

Pipeline Hazards

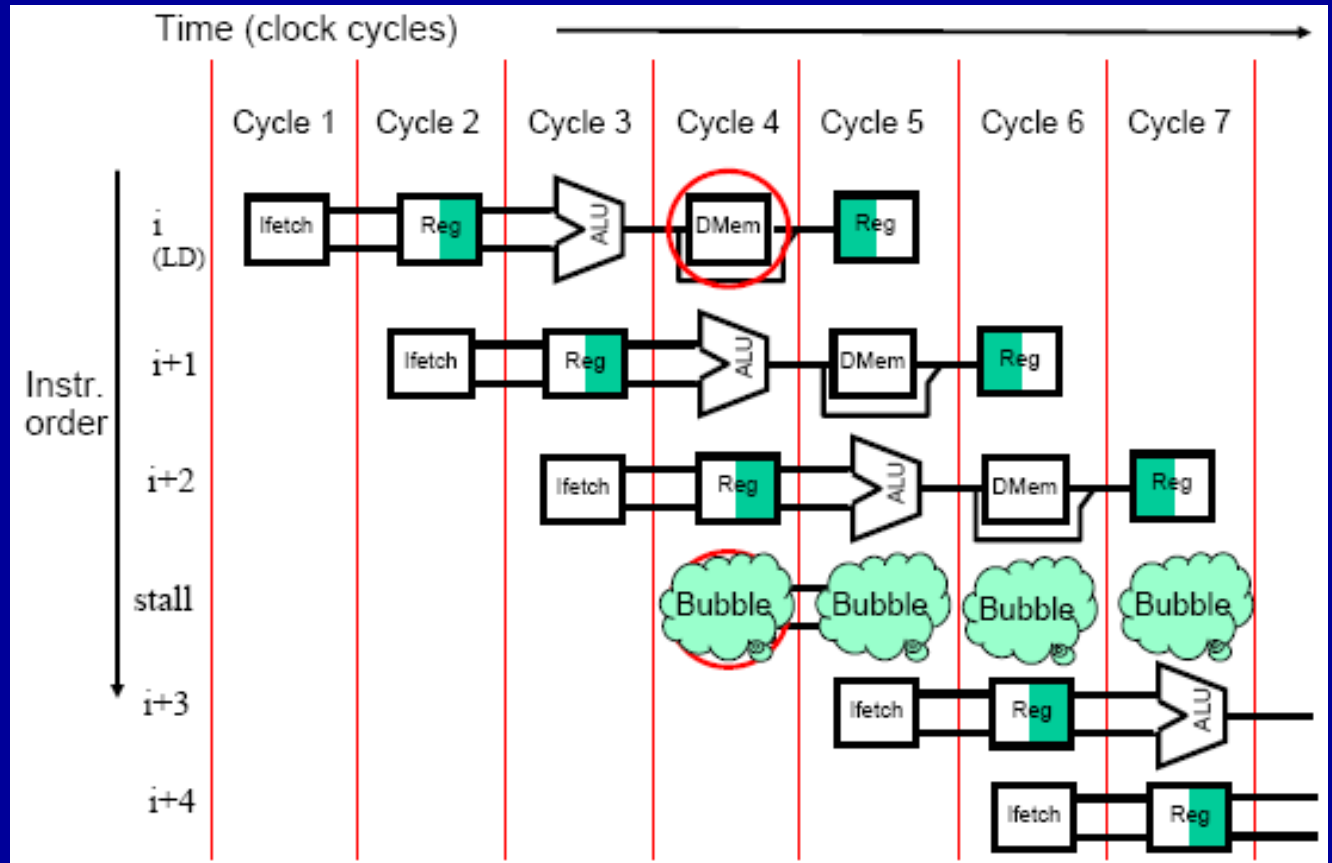
Pipeline Hazards

- Should we expect a **CPI of 1** in practice?
 - Unfortunately, the answer to the question is **NO**.
 - Limit to pipelining: **Hazards**
- **Hazards** are circumstances that prevent the next instruction in the instruction stream from executing during the designated clock cycle
- Three major types:
 - **Structural hazards:** Not enough HW resources to keep all instructions moving
 - **Data hazards:** Data results of earlier instructions not available yet
 - **Control hazards:** Control decisions resulting from earlier instruction (branches) not yet made; don't know which new instruction to execute

Structural Hazard

Structural Hazard Example

- Occurs when two or more instructions need the same resource
- A pipeline stalled for a structural hazard when both instructions and data shares the same memory



Common methods for eliminating structural hazards

- Duplicate resources
- Pipeline the resource
- Reorder the instructions
- It may be too expensive to eliminate a structural hazard, in which case the pipeline should stall
- No new instruction is issued until the hazard has been resolved

Data Hazard

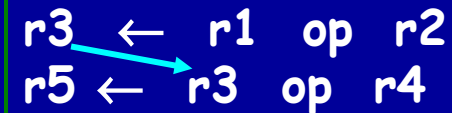
Program Dependences

- Dependences are **property of programs**
- Whether a dependence leads to an actual Hazards and whether that hazard actually causes a stall are properties of the **pipeline organization**
- There are three main types of dependences in a program:
 - **Data dependences**
 - **Name dependences**
 - **Control dependences**

Data Dependences

➤ An instruction **j** is **data dependent** on instruction **i**, iff either of:

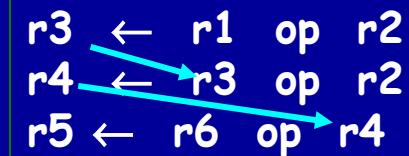
- **Direct:** Instruction **i** produces a result that is used by instruction **j**



A diagram showing two instructions: $r3 \leftarrow r1 \text{ op } r2$ and $r5 \leftarrow r3 \text{ op } r4$. A red arrow points from the $r3$ in the first instruction to the $r3$ in the second instruction, indicating that the second instruction depends on the result of the first.

- **Transitive:**

- Instruction **j** is data dependent on instruction **k** and
- Instruction **k** is data dependent on instruction **i**



A diagram showing three instructions: $r3 \leftarrow r1 \text{ op } r2$, $r4 \leftarrow r3 \text{ op } r2$, and $r5 \leftarrow r6 \text{ op } r4$. A red arrow points from the $r3$ in the first instruction to the $r3$ in the second instruction. Another red arrow points from the $r4$ in the second instruction to the $r4$ in the third instruction, illustrating a transitive data dependence from the first instruction to the third.

- Dependence within a single instruction (such as **ADDD R1, R1, R1**) is not considered as a dependence

Data Dependences

- If two instructions are **data dependent**, they cannot be executed simultaneously or be completely overlapped

- Example:

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

- A data dependence conveys three things:
 - The possibility of a hazard
 - The order in which the results must be calculated
 - An upper bound on how much parallelism can be exploited

Data Dependences

- Data dependence can limit the instruction level parallelism
- A dependence can be overcome in two different ways
 - Maintaining the dependence by avoiding a hazard
 - Eliminating a dependence by transforming the code

Detecting Data Dependences

- A data value may flow between instructions:
 - through registers
 - through memory locations
- When data flow is through a register:
 - Detection is rather straightforward
- When data flow is through a memory location:
 - Detection is difficult.
 - Two addresses may refer to the same memory location but look different. **100(R4) and 20(R6)**
 - The effective address may differ from one execution to another; **20 (R4) in one instruction may differ from 20(R4) in another**

Name Dependences

➤ Name dependence:

- Two instructions use the same register or memory location (called a name)
- There is no true flow of data between the two instructions
- Example: $A = B + C$; $A = P + Q$;

➤ Two types of name dependences:

- Anti-dependence
- Output dependence

Antidependence

- Antidependence occurs between two instructions **i** and **j**, iff:
 - **j** writes to a register or memory location that **i** reads.
 - Original ordering must be preserved to ensure that **i** reads the correct value.
- Example:
 - **ADD F0,F6,F8**
 - **SUB F8,F4,F5**

Output Dependences

- Output dependence occurs between two instructions **i** and **j**, iff:
 - The two instructions write to the same register or memory location.
- Ordering of the instructions must be preserved to ensure:
 - Finally written value corresponds to **j**.
- Name dependences are not true data dependences
- Instructions involving name dependences can be executed simultaneously by **register renaming**

Control Dependences

- Determines the ordering the of an instruction with respect to a branch instruction. Example:

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

- S1 is control dependent on p1, but S2 is not control dependent on p1
- An instruction that is control dependent on a branch cannot be moved **before** the branch
- An instruction that is not control dependent on a branch cannot be moved **after** the branch

Control Dependences

- When program order is strictly preserved, it ensures that control dependences are also preserved
- Example:

| | |
|-------|----------|
| DADDU | R2,R3,R4 |
| BEQZ | R2,L1 |
| LW | R1,0(R2) |

L1:

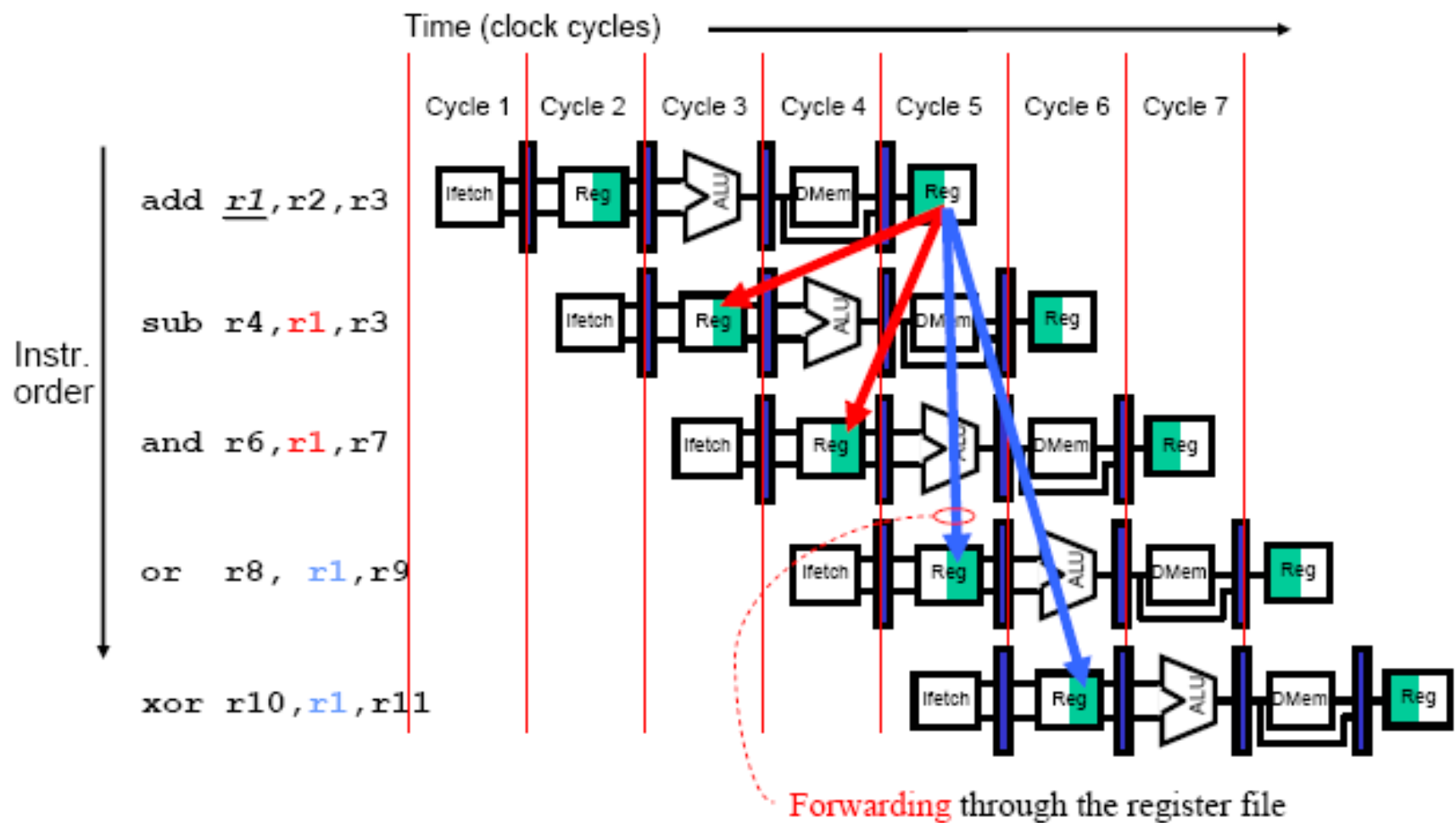
Data Hazard

➤ Arises out of change of relative timing of instructions by overlapping their execution

➤ Example:


| | |
|-----|--------------|
| add | r1, r2, r3 |
| sub | r4, r1, r5 |
| and | r6, r1, r7 |
| or | r8, r1, r9 |
| xor | r10, r1, r11 |

Data Hazard Example




Classification of Data Hazards


- Let **I** be an earlier instruction, **J** a later one
- **RAW** (read after write)
 - **J** tries to read a value before **I** writes into it
- **WAW** (write after write)
 - **I** and **J** write to the same place, but in the wrong order
 - Only occurs if more than one pipeline stage can write (in-order)
- **WAR** (write after read)
 - **J** writes a new value to a location before **I** has read the old one
 - Only occurs if writes can happen before reads in pipeline (in-order)



```
I: add r1, r2, r3
J: sub r4, r1, r3
```



```
I: sub r1, r4, r3
J: add r1, r2, r3
```



```
I: sub r4, r1, r3
J: add r1, r2, r3
```

No WAR and WAW hazards

