed

Dependencies via register/memory

- Data values may flow from instruction to instruction
  – via registers (in which case dependence detection is reasonably straightforward since register names are fixed in the instructions, although intervening branches may cause correctness concerns), or
  – via memory (in which case dependences are more difficult to detect because of aliasing i.e. 100(R4) = 20(R6) and effective addresses such as 20(R6) may change from one execution of an instruction to the next)
- We will examine hardware for detecting data dependences that involve memory locations and will see the limitations of these techniques
- Compiler techniques for detecting dependences (Ch. 4) may be examined later

Name dependences

- A *name dependence* occurs when two instructions use the same register or memory location, called a *name* but there is no flow of data between the instructions associated with that name
- Two types, defined between an instruction *i* that *precedes* instruction *j*:
  1. An *antidependence* occurs when *j* writes to a name that *i* reads – the original ordering must be preserved
     e.g.
     ```
     ADD   R1, R3, R4
     LD    R4, 0(R0)
     ```
  2. An *output dependence* occurs when *i* and *j* write to the same name – also requires order to be preserved
     e.g.
     ```
     ADD   R4, R3, R1
     LD    R4, 0(R0)
     ```

Avoiding name dependencies

- Since a name dependence is not a true (data) dependence, instructions involved in a name dependence can be executed simultaneously or be reordered if the name (register or memory location) is changed to avoid the conflict
- Renaming is more easily done for registers, and it can be done either statically by the compiler, or dynamically by hardware

Data hazards

- A hazard is created whenever there is a dependence between instructions and they are close enough that the overlap caused by pipelining or reordering would change the order of access to the operand involved in the dependence
- We must then preserve *program order* i.e., the order instructions would execute in if executed sequentially
- Our goal is to exploit parallelism by preserving program order *only where it affects the outcome of the program*
- Detecting & avoiding hazards ensures the necessary program order is preserved

Categorizing data hazards

- Three types of data hazards depending upon the order of read and write accesses
- By convention, hazards are named by the ordering that must be preserved
- Consider two instructions *i* and *j* with *i* occurring before *j* in program order. The possible hazards are:
  - *RAW (read after write) – j* tries to read a source before *i* writes it. This hazard is most common and corresponds to true data dependence

    e.g.
    ```
    LD    R4, 0(R0)
    ADD   R1, R3, R4
    ```

Data hazard categories

  - *WAW (write after write) – j* tries to write an operand before it is written by *i*. This hazard corresponds to output dependence. They occur in pipelines that write in more than one stage or allow instructions to proceed when previous ones are stalled. It is not present in the simple statically scheduled 5 stage pipeline that only writes in the WB stage.
  - *WAR (write after read) – j* tries to write a destination before it has been read by *i*. Arises from antidependence. Cannot occur in static issue pipelines when reads are earlier in the pipeline than writes. Can occur when some instruction writes early in the pipeline and another reads late, or when instructions can be reordered
- Note that RAR (read after read) is not a hazard

Control dependence

- A control dependence determines the ordering of an instruction *i* with respect to a branch instruction so that *i* is executed in correct program order, and only when it should be

  e.g.
```
S1;
if P2 {
     S3;
}
```
- S1 cannot be moved under the control of P2, nor can S3 be moved out of control of P2

Control Dependence Ignored

- Control dependence need not be preserved
  – willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program
- Instead, 2 properties critical to program correctness are exception behavior and data flow

Overcoming data hazards with dynamic scheduling

- Simple statically scheduled pipelines fetch instructions and issue them unless stalled due to some data dependence that cannot be hidden by forwarding
- Once stalled, no further instructions are fetched or issued until the dependence is cleared

- From now on we explore *dynamic scheduling* in which the hardware rearranges instruction execution to reduce stalls while maintaining data flow and exception behaviour
- This technique allows us to handle dependences that are unknown at compile time (e.g. a memory reference) and allows code that was compiled with one pipeline in mind to be efficiently executed on a different pipeline
- Unfortunately, the benefits of dynamic scheduling are gained at the cost of a significant increase in hardware complexity

Scheduling to minimize hazards

- A dynamically scheduled processor attempts to avoid stalls in the presence of dependences.
- In contrast, static pipeline scheduling by the compiler (Ch. 4) tries to minimize stalls by separating dependent instructions to avoid hazards

For next week:

- **Appendix A.8 – Scoreboarding**
- **Ch 3.2,3.3 – Tomasulo’s Algorithm**