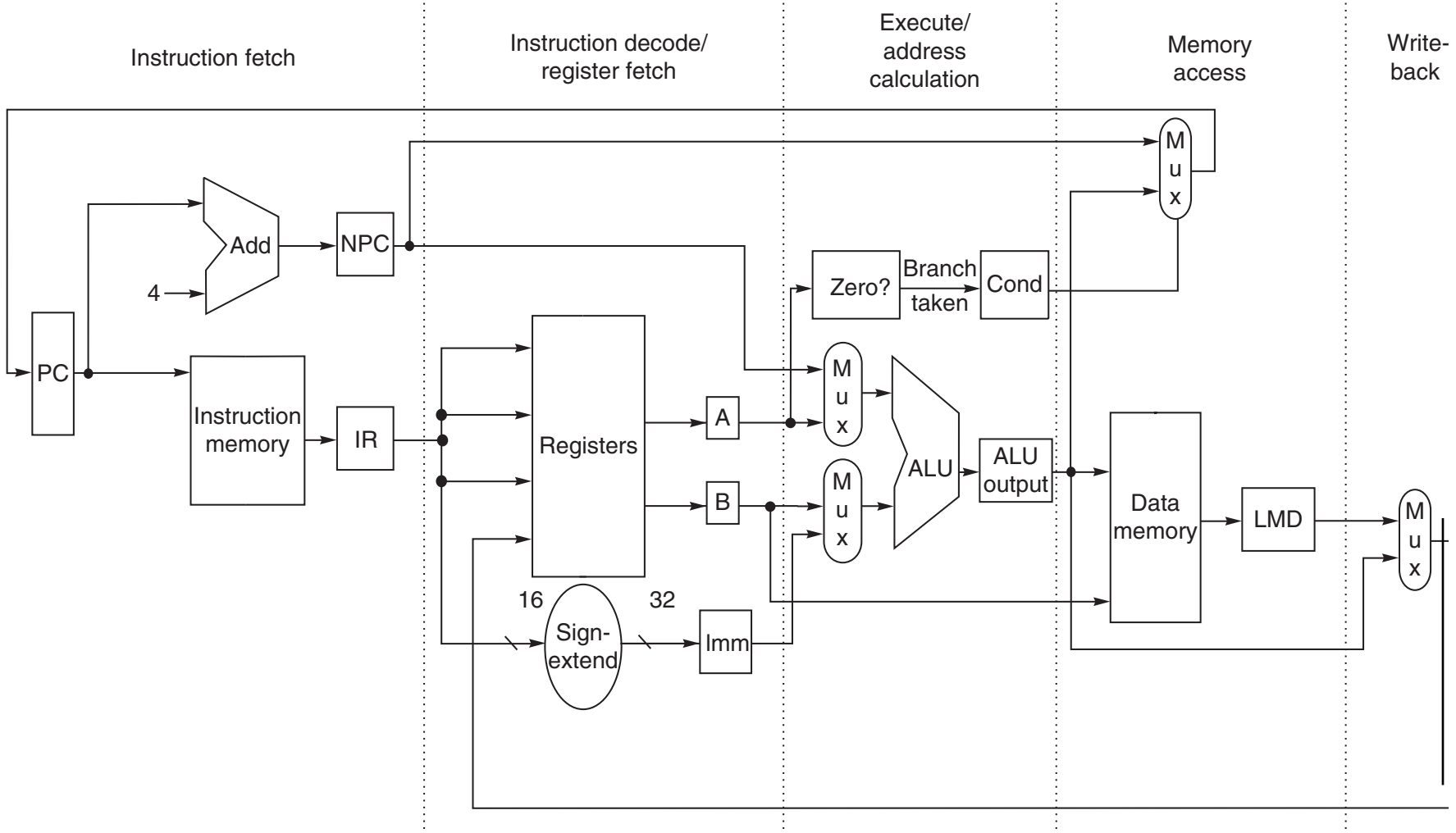
EEL 5764 Computer Architecture

Sandip Ray
Department of Electrical and Computer Engineering
University of Florida

