CS6L047: Advanced Computer Architecture

Dynamic Instruction Scheduling
(Appendix C, Chap 3)

Static vs Dynamic Instruction Scheduling

- **Static Scheduling (Compiler-based):**
  - Performed at **compile time**
  - Avoids some pipeline hazards
  - Example: **Filling branch delay slots**
  - Limited by compile-time knowledge of dependencies
- **Dynamic Scheduling (Hardware-based):**
  - Performed at **runtime by hardware**
  - Rearranges instruction execution to reduce stalls
  - More flexible and responsive to actual runtime behavior

Why Dynamic Scheduling in Hardware?

- **Unknown dependencies at compile time**:
  - Data-dependent branches
  - Indirect memory references
- **Portability**:
  - Simple compiler → Code runs efficiently on different machines
- **Latency hiding**:
  - Tolerates unpredictable delays (e.g., cache misses)
  - Executes independent instructions while waiting

Limitations of In-Order Execution

- **In-order issue and execution**:
  - If one instruction stalls, **all subsequent instructions stall**
  - Even independent instructions are delayed

- **Hazard example**:
  - Two instructions with a hazard between them → pipeline stalls
  - Later independent instructions cannot proceed

- **Out – of –order Execution in Dynamic Scheduling (Hardware):**
  - Rearranges instruction execution at runtime
  - Allows instructions **behind a stall** to proceed if they're independent

Motivation for Out-of-Order Execution

- Dynamic scheduling enables:
  - **Out-of-order issue and execution**
  - Better **instruction-level parallelism**
  - Reduced pipeline stalls
  - Improved CPU throughput

Dynamic Scheduling (Hardware)

- Key idea: Allow instructions behind stall to proceed even if there is dependency. Check it at the time of execution. Then independent instructions, which are behind, can be executed.
- Example instruction sequence:

```
DIVD   F0, F2, F4      ; Long latency
ADDD   F10, F0, F8     ; Depends on DIVD result
SUBD   F8, F8, F14     ; Independent of DIVD
```

DIVD starts execution
ADDD is stalled waiting for **F0**
SUBD is independent but **cannot proceed** in in-order pipeline

```
Inst#1:DIVD  F0, F2, F4      ; Long latency (400 cycles)
Inst#2: ADDD  F10, F0, F8     ; Depends on DIVD result (4 cycles)
Inst#3: SUBD  F8, F8, F14     ; Independent of DIVD (4 cycles)
```

Draw a pipeline diagram for the code example.

Calculate the total number of cycles for all 3 instructions.

Instruction execution latency is given. Assume data forwarding is ON

```
Inst#1:DIVD  F0, F2, F4      ; Long latency (400 cycles)
Inst#2: ADDD  F10, F0, F8     ; Depends on DIVD result (4 cycles)
Inst#3: SUBD  F8, F8, F14     ; Independent of DIVD (4 cycles)
```

Draw a pipeline diagram for the code example.

Calculate the total number of cycles for all 3 instructions.

Instruction execution latency is given. Assume data forwarding is ON

cc	1	2	3	4	5	. . .	402	403	404	405	406	407	408	409	410	411	412
Inst#1	F	D	X_1	X_2	X_3	…	X_{400}	M	W								
Inst#2		F	D	stall	stall	stall	stall	X_1	X_2	X_3	X_4	M	W				
Inst#3			F	D	stall	stall	stall	stall	stall	stall	stall	X_1	X_2	X_3	X_4	M	W

- stall due to RAW in inst#1 and inst#2
  stall due to structural hazard with inst#1 and inst#2
  Total cycles = 412

```
Inst#1:DIVD  F0, F2, F4      ; Long latency (400 cycles)
Inst#2: ADDD  F10, F0, F8     ; Depends on DIVD result (4 cycles)
Inst#3: SUBD  F8, F8, F14     ; Independent of DIVD (4 cycles)
```

Draw a pipeline diagram for the code example.

Calculate the total number of cycles for all 3 instructions.

Instruction execution latency is given. Assume data forwarding is ON

cc	1	2	3	4	5	. . .	402	403	404	405	406	407	408	409	410	411	412
Inst#1	F	D	X_1	X_2	X_3	…	X_{400}	M	W								
Inst#2		F	D	stall	stall	stall	stall	X_1	X_2	X_3	X_4	M	W				

Can you reduce the total cycles by using any of the optimization techniques you have studied so far?

