

Dynamic scheduling

- Last time: data hazards that prevent instruction issue were hidden by:
 - Forwarding
 - Static scheduling by the compiler
- Dynamic scheduling is also possible:
 - CPU rearranges the instructions (while preserving dependences) to reduce stalls
- Dynamic scheduling has several advantages over static
 - Handles dependencies that are *UNKNOWN* at compile time such as
 - Memory references
 - Branches
 - Allows code compiled with one pipeline in mind to run efficiently on a different pipeline

Out-of-order execution: basics

- Until now, all techniques require in-order instruction issue
 - A stalled instruction holds up those behind it
- What if following instructions could “pass” the stalled one?

```
DIVD F0,F2,F4      ; long latency
```

```
ADDD F10,F0,F8     ; stalled waiting for F0
```

```
SUBD F12,F8,F14    ; could proceed with this one!
```

- Out-of-order execution: allow instructions to issue in any order as long as dependencies aren't violated
 - Execute SUBD before ADDD in above example, reducing stalls
 - Handle out-of-order completion
 - May cause problems handling exceptions
 - May not gain if there are long dependence chains

Implementing out-of-order execution

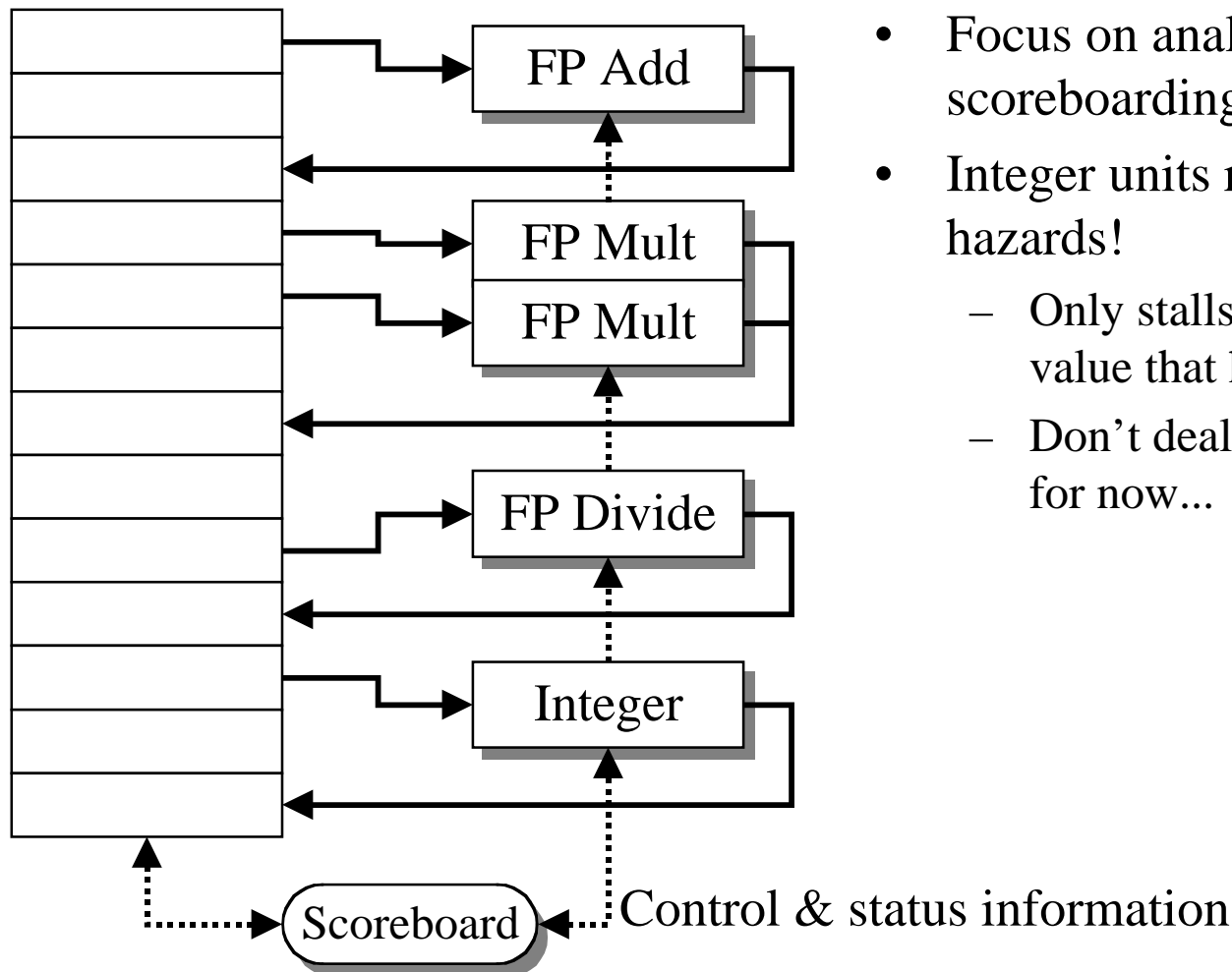
- Split the ID stage into two halves
 - Issue: decode instructions and check for structural hazards
 - Read operands
 - Wait until there are no data hazards, then
 - Read the operands
 - Continue to use forwarding to remove data hazards
- Designs of this type may use an instruction queue to hold instructions that have been fetched but are waiting to be executed
 - An instruction is considered to be in **execution** at any time that it's in an EX stage
 - Multiple instructions can be in execution at any given time

