Instruction-Level Parallelism

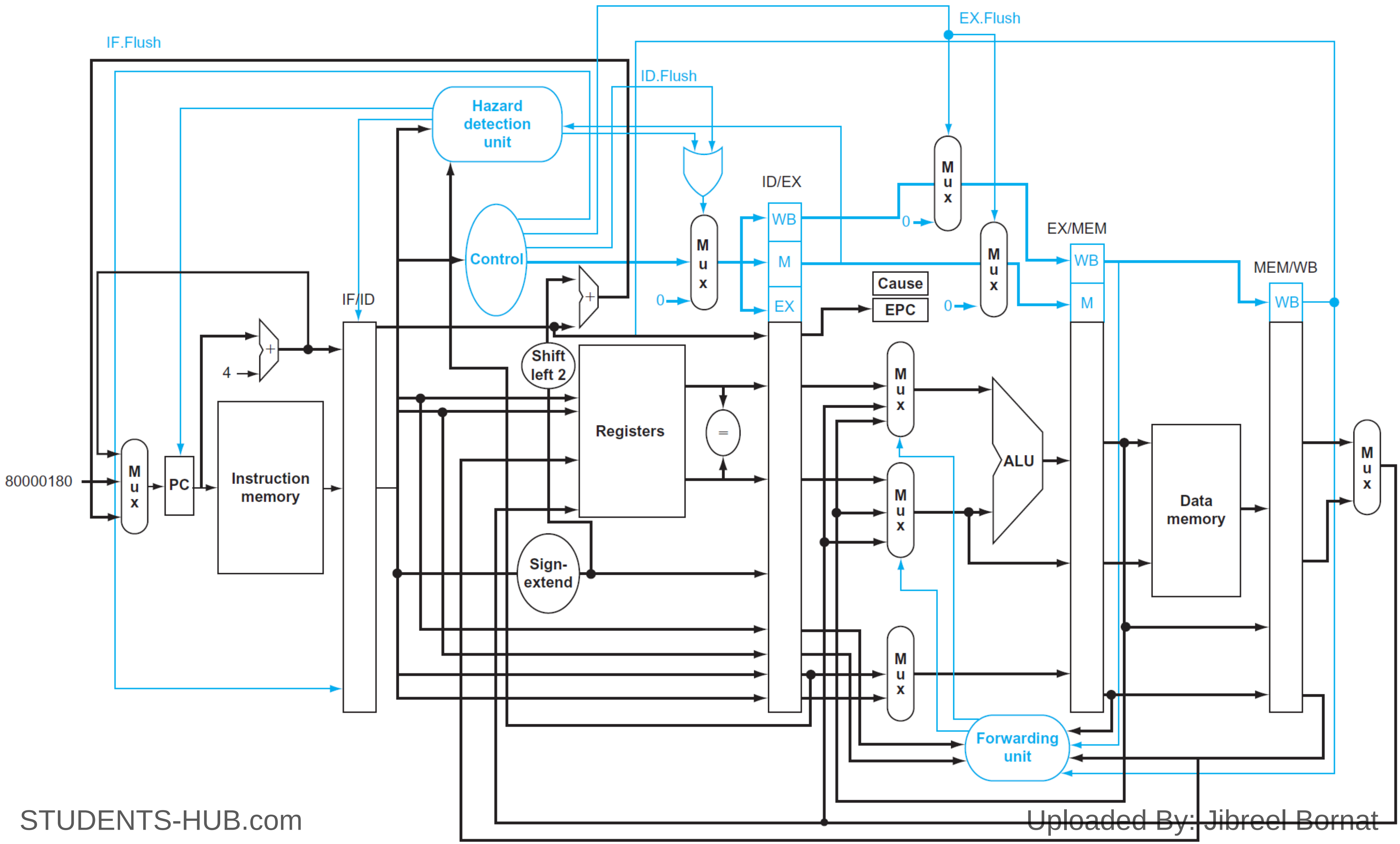
STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Presentation Outline

- ❖ **Diversified Pipeline**
- ❖ Out of Order Execution
- ❖ Tomasulo's Dynamic Scheduling
- ❖ Dealing with Exceptions
- ❖ Reorder Buffer
- ❖ Out of Order Memory Access

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Pipelined Datapath Review



STUDENTS-HUB.com

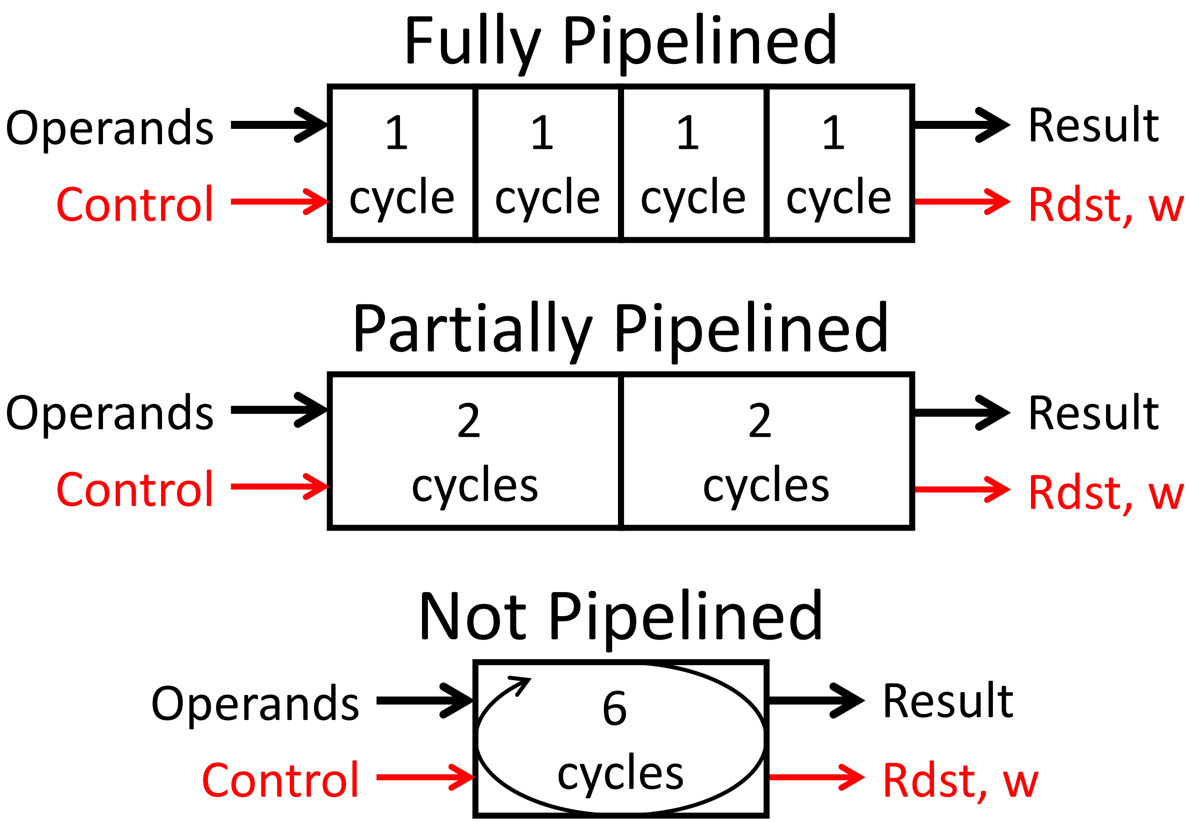
Uploaded By: Jibreel Bornat

What Makes Pipelining Complex?

- Multiple Execution Units
  - Integer ALU, Floating-Point Unit, Multiply, Divide, etc.
- Long-Latency Execution Units
  - Integer Multiplication and Division take longer to execute
  - FP Addition, FP Multiplication, and FP division (multiple cycles)
- Fully pipelined versus non-pipelined arithmetic units
  - Fully pipelined: can issue one instruction every cycle
  - Not pipelined: must wait until arithmetic unit is not busy
- Cache memory system with variable access time
  - Cache miss: must wait for instruction or data
- Dealing with Exceptions
  - Imprecise versus Precise exceptions

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Function Unit Characteristics



Initiate 1 op per cycle
Result after 4 cycles

Initiate 1 op per 2 cycles
Result after 4 cycles

Initiate 1 op per 6 cycles
Result after 6 cycles

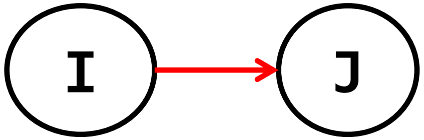
- Function units have internal pipeline registers
  - Operands are latched when an instruction enters a function unit
  - Control specifies operation + destination register
  - A busy signal can be added to control the initiation of instructions

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Program Order and Dependences

- Program Order
  - Sequential order of instructions as defined by source program
  - Hardware and software must preserve program order
- True data dependence
  - Instruction **`I`** computes a result that is used by instruction **`J`**

```
I: add r1, r2, r3
J: sub r4, r1, r5
```



Dependency Graph

  - If instruction **`J`** depends on **`I`** ➔ Cannot execute in parallel
- Effect of true data dependence must be preserved
- If data dependence causes a hazard then it is called
  - Read-After-Write (RAW) data hazard

Uploaded By: Jibreel Bornat

Name Dependences

❖ Name Dependence

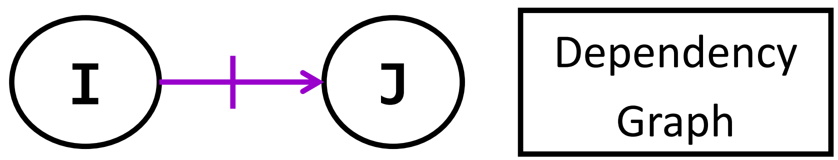
✧ When two instructions use the same register or memory location

✧ But no flow of data between the two instructions

❖ Two types of Name Dependences

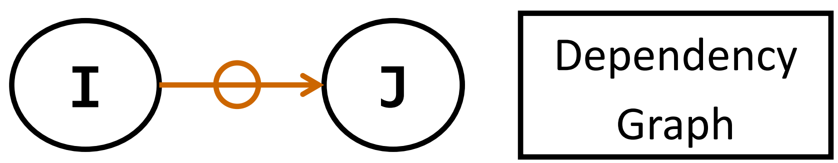
1. Anti-Dependence: **I** reads **r2**, which is later written by **J**

```
I: add r4, r2, r3
J: sub r2, r1, r5
```



2. Output Dependence: **I** writes **r4**, later re-written by **J**

```
I: add r4, r2, r3
J: sub r4, r1, r5
```



❖ Caused by the reuse of same register name

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

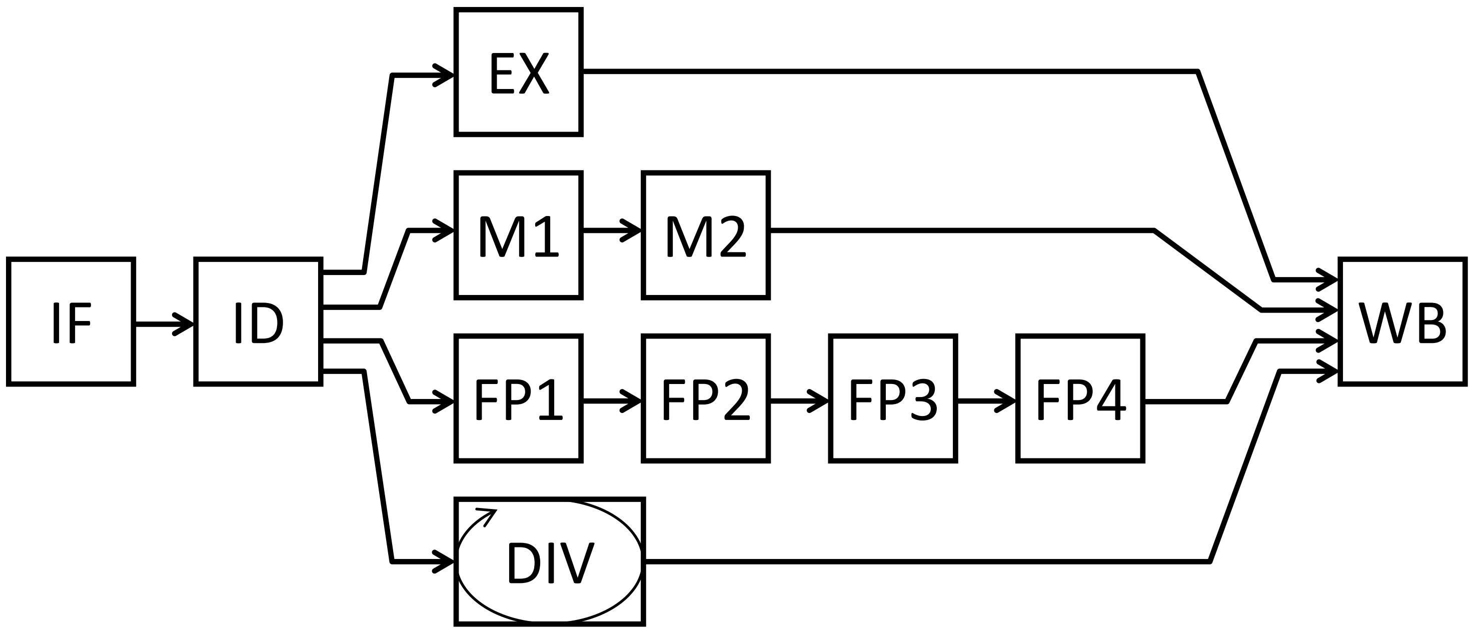
Name Dependences and Hazards

- If anti-dependence causes a hazard then it is called
  - Write-After-Read (WAR) data hazard
- If output-dependence causes a hazard then it is called
  - Write-After-Write (WAW) data hazard
- Dependences are a property of programs
- Hazards are a property of pipeline implementation
- Name dependences can be eliminated by renaming
  - Instructions can execute in parallel if a different name is used
- Renaming can be done by the compiler at compile-time
- Can be done also by the hardware during execution time

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Example of a Diversified Pipeline

- Four functional units are used:
  - Integer Execution unit: 1 pipeline stage (EX)
  - Memory Unit (Data Cache): 2 pipeline stages (M1 and M2)
  - Floating-Point Multiply-Add: 4 pipeline stages (FP1 thru FP4)
  - Divide Unit (Integer and FP): 6 cycles, NOT pipelined (DIV)



STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Timing of Long-Latency Operations

- Timing of Independent Floating-Point Instructions
- In-Order Instruction Fetch, Decode, and Execute
- **Out-of-Order Completion** of Instructions
  - The **FDIV** instruction has the longest execution time (6 cycles)
  - The **FADD** and **LD** complete before **FDIV**
  - Structural Hazards occur if multiple write-backs during same cycle

	Clock Cycle Number											
	1	2	3	4	5	6	7	8	9	10	11	12
FDIV	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	**WB**			
FADD		IF	ID	FP1	FP2	FP3	FP4	**WB**				
LD			IF	ID	M1	M2	**WB**					
FMUL				IF	ID	FP1	FP2	FP3	FP4	**WB**		

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Structural Hazards

- Structural hazards occur when a unit is not pipelined
- The divide (DIV) unit is not pipelined
  - Cannot issue second divide instruction during next clock cycle
- Stall pipeline as long as the DIV functional unit is busy
  - Cannot execute second **FDIV** instruction until the DIV function unit is free
  - Freeze the IF and ID stages until DIV unit is free (5 stall cycles)

	Clock Cycle Number											
	1	2	3	4	5	6	7	8	9	10	11	12
FDIV	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	WB			
FDIV		IF	ID	ID	ID	ID	ID	ID	DIV	DIV	DIV	DIV
LD			IF	IF	IF	IF	IF	IF	ID	M1	M2	WB
FMUL				**5 Stall Cycles**					IF	ID	FP1	FP2

Uploaded By: Jibreel Bornat

Conflicting Writes

- **Out-of-Order completion** causes **conflicting writes**
- Multiple write-backs during the same cycle
- Must detect hazard and delay conflicting writes

	Clock Cycle Number											
	1	2	3	4	5	6	7	8	9	10	11	12
FDIV	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	**WB**			
LD		IF	ID	M1	M2	**WB**						
FMUL			IF	ID	FP1	FP2	FP3	FP4	**WB**			
FADD				IF	ID	FP1	FP2	FP3	FP4	**WB**		
LD					IF	ID	M1	M2	**WB**			
ADD						IF	ID	EX	**WB**			
SUB							IF	ID	EX	**WB**		

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Detecting Conflicting Writes

- Conflicting writes to the register file is a **structural hazard**
  - Function units can produce multiple results in parallel
  - However, one write port to the register file can write one result
- Conflicting writes can be detected early in the ID stage
  - Preventing conflicting writes early in the ID stage
  - Knowing the latency of each function unit
- Conflicting writes can also be detected late in the WB stage
  - The Write-Back stage must resolve the conflict
  - Forcing some function units to stall until write-back is completed
- A third choice is to balance the depth of all function units
  - By adding delay stages and **ensuring In-Order Completion**
  - Prevents conflicting writes, but increases the delay of function units

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Resolving Conflicting Writes

- An instruction in the ID stage is not issued for execution
  - Until it is clear there is no conflicting write
  - A conflicting write causes the instruction in the ID stage to stall

	Clock Cycle Number														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
FDIV	IF	ID	DIV	DIV	DIV	DIV	DIV	DIV	**WB**						
LD		IF	ID	M1	M2	**WB**									
FMUL			IF	ID	ID	FP1	FP2	FP3	FP4	**WB**					
FADD				IF	IF	ID	FP1	FP2	FP3	FP4	**WB**				
LD						IF	ID	ID	ID	M1	M2	**WB**			
ADD							IF	IF	IF	ID	ID	EX	**WB**		
SUB										IF	IF	ID	EX	**WB**	
OR												IF	ID	EX	**WB**

Write-After-Write (WAW) Hazards

- **Out-of-Order Completion** also causes **WAW hazards**
- Write-Backs to the same register can occur out-of-order
- In the example shown below:
  - The **LD** instruction completes **before** **FADD**
  - Final value of **F5** is written by **FADD** not **LD**
  - Causing WAW Hazard
- Writes to the same register should be in program order

		Clock Cycle Number									
		1	2	3	4	5	6	7	8	9	10
FADD	**F5**, F2, F3	IF	ID	FP1	FP2	FP3	FP4	**WB**			
LD	**F5**, 16(R6)		IF	ID	M1	M2	**WB**				

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

RAW Hazards: Stall and Forward

- RAW hazards are caused by long-latency operations
- Stall cycles: waiting for results of long-latency operations
- Implementing Stall-and-Forward can be complex
  - Number of stall cycles can vary according to the unit latencies
  - Forwarding is needed across multiple functional units

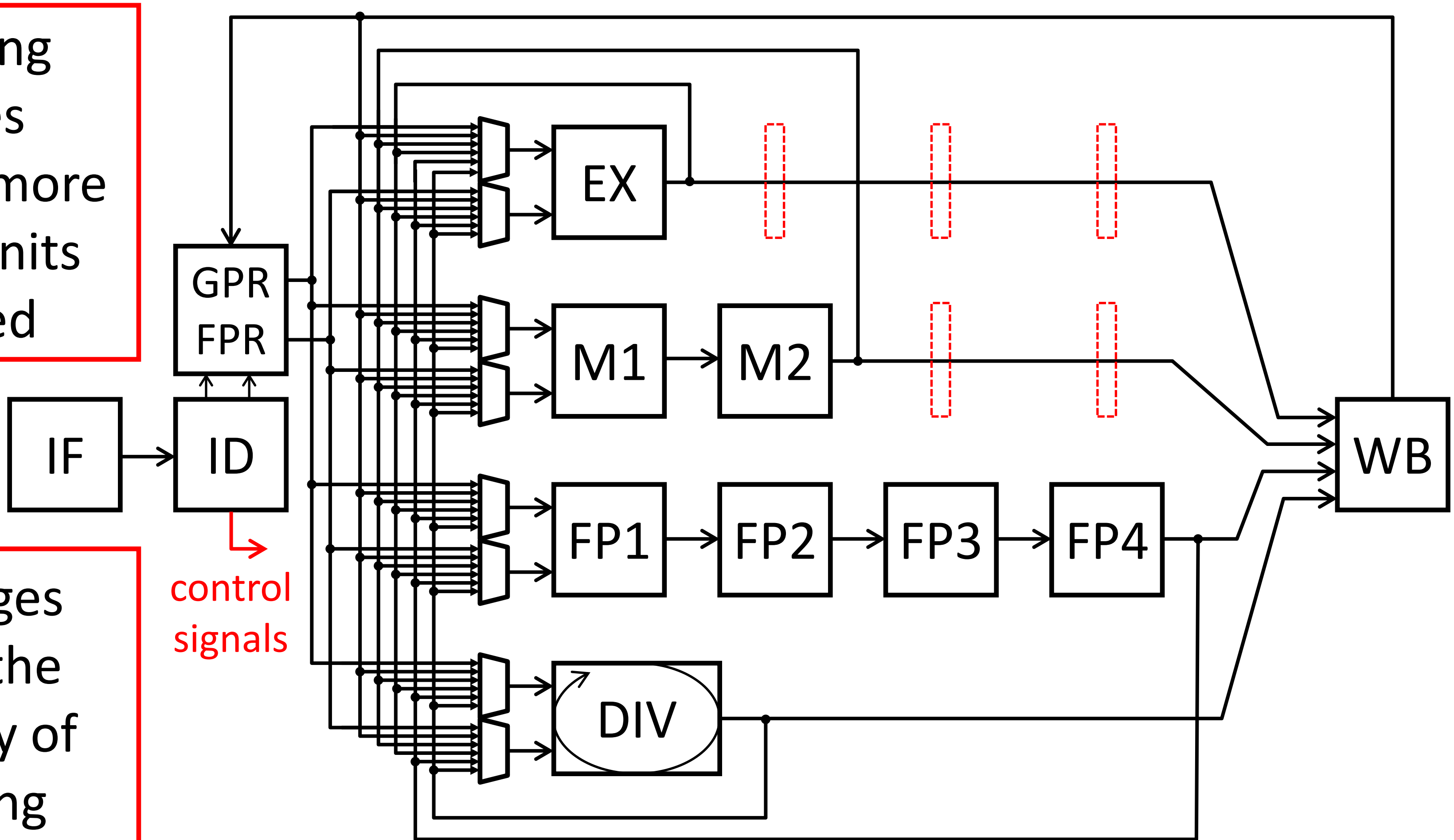
Clock Cycle Number														
		1	2	3	4	5	6	7	8	9	10	11	12	13
LD	**F4**, 0(R7)	IF	ID	M1	M2	WB								
FADD	**F6**, **F4**, F2		IF	ID	ID	FP1	FP2	FP3	FP4	WB				
FMUL	**F5**, **F6**, F3			IF	IF	ID	ID	ID	ID	FP1	FP2	FP3	FP4	WB
ADDI	R7, R7, 8					IF	IF	IF	IF	ID	EX	WB		
SD	**F5**, 16(R8)									IF	ID	ID	ID	M1

Uploaded By: Jibreel Bornat

Complex Forwarding Network

- ❖ Wide multiplexers at the input of each function unit
- ❖ Must detect RAW hazards and control each multiplexer

Forwarding becomes complex if more function units are added



Delay stages increase the complexity of forwarding

Uploaded By: Jibreel Bornat

Presentation Outline

- **Diversified Pipeline**
- **Out of Order Execution**
- Tomasulo's Dynamic Scheduling
- Dealing with Exceptions
- Reorder Buffer
- Out of Order Memory Access

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Motivation for Out-of-Order Execution

- So far, instructions are …
  - Fetched, decoded, and executed in program order
  - But complete out-of-order
- Consider the following example:

```
DIV.D    F0, F1, F2   ; Long latency operation
ADD.D    F4, F0, F3   ; Wait for DIV.D
L.D      F5, 0(R4)
L.D      F6, 8(R4)
SUB.D    F7, F6, F5   ; Wait for both L.D
```

Do not depend on the result of DIV.D or ADD.D

- If in-order execution then **ADD.D** will stall the pipeline
- But the two **L.D** instructions can execute

Uploaded By: Jibreel Bornat

Out-of-Order Execution

- Also known as Dynamic Scheduling
  - Done by the processor pipeline at runtime
  - Instructions execute based on the availability of their operands
  - In contrast, compiler scheduling is called Static Scheduling
- OOO execution must preserve sequential semantics
  - Instructions with dependences **must wait**
  - But should allow later independent instructions to execute
- Must have an Issue Stage in the processor pipeline
  - Issue stage detects when an instruction can begin execution
  - Based on the availability of its operands
- No complex forwarding network
  - Only the WB stage forwards the result to the Issue Stage

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Example of Out-of-Order Execution

Clock Cycle Number																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
DIV.D	**F0**, F1, F2	IF	ID	**IS**	DIV	DIV	DIV	DIV	DIV	DIV	WB					
ADD.D	F4, **F0**, F3		IF	ID	**IS**	**IS**	**IS**	**IS**	**IS**	**IS**	**IS**	FP1	FP2	FP3	FP4	WB
L.D	**F5**, 0(R4)			IF	ID	**IS**	M1	M2	WB							
L.D	**F6**, 8(R4)				IF	ID	**IS**	M1	M2	WB						
SUB.D	F7, **F6**, **F5**					IF	ID	**IS**	**IS**	**IS**	FP1	FP2	FP3	FP4	WB	

- A new Issue Stage (**IS**) is added to the pipeline
- Instructions wait in the Issue Stage if they cannot execute
  - ADD.D waits for the result of DIV.D (RAW hazard)
  - SUB.D waits for the two L.D instructions (RAW hazard)
  - Can have **many waiting instructions** in **IS without stalling pipeline**

 Uploaded By: Jibreel Bornat

Advantages of Out-of-Order Execution

- Overcomes the limitations of in-order execution pipelines
- Allows non-dependent instructions that come after to proceed
- Hardware rearranges instructions to **reduce stall cycles**
- Hardware detects and handles all types of dependences
- Works even when dependences are not known at compile time
- Multiple execution units can be used in parallel
- Simplifies the compiler
- Code for one pipeline runs well on another pipeline

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Write-After-Read (WAR) Hazards

- Out-of-Order execution can cause **WAR hazards**
- In the example shown below:
  - The ADD.D instruction depends on the result of DIV.D
  - ADD.D waits in the Issue Stage and begins execution at cycle 11
  - L.D does NOT depend on the result of DIV.D or ADD.D
  - L.D completes and writes **F3 before** ADD.D begins execution
  - ADD.D reads the wrong value of **F3** (value written by L.D)

Clock Cycle Number																
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
DIV.D	**F0**, F1, F2	IF	ID	**IS**	DIV	DIV	DIV	DIV	DIV	DIV	WB					
ADD.D	F5, **F0**, **F3**		IF	ID	**IS**	**IS**	**IS**	**IS**	**IS**	**IS**	**IS**	FP1	FP2	FP3	FP4	**WB**
L.D	**F3**, 8(R4)			IF	ID	**IS**	M1	M2	**WB**							

Uploaded By: Jibreel Bornat

Register Renaming

- OOO execution causes **WAR and WAW hazards**
  - Because of name dependences, NOT true data dependences
- Name dependences can be eliminated with **Register Renaming**
  - **Eliminates WAR and WAW hazards**
  - Done by the hardware at runtime using extra registers: **X0**, **X1**, etc.
- Example on register renaming

```
DIV.D   F0, F1, F2
ADD.D   F5, F0, F3
L.D     F0, 0(R4)
L.D     F3, 8(R4)
MUL.D   F5, F0, F3
```

```
DIV.D   F0, F1, F2
ADD.D   F5, F0, F3
L.D     X0, 0(R4)
L.D     X1, 8(R4)
MUL.D   X2, X0, X1
```

- With register renaming, only true data dependences remain

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Presentation Outline

- ❖ **Diversified Pipeline**
- ❖ **Out of Order Execution**
- ❖ **Tomasulo's Dynamic Scheduling**
- ❖ Dealing with Exceptions
- ❖ Reorder Buffer
- ❖ Out of Order Memory Access

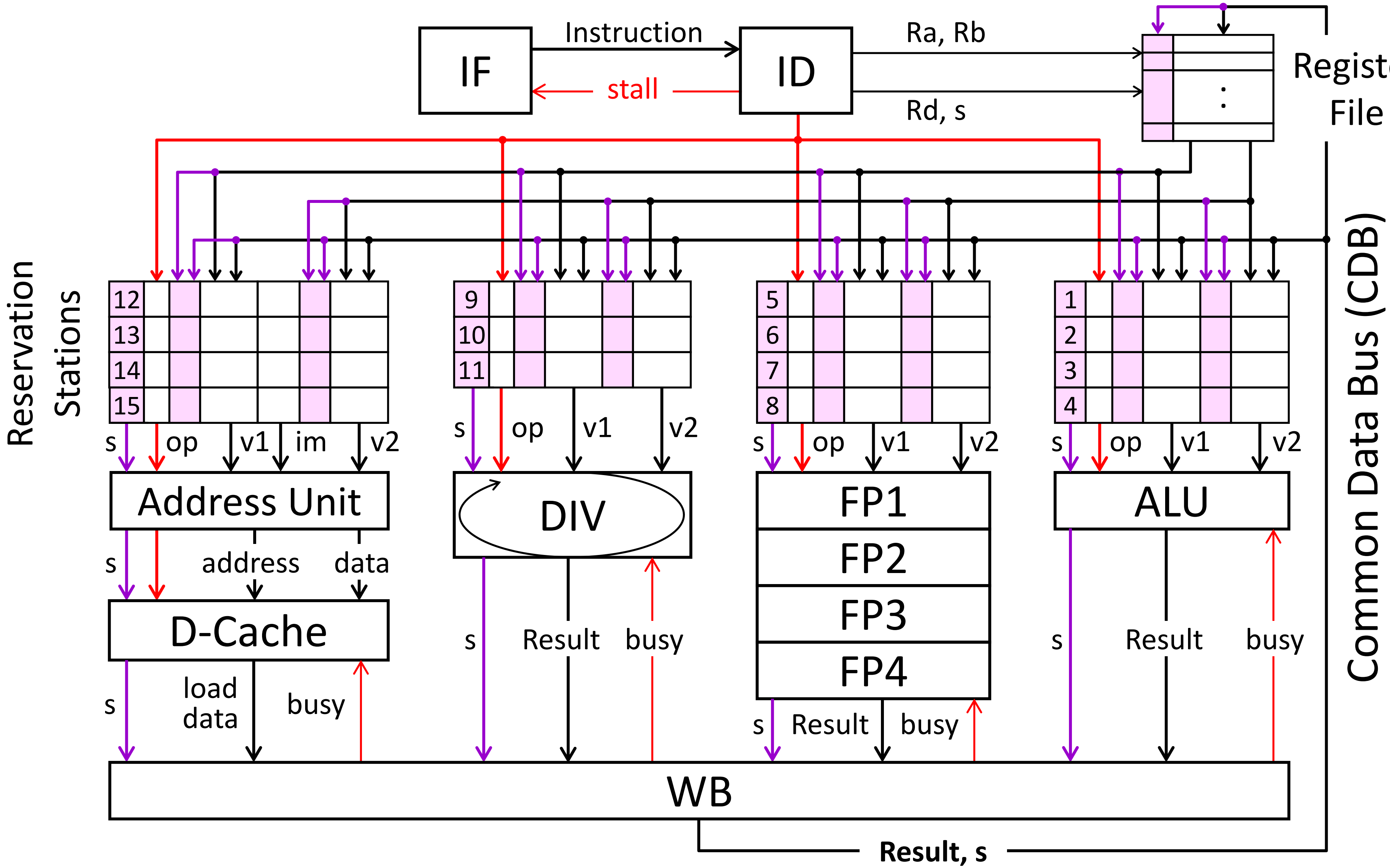
STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Tomasulo's Dynamic Scheduling

- Reservation Stations (RS)
  - A reservation station is a buffer for an instruction and its operands
  - An instruction **waits** in a reservation station until its operands are available
  - A reservation station number renames the destination register
  - Number of reservation stations can exceed the number of registers
- Common Data Bus (CDB)
  - Broadcasts a result to all waiting reservation stations
- First implementation: IBM 360/91 (1967)
  - Dynamic scheduling for Floating-Point units only
  - IBM 360/91 had 4 Floating-Point registers only!
- Our example of a Tomasulo Pipeline
  - Dynamic scheduling for everything: ALU, FPU, and load/store
  - One register file for general-purpose and floating-point registers

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Example of a Tomasulo Pipeline



STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Reservation Stations

- Each reservation station RS[s] has a unique number **s**
  - Used as a **tag** to **rename** and **identify** the result of an instruction
  - There is **NO zero tag** (reserved for special use)
- Each reservation station has the following fields:
  - **T: Tag** = a unique number **s** identifying each reservation station
  - **Busy:** Indicates whether the reservation station is busy or free
  - **Op:** Operation of the instruction waiting in this reservation station
  - **V1, V2:** Source operand values
    - I-type ALU instructions put the **immediate constant in V2**
  - **T1, T2:** Source operand tags (stations that will produce value)
    - If **T1** (or **T2**) is **zero** then the value is **present** in **V1** (or **V2**)
    - If both **T1** and **T2** are **zero**, the instruction is **ready to execute**
  - Load/store instructions need an immediate field to compute address

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Register File

- ❖ One register file is used for both integer and FP registers
- ❖ Each entry in the register file has two fields:
  - ✧ **T = Tag** (rename) of a register = reservation station number **s**
  - ✧ **V = Value** of a register in the register file
- ❖ The register Tags are also known as the Rename table
  - ✧ If **T** is **s** then the instruction waiting in RS[**s**] will compute value
  - ✧ If **T** is **zero** then the register value **V** is present

Register File for general-purpose and floating-point registers

R1	T = tag	V = value
:	:	:
R31	T = tag	V = value
F0	T = tag	V = value
:	:	:
F31	T = tag	V = value

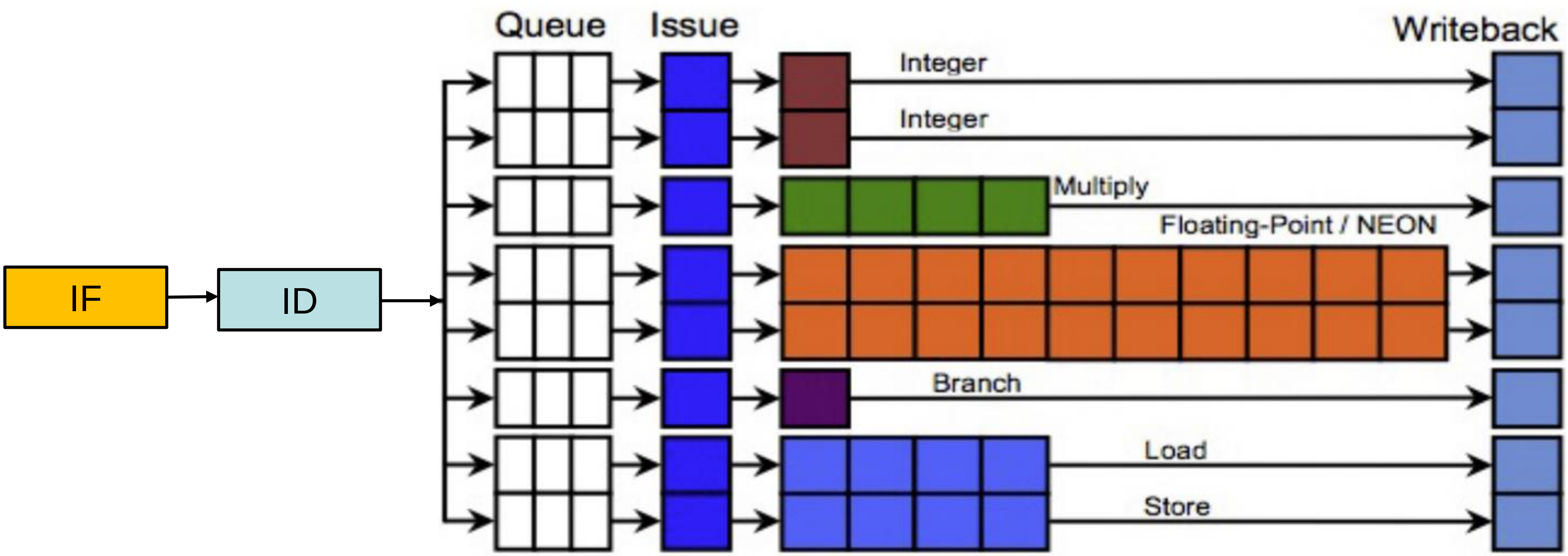
STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Tomasulo's Algorithm: Four Steps

1. Dispatch: in-order
   - Instruction in ID allocates free reservation station **s**: Rs[**s**].Busy ← **1**
   - Register file is read and operand values are copied to RS[**s**]
   - Destination register **Rd** is **renamed** to reservation station **s**: Reg[**Rd**].T ← **s**
2. Issue: out-of-order
   - If input operands are **ready** in RS[**s**], issue and begin execution
   - If not, instruction **waits** in RS[**s**] and monitors the common data bus
3. Execute: out-of-order
   - Execute instruction in RS[**s**] and compute its Result
4. Write Back: out-of-order
   - Forward Result to stations with a source operand matching tag **s**
   - Write back Result to register with matching tag **s**
   - Free reservation station **s**: Rs[**s**].Busy ← **0**

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Tomasulo's Algorithm: Four Steps



Uploaded By: Jibreel Bornat

Instruction Dispatch Stage

- ID Stage = Instruction Decode and Dispatch Stage
- Instructions are decoded and dispatched in program order
- Check for a free reservation station **s**
  - If station **s** is free (Busy is **0**) then Rs[**s**].Busy ← **1**; RS[**s**].Op ← Operation
  - Let **Ra**, **Rb** = source registers in the register file
  - Copy the source register tags and values into station **s**
  - RS[**s**].T1 ← Reg[**Ra**].T RS[**s**].V1 ← Reg[**Ra**].V
  - RS[**s**].T2 ← Reg[**Rb**].T RS[**s**].V2 ← Reg[**Rb**].V
  - If instruction has no 2nd source register then: RS[**s**].T2 ← 0
  - Load and Store instructions: RS[**s**].imm ← immediate (offset)
  - Let **Rd** = destination register, then Reg[**Rd**].T ← **s**
- If no free reservation station then **stall** (structural hazard)

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Issue Stage

- Instructions wait in reservation stations in the Issue stage
- An instruction in station **s** is ready to execute if
  - Both operands are present: (RS[**s**].T1 == 0 and RS[**s**].T2 == 0)
- Issue instruction if corresponding function unit is **NOT busy**
  - If corresponding function unit is busy then **wait** (structural hazard)
- Issued Instruction = (**s**, RS[**s**].op, RS[**s**].V1, RS[**s**].V2)
- Can issue instructions in parallel to different function units
  - However, cannot issue multiple instructions to same function unit
- After an instruction is issued, then it is executed
  - Execution might take one or multiple cycles
  - Function unit may or may not be pipelined

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Write Back Stage

- Write Back stage receives the results of all function units
- Write Back stage detects and resolves conflicting writes
  - Each cycle, the WB stage selects the result of one function unit
  - Places **Result** and its station number **s** on the common data bus
  - Forces other function units to wait if conflicting write (busy signal)
- All reservation stations monitor the common data bus
  - ∀ **x**, if (RS[**x**].T1 == **s**) { RS[**x**].V1 ← **Result**; RS[**x**].T1 ← 0 }
  - ∀ **x**, if (RS[**x**].T2 == **s**) { RS[**x**].V2 ← **Result**; RS[**x**].T2 ← 0 }
- The register file also monitors the common data bus
  - ∀ **x**, if (Reg[**x**].T == **s**) { Reg[**x**].V ← **Result**; Reg[**x**].T ← 0 }
- Free reservation station **s**: Busy[**s**] ← 0

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Improving the Write-Back Stage

- Drawback of Write-Back: tag **s** must be compared against
  - **All tags** in all reservation stations to **detect data dependencies**
  - **All tags** in the rename table of the register file
- Requires many comparators and consumes a lot of energy
- Writing result to the register file **can be improved**
  - Result must be written to destination register **Rd**
  - During instruction dispatch, store **Rd** in reservation station entry **s**
  - During write-back: **Result**, **s**, and **Rd** should appear on the result bus
- Check the tag of register **Rd** in the register file
  - if (Reg[**Rd**].T == **s**) { Reg[**Rd**].V ← **Result**; Reg[**Rd**].T ← **0** }
- Only **one tag** is checked in the register file, not all of them

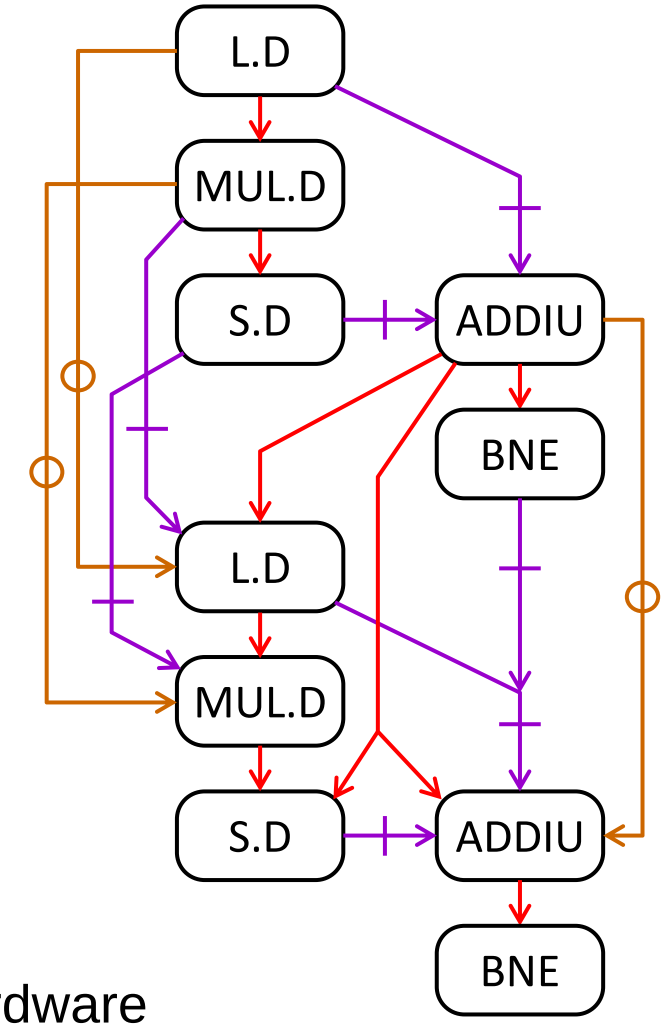
STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Tomasulo Example

- Consider the following loop example:

```
Loop:  L.D      F1, (R4)
       MUL.D    F2, F1, F0
       S.D      F2, (R4)
       ADDIU    R4, R4, 8
       BNE      R4, R5, Loop
```

- Dependence Graph (for 2 iterations)
  - Data + Name Dependences within iteration
  - Data + Name Dependences across iterations
- Assume that BNE is predicted to be taken
  - The loop can be unrolled dynamically by the hardware
  - Renaming eliminates all WAW and WAR hazards



Uploaded By: Jibreel Bornat

Tomasulo Example: Cycles 3 to 5

Cycle 3

	B	Op	T1	V1	T2	V2	Rd
12	1	L.D	0	1000	0	-	F1
13	0	-	0	-	0	-	-

Cycle 4

	B	Op	T1	V1	T2	V2	Rd
5	1	MUL.D	12	?	0	2.0	F2
6	0	-	0	-	0	-	-

Cycle 5

	B	Op	T1	V1	T2	V2	Rd
12	1	L.D	0	1000	0	-	F1
13	1	S.D	0	1000	5	?	-

Register File

Cycles: 3, 4, and 5

	T	Value
R4	0	1000
R5	0	1800
F0	0	2.0
F1	12	?
F2	0	-

	T	Value
R4	0	1000
R5	0	1800
F0	0	2.0
F1	12	?
F2	5	?

	T	Value
R4	0	1000
R5	0	1800
F0	0	2.0
F1	12	?
F2	5	?

Clock Cycle Number

		1	2	3	4	5	6	7	8	9	10
L.D	F1, (R4)	IF	ID	IS	M1	M2	WB				
MUL.D	F2, F1, F0		IF	ID	IS	IS	IS	FP1	FP2	FP3	FP4
S.D	F2, (R4)			IF	ID	IS	IS	IS	IS	IS	IS
ADDIU	R4, R4, 8				IF	ID	IS	EX	WB		
BNE	R4, R5, Loop					IF	ID	IS	IS	EX	

STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Tomasulo Example: Cycles 6 to 8

Cycle 6

	B	Op	T1	V1	T2	V2	Rd
1	1	ADDIU	0	1000	0	8	R4
2	0	-	0	-	0	-	-

Cycle 7

	B	Op	T1	V1	T2	V2	Rd
1	1	ADDIU	0	1000	0	8	R4
2	1	BNE	1	?	0	1800	-
5	1	MUL.D	0	5.3	0	2.0	F2

Cycle 8

	B	Op	T1	V1	T2	V2	Rd
12	0	L.D	0	1000	0	-	F1
13	1	S.D	0	1000	5	?	-
14	1	L.D	1	?	0	-	F1

Register File

Cycles: 6, 7, and 8

	T	Value
R4	1	?
R5	0	1800
F0	0	2.0
F1	12	?
F2	5	?

	T	Value
R4	1	?
R5	0	1800
F0	0	2.0
F1	0	5.3
F2	5	?

	T	Value
R4	0	?
R5	0	1800
F0	0	2.0
F1	14	?
F2	5	?

Clock Cycle Number

		4	5	6	7	8	9	10	11	12	13
L.D	F1, (R4)	M1	M2	WB							
MUL.D	F2, F1, F0	IS	IS	IS	FP1	FP2	FP3	FP4	WB		
S.D	F2, (R4)	ID	IS	IS	IS	IS	IS	IS	IS	M1	M2
ADDIU	R4, R4, 8	IF	ID	IS	EX	WB					
BNE	R4, R5, Loop		IF	ID	IS	IS	EX				
L.D	F1, (R4)			IF	ID	IS	M1	M2	WB	WB	

Conflicting Write

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Tomasulo Example: Cycles 9 and 10

Cycle 9

	B	Op	T1	V1	T2	V2	Rd
2	1	BNE	0	1008	0	1800	-
5	1	MUL.D	0	5.3	0	2.0	F2
6	1	MUL.D	14	?	0	2.0	F2
14	1	L.D	0	1008	0	-	F1

Cycle 10

	B	Op	T1	V1	T2	V2	Rd
13	1	S.D	0	1000	5	?	-
14	1	L.D	0	1008	0	-	F1
15	1	S.D	0	1008	6	?	-

	T	Value
R4	0	1008
R5	0	1800
F0	0	2.0
F1	14	?
F2	6	?

Register File

Cycles: 9 and 10

	T	Value
R4	0	1008
R5	0	1800
F0	0	2.0
F1	14	?
F2	6	?

Clock Cycle Number

		7	8	9	10	11	12	13	14	15	16	17
MUL.D	F2, F1, F0	FP1	FP2	FP3	FP4	WB						
S.D	F2, (R4)	IS	IS	IS	IS	**IS**	M1	M2				
ADDIU	R4, R4, 8	EX	WB									
BNE	R4, R5, Loop	IS	**IS**	EX								
L.D	F1, (R4)	ID	**IS**	M1	M2	WB	WB					
MUL.D	F2, F1, F0	IF	ID	IS	IS	IS	**IS**	FP1	FP2	FP3	FP4	WB
S.D	F2, (R4)		IF	ID	IS	IS	IS	IS	IS	IS	IS	**IS**

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Tomasulo Example: Cycles 1 to 20

Iteration 1, 2		1	2	3	4	5	6	7	8	9	10	11	12	13
L.D	F1, (R4)	IF	ID	**IS**	M1	M2	WB							
MUL.D	F2, F1, F0		IF	ID	IS	IS	**IS**	FP1	FP2	FP3	FP4	WB		
S.D	F2, (R4)			IF	ID	IS	IS	IS	IS	IS	IS	**IS**	M1	M2
ADDIU	R4, R4, 8				IF	ID	**IS**	EX	WB					
BNE	R4, R5, Loop					IF	ID	IS	**IS**	EX				
L.D	F1, (R4)						IF	ID	**IS**	M1	M2	WB	WB	
MUL.D	F2, F1, F0							IF	ID	IS	IS	IS	**IS**	FP1

Conflicting Write

Iteration 2, 3		8	9	10	11	12	13	14	15	16	17	18	19	20
MUL.D	F2, F1, F0	ID	IS	IS	IS	**IS**	FP1	FP2	FP3	FP4	WB			
S.D	F2, (R4)	IF	ID	IS	IS	IS	IS	IS	IS	IS	**IS**	M1	M2	
ADDIU	R4, R4, 8		IF	ID	**IS**	EX	WB							
BNE	R4, R5, Loop			IF	ID	IS	**IS**	EX						
L.D	F1, (R4)				IF	ID	**IS**	M1	M2	WB				
MUL.D	F2, F1, F0					IF	ID	IS	IS	**IS**	FP1	FP2	FP3	FP4
S.D	F2, (R4)						IF	ID	IS	IS	IS	IS	IS	IS
ADDIU	R4, R4, 8							IF	ID	**IS**	EX	WB		
BNE	R4, R5, Loop								IF	ID	IS	**IS**	EX	

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Advantages of the Tomasulo Approach

- Register Renaming
  - Uses reservation station numbers to rename destination registers
  - Eliminates all WAR and WAW hazards
  - Preserving true data dependences only
- Allows out-of-order execution based on data availability
  - Instructions wait in reservation stations
  - Until their operands are available (RAW hazards)
  - Tomasulo builds data dependency graph on the fly
- Broadcasts a result to many waiting instructions
  - Uses a common data bus to forward a result to waiting instructions
  - Uses tags to identify which instructions should grab the result
- Allows dynamic loop unrolling
  - This requires an accurate branch predictor

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Drawbacks of the Tomasulo Approach

- Implementation Complexity
  - Delays in the introduction of IBM 360/91, MIPS 10000, etc.
- Fully associative reservation stations
  - Should compare (at high speed) the result tag against all the tags stored in the reservation stations
  - Resulting in high energy consumption
- Performance limited by one Common Data Bus
  - Can have multiple busses for multiple results computed in parallel
  - However, this will increase the complexity of the reservation stations and requires more comparators for tag checking
- Out-of-Order Completion
  - Causes Imprecise Exceptions (discussed next)

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Presentation Outline

- ❖**Diversified Pipeline**
- ❖**Out of Order Execution**
- ❖**Tomasulo's Dynamic Scheduling**
- ❖**Dealing with Exceptions**
- ❖Reorder Buffer
- ❖Out of Order Memory Access

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Exceptions and Interrupts

- Unexpected events requiring change in flow of control
  - Different architectures use different terminology
- Exceptions (Synchronous)
  - Caused by the program instructions
  - Undefined opcode, overflow, page fault, system call, etc.
- Interrupts (Asynchronous)
  - Caused by I/O devices requesting the processor
  - Not related to program execution
- Exceptions and Interrupts complicate the implementation
  - They require immediate attention and special handling

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Types of Exceptions

- Undefined Instruction
- Memory protection violation (illegal access to memory)
- Page fault (requested page is not in memory)
- Misaligned memory access (e.g. misaligned word)
- Integer arithmetic overflow
- Floating-point exceptions (overflow, underflow, etc.)
- System call instruction (calling the operating system)
- Breakpoint instruction (calling the debugger)
- Tracing instruction execution (trap after each instruction)

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Types of Interrupts

- I/O Device Request
  - Called hardware or device interrupt
  - Caused by devices external to the CPU
    - Such as disk controller, keyboard, mouse, timer, etc.
  - Can be handled after the completion of the current instructions
  - Current instructions in the pipeline are allowed to complete
  - Program is resumed after handling the interrupt
- Hardware Malfunctions or Power Failure
  - Terminate program execution

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Detecting Exceptions in the Pipeline

- Instruction TLB for I-Cache
  - Translates instruction addresses for instruction fetch
  - Detects page faults and memory protection violation
- Instruction Decode
  - Examines instruction opcode and detects undefined instructions
- Arithmetic and Logic Unit
  - Detects integer arithmetic overflow
- Floating-Point Unit
  - Detects floating-point arithmetic exceptions
- Data TLB for D-Cache
  - Translates data addresses for load/store instructions
  - Detects page faults, memory protection, and misaligned access

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Handling Exceptions and Interrupts

- Exceptions and Interrupts are handled by the OS kernel
- Processor saves the address of the offending instruction
  - In a special register, called the Exception Program Counter (EPC)
- Processor saves the cause of the exception
  - In a special register, called the Cause register
  - For address faults, processor also saves the Virtual Address
- Processor jumps to the exception (or interrupt) handler
  - The handler is a program that runs in the OS kernel (supervisor level)
  - The handler saves the EPC and the register state
  - When done, the handler restores the EPC and the register state
- Return From Exception instruction
  - This instruction restarts the program at PC ← EPC (in user level)

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Precise Exceptions and Interrupts

- Can be viewed as an implicit call to a handling routine
- Inserted between two instructions: $\mathbf{I}_{n-1}$ and $\mathbf{I}_n$
- All instructions up to and including $\mathbf{I}_{n-1}$ have completed
- No effect on instruction $\mathbf{I}_n$ or after has taken place
- The handling routine either aborts the program
- Or restarts the program at instruction $\mathbf{I}_n$
- I/O Device Interrupts can be forced to be precise
  - Even though instructions complete out-of-order
  - Wait for all instructions in the pipeline to complete (including $\mathbf{I}_{n-1}$)
- However, it is difficult to achieve precise exceptions
  - When instructions complete out-of-order

Imprecise Exceptions

- Caused by Out-of-Order Completion
- Consider the following example:

```
MUL.D   F4, F2, F3
SUB.D   F5, F4, F1
ADDU    R8, R8, R5
```

Handle MUL.D exception
Then restart at SUB.D
ADDU will re-compute R8
Wrong value in R8!

- Suppose that **MUL.D** causes a floating-point exception
  - **ADDU** has completed but **SUB.D** did not start execution yet!

		Clock Cycle Number										
		1	2	3	4	5	6	7	8	9	10	11
MUL.D	F4, F2, F3	IF	ID	IS	FP1	FP2	FP3	FP4	**FP Exception**			
SUB.D	F5, F4, F1		IF	ID	IS	IS	IS	IS	**Did not execute**			
ADDU	R8, R8, R5			IF	ID	IS	EX	**WB**	**Completed**			

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Dealing with Imprecise Exceptions

- Do nothing
  - Makes debugging difficult → NOT acceptable!
  - Makes page faults difficult → NOT acceptable!
  - IEEE FP standard strongly suggests precise exceptions
- Let the exception handler fix the problem
  - Exception handler can create a precise sequence
  - Knowing what instructions were in the pipeline and their addresses
  - Handler can simulate and complete these instructions
  - Major difficulty is the complexity added to the exception handler
- Ensure In-Order Completion
  - Instructions execute out-of-order, but complete in program order
  - Use a Reorder buffer and a new Commit Stage (discussed next)

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Presentation Outline

- ❖ **Diversified Pipeline**
- ❖ **Out of Order Execution**
- ❖ **Tomasulo's Dynamic Scheduling**
- ❖ **Dealing with Exceptions**
- ❖ **Reorder Buffer**
- ❖ Out of Order Memory Access

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

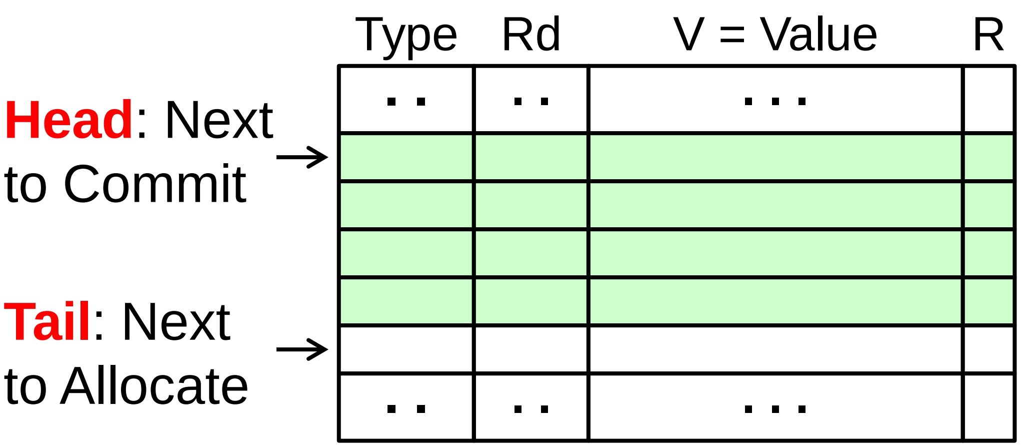
Reorder Buffer (ROB)

- Must split the Write stage
- Instructions can finish execution out of program order
- However, should only commit results in program order
- Commit ➔ Update register file or memory in program order
  - To support precise exceptions
  - To support speculation, such as dynamic branch prediction
- Requires an additional stage, called Commit or Retire
  - Commit is the last stage in the pipeline, after Write-Back
  - Instructions commit their results in program order
- Requires an additional Reorder Buffer to hold results
  - Results of instructions that finished execution
  - But have not updated the register file or memory

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Reorder Buffer: a Circular Queue

- Reorder Buffer holds instructions in FIFO order
  - Implemented as a **circular queue**
- Dispatch stage allocates an entry at the **Tail** of the ROB
  - If reorder buffer is full then stall instruction fetching
- Commit stage frees the entry at the **Head** of the ROB
  - The result value is written in the register file or stored in memory



Reorder Buffer (ROB)

If the instruction at the Head of ROB causes an exception, then clear ROB by changing the tail pointer: Tail = Head

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Reorder Buffer Entry

❖ Each entry in the ROB contains four fields:

1. Instruction Type
   - ✧ Branch type: indicates prediction (No destination register)
   - ✧ Store type: write to memory (No destination register)
   - ✧ Load, ALU, or FPU type: write result to destination register
2. Destination Register: Rd
   - ✧ Destination register for Load, ALU and FPU instructions
3. Value: V
   - ✧ Value of instruction result until the instruction commits
4. Ready Bit: R
   - ✧ Whether instruction has finished execution and **value is present**

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Register File

- ❖ One register file is used for both integer and FP registers
- ❖ Each entry in the register file has two fields:
  - ✧ T = Tag (rename) of a register = **reorder buffer entry**
  - ✧ V = Value of a register in the register file
- ❖ The register Tags are also known as the Rename table
  - ✧ If **T** is **zero** then the value V is present in the register file
  - ✧ If **T** is **non-zero** then the instruction result will be written to ROB[**T**]

Register File for general-purpose and floating-point registers

R1	T = tag	V = value
:	:	:
R31	T = tag	V = value
F0	T = tag	V = value
:	:	:
F31	T = tag	V = value

Uploaded By: Jibreel Bornat

Renaming Registers with ROB entries

- Each ROB entry has a unique index **T**
- Tag **T** is used to rename the destination register of an instruction
  - **Tag zero is NOT used** ➔ Indicates that value is present in register file
- If the Tag **T** of a source register is **zero** in the rename table then
  - Read the source operand value from the **register file**
- If the Tag **T** of a source register is **non-zero** in the rename table
  - Read the source operand value from ROB[**T**] entry
  - If the Value is **NOT ready** in ROB[**T**] then it has not been computed yet
- Execution units use the tag **T** to write computed result in ROB[**T**]

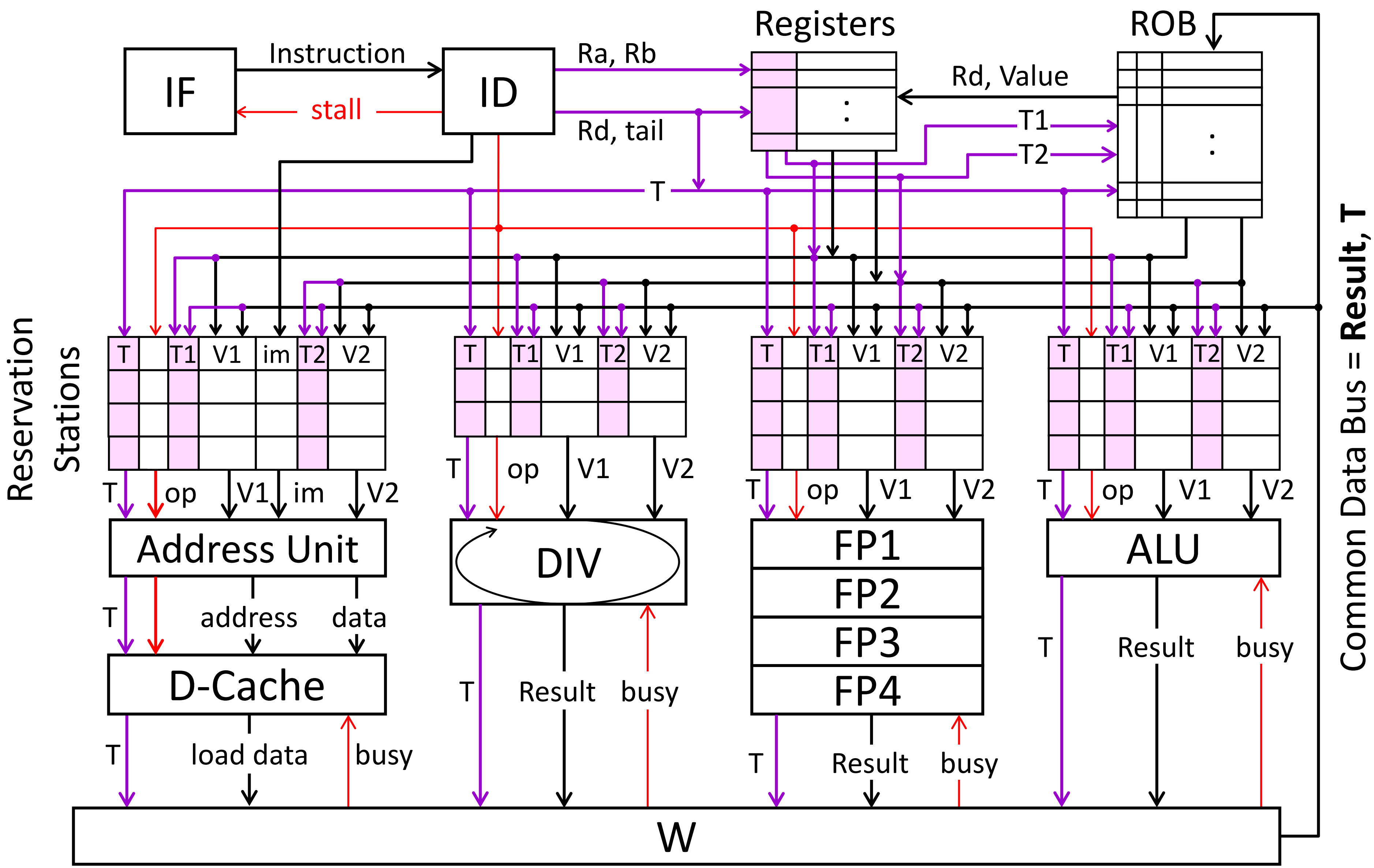
STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Reservation Stations

- Instructions wait in reservation stations
  - Until their source operands are present and function unit is not busy
- Each reservation station has the following fields:
  - Busy: Indicates whether the reservation station is busy or free
  - T: Tag = Destination ROB entry
  - Op: Operation of the instruction waiting in this reservation station
  - V1, V2: Source operand values
    - I-type ALU instructions put the immediate constant in V2
  - T1, T2: Source operand tags
  - If T1 (or T2) is zero then the value is present in V1 (or V2)
    - If both T1 and T2 are zeros, the instruction is ready to execute
  - Load/store instructions need an immediate field to compute address

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

New Pipeline with Reorder Buffer



STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

New Pipeline Stages: IF, ID, IS, EX, W, C

IF: Instruction Fetch stage (In-Order)

ID: Instruction Decode and Dispatch stage (In-Order)

- ✧ Allocate **tail** ROB entry **T** and a free reservation station **s**
- ✧ Read operands into RS[**s**]; Rename the destination register as **T = tail**

IS: Issue Stage (Out-of-Order)

- ✧ If operands are available in RS[**s**] then begin execution **else wait**

EX: Execute Stage using different function units (Out-of-Order)

W: Write Result to Reorder Buffer (Out-of-Order)

- ✧ Write result on the common data bus to ROB entry **T**
- ✧ Forward Result to stations with a source operand matching tag **T**

C: Commit stage (In-Order)

- ✧ Write value in ROB[**head**] to register file and free ROB[**head**]

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Instruction Dispatch Stage

- Allocate ROB[tail] and a free reservation station RS[s]
  - **ROB[tail].Type** ← Type; **ROB[tail].Rd** ← Rd; **ROB[tail].R** ← **0**
  - **RS[s].busy** ← **1**; **RS[s].Op** ← Operation; **RS[s].T** ← **tail**
- If no free ROB or RS entry then stall (structural hazard)
- Reading register (rename) tags and values
  - Let **Ra**, **Rb** = source registers; **T1** = **Reg[Ra].T**; **T2** = **Reg[Rb].T**
- Renaming Destination Register
  - Let **Rd** = destination register; **Reg[Rd].T** ← **tail**
- Advancing **tail** index
  - if (**tail** == **last** ROB index) then **tail** ← **1** else **tail** ← **tail + 1**

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Reading Register Values

- Can be done during Instruction Dispatch or next cycle
- Read values from Register file or ROB
- If (**T1 == 0**) **RS**[**s**]**.V1 ← Reg**[**Ra**]**.V** else **RS**[**s**]**.V1 ← ROB**[**T1**]**.V**
- If (**T2 == 0**) **RS**[**s**]**.V2 ← Reg**[**Rb**]**.V** else **RS**[**s**]**.V2 ← ROB**[**T2**]**.V**
- **RS**[**s**]**.T1 ← T1**; If (**ROB**[**T1**]**.R**) **RS**[**s**]**.T1 ← 0**
- **RS**[**s**]**.T2 ← T2**; If (**ROB**[**T2**]**.R**) **RS**[**s**]**.T2 ← 0**
- If instruction has **NO 2nd source** register then: **RS**[**s**]**.T2 ← 0**
- Load and Store instructions: **RS**[**s**]**.imm ←** immediate (offset)

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Issue Stage

- Instructions wait in reservation stations in the Issue stage
- An instruction in station **s** is ready to execute if
  - Both operands are present: (**RS[s].T1 == 0** and **RS[s].T2 == 0**)
- Issue instruction if corresponding function unit is NOT busy
  - If corresponding function unit is busy then wait (structural hazard)
- Issue Instruction: (RS[**s**].T, RS[**s**].Op, RS[**s**].V1, RS[**s**].V2)
- Free reservation station **s**: RS[**s**].busy ← **0**
- Can issue instructions in parallel to different function units
  - However, cannot issue multiple instructions to same function unit
- After an instruction is issued, then it is executed
  - Execution might take one or multiple cycles

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Write Stage

- Write stage writes to reorder buffer (Not register file)
- Write stage detects and resolves conflicting writes
  - Each cycle, the write stage selects the result of one function unit
  - Forces other function units to wait if conflicting write (busy signal)
- Place **Result** and ROB target **T** on the common data bus
- Write to ROB[**T**]:   ROB[**T**]**.V** ← **Result**;   ROB[**T**]**.R** ← **1**
- All reservation stations monitor the common data bus
  - ∀ **x**, if (RS[**x**].T1 == **T**) { RS[**x**].V1 ← **Result**; RS[**x**].T1 ← **0** }
  - ∀ **x**, if (RS[**x**].T2 == **T**) { RS[**x**].V2 ← **Result**; RS[**x**].T2 ← **0** }

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Commit Stage

- Check the instruction type at the **head** of the ROB

  If (**ROB**[**head**]**.Type** == Branch)

  - If (Branch is mispredicted)
    - Clear ROB, Reservation Stations, Store Queue, Rename table
    - Modify the PC to change the branch direction

  Else if (**ROB**[**head**]**.Type** == Store)

  - Commit store instruction and write value to memory

  Else if (**ROB**[**head**]**.R**) // Ready to commit

  - Let **Rd** = **ROB**[**head**]**.Rd**; **Reg**[**Rd**]**.V** ← **ROB**[**head**]**.V**;
  - If (**Reg**[**Rd**]**.T** == **head**) then **Reg**[**Rd**]**.T** ← **0**

  Else wait until (**ROB**[**head**]**.R**)

- If (**head** == last index)   **head** ← **1** else **head** ← **head** + **1**

STUDENTS-HUB.com   Uploaded By: Jibreel Bornat

Dealing with Branch Mispredictions

- ❖ Mispredicted branches should be recovered quickly
- ❖ Should NOT wait until the Commit Stage
  - ✧ Until the mispredicted branch appears at the head of the ROB
- ❖ Recover as soon as a branch misprediction is detected

1. Clear the Reorder Buffer
   - ✧ For all entries that **appear after the mispredicted branch**
   - ✧ Allow instructions **before the branch to continue**
2. Restart at the correct branch successor

- ❖ Performance is sensitive to
  - ✧ Accuracy of the branch predictor
  - ✧ How quickly a processor can recover from a branch misprediction

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Ensuring Precise Exceptions

- Exceptions are recorded in the Reorder Buffer
  - An exception bit is raised and the cause is also stored in the ROB
  - Can have multiple exceptions raised in the ROB
- An exception does not take place until …
  - The instruction that raised the exception is **at the head of ROB**
  - The instruction is not speculative (branch predictions are resolved)
- A mispredicted branch fetches instructions along wrong path
  - These mispredicted instructions can also raise exceptions
  - However, these exceptions **are never taken**
- The processor pipeline takes an exception by …
  - Flushing the ROB and jumping to the exception handler
- A store instruction can update memory only when …
  - It reaches the **head of ROB** and there is no raised exception

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Presentation Outline

- **Diversified Pipeline**
- **Out of Order Execution**
- **Tomasulo's Dynamic Scheduling**
- **Dealing with Exceptions**
- **Reorder Buffer**
- **Out of Order Memory Access**

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Load and Store Instructions

- Load instruction:
  - Requires the address to read the D-Cache
- Store instruction:
  - Requires the address and data to write the D-Cache
- Address Calculation:
  - Compute the effective memory address (addressing mode)
  - Do the address translation to support virtual memory
- Can loads and stores access the D-Cache **out-of-order ?**
- Consider the following example:

```
S.D  F4, 8(R6)
L.D  F5, 0(R8)
```

Can **L.D** access D-Cache before **S.D**?

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Memory Disambiguation

- Consider again the same example:

```
S.D  F4, 8(R6)
L.D  F5, 0(R8)
```

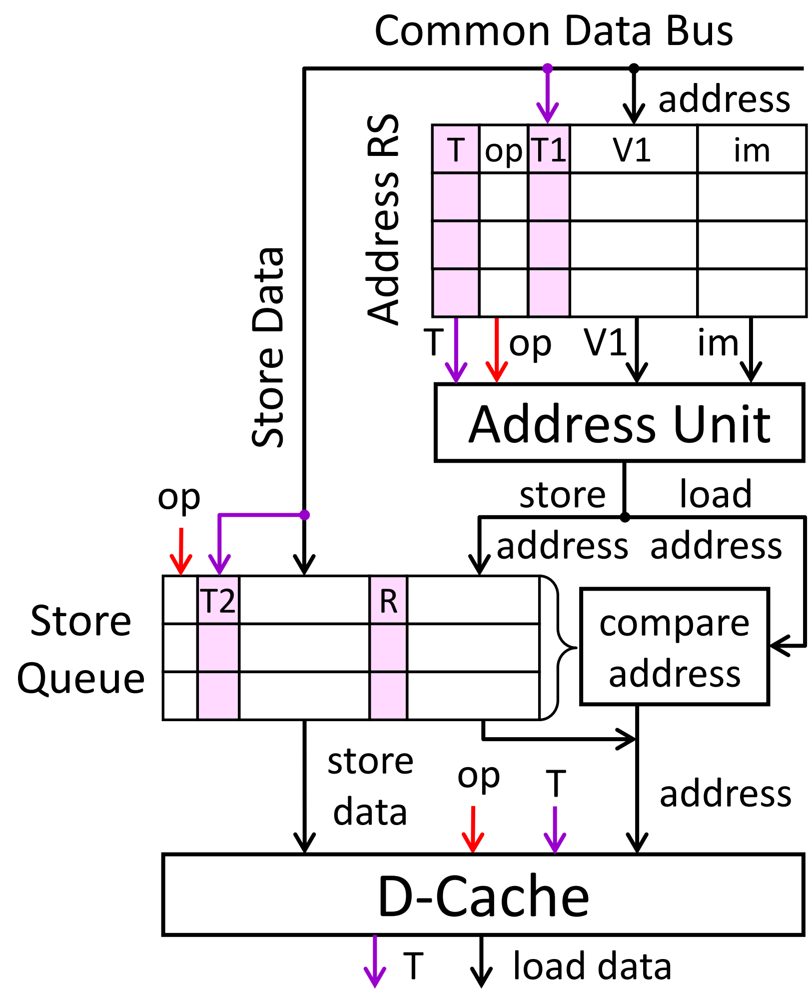
Can **L.D** access D-Cache before **S.D**?

- Can **L.D** execute **before** **S.D** (out of program order)?
- If **S.D** and **L.D** access **different memory addresses …**
  - Then **YES**, it is safe to execute them out-of-order
- If **S.D** and **L.D** access the **same memory address …**
  - Then **NO**, executing **L.D** before **S.D** is a **RAW memory hazard**
  - However, **can forward data** from **S.D** to **L.D**
- The **S.D** cannot write D-Cache until it is retired (head of ROB)
- The **L.D** can bypass the **S.D** instruction **if different address**

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Allowing Loads to Bypass Stores

- Address Reservations Stations
  - Queue all load and store addresses
  - Wait until the address is available
- Stores wait in a **Store Queue**
  - Queue store address and data
  - Until the store is allowed to commit
- Load can bypass waiting stores
  - If the **load address is different**
  - Compare load address against all the waiting store addresses in Store Queue
- Forward the store data to a load
  - If they have **identical addresses**

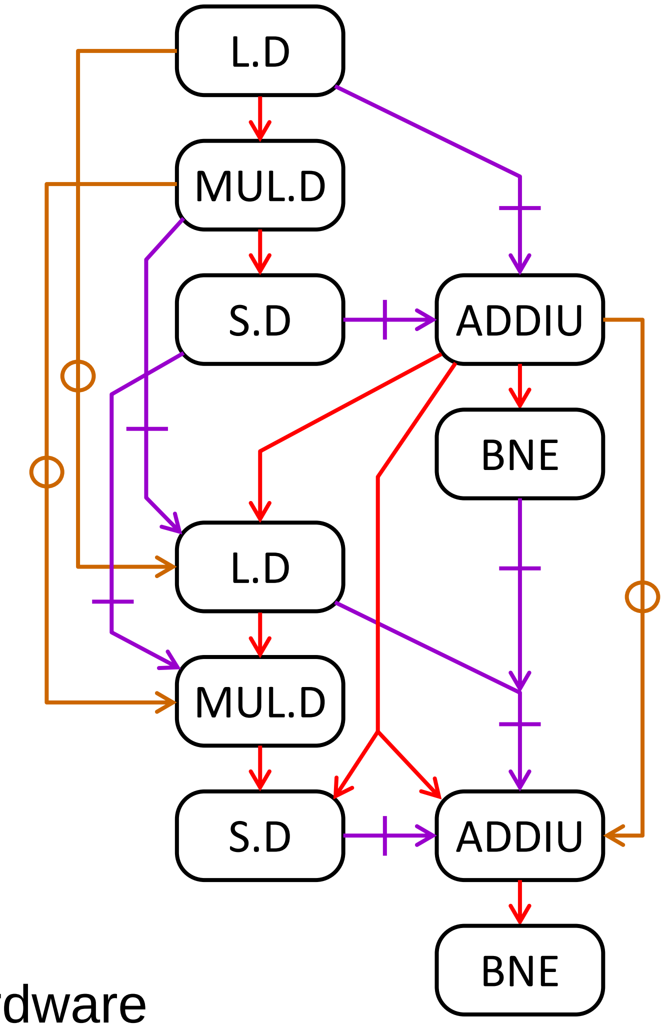


 Uploaded By: Jibreel Bornat

ROB Example

- Consider the same loop example:

```
L:  L.D     F1, (R4)
    MUL.D   F2, F1, F0
    S.D     F2, (R4)
    ADDIU   R4, R4, 8
    BNE     R4, R5, L
```



- Dependence Graph (for 2 iterations)
  - Data + Name Dependences within iteration
  - Data + Name Dependences across iterations
- Assume that BNE is predicted to be taken
  - The loop can be unrolled dynamically by the hardware
  - Renaming eliminates all WAW and WAR hazards

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

ROB Example: Cycles 1 to 30

Iteration 1, 2		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
L.D	F1, (R4)	IF	ID	IS	A	M	W	**C**													
MUL.D	F2, F1, F0		IF	ID	IS	IS	IS	FP1	FP2	FP3	FP4	W	**C**								
S.D	F2, (R4)			IF	ID	IS	A	SQ	SQ	SQ	SQ	SQ	SQ	**C**							
ADDIU	R4, R4, 8				IF	ID	IS	EX	W	rob	rob	rob	rob	rob	**C**						
BNE	R4, R5, L					IF	ID	IS	IS	EX	rob	rob	rob	rob	rob	**C**					
L.D	F1, (R4)						IF	ID	IS	A	M	W	W	rob	rob	rob	**C**				
MUL.D	F2, F1, F0							IF	ID	IS	IS	IS	IS	FP1	FP2	FP3	FP4	W	**C**		
S.D	F2, (R4)								IF	ID	IS	A	SQ	SQ	SQ	SQ	SQ	SQ	SQ	**C**	
ADDIU	R4, R4, 8									IF	ID	IS	EX	W	rob	rob	rob	rob	rob	rob	**C**
BNE	R4, R5, L										IF	ID	IS	IS	EX	rob	rob	rob	rob	rob	rob

SQ = Store Queue

W = Conflicting Write

Iteration 3, 4		11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
L.D	F1, (R4)	IF	ID	IS	A	M	W	rob	rob	rob	rob	rob	**C**								
MUL.D	F2, F1, F0		IF	ID	IS	IS	IS	FP1	FP2	FP3	FP4	W	rob	**C**							
S.D	F2, (R4)			IF	ID	IS	A	SQ	SQ	SQ	SQ	SQ	SQ	SQ	**C**						
ADDIU	R4, R4, 8				IF	ID	IS	EX	W	rob	rob	rob	rob	rob	rob	**C**					
BNE	R4, R5, L					IF	ID	IS	IS	EX	rob	rob	rob	rob	rob	rob	**C**				
L.D	F1, (R4)						IF	ID	IS	A	M	W	W	rob	rob	rob	rob	**C**			
MUL.D	F2, F1, F0							IF	ID	IS	IS	IS	IS	FP1	FP2	FP3	FP4	W	**C**		
S.D	F2, (R4)								IF	ID	IS	A	SQ	SQ	SQ	SQ	SQ	SQ	SQ	**C**	
ADDIU	R4, R4, 8									IF	ID	IS	EX	W	rob	rob	rob	rob	rob	rob	**C**
BNE	R4, R5, L										IF	ID	IS	IS	EX	rob	rob	rob	rob	rob	rob

At steady state:
One Instruction is
Committed Per Cycle

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Making the Pipeline Superscalar

- Pipelines studied so far are scalar
  - Fetch, decode, and dispatch one instruction per cycle
  - Write-back and Commit one instruction per cycle
- Fundamentally limited to CPI ≥ 1
- Superscalar pipelines can do more …
  - Can fetch, decode, and dispatch multiple instructions per cycle
  - Can execute, write-back, and commit multiple instructions per cycle
  - Can reduce the CPI below 1 (CPI < 1)
  - **IPC = Instructions per Cycle = 1 / CPI**
- Two types of superscalar processors
  - In-order execution: based on program order
  - Out-of-order execution: based on data dependences

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Superscalar Complexities

- Must fetch a group of instructions each cycle (not just one)
  - An accurate branch predictor is essential for correct fetching
- Must decode a group of instructions each cycle
  - Must check dependences between instructions in a single group
- Must dispatch a group of instructions each cycle
  - Requires more read ports for the register file and reorder buffer
  - Read ports increases with the number of dispatched instructions
- Must write multiple results each cycle
  - Requires multiple result busses (not just one common data bus)
  - Requires multiple write ports for the ROB (not just one)
  - More comparators for tag matching in reservation stations
- Must commit multiple results each cycle
  - Requires multiple write ports for the register file

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Pipelining Delivers CPI = 1 at Best

- How to get CPI < 1?
  - Issuing and completing multiple instructions/cycle
  - Challenge: checking and resolving dependences among the instructions issued at the same cycle
- Dynamic multiple Issue
  - Fetch, decode, and commit multiple instructions
  - Use dynamic pipeline scheduling, HW-based speculation and recovery
  - Always beneficial if compiler can help, but recompiling not required for new machines
- Static multiple issue
  - Compiler handles data and control hazards and decides what instructions can be issued in the same cycle
  - Often restrict mix of instructions can be initiated in a clock
  - Recompilation required for machines with diff. pipelines

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Hardwere Strategies for Multiple Issue

- **Superscalar:** varying number of instructions/cycle (1 to 8), scheduled by compiler (static) or by HW (dynamic)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium 4, i7
- **Multithreading:** exploiting thread-level parallelism to improve uniprocessor throughput
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium 4, i7
- **Vector Processing:** Explicit coding of independent loops as operations on large vectors of numbers
  - Multimedia instructions being added to many processors

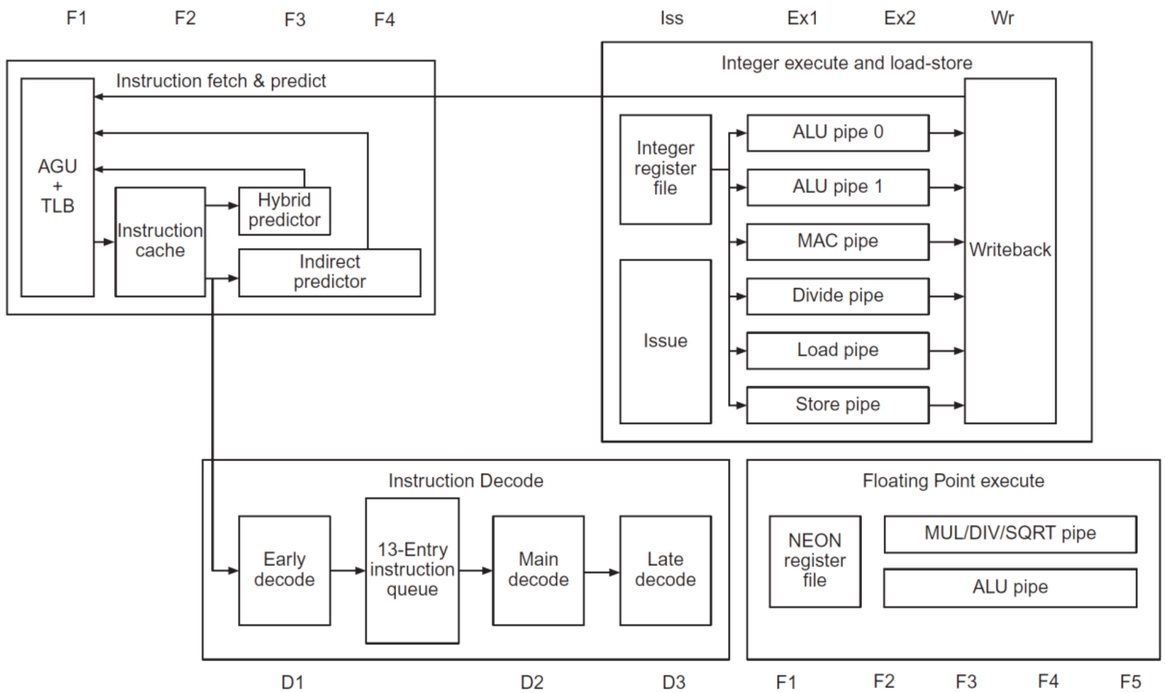
STUDENTS-HUB.com Uploaded By: Jibreel Bornat

Software Strategies for Multiple Issue

- **Rely on compiler technology to extract parallelism**
  - Loop Level Parallelism
  - Loop Unrolling
  - Software Pipelining
  - Predicated Instructions
  - VLIW Approach

STUDENTS-HUB.com Uploaded By: Jibreel Bornat

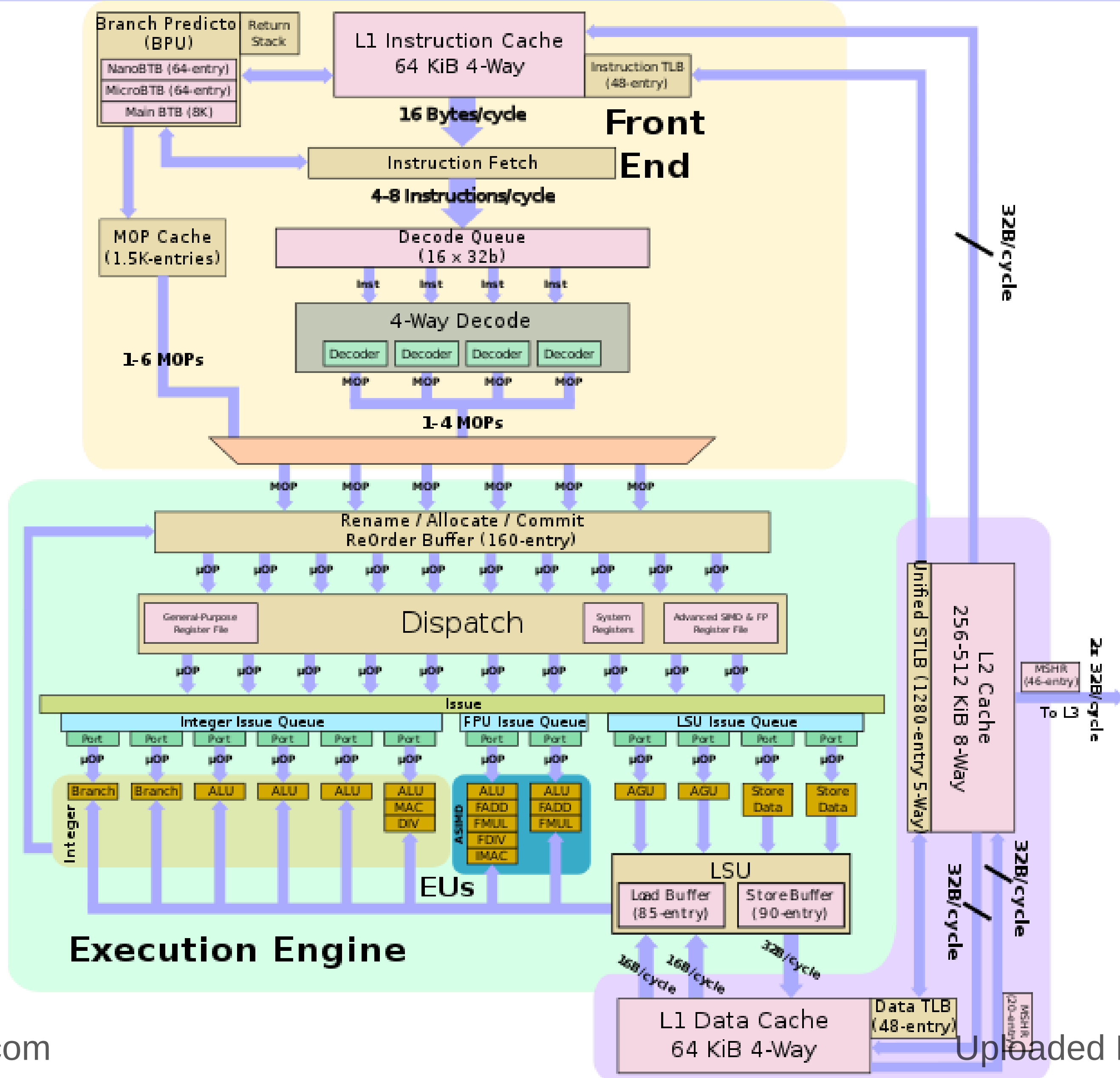
Example: ARM Cortex-A53 Pipeline Structure



- F1 and F2 fetch the instruction,
- D1 and D2 do the basic decoding, and D3 decodes some more complex instructions and is overlapped with the first stage of the execution pipeline (ISS).
- After ISS, the Ex1, EX2, and WB stages complete the integer pipeline.
- Branches use four different predictors, depending on the type.
- The floating-point execution pipeline is 5 cycles deep, in addition to the 5 cycles needed for fetch and decode, yielding 10 stages in total.

STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Example: ARM Cortex-A77 Pipeline Structure



STUDENTS-HUB.com

Uploaded By: Jibreel Bornat

Example: Intel Core i7 Pipeline Structure

Pipeline depth = 14 stages

Branch penalty = 15 cycles

Intel x86 instructions translated into micro-operations

Micro-ops: RISC-like instructions

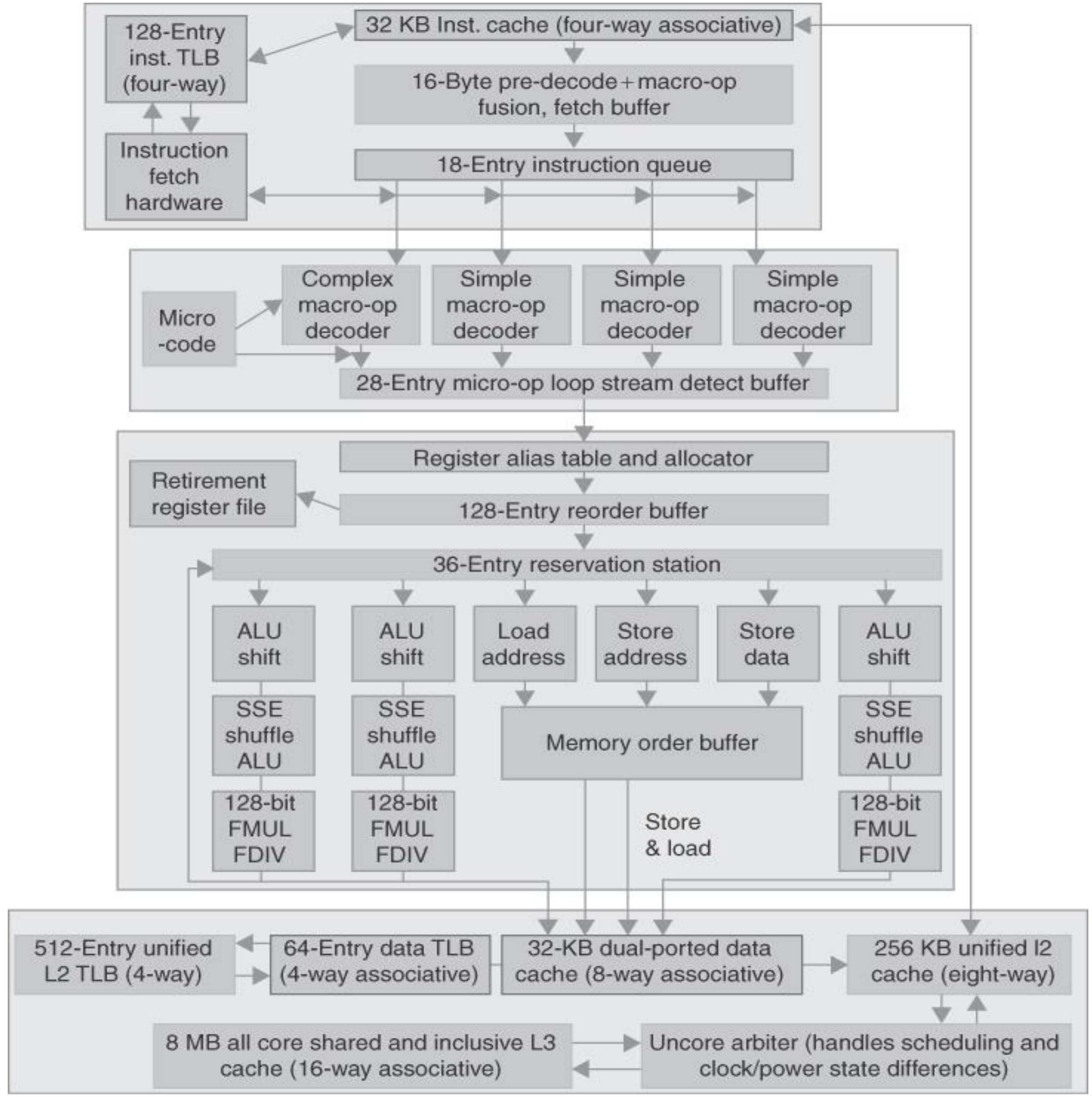
4 instructions decoded per cycle

Loop stream detector and buffer

36-entry centralized RS

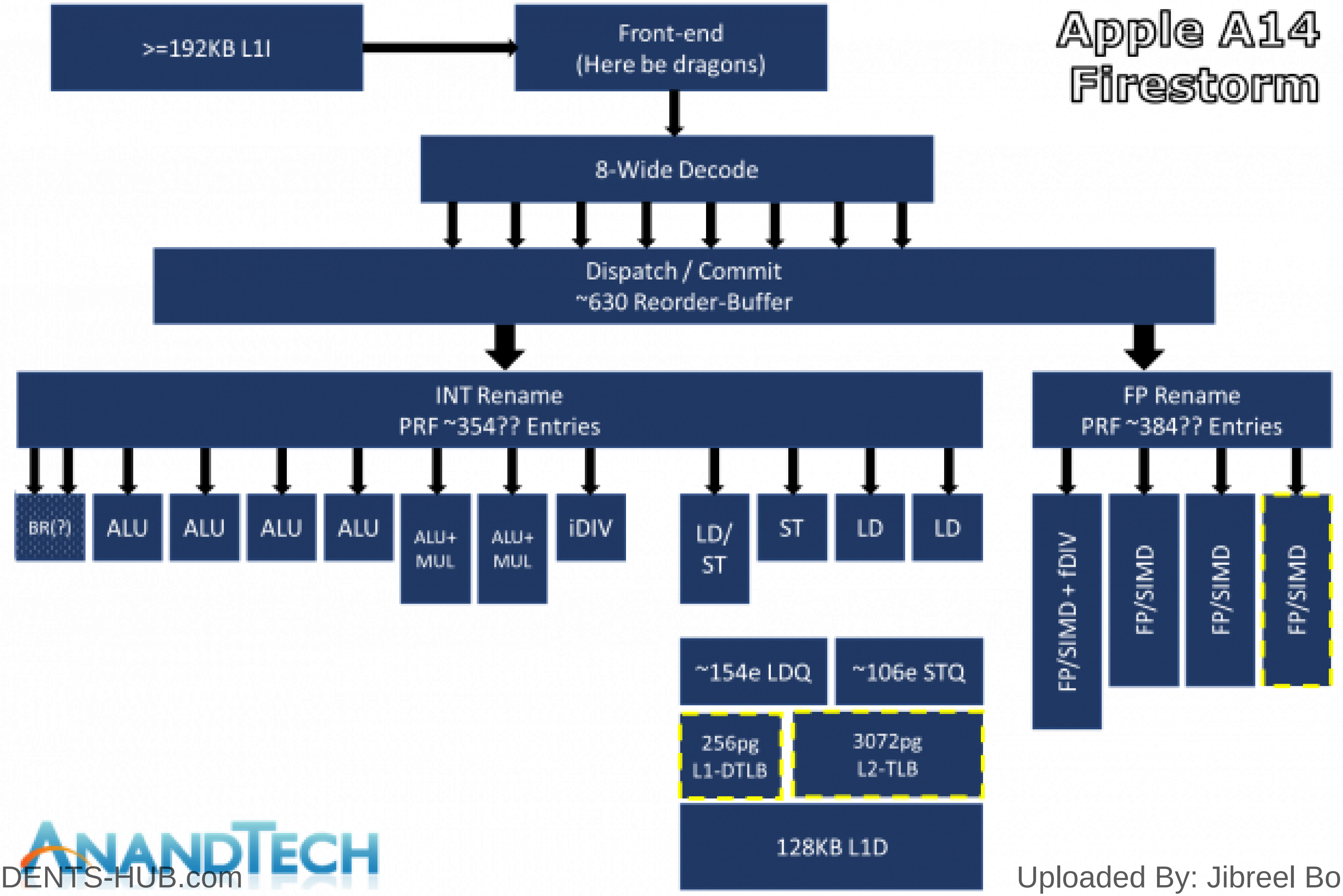
Six independent function units

Up to 6 micro-ops can be executed per cycle



STUDENTS-HUB.com
Uploaded By: Jibreel Bornat

Example: Apple M1 Pipeline Structure



STUDENTS-HUB.com

Uploaded By: Jibreel Bornat