Can you reduce the total cycles by using any of the optimization techniques you have studied so far?

```
Inst#1:DIVD   F0, F2, F4      ; Long latency (400 cycles)
Inst#2:ADDD   F10, F0, F8     ; Depends on DIVD result (4 cycles)
Inst#3:SUBD   F8, F8, F14     ; Independent of DIVD (4 cycles)
```

Solution#1: Code Scheduling + register renaming

```
Inst#1:DIVD   F0, F2, F4      ; Long latency (400 cycles)
Inst#3:SUBD   F18, F18, F14   ; Independent of DIVD (4 cycles)
Inst#2:ADDD   F10, F0, F8     ; Depends on DIVD result (4 cycles)
```

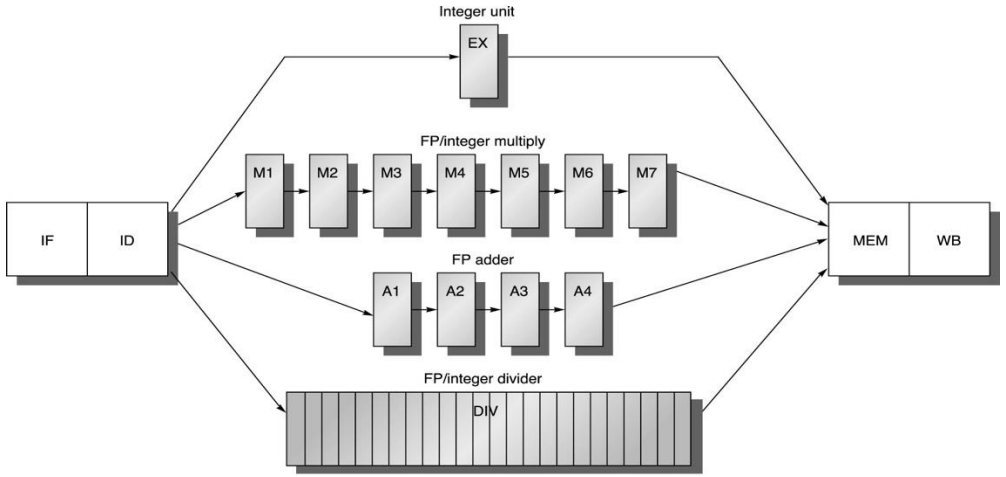
cc	1	2	3	4	5	. . .	402	403	404	405	406	407	408	409	410	411	412
Inst#1	F	D	X_1	X_2	X_3	…	X_{400}	M	W								
Inst#3		F	D	stall	stall	stall	stall	X_1	X_2	X_3	X_4	M	W				
Inst#2			F	D	stall	stall	stall	stall	stall	stall	stall	X_1	X_2	X_3	X_4	M	W

- stall due to structural hazard with in inst#1
  stall due to RAW in inst#1
  stall due to structural hazard with inst#3
  Total cycles = 412

Can you reduce the total cycles by using any of the optimization techniques you have studied so far?

```
Inst#1:DIVD   F0, F2, F4      ; Long latency (400 cycles)
Inst#2:ADDD   F10, F0, F8     ; Depends on DIVD result (4 cycles)
Inst#3:SUBD   F8, F8, F14     ; Independent of DIVD (4 cycles)
```

Solution#2: Super Scalar [with DIVD, SUBD and ADDD having separate execution units]



Representative diagram to remind you about super scalar and long – execution pipelines

Draw a pipeline diagram for the code example.

Calculate the total number of cycles for all 3 instructions.

Can you reduce the total cycles by using any of the optimization techniques you have studied so far?

```
Inst#1:DIVD   F0, F2, F4      ; Long latency (400 cycles)
Inst#2:ADDD   F10, F0, F8     ; Depends on DIVD result (4 cycles)
Inst#3:SUBD   F8, F8, F14     ; Independent of DIVD (4 cycles)
```

Solution#2: Super Scalar

cc	1	2	3	4	5	. . .	402	403	404	405	406	407	408	409	410	411	412
Inst#1	F	D	X_1	X_2	X_3	…	X_{400}	M	W								
Inst#3		F	D	X_1	X_2	…											
Inst#2			F	D	stall	stall	stall	X_1	X_2	X_3	X_4	M	W				

stall due to RAW in inst#1

Total cycles = 408

Dynamic Scheduling (Example)

- `DIVD  F0, F2, F4      ; Long latency`
- `ADDD  F10, F0, F8     ; Depends on DIVD result`
- `SUBD  F8, F8, F14     ; Independent of DIVD`
- **Limitation of In-Order Issue**
  - Structural and data hazards are checked at **ID (Instruction Decode)** stage
  - If **ADDD** is stalled at ID, **SUBD** cannot even enter ID
  - Pipeline stalls unnecessarily
- **Dynamic Scheduling Enables:**
  - **Out-of-order execution**: SUBD can execute while ADDD waits
  - **Out-of-order completion**: Instructions finish as soon as they're ready
  - Better **latency hiding** and **throughput**

Enabling Out-of-Order Execution in MIPS/RISC-V Pipeline

- **Problem with Traditional ID Stage:**
  - In classic MIPS/RISC-V pipeline:
    - **Structural and data hazards** are checked during **Instruction Decode (ID)**
    - If an instruction is stalled at ID, **later instructions cannot proceed**
- **Goal:**
  - Allow **independent instructions** to begin execution **even if earlier ones are stalled**
  - Enables **out-of-order execution** and **out-of-order completion**
  - **Out-of-order execution, typically implies out-of-order completion**.

Solution: Split the ID Stage

- To support dynamic scheduling:

1. Issue Stage

- Decode instruction
- Check **structural hazards**
- Issue instruction if resources are available

2. Read Operands Stage

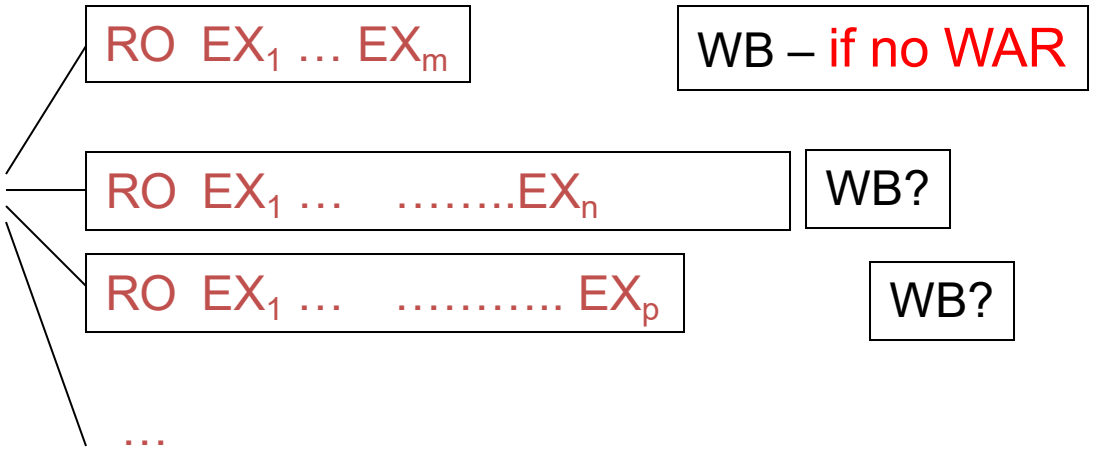
- Wait until **data hazards** are resolved
- Read operands when they are ready
- Begin execution immediately

Benefits of Splitting ID Stage

- Instructions are **issued in order**
- Execution begins **out of order** as soon as operands are ready
- Improves **pipeline utilization**
- Reduces **stalling due to data dependencies**

Scoreboard Technique in CDC 6600

- The Scoreboard replaces the traditional **Instruction Decode (ID)** stage with two distinct stages:
  - **1. Issue Stage**
    - Decodes the instruction.
    - Checks for **structural hazards**.
    - Issues instructions **in program order** if:
      - The required functional unit is available.
      - There is **no Write After Write (WAW)** hazard.
  - **2. Read Operands (RO) Stage**
    - Waits until there are no Read After Write (RAW) hazards.
    - Once safe, reads operands and proceeds to execution.



Scoreboard Technique in CDC 6600

- Instructions can **execute out-of-order** as long as:
  - The functional unit is free and no WAW hazard exists.
  - There are no RAW hazards.
- This allows **greater parallelism** and **better utilization** of hardware resources.
- Pipeline stages with Issue, RO, Execute, Write Result.
- A scoreboard table tracking:
  - Instruction status
  - Functional unit usage
  - Operand availability
  - Hazards

HW Schemes: Scoreboard CDC 6600 (Appendix C)

- Scoreboarding is a technique for allowing instructions to **execute out of order**
  - when there are sufficient resources and
  - no data dependences;
- it is named after the CDC 6600 scoreboard

Scoreboard Technique

Stage	Scoreboarding Behavior
Issue	In-order
Execution	Out-of-order
Completion	Out-of-order
Write-back	In-order

- In-order issue
  - Instructions are issued in the order they appear in the program.
  - The scoreboard checks if an instruction can proceed based on resource availability and hazards.
- Out-of-order execution
  - Once issued, instructions can execute out of order if their operands are ready and there are no hazards.
  - This allows independent instructions to proceed even if earlier ones are stalled.
- Out-of-order completion
  - Instructions can also complete **out of order**, but results are written in order to maintain precise exceptions.

Out-of-Order Completion: Hazards and Scoreboard Solutions

- ⚠ Hazards Introduced:
  - Write After Read (WAR)
  - Write After Write (WAW)
- ✅ Solutions for WAR Hazards
  - Stall the Write operation to allow Reads to complete.
  - Restrict register reads to the Read Operands (RO) stage only.
- ✅ **Solutions for WAW Hazards**
  - Detect hazard during **Issue** stage.
  - Stall issuing instruction until the earlier one completes.
  - Requires **multiple execution units** or **pipelined execution units** to support concurrent execution.

Role of the Scoreboard

- Tracks **instruction dependencies** and **hardware state**.
- Monitors all changes in the system.
- Determines:
  - When operands can be read.
  - When execution can begin.
  - When results can be written back.
  - Centralized **hazard detection and resolution**.

Benefits of Splitting ID Stage

- Instructions are **issued in order**
- Execution begins **out of order** as soon as operands are ready
- Improves **pipeline utilization**
- Reduces **stalling due to data dependencies**

```
DIVD   F0, F2, F4      ; Long latency (400 cycles)
ADDD   F10, F0, F8     ; Depends on DIVD result (4 cycles)
SUBD   F8, F8, F14     ; Independent of DIVD (4 cycles)
```

Draw a pipeline diagram for the code example with **Split ID stage.**
Instruction latency is given.