Scoreboarding

- This technique issues instructions in order (in-order issue)
 - Instructions can pass other waiting instructions in the “read operands” phase
 - WAR hazards are now possible (didn’t exist in previous pipelines)
- Scoreboarding first used in the CDC 6600 (the designers named it)
- Goal: maintain an execution rate of one instruction per cycle
 - Execute instructions as soon as possible
 - Use either multiple functional units or pipelined functional units (they’re equivalent for the purposes of pipeline control)
 - We’ll assume multiple functional units

```
DIVD F0,F2,F4    ; divide takes a long time
ADDD F10,F0,F8   ; stalled waiting for F0 from divide
SUBD F8,F8,F14   ; stalled waiting for ADDD to read F8 (WAR)
```

DLX implementation using a scoreboard



- Focus on analysis of scoreboarding in the FP units
- Integer units rarely encounter hazards!
 - Only stalls when waiting for a value that has just been loaded
 - Don't deal with integer hazards for now...

Pipeline changes for scoreboarding

- Every instruction goes through the scoreboard
 - Scoreboard determines when an instruction can read its operands and write its results
 - ⇒ All hazard detection and resolution is centralized
- ID stage replaced with two stages:
 - Issue (IS)
 - An instruction is issued if:
 - The functional unit is available and
 - No other active instruction has the same destination register
 - This avoids WAW hazards and structural hazards
 - During a stall, this causes the buffer between IF and IS to fill
 - A one-entry buffer fills quickly!
 - Read operands (RD)

More pipeline changes for scoreboarding

- Read operands (RD)
 - Read operation is delayed until operands are available
 - ⇒ No previously issued but uncompleted instruction has the operand as its destination
 - RAW hazards resolved dynamically
- Execution (EX) stage changed
 - Notify the scoreboard when EX is completed
 - Allow a new instruction to use the functional unit
 - EX may take multiple cycles if necessary

Writeback (WB) with scoreboarding

- The scoreboard checks for WAR hazards and stalls the completing instruction if necessary
 - In the earlier example, SUBD would be stalled in WB until ADDD reads its operands
- Writeback is stalled if
 - A preceding instructions has not read its operands and
 - One of the operands is the same register as the destination of the completing instruction
- The DLX pipeline is now six cycles long
IF IS RD EX MEM WB
 - Forwarding is not used here: not a large penalty since write-back occurs as soon as the result is available
 - Instructions that do NOT need the MEM stage don't execute it

Scoreboard components

- Instruction status
 - Keeps track of the current stage of each instruction
 - There is one entry for each instruction that has passed the IF stage but has not yet completed
- Functional unit status
 - Holds the status of each functional unit
 - “Busy” indicates whether or not the unit is busy
 - “Op” indicates the operation being performed (some functional units can do more than one operation)
 - F_i , F_j and F_k indicate the instruction’s source and destination registers
 - Q_j and Q_k indicate the functional units producing the instruction’s source registers
 - R_j and R_k indicate whether the values are ready (avoid WAR hazards)

Scoreboarding: sample code

- Register result status
 - Holds the ID of the functional unit that will eventually write a register
 - If the register is not the destination of an issued instruction, the field will indicate no functional unit
- For this example, use the code on the right
 - Examine snapshots of the three components of the scoreboard during execution
 - See how hazards are handled

LD F6, 40(R2)

LD **F2**, 52(R3)

MULTD F0, **F2**, F4

SUBD **F8**, F6, **F2**

DIVD F10, F0, F6

ADDD F6, **F8**, **F2**

F0: RAW hazard

F2: RAW hazard

F6: RAW hazard

F8: RAW hazard

Scoreboard: snapshot #1

Instruction status

| Instruction | Issue | Read operands | Exec complete | Write result |
|--|-------|---------------|---------------|--------------|
| LD F6 , 40(R2) | X | X | X | X |
| LD F2 , 52(R3) | X | X | X | |
| MULTD F0 , F2 , F4 | X | | | |
| SUBD F8 , F6 , F2 | X | | | |
| DIVD F10, F0 , F6 | X | | | |
| ADDD F6, F8 , F2 | | | | |

Mult & Sub
waiting for WB

First load complete

Function unit status

| Name | Busy | Op | F _i | F _j | F _k | Q _j | Q _k | R _j | R _k |
|---------|------|------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| Integer | Yes | Load | F2 | R3 | - | - | - | No | - |
| Mult1 | Yes | Mult | F0 | F2 | F4 | Int | - | No | Yes |
| Mult2 | No | | - | - | - | - | - | - | - |
| Add | Yes | Sub | F8 | F6 | F2 | - | Int | Yes | No |
| Divide | Yes | Div | F10 | F0 | F6 | Mult1 | - | No | Yes |

Register result status

| FuncUnit | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 |
|----------|-------|-----|----|----|-----------|-----|-----|-----|-----|
| | Mult1 | Int | | | Sub | Div | | | |

Scoreboard: snapshot #2

Instruction status

| Instruction | Issue | Read operands | Exec complete | Write result |
|--|-------|---------------|---------------|--------------|
| LD F6 , 40(R2) | X | X | X | X |
| LD F2 , 52(R3) | X | X | X | X |
| MULTD F0 , F2 , F4 | X | X | X | X |
| SUBD F8 , F6 , F2 | X | X | X | X |
| DIVD F10, F0 , F6 | X | X | X | X |
| ADDD F6, F8 , F2 | X | X | X | X |

Why can't ADDD finish?

Function unit status

| Name | Busy | Op | F _i | F _j | F _k | Q _j | Q _k | R _j | R _k |
|---------|------|------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| Integer | No | - | - | - | - | - | - | - | - |
| Mult1 | Yes | Mult | F0 | F2 | F4 | - | - | No | No |
| Mult2 | No | - | - | - | - | - | - | - | - |
| Add | Yes | Add | F6 | F8 | F2 | - | - | No | No |
| Divide | Yes | Div | F10 | F0 | F6 | Mult1 | - | No | Yes |

Register result status

| FuncUnit | F0 | F2 | F4 | F6 | F8 | F10 | F12 ... F30 |
|----------|----|----|----|-----|----|-----|-------------|
| Mult1 | | | | Add | | Div | |

