Limitations of Static Scheduling

The Problem

Simple pipelining uses in-order instruction issue and execution. If an instruction stalls due to a dependency, all later instructions must also stall - even if they have no dependencies.

Example

```
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F12,F8,F14
```

The SUB.D must wait for ADD.D, which waits for DIV.D, despite SUB.D having no dependency on DIV.D.

This creates a performance limitation that can be eliminated by allowing instructions to execute out of program order.

Dynamic Scheduling: The Concept

Key Innovations

- Split instruction decode into two parts:
    - checking for structural hazards and
    - waiting for data hazards to clear
- Instructions still issue in-order
- Instructions begin execution as soon as operands are available
- Results in out-of-order execution and completion

Benefits

- Code compiled for one pipeline runs efficiently on different pipelines
- Handles dependencies unknown at compile time
- Tolerates unpredictable delays like cache misses

These advantages come at the cost of significant hardware complexity.

Out-of-Order Execution

- Split the ID (decode) stage into:
  - Issue: Decode instructions, check for structural hazards.
  - Read operands: Wait until no data hazards, then read operands.
- Dynamically scheduled pipeline
  - All instructions pass through the issue stage in order
  - They can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order

Challenges of Out-of-Order Execution

1

New Hazard Types

Out-of-order execution introduces Write-After-Read (WAR) and Write-After-Write (WAW) hazards that don't exist in simple in-order pipelines.

2

Exception Handling

Must preserve exception behavior as if instructions executed in strict program order, despite out-of-order completion.

3

Register Conflicts

Limited register sets (like IBM 360's four FP registers) create artificial dependencies that limit performance.

These challenges are addressed through register renaming and careful exception management.

Register Renaming: Eliminating False Dependencies

Original Code with Dependencies

```
DIV.D F0,F2,F4
ADD.D F6,F0,F8
S.D   F6,0(R1)
SUB.D F8,F10,F14
MUL.D F6,F10,F8
```

Contains WAR hazards between ADD.D/SUB.D and S.D/MUL.D, plus WAW hazard between ADD.D/MUL.D.

After Register Renaming

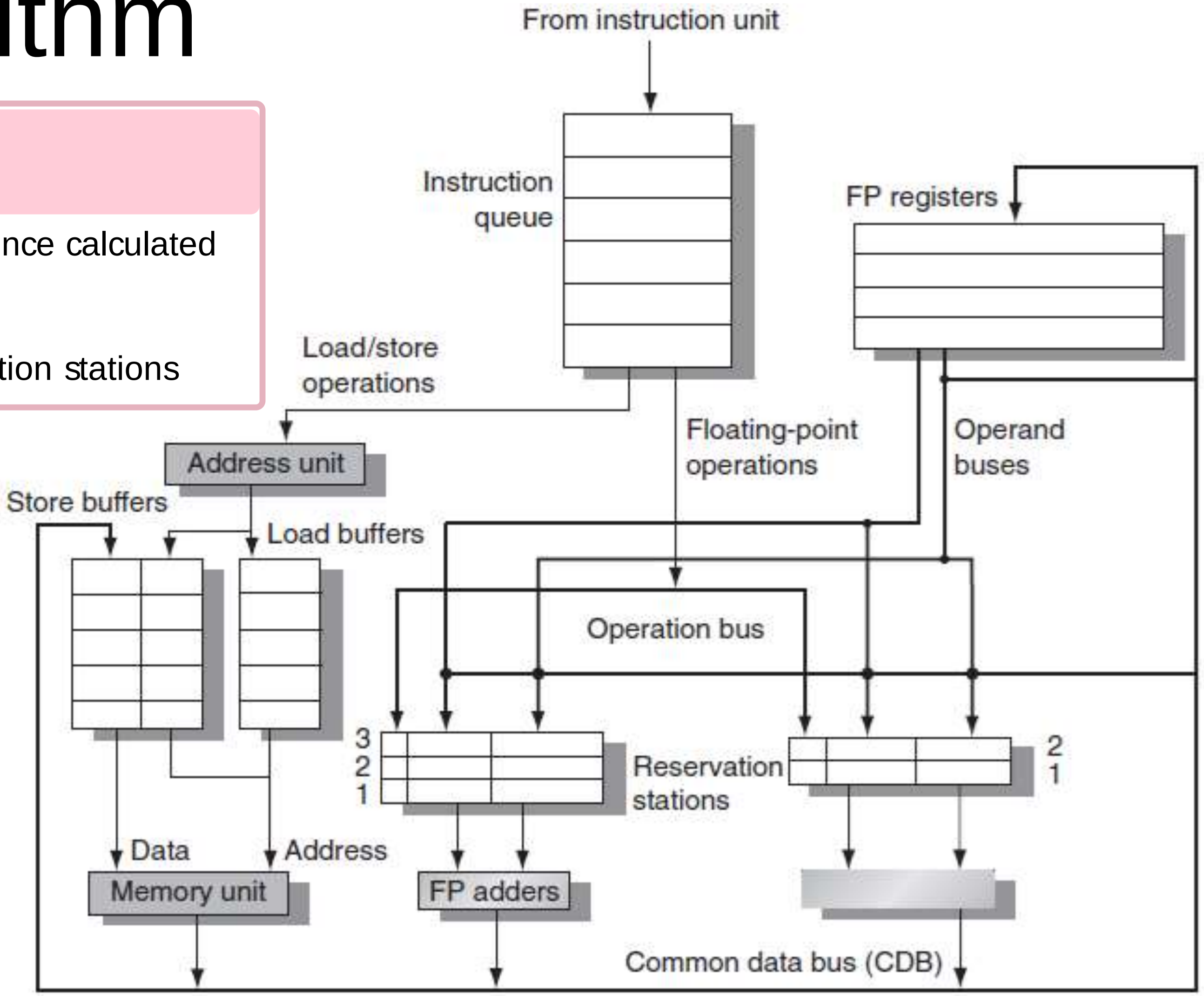
```
DIV.D F0,F2,F4
ADD.D S,F0,F8
S.D   S,0(R1)
SUB.D T,F10,F14
MUL.D F6,F10,T
```

Using temporary registers S and T eliminates all name dependencies, allowing parallel execution.

Floating-Point Unit using Tomasulo's Algorithm

Load/Store Buffers

- **A**: Effective address once calculated
- Functions like reservation stations



Register File

- **Qi**: Reservation station computing result for this register

Reservation Stations

- **Op**: Operation to perform
- **Qj, Qk**: Stations producing source operands
- **Vj, Vk**: Source operand values
- **A**: Address information for loads/stores
- **Busy**: Station occupancy status

Instruction Lifecycle in Tomasulo's Algorithm

Issue

- Get next instruction from queue (FIFO order)
- If empty reservation station exists, issue instruction
- Record which functional units will produce needed operands
- Rename registers to eliminate WAR/WAW hazards

Execute

- Monitor common data bus for needed operands
- When all operands available, execute at functional unit
- Loads/stores require two-step execution process
- Delay until preceding branches complete (for exception handling)

Write Result

- Broadcast result on common data bus
- Update registers and any waiting reservation stations
- Store results wait in buffer until address and value available