Pipelining: Basic and Intermediate Concepts

Appendix C

- Basics of Pipelining
- Implementation of Pipeline
- Hazards and their Solutions
- Exception Handling
- Pipeline with Floating-Point Instructions
- Dynamic Scheduling of Pipelines

Slides credit : Prof. Dr. Hasina Khatoon

Principles of Pipelining

- **How is Pipelining Implemented?**

  Implementation of RISC-V without pipelining

- **A Simple Implementation of RISC-V**

  Every instruction takes at most 5-cycles

1. ***Instruction Fetch Cycle (IF)***

   IR ← Mem [PC]

   NPC ← PC + 4 NPC: Next Sequential Instruction

2. ***Instruction Decode/Register Fetch Cycle (ID)***

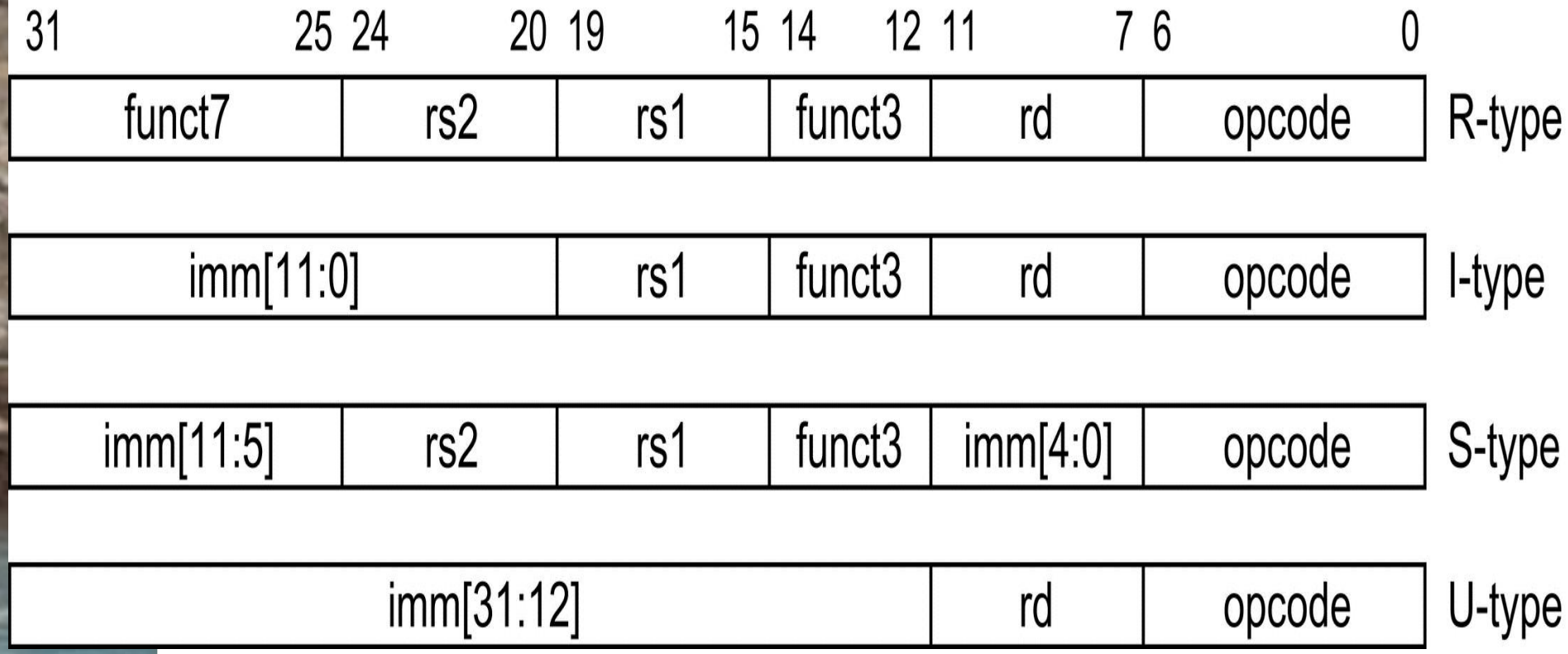
   A ← Regs [rs1]

   B ← Regs [rs2]

   Imm ← sign-extended immediate field of IR

- **Imm, A and B are temporary registers**
- **Fixed-field decoding is done**

Instruction layout for RISC-V



Four major types of instructions

A Simple Implementation of RISC-V

3. *Execution/Effective Address Cycle (EX)*

- Only one of following four functions is performed depending on the type of instruction

Memory Reference Instruction

$$ALU_{output} \leftarrow A + Imm$$

Register-Register ALU Operation

$$ALU_{output} \leftarrow A \text{ func } B$$

Register-Immediate ALU Operation

$$ALU_{output} \leftarrow A \text{ op } Imm$$

Branch Operation

$$ALU_{output} \leftarrow NPC + (Imm<<2);$$

$$Cond \leftarrow (A == B)$$ (Only BEQ is considered)

A Simple Implementation of RISC-V

4. Memory access/branch completion cycle (MEM)

PC is updated for all instructions

$$PC \leftarrow NPC;$$

- For load, store and branch instructions

  For Memory Reference instructions

  $$LMD \leftarrow Mem[ALU_{output}] \quad OR \quad Mem[ALU_{output}] \leftarrow B$$

  For Branch instructions

  $$\text{If (cond)} \quad PC \leftarrow ALU_{output}$$

5. Write-Back Cycle (WB)

Register-Register or Register-Immediate ALU Instructions

$$Regs[rd] \leftarrow ALU_{output}$$

Load Instruction

$$Regs[rd] \leftarrow LMD$$

RISC-V data path allows every instruction to be executed in 4 or 5 clock cycles

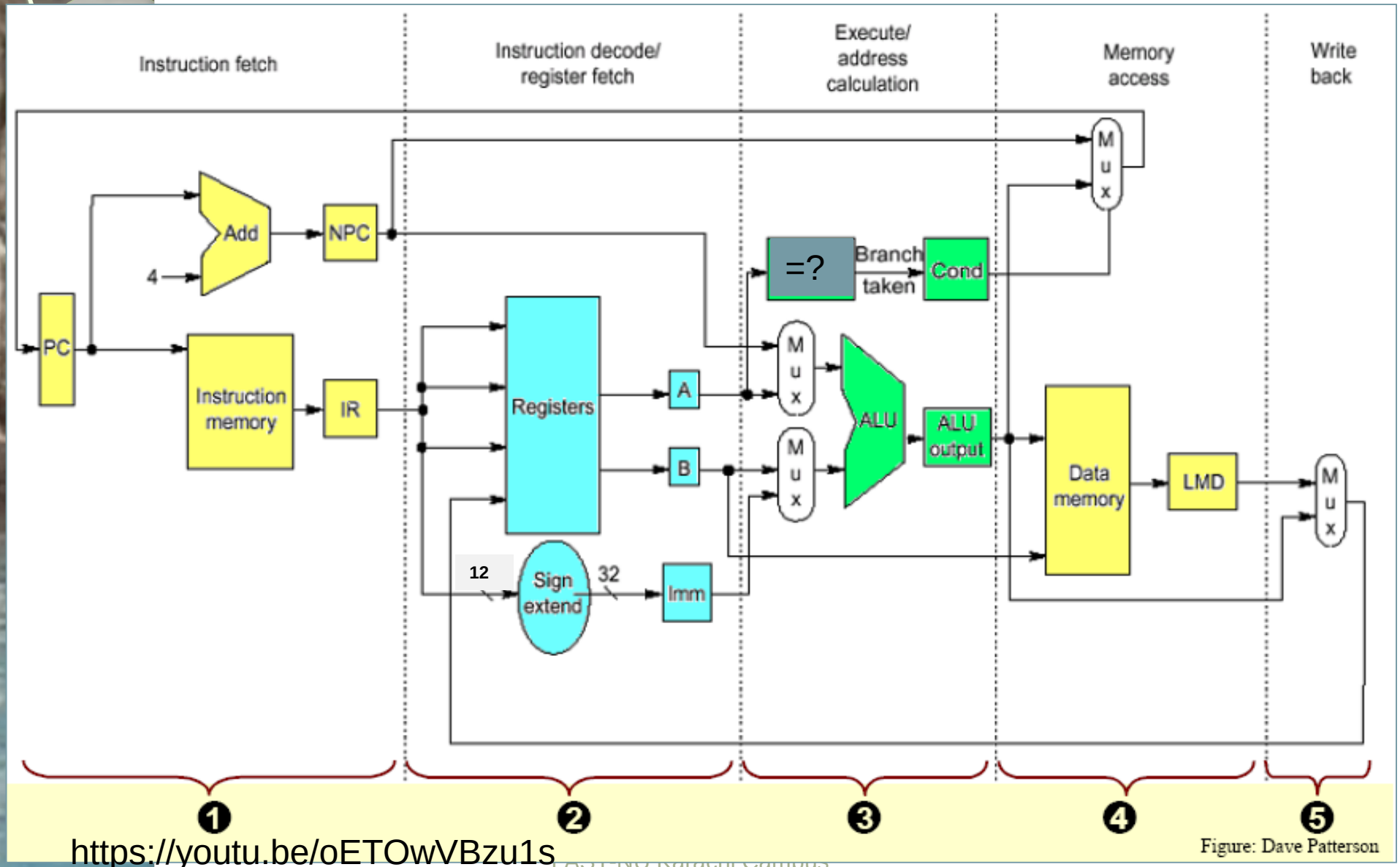


Figure: Dave Patterson

https://youtu.be/oETOwVBzu1s

A Simple Implementation of RISC-V

- At the end of each clock cycle every value that is computed is written into a storage device

  LMD, Imm, A, B, IR, etc.

- The temporary registers hold values between clock cycles

- Hardware Redundancies

  Two ALUs can be merged as one

  Data and instructions can be stored in the same memory

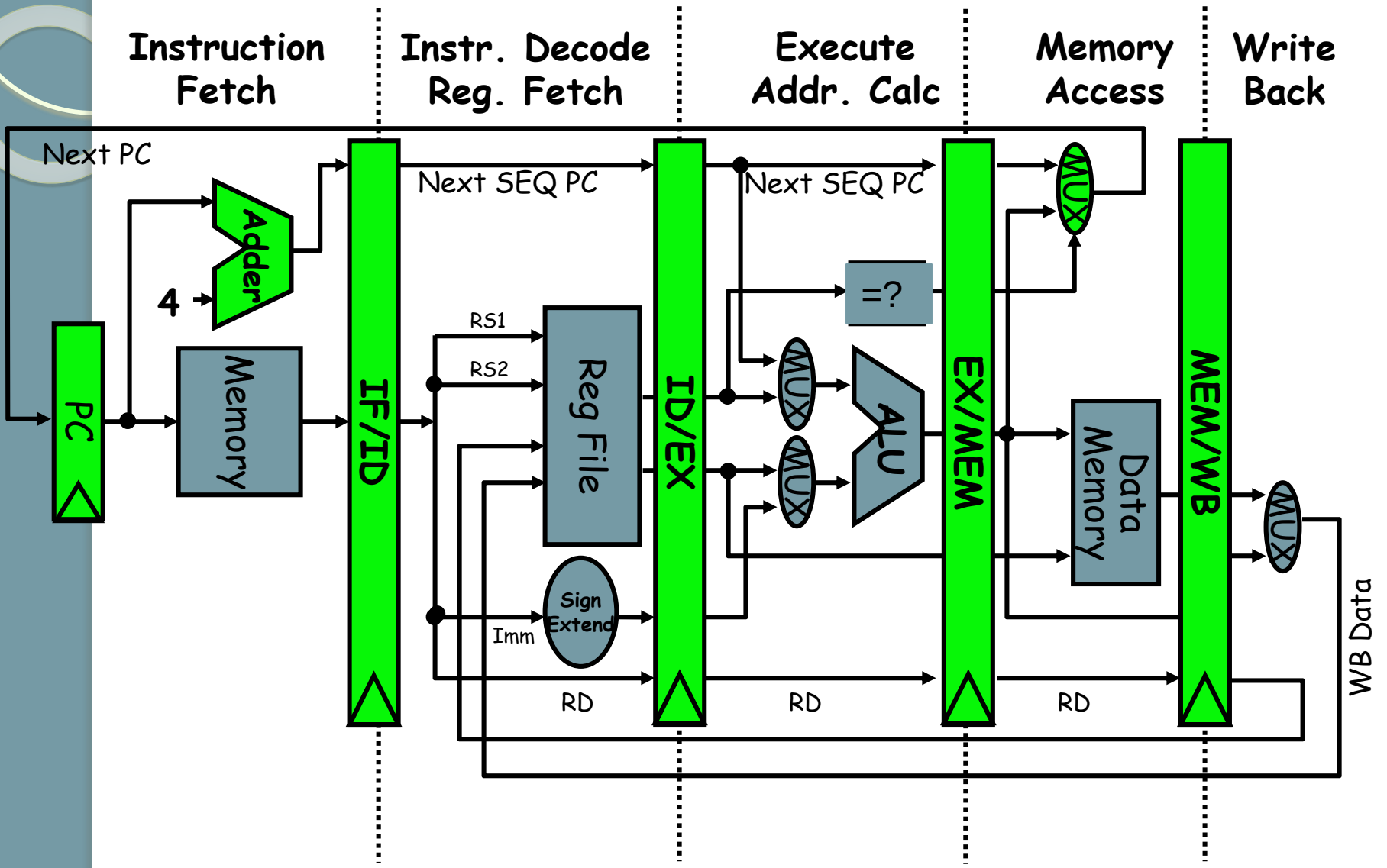
- Design provides a better base for pipelined implementation

Implementation of Pipelining

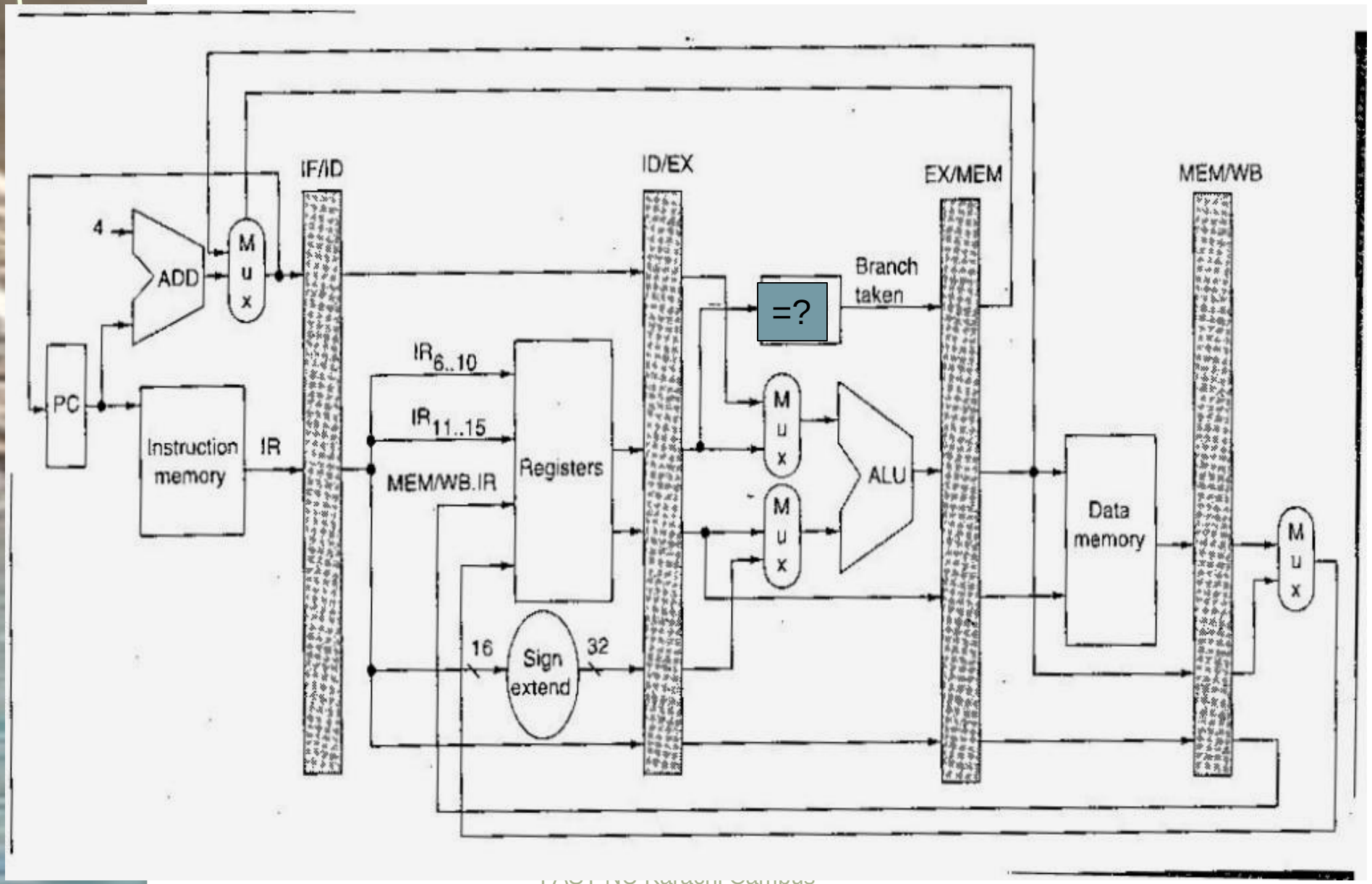
- **A Basic Pipeline for RISC-V**
- **Pipeline the data path with almost no changes**
- ***Pipeline registers* or *pipeline latches* pass the values from one pipe stage to another**
  - **All information for an instruction required for its correct execution is submitted in these registers**

    Contains both data and control information
- **Events on every pipe stage of the RISC-V pipeline**
- ***Operations in the first two stages are independent of the current instruction***

Pipelined data path after adding a set of registers, one between each pair of pipe stages



Data path is pipelined by adding a set of pipeline registers



Events on every pipe stage of the RISC-V pipeline

Stage	Any Instruction		
IF	IF/ID.IR ←MEM[PC] ; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond) {EX/MEM.ALUOutput } else { PC + 4 }) ;		
ID	ID/EX.A = Regs[IF/ID. IR $_{6..10}$]; ID/EX.B ←Regs[IF/ID. IR $_{11..15}$]; ID/EX.NPC ←IF/ID.NPC ; ID/EX.IR ←IF/ID.IR; ID/EX.Imm ← (IF/ID. IR $_{16}$) ^{16} ## IF/ID. IR $_{16..31}$;		
	ALU	***Load or Store***	***Branch***
EX	EX/MEM.IR = ID/EX.IR; EX/MEM. ALUOutput ← ID/EX.A func ID/EX.B; Or EX/MEM.ALUOutput ← ID/EX.A op ID/EX.Imm; EX/MEM.cond ← 0;	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.cond ← 0; EX/MEM.B ←ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + ID/EX.Imm; EX/MEM.cond ← (ID/EX.A op 0);
MEM	MEM/WB.IR ←EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput] ; Or Mem[EX/MEM.ALUOutput] ← EX/MEM.B ;	
WB	Regs[MEM/WB. IR $_{16..20}$] ← EM/WB.ALUOutput; Or Regs[MEM/WB. IR $_{11..15}$] ← MEM/WB.ALUOutput ;	For load only: Regs[MEM/WB. IR $_{11..15}$] ← MEM/WB.LMD;	

Implementation of Pipelining

- Controls for the four multiplexers
- Top MUX of ALU

  Set by whether an instruction is a branch or not
- Bottom MUX of ALU

  Set by whether the instruction is a register-register ALU or any other type of operation
- IF or MEM stage MUX

  Chooses between the incremented PC or the branch target address
- WB stage MUX

  Chooses between whether the instruction in WB is a Load or an ALU instruction

- Implementing the Control for the RISC-V Pipeline

Implementation of Pipelining

Instruction Issue: The process of letting an instruction move from ID to EX stage

- All data hazards are checked in ID stage
- Decision to *stall* the pipeline or *forwarding* operation is taken in this stage

  Detecting interlocks early in the pipeline is easier and it reduces hardware complexity
- A number of tests are performed in ID stage

  A number of fields are tested
- Forwarding logic and the comparisons done when the destination of the forwarded result is the ALU input (for the instruction currently in EX)
- Enlarge the multiplexers needed at the ALU input to accommodate additional logic

Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions

Situation	Example code sequence	Action

No dependence	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		No hazard possible because no dependence exists on x1 in the immediately following three instructions

Dependence requiring stall	`ld x1,45(x2)`
	`add x5,x1,x7`
	`sub x8,x6,x7`
	`or x9,x6,x7`
		Comparators detect the use of x1 in the `add` and stall the `add` (and `sub` and `or`) before the `add` begins EX

Dependence overcome by forwarding	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x1,x7`
	`or x9,x6,x7`
		Comparators detect use of x1 in `sub` and forward result of load to ALU in time for `sub` to begin EX

Dependence with accesses in order	`ld x1,45(x2)`
	`add x5,x6,x7`
	`sub x8,x6,x7`
	`or x9,x1,x7`
		No action required because the read of x1 by `or` occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half

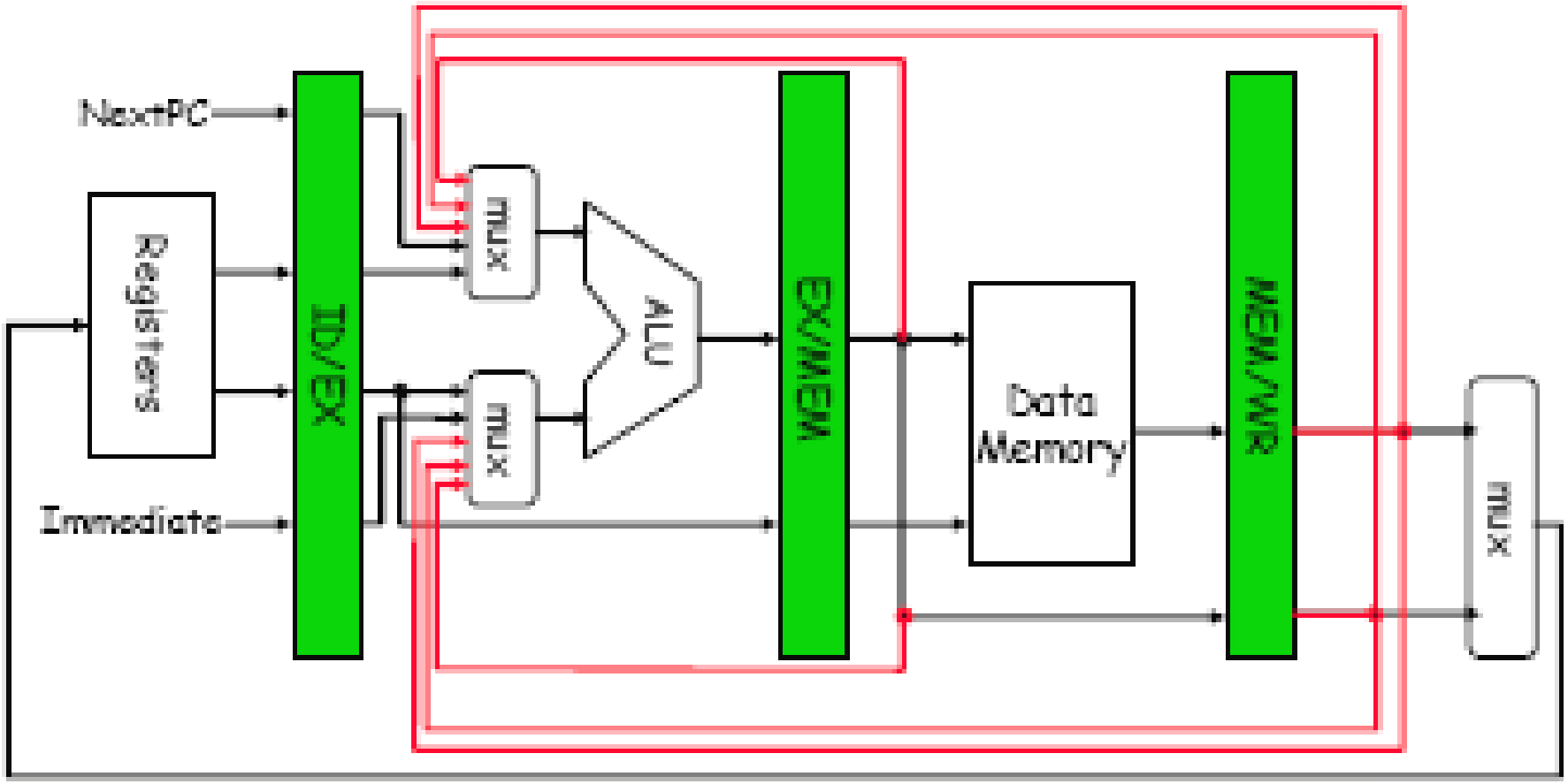
Logic to detect the need for load interlocks in ID stage - Requires two comparisons

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..6}$)	Matching operand fields
Load	Register-register ALU, load, store, ALU immediate, or branch	`ID/EX.IR[rd] == IF/ID.IR[rs1]`
Load	Register-register ALU, or branch	`ID/EX.IR[rd] == IF/ID.IR[rs2]`

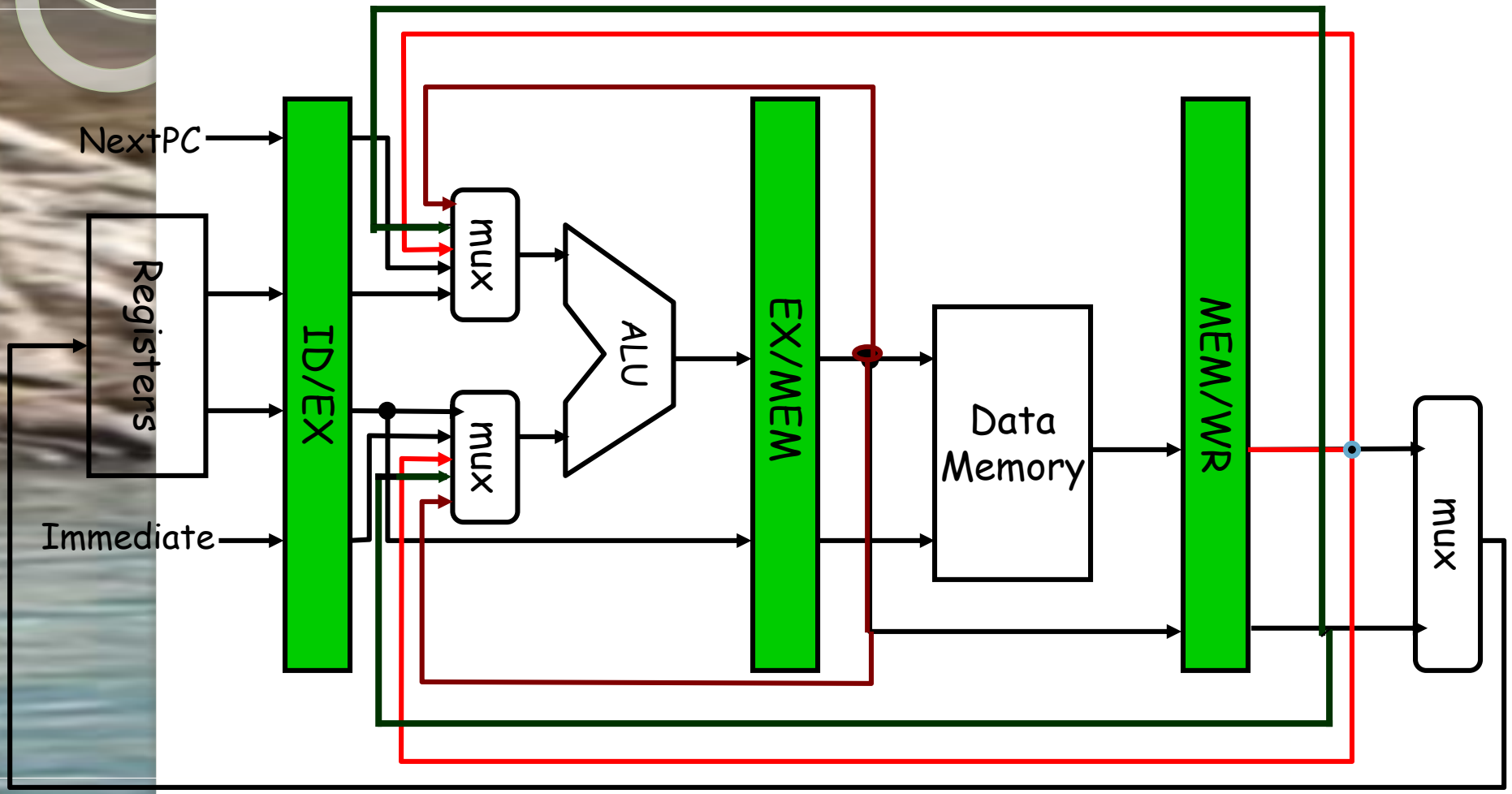
Forwarding of data to the two ALU inputs

Pipeline register of source instruction	Opcode of source instruction	Pipeline register of destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	`EX/MEM.IR[rd] == ID/EX.IR[rs1]`
EX/MEM	Register-register ALU, ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	`EX/MEM.IR[rd] == ID/EX.IR[rs2]`
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	`MEM/WB.IR[rd] == ID/EX.IR[rs1]`
MEM/WB	Register-register ALU, ALU immediate, Load	ID/EX	Register-register ALU	Bottom ALU input	`MEM/WB.IR[rd] == ID/EX.IR[rs2]`

Forwarding of results to ALU adds three extra inputs to ALU MUXes



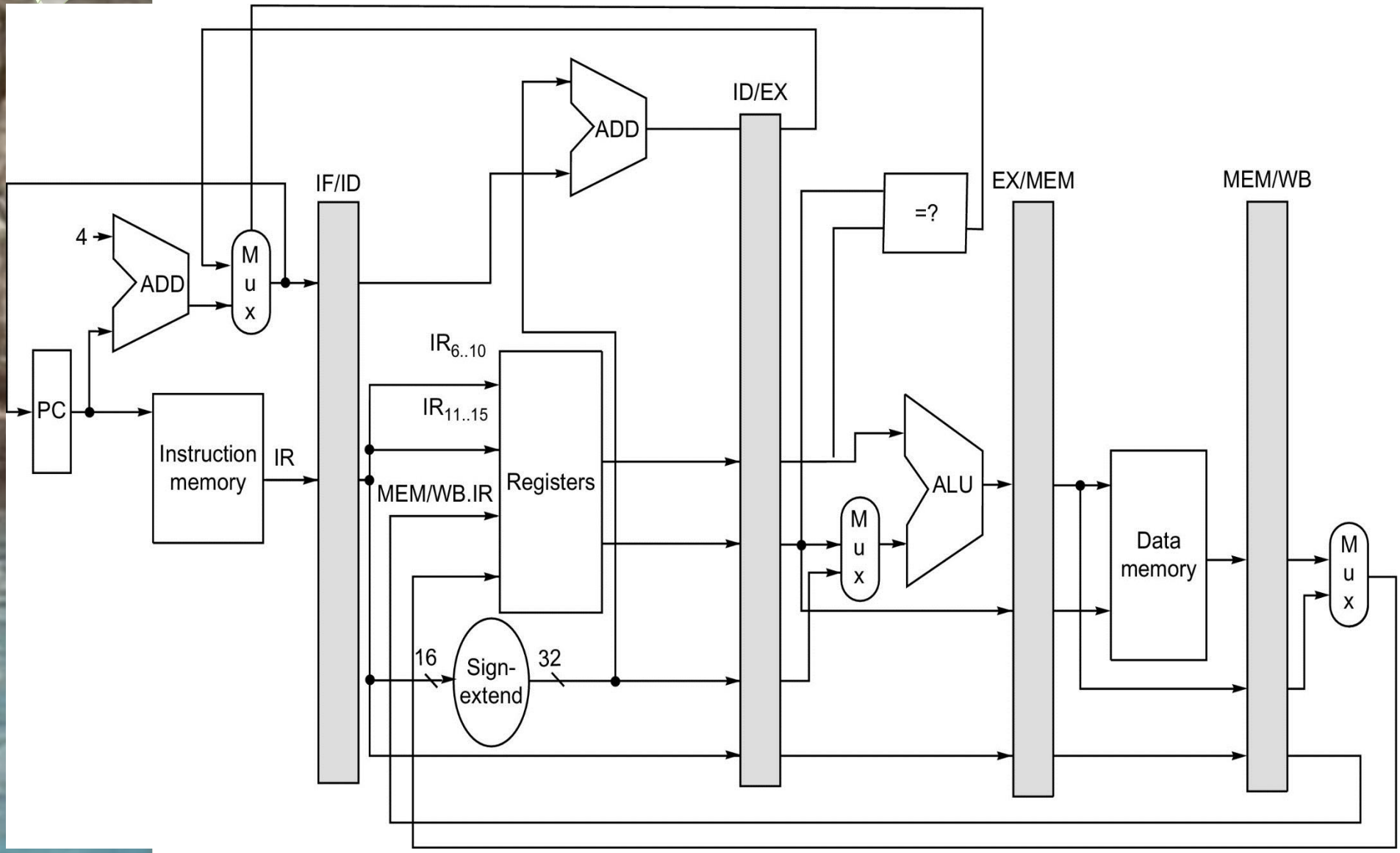
Hardware Changes for Forwarding



Implementation of Pipelining

- **Hazard detection and forwarding hardware is simple in RISC-V**
  - Floating-point instructions are not included
- **Dealing with Branches in the Pipeline**
  - Revised pipelined data path where branch is completed in the EX cycle
    - Additional ALU is required in ID stage
- The branch penalty is now 2 cycles
- With the increase in pipeline depth, branch penalty increases
  - Dynamic branch prediction is becoming more important and necessary

Revised pipeline data path



What Makes Pipelining Hard to Implement?

- **Exceptions:** ***When the instruction order is changed in unexpected ways***
- **Challenges raised by different instruction sets**

- **Dealing with Exceptions**

  Overlapped execution of instructions make exception handling more difficult to implement

Types of Exceptions and Requirements

- **I/O device request**
- **Invoking an OS system service from a user program**
- **Tracing instruction execution**
- **Breakpoint (programmer requested interrupt)**
- **Integer arithmetic overflow**
- **FP arithmetic anomaly**
- **Page fault**

What Makes Pipelining Hard to Implement?

- Misaligned memory access
- Memory protection violation
- Using an undefined or unimplemented instruction
- Hardware malfunctions
- Power failure

- Names of common exceptions vary among different architectures

➢ *The requirements on exceptions can be characterized on five semi-independent axes*

1. *Synchronous versus asynchronous*

   If the event occurs at the same place every time the program is executed with the same data and memory allocation, it is said to be *synchronous*

What Makes Pipelining Hard to Implement?

Asynchronous events **are caused by devices external to the processor and memory (except hardware malfunction)**

2. **User requested versus coerced**
3. **User maskable versus user nonmaskable**
4. **Within versus between instructions**

   Whether the event prevents instruction completion by occurring in the middle of execution or whether it is recognized between instructions

5. **Resume versus terminate**

   If the program execution stops after the interrupt, it is a *terminating* event

Most difficult exception to handle

Exceptions that occur *within* an instruction and are expected to be *resumed*

Five categories are used to define actions needed for different exception types

Exc. Type	Synchronous - Asynch.	Requested - Coerced	mask - non-mask	within - between	resume - terminate
I/O Device Req.	Asynch	Coerced	Non-maskable	Between	Resume
Invoke OS svc.	Synch	User Requested	Non-maskable	Between	Resume
Trace/Bkpoint	Synch	User Requested	Maskable	Between	Resume
Arith. exception	Synch.	Coerced	Maskable	Within	Resume
Page Fault	Synch.	Coerced	Non-maskable	Within	Resume
Misaligned addr.	Synch	Coerced	Maskable	Within	Resume
Mem. prot. viola-tion	Synch	Coerced	Non-maskable	Within	Resume
Undefined Inst.	Synch.	Coerced - ???	Non-maskable	Within	Terminate - ???
HW error	Asynch.	Coerced	Non-maskable	Within	Terminate
Power Failure	Asynch	Coerced	Non-maskable	Within	Terminate

What Makes Pipelining Hard to Implement?

Restartable processors

If a processor's pipeline provides to it the ability to handle the exception and restart the program without affecting its correctness and execution

- Stopping and Restarting Execution
- Most difficult exceptions have two properties
  - They occur within instructions
  - They must be restartable
- When an exception occurs, save the pipeline state by the following steps

1. Force a trap instruction into the pipeline on the next IF
2. Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline

What Makes Pipelining Hard to Implement?

Prevents any state changes for instructions that will not be completed until the exception is handled

3. After the exception handling routine receives control, it immediately saves the PC of the faulting instruction

- A pipeline is said to have *precise exception*, if the pipeline can be stopped so that the instructions before the faulting instruction are completed and those after it can be restarted from scratch

  Simpler for integer instructions but is complex to maintain for floating point instructions

- Floating point instructions run for many cycles

  Operations often complete out-of-order

What Makes Pipelining Hard to Implement?

- *Precise exception mode and fast mode of execution*
  - Supporting precise exception is a requirement in many systems

- **Exceptions in RISC-V**
  - **RISC-V pipeline stages and the 'problem' exceptions raised**
- Multiple exceptions can occur in the same pipeline cycle
  - Exceptions may occur out-of-order
- Hardware posts all exceptions in a status vector and is checked in the WB stage
  - Turn off all writes before the WB stage
    - Exceptions are handled in order
- **Works well for int. pipeline but not for fl-pt. pipeline**

Exceptions that may occur in RISC-V pipeline

Pipeline Stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory protection violation
WB	None

What Makes Pipelining Hard to Implement?

- **Instruction Set Complications**
  - **When an instruction is guaranteed to complete, it is said to be *committed***
- **RISC-V writes at the end of integer pipeline**
  - **End of MEM or beginning of WB cycle**
    - ***Precise exceptions are easy to implement and straightforward for RISC-V integer pipeline***
- **Processors that write in the early stages of pipeline**
  - ***Rollback the instruction or write is deferred till the end***
- **Memory state changes?**
  - **String copy operations of Intel 80x86, IBM 360**
    - ***States are kept in registers so that restoration is possible***

What Makes Pipelining Hard to Implement?

- Some processors may need extra states to be saved
- Condition code and its setting time creates more problems

  Condition code is also treated as an operand

- Instruction taking varying number of cycles to execute

  Pipeline the micro-instruction execution

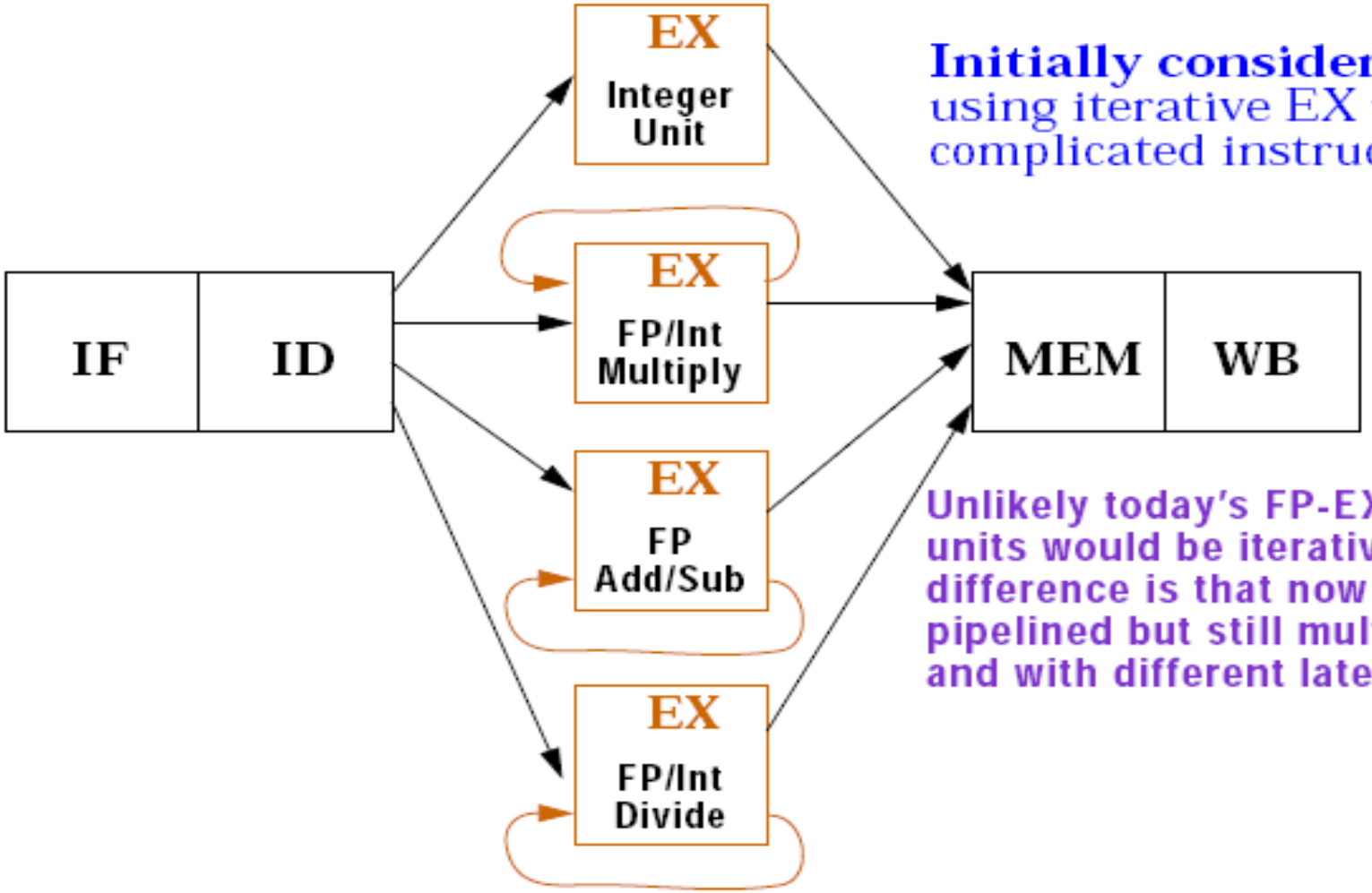
- Micro-instructions are simple instructions used in sequence to implement a more complex instruction set
- Used in the Intel processors to give it RISC like characteristic
- Simpler instruction sets avoid many of the complications

Pipeline for Multicycle Operations

- **Extending the RISC-V Integer Pipeline to Handle Multicycle Operations**
  - Basic approach and design alternatives
  - Performance measurements of a RISC-V FP pipeline
- **Same pipeline of RISC-V with two changes for FP instructions**
  - EX cycle is repeated as many times as needed to complete the operation
  - **There are multiple floating-point functional units**
- **Assume 4-separate functional units**

1. ***Main integer unit*:** Handles loads, stores, integer ALU operations and branches
2. ***FP and integer multiplier***
3. ***FP adder*:** **Handles FP add, subtract and conversion**
4. ***FP and integer divider***

RISC-V pipeline with three additional unpipelined, floating point, functional units



Pipeline for Multicycle Operations

- If the execution stages of functional units are not pipelined

  EX stage has number of clock delays larger than 1

- Allow pipelining of some stages and allow multiple ongoing operations
- *Latency* and *initiation interval* or *repeat interval*
  - *Latency*: The number of intervening cycles between an instruction that produces a result and an instruction that uses the result
  - *Initiation* or *repeat interval*: The number of cycles that must elapse between issuing two operations of the same type
- Latencies and initiation intervals for the given functional units

Latencies and Initiation Intervals for Functional Units

Functional Unit	Latency	Initiation Interval
Integer ALU	0	1
Data Memory (Integer and FP loads(1 less for store latency))	1	1
FP Add/SUB	3	1
FP & Integer Multiply	6	1
FP & Integer Divide + FP sqrt	24	25

Pipeline for Multicycle Operations

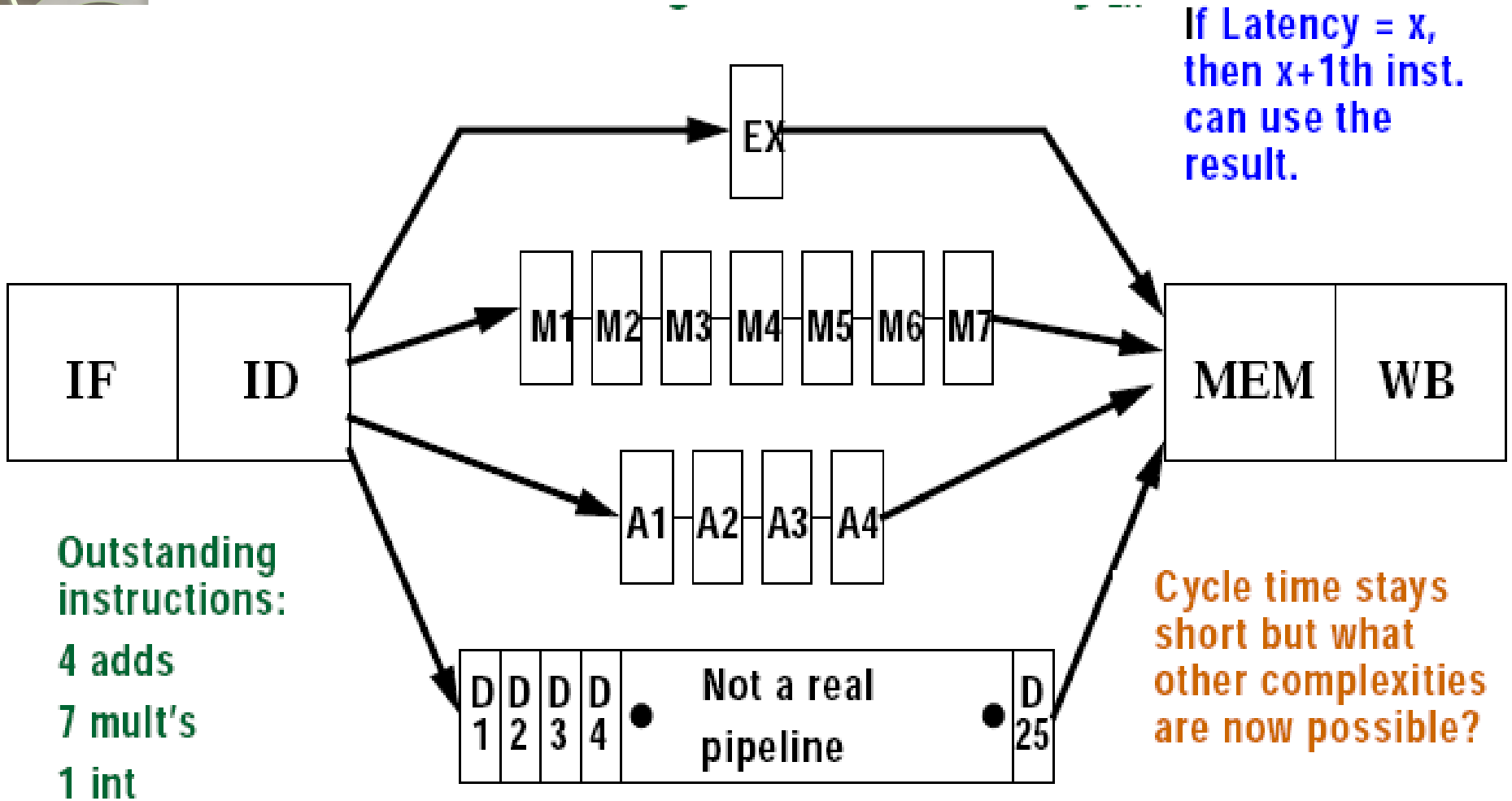
- The latency and repeat interval added will result in pipelining of some functional units
- Additional pipeline registers are introduced and modification of the connections to those registers are done

  A1/A2, A2/A3, A3/A4, etc.

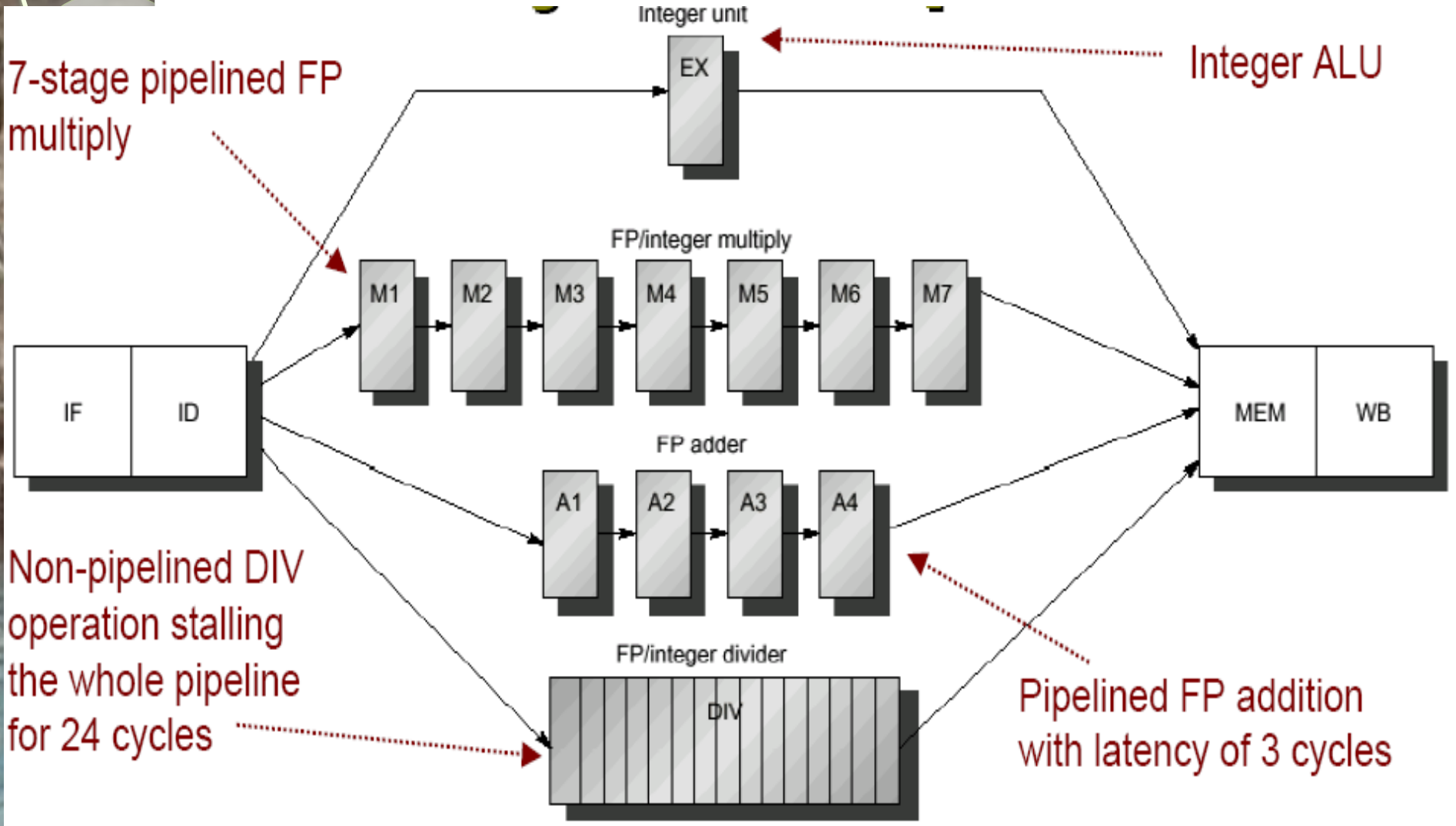
- Hazards and Forwarding in Longer Latency Pipelines
- A number of different aspects to the hazards detection and forwarding logic

1. Structural hazards can occur because the divide unit is not fully pipelined
2. Since instructions have varying running times, the number of register writes per cycle may be larger than 1
3. WAW hazards are possible but WAR hazard is not possible

A pipeline that supports multicycle FP operations



Multicycle FP operations



Pipeline timing of a set of independent instructions

fmul.d	IF	ID	*M1*	M2	M3	M4	M5	M6	M7	MEM	WB
fadd.d		IF	ID	*A1*	A2	A3	A4	MEM	WB		
fld.d			IF	ID	*EX*	MEM	WB				
fsd				IF	ID	*EX*	MEM	WB			

Pipeline for Multicycle Operations

4. **Instructions can complete in a different order than issued, causing problems with exceptions**
5. **Stalls for RAW hazards will be more frequent because of longer latency of operations**

- ***Potential impact of RAW hazards***
- ***Problems arising from writes***
- **Since floating point register file has only one write port, it may result in higher number of structural hazards**

  Increase the number of write ports

  Additional write port will be used rarely

  Structural hazard is acceptable

- ***Implement Interlock to handle the above***
- **There are two ways to implement this interlock**

FP code sequence with stalls for RAW hazards

	Clock cycle number																
Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
fld f4,0(x2)	IF	ID	EX	MEM	WB												
fmul.d f0,f4,f6		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
fadd.d f2,f0,f8			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
fsd f2,0(x2)					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

Three instructions want to perform WB to FP register file simultaneously

	Clock cycle number										
Instruction	1	2	3	4	5	6	7	8	9	10	11
fmul.d f0,f4,f6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
fadd.d f2,f4,f6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
fld f2,0(x2)							IF	ID	EX	MEM	WB

Pipeline for Multicycle Operations

1. *Track the use of write port in the ID stage and stall an instruction before it issues*
   - Implemented with a shift register
     - Indicates when already issued instructions will use the register file
   - Maintains the property of stalling instructions only in the ID stage
     - Cost is the shift register and the write conflict logic
2. *Stall the conflicting instruction when it tries to enter either MEM or WB stage*
   - Give priority to the unit with the longest latency

Advantage - Detect the conflict at MEM stage where it is easy to see

Disadvantage - Complicates pipeline control as stalls can now arise from two places

Pipeline for Multicycle Operations

- ***Possibility of WAW hazards:***

  WAW hazard occur only when a useless instruction is executed

  Need to detect and handle it

There are two possible ways to handle WAW hazards

1. **Delay the issue of load instruction until the ADD.D enters MEM**
2. **Stamp out the result of ADD.D by detecting the hazard and changing the control so that ADD.D does not write its result**

- **It is difficult to detect this hazard**
- **Use a simpler solution**

  If an instruction in ID wants to write the same register as an instruction already issued, do not issue the instruction

Pipeline for Multicycle Operations

- **Possible Hazards to deal with**

1. **Hazards among FP instructions**
2. **Hazards between a FP and an integer instruction**

- **Only FP load-stores and FP-integer register moves**

Separate register files

- **Three checks that must be performed in the ID stage before an *instruction issue* can take place**

i. ***Check for structural hazards*:**

- **Wait until the required functional unit is not busy**
- **Make sure the register write port is available when it will be needed**

ii. ***Check for a RAW data hazard*:**

- **Wait until the source registers are not listed as pending destinations in a pipeline register that will not be available when this instruction needs the result**

Pipeline for Multicycle Operations

iii. ***Check for a WAW data hazard*: Determine if any instruction in A1, A2,.., A4; D; M1,.., M7 has the same destination register as this instruction**

- **Hazard detection and forwarding logic are similar to the RISC-V integer pipeline although it is more complex than the former**
- **Multicycle FP operations also introduce problems for exception mechanisms**

- **Maintaining Precise Exceptions**

  An instruction issued earlier completes after an instruction issued later

  ***Out-of-order* completion**

  Imprecise exceptions can occur

Pipeline for Multicycle Operations

Four approaches are used to deal with this problem

1. Ignore the problem and settle for imprecise exception
   - IEEE floating point standard requires precise exception
   - Two-mode implementation is seen in most processors
2. Buffer the results of an operation until all the operations that were issued earlier are complete
   - Expensive when difference in running times among operations is large
3. Two viable variations
   - *History file*: Keep track of original values of registers
   - *Future file*: Keeps the newer values of registers
     - Update later from the future file

Pipeline for Multicycle Operations

3. **Allow the exception to become imprecise**
   - Keep enough information so that trap handling routines can create a precise sequence for the exception
- **Need to know the operations in the pipeline and their PCs**
   - Only floating point instructions overlap need to be handled
4. **A hybrid scheme**
   - Allows the instruction issue to continue only if it is certain that all instructions before the issuing instruction will complete without causing an exception
     - **May require to stall the CPU to maintain precise exceptions**