Distinguishes RAW from WAR

Scoreboard: snapshot #3

Instruction status

| Instruction | Issue | Read operands | Exec complete | Write result |
|--|-------|---------------|---------------|--------------|
| LD F6 , 40(R2) | X | X | X | X |
| LD F2 , 52(R3) | X | X | X | X |
| MULTD F0 , F2 , F4 | X | X | X | X |
| SUBD F8 , F6 , F2 | X | X | X | X |
| DIVD F10, F0 , F6 | X | X | X | |
| ADDD F6, F8 , F2 | X | X | X | X |

Function unit status

| Name | Busy | Op | F _i | F _j | F _k | Q _j | Q _k | R _j | R _k |
|---------|------|------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| Integer | No | - | - | - | - | - | - | - | - |
| Mult1 | Yes | Mult | - | - | - | - | - | - | - |
| Mult2 | No | - | - | - | - | - | - | - | - |
| Add | Yes | Add | - | - | - | - | - | - | - |
| Divide | Yes | Div | F10 | F0 | F6 | - | - | No | No |

Register result status

| FuncUnit | F0 | F2 | F4 | F6 | F8 | F10 | F12 ... F30 |
|----------|----|----|----|----|----|-----|-------------|
| Div | | | | | | | |

Handling hazards with a scoreboard

- RAW hazards
 - Detect RAW hazards by checking to see if a source register is listed in the Register Result Status table
 - ⇒ If it is, we have a RAW hazard
 - If the pending instruction is receiving a value from the current instruction, then set one of the pending instruction's R_j/R_k fields to **No**
- WAR hazards
 - Before writing the value, check to make sure that no pending instruction is using a previous value for the register to be modified
 - If some pending instruction has already “received” the value it needs but hasn't yet read it, then R_j/R_k is set to **Yes** and any instruction writing the register must stall (WAR)
- This is how we distinguish between a RAW and WAR

Scoreboard limitations

- ILP: if there aren't any independent instructions to execute, scoreboarding and other dynamic techniques don't help much
- Size of the “issued” queue (the **window**)
 - Determines how far ahead the CPU can look for instructions
 - For now, assume that a window cannot span a branch
 - Window includes instructions only within basic blocks
 - The window can be extended beyond the branch: details later
- Number, types, and speed of the functional units
- Presence of antidependences and output dependences
 - WAR and WAW hazards limit scoreboard more than RAW hazards
 - RAW hazards are problems for any technique
 - WAR and WAW hazards can be solved using other mechanisms

Tomasulo's approach

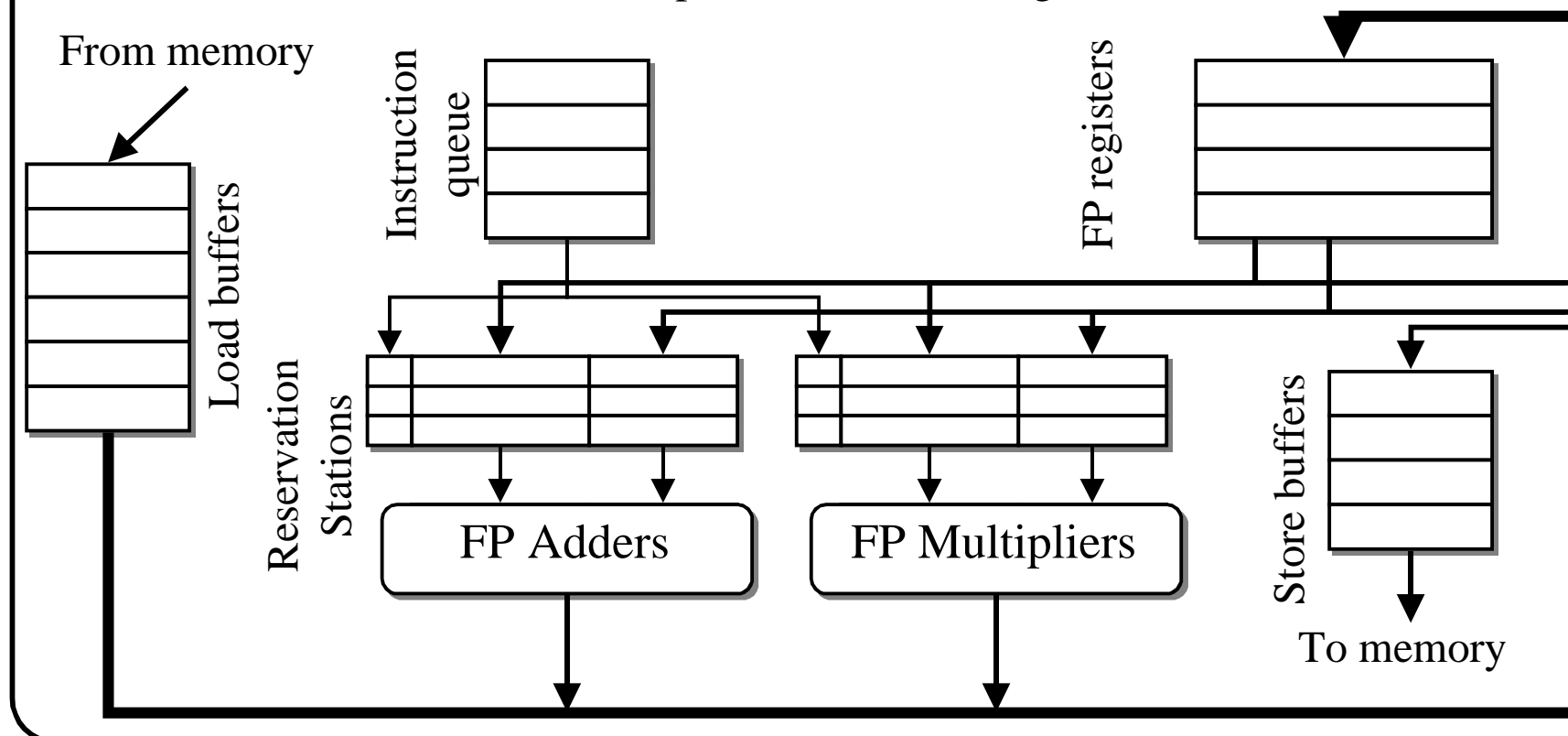
- Tomasulo's approach is a technique to allow execution to proceed in the presence of hazards
 - First introduced in the IBM 360/91
 - Applied only to floating-point operations (including FP memory ops)
- Uses renaming to avoid **WAW** and **WAR** hazards
 - Compiler can rename registers (statically) to avoid **WAW** and **WAR** hazards
 - Tomasulo's scheme performs this function dynamically
 - Buffers operands of instructions waiting to issue, fetching them as soon as they are available, avoiding the register file
 - The register specifiers of instructions are renamed to reservation station numbers as they are issued, *eliminating* **WAW** and **WAR** hazards

Scoreboarding vs. Tomasulo's approach

- Register renaming
 - Register renaming is used to eliminate **WAR** and **WAW** hazards
 - Scoreboarding must wait for **WAR** and **WAW** hazards to clear
- Distributed control
 - Hazard detection and execution control are distributed to each functional unit
 - Scoreboarding has a centralized control unit
- Common Data Bus
 - Used to forward results directly to the functional units without going through the register file
 - Scoreboarding connects each functional unit to the register file

Tomasulo's approach: design

- Reservation stations are the heart of Tomasulo's approach
 - Located at each functional unit (may be more reservations than func units)
 - Hold values for each computation before it begins



Tomasulo's approach: issue stage

- Take an instruction from the instruction queue
 - If there's a station available for it, send the instruction to the station
 - Otherwise, stall for a structural hazard
- This step checks to see if the source operands will be produced by a current instruction
 - If so, renaming is done by checking to see if the desired register is being written by an instruction already at a reservation station
 - If the value is not being generated by a functional unit, it is fetched from the register file
 - If the value is being generated, the name of the reservation station generating the result is used instead
 - If the operation is a load or a store, it can issue if there is an available load or store buffer

Tomasulo's approach: execute & WB

- Execute
 - If at least one operand is missing, monitor the CDB until it is generated
 - When a needed operand is put out onto the CDB, it is placed into the appropriate reservation station
 - When both operands are ready, the operation is executed
 - ⇒ RAW hazards are handled here
- Write result
 - When the result is ready, write it on the CDB and into the register file and any waiting reservation station
 - ⇒ Only one value can be written on the CDB in any single cycle!
 - Indicate that the reservation station is no longer busy

Tomasulo's approach: design details

- Control structures
 - Operation (Op): the operation to be performed.
 - Operand sources (Q_j, Q_k): the reservation stations that will produce the values for the two operands
 - A 0 in either slot means the source operand is already in V_j or V_k , or that the slot is not needed
 - Operand values (V_j, V_k): the values for the two operands.
 - They are valid if and only if the corresponding Q is 0
 - Busy: indicates the reservation station and the accompanying functional unit are busy
- Register file & store buffer
 - Field Q_i for each element: indicates which reservation station is producing the result that will go into this element (0 if blank)

Tomasulo's approach: control example

| Instruction | Issue | Instruction status | | |
|--|-------|--------------------|---------------|--------------|
| | | Read operands | Exec complete | Write result |
| LD F6 , 40(R2) | X | X | X | X |
| LD F2 , 52(R3) | X | X | X | |
| MULTD F0 , F2 , F4 | X | | | |
| SUBD F8 , F6 , F2 | X | | | |
| DIVD F10, F0 , F6 | X | | | |
| ADD F6, F8 , F2 | X | | | |

MULTD & SUBD
waiting for WB
DIVD waiting on MULTD

| Name | Busy | Op | Function unit status | | | |
|-------|------|------|----------------------|----------------|----------------|----------------|
| | | | V _j | V _k | Q _j | Q _k |
| Add1 | Yes | Sub | M[40+Regs[R2]] | - | - | Load2 |
| Add2 | Yes | Add | - | - | Add1 | Load2 |
| Add3 | No | - | - | - | - | - |
| Mult1 | Yes | Mult | - | Regs[F4] | Load2 | - |
| Mult2 | Yes | Div | - | M[40+Regs[R2]] | Mult1 | - |

| | | Register result status | | | | | | | | | |
|----------|-------|------------------------|----|----|------|------|-------|-----|-----|-----|--|
| | | F0 | F2 | F4 | F6 | F8 | F10 | F12 | ... | F30 | |
| FuncUnit | Mult1 | Load2 | | | Add2 | Add1 | Mult2 | | | | |

Tomasulo's approach: advantages

- Hazard detection logic is distributed
 - If multiple instructions are waiting on the second of two operands, the instructions can be released simultaneously broadcasting on the CDB
- WAW and WAR hazards are eliminated because
 - Register renaming is performed using the reservation stations.
 - Operands are stored into the reservation tables as soon as they are available
- The WAR hazard was eliminated because the reservation station held the value of F6 for the DIVD instruction
 - Even if **LD F6, 40(R2)** hadn't completed before the **DIVD** had issued
 - The **WAR** hazard & potential WAW hazard are eliminated
 - Q_k would point to the Load1 reservation table for the value of F6

Tomasulo's approach: loop unrolling

- Loop unrolling is performed dynamically !
 - With only 4 FP registers, WAW and WAR hazards would severely limit loop unrolling, even by the compiler
 - Virtual registers provided by the reservation stations make it possible to execute multiple iterations of some loops simultaneously
 - Memory disambiguation
 - Since the store functional unit keeps a memory address as well as a value, it's possible to do disambiguation
 - When a memory operation is issued, check to see if that location is already involved in an operation
- ⇒ LOADs and STOREs from different iterations of the loop can be executed *non-sequentially*