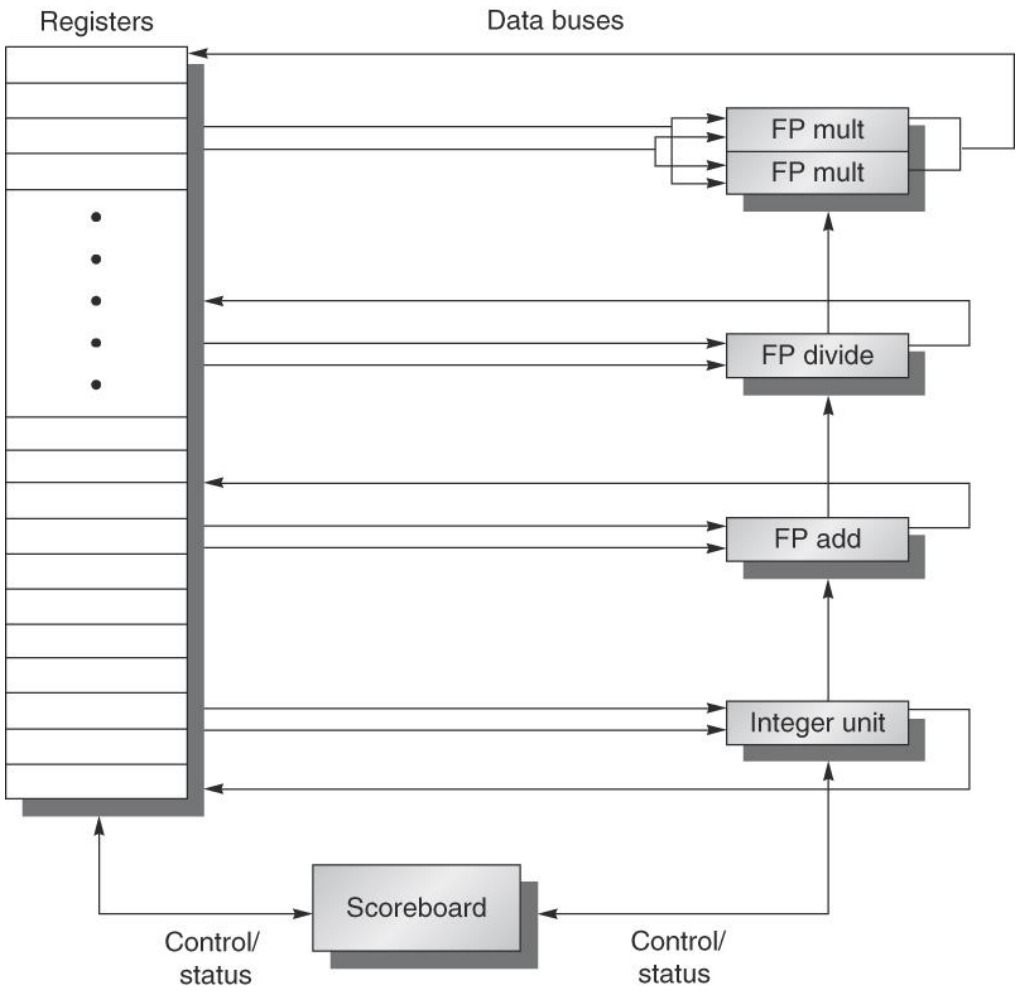
For next class…. Come prepared with -

Scoreboard CDC 6600

Solve the practice problem given in the slide:

https://people.eecs.berkeley.edu/~randy/Courses/CS252.S96/Lecture10.pdf

MIPS Processor Architecture with Scoreboard



The **scoreboard** manages instruction execution using **vertical control lines**.

Data flow between the **register file** and **functional units** occurs over **horizontal buses**, known as **trunks** in the CDC 6600.

The architecture includes:

- **Two FP multipliers**
- **One FP divider**
- **One FP adder**
- **One integer unit**

A **single bus set** (two inputs and one output) serves a group of functional units.

Detailed operation of the scoreboard in **Figures C.55 to C.58**.

Scoreboarding: 4 Stages

- **1. Issue Stage (ID1)**
  - **Decodes instruction** and checks for **structural hazards**.
  - If:
    - A **functional unit is free**, and
    - No active instruction has the **same destination register** (avoiding **WAW hazard**),
  - → The scoreboard **issues** the instruction and **updates its internal state**.
  - If a **structural or WAW hazard** exists:
    - Instruction **issue stalls**.
    - **No further instructions** are issued until the hazard is resolved.

Scoreboarding: 4 Stages

- **2. Read Operands Stage (ID2)**
  - Waits until **no RAW hazards** exist.
  - A source operand is available if:
    - No earlier active instruction will **write to it**, or
    - It is being **written by an active functional unit**.
  - Once operands are available:
    - Scoreboard signals the functional unit to **read operands** and **begin execution**.
    - **RAW hazards are resolved dynamically**.
    - Instructions may **enter execution out-of-order**.

Scoreboarding: 4 Stages

- **3. Execution Stage (EX)**
  - The **functional unit begins execution** once operands are received.
  - Upon completion, it **notifies the scoreboard** that execution is finished.
- **4. Write Result Stage (WB)**
  - The scoreboard checks for **Write After Read (WAR)** hazards.
  - If **no WAR hazard**, the result is **written back**.
  - If a **WAR hazard** exists, the instruction **stalls** until it is resolved.

Example:

```
DIVD.   F0,F2,F4
ADDD    F10,F0,F8
SUBD    F8,F8,F14
```

CDC 6600 scoreboard would stall SUBD until ADDD reads operands

Scoreboard summary

- ❌ **No forwarding hardware**
  → Results cannot be directly passed to dependent instructions.
- **Limited to instructions within a basic block**
  → Small instruction window restricts dynamic scheduling.
- **Small number of functional units**
  → Increases chances of **structural hazards**.
- ⛔ **Wait for WAR hazards**
  → Write operations stall if a read is pending on the same register.
- ⛔ **Prevent WAW hazards**
  → Instruction issue stalls to avoid multiple writes to the same register.