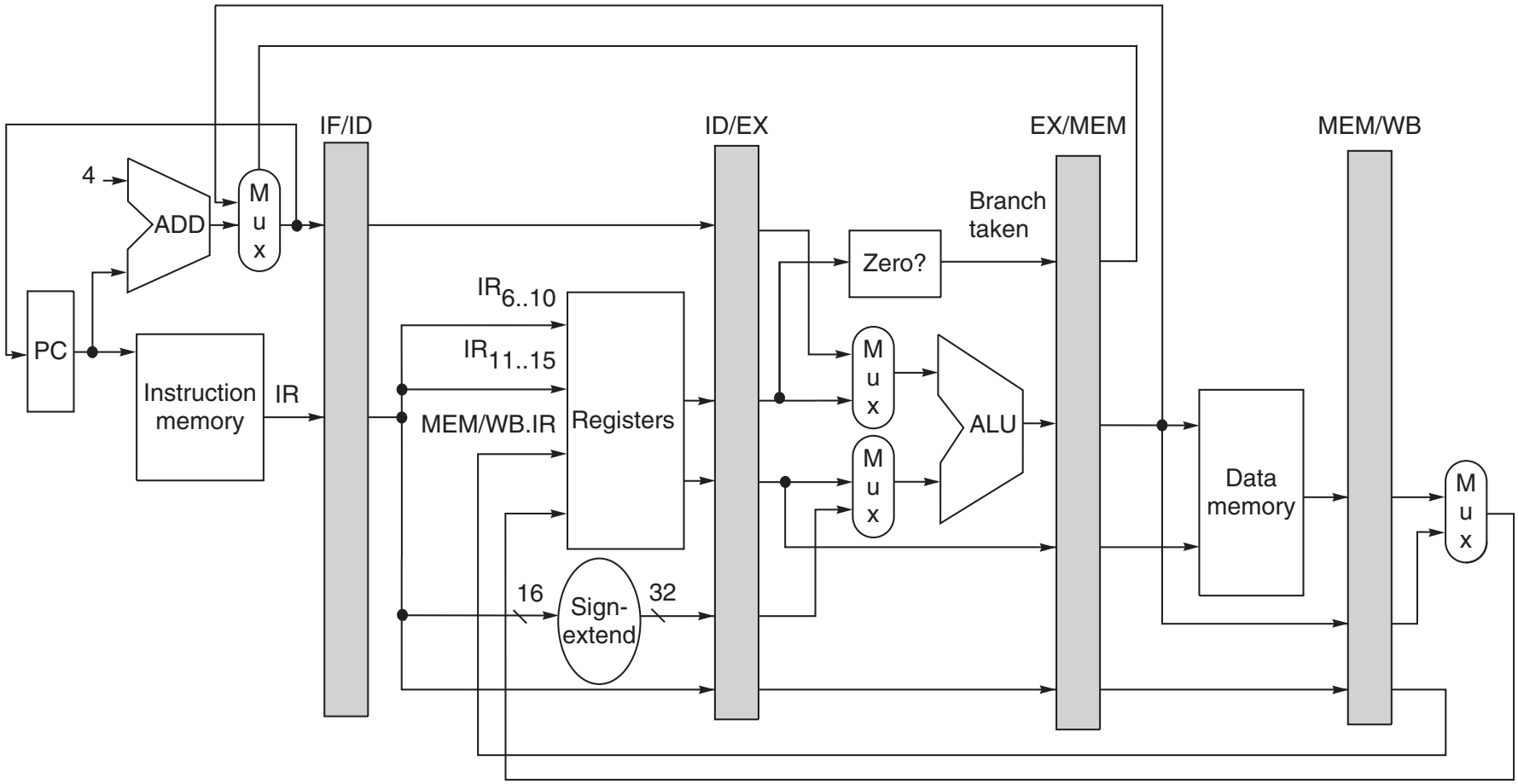
Lecture 21:

- Pipeline Implementation
- Techniques for Instruction-level Parallelism

Non-Pipelined Version



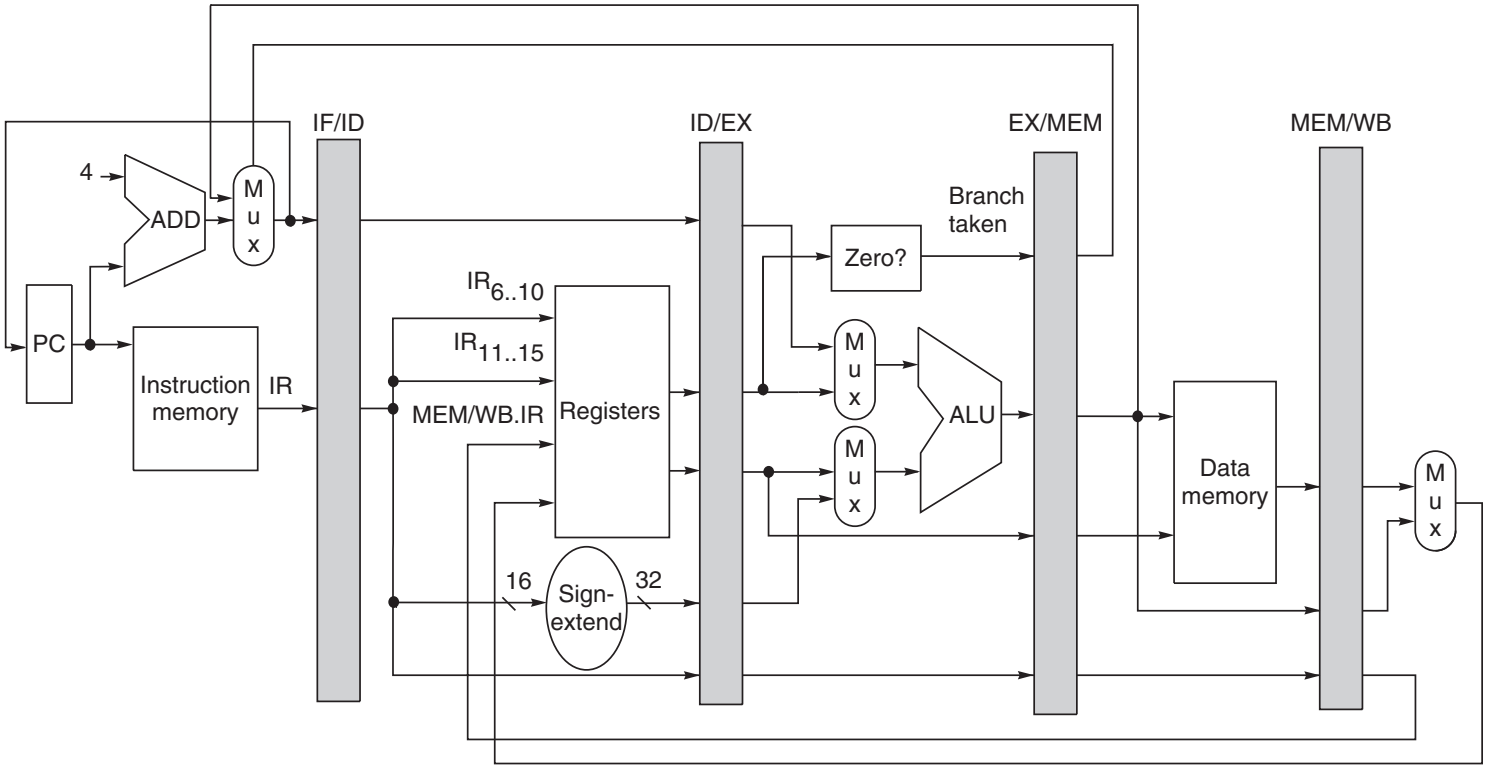
A Basic MIPS Pipeline



Temporary registers become part of PL registers.

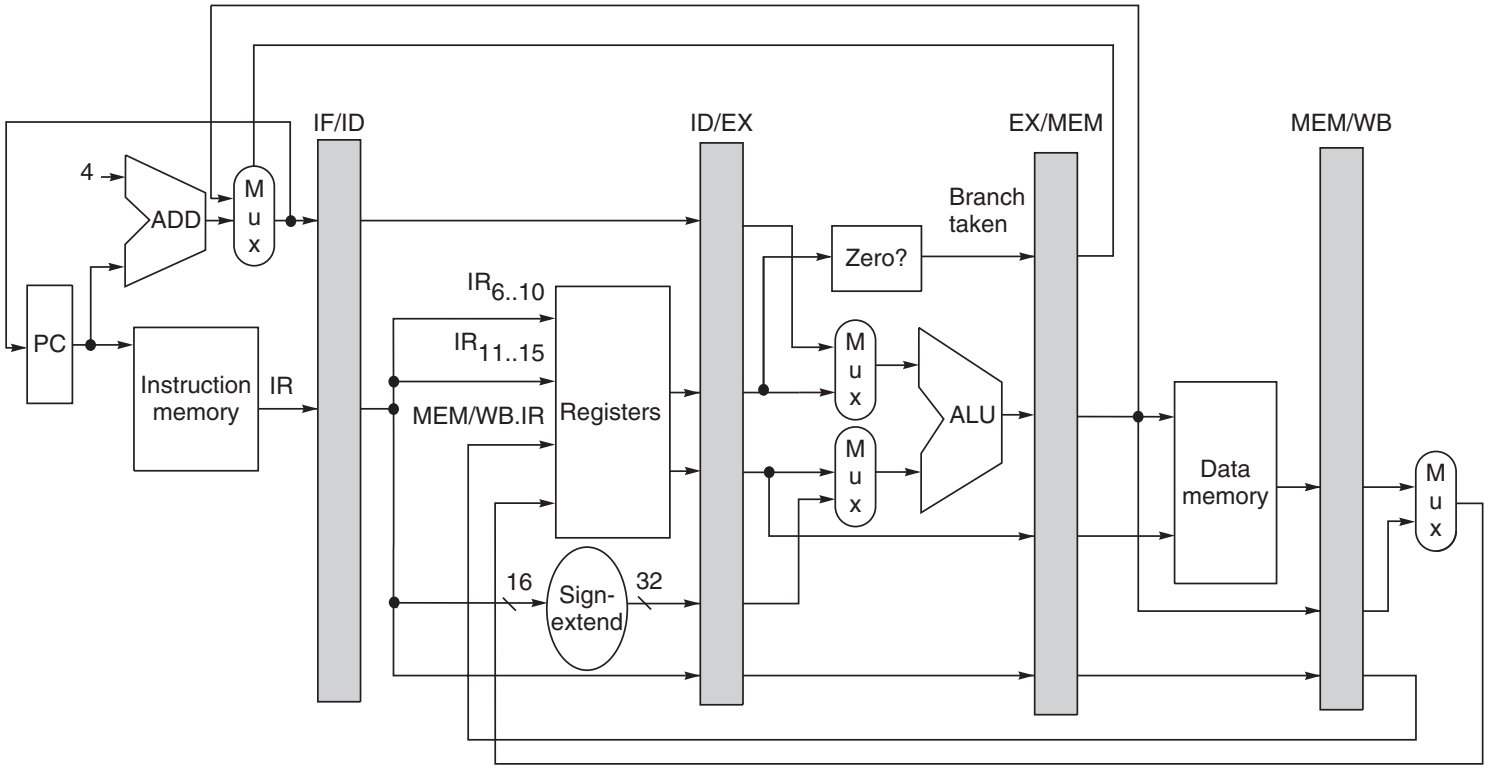
PL registers carry partial results from one stage to the next.

A Basic MIPS Pipeline



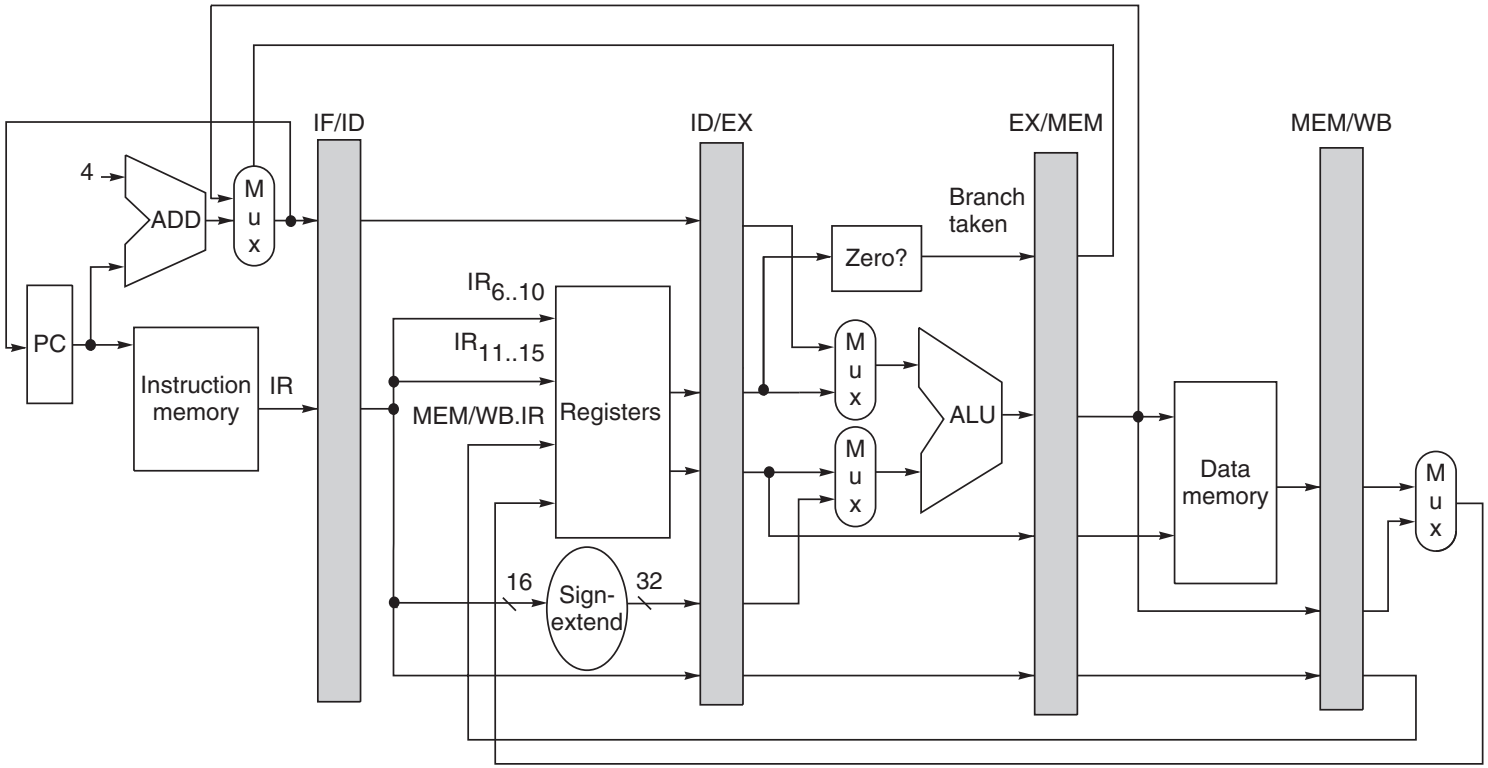
Stage	Any instruction
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM.ALUOutput} else {PC+4});
ID	ID/EX.A ← Regs[IF/ID.IR[rs]]; ID/EX.B ← Regs[IF/ID.IR[rt]]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);

A Basic MIPS Pipeline



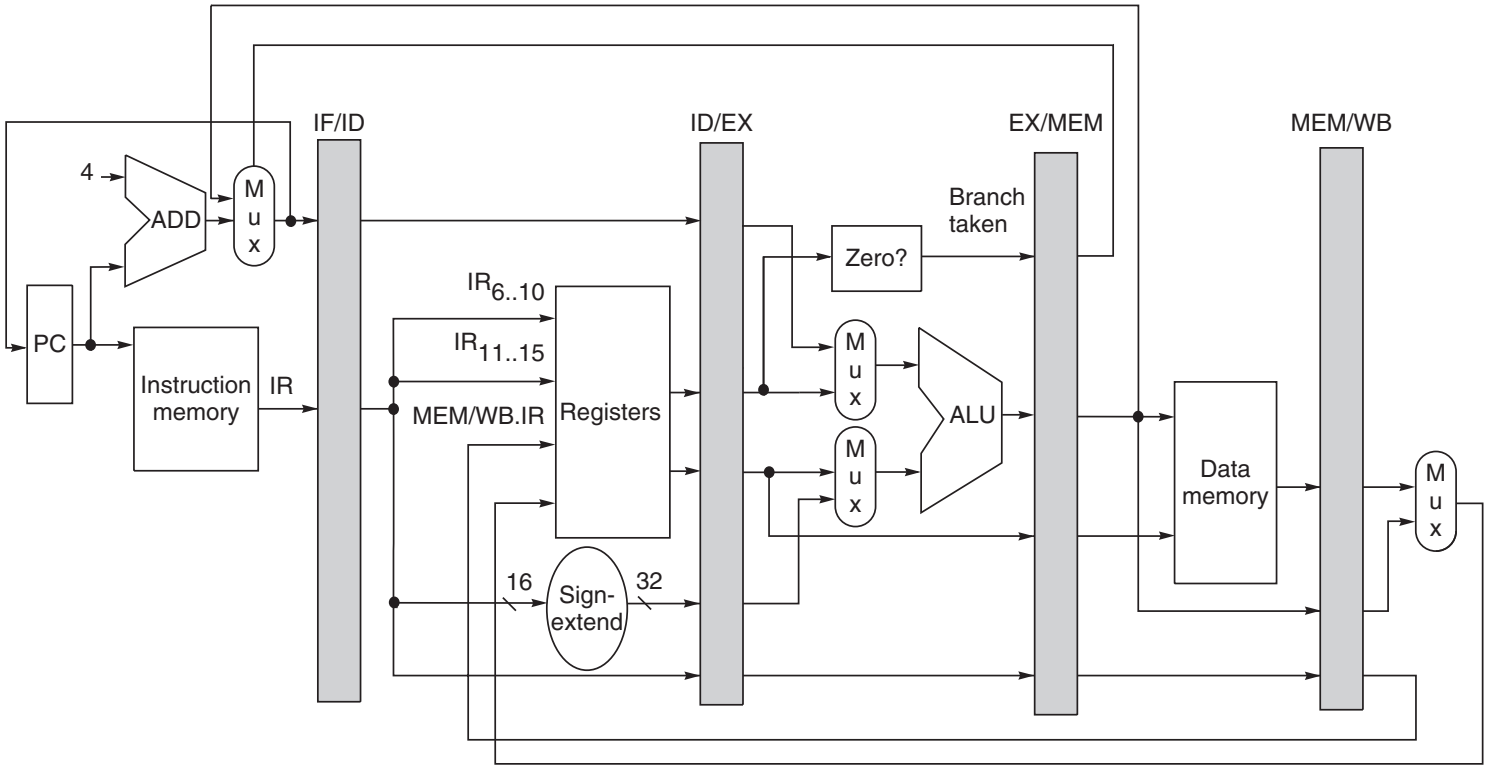
	ALU instruction	Load or store instruction	Branch instruction
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← ID/EX.A func ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm;	EX/MEM.IR to ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.B ← ID/EX.B;	EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm << 2); EX/MEM.cond ← (ID/EX.A == 0);

A Basic MIPS Pipeline



	ALU instruction	Load or store instruction	Branch instruction
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;	

A Basic MIPS Pipeline



	ALU instruction	Load or store instruction	Branch instruction
WB	Regs[MEM/WB.IR[rd]] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR[rt]] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR[rt]] ← MEM/WB.LMD;	

Hazard Detection

- **Instruction issue**: move from ID to EX.
- Detect data hazards during ID
  - → Stall the instruction before it is issued
  - → Insert pipeline bubbles (no-ops) by changing control fields to 0s
- Early detection of interlocks (e.g., due to a load) reduces complexity
- Detect load interlock by comparing:
  - → Source registers in IF/ID (consumers) with
  - → Destination register in ID/EX or EX/MEM (producer)

Hazard Detection – Example

Situation	Example code sequence	Action

No dependence	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R6,R7`
	`OR R9,R6,R7`
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

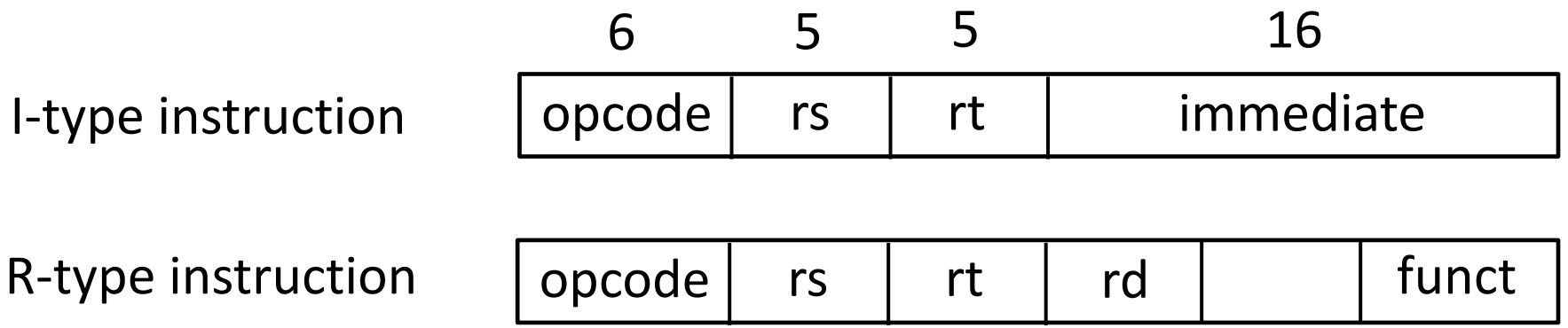
Dependence requiring stall	`LD R1,45(R2)`
	`DADD R5,R1,R7`
	`DSUB R8,R6,R7`
	`OR R9,R6,R7`
		Comparators detect the use of R1 in the DADD and stall the DADD (and DSUB and OR) before the DADD begins EX.

Dependence overcome by forwarding	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R1,R7`
	`OR R9,R6,R7`
		Comparators detect use of R1 in DSUB and forward result of load to ALU in time for DSUB to begin EX.

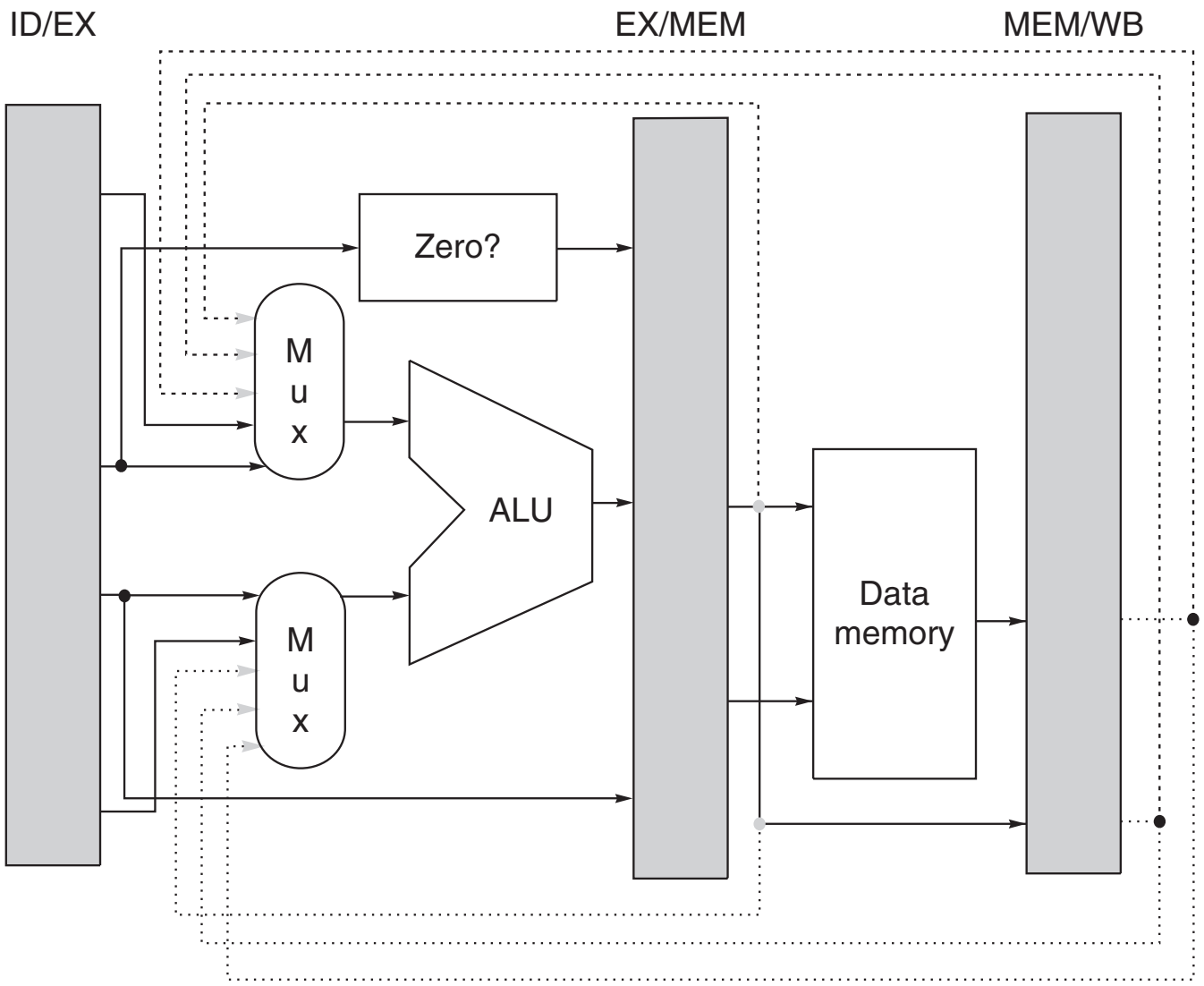
Dependence with accesses in order	`LD R1,45(R2)`
	`DADD R5,R6,R7`
	`DSUB R8,R6,R7`
	`OR R9,R1,R7`
		No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

Hazard Detection – Example

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



Forwarding



Forwarding

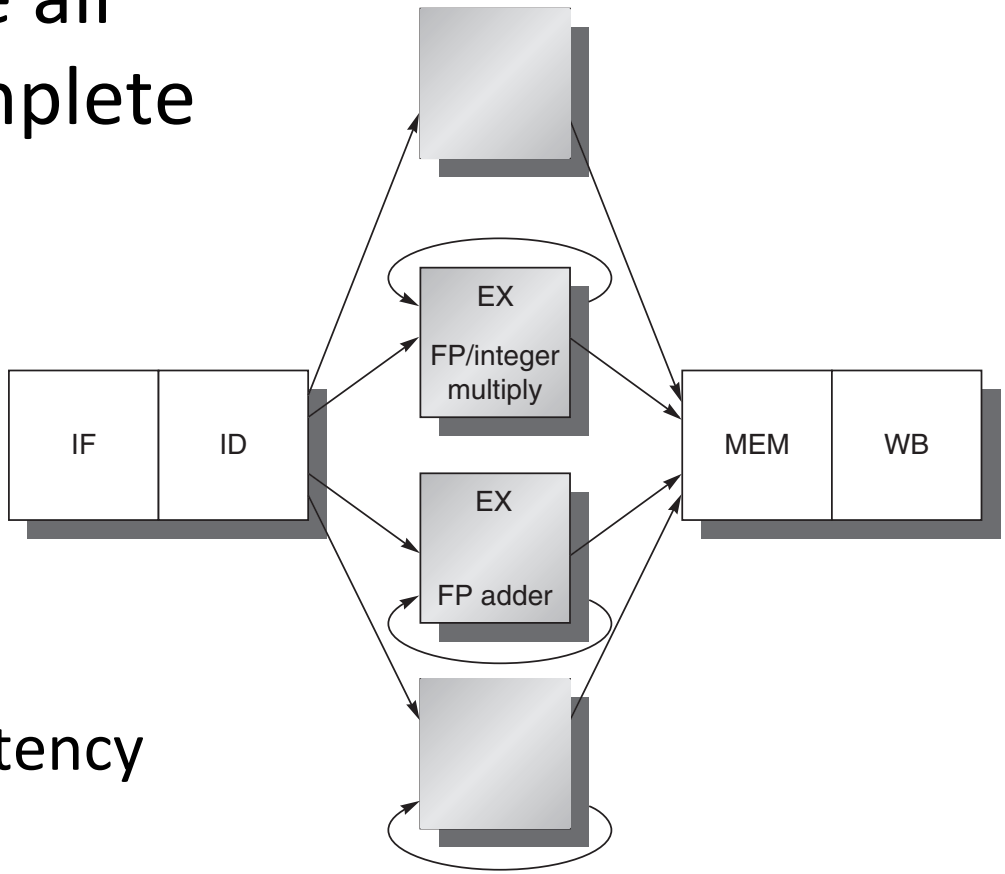
- Sources: outputs of
  → ALU or Memory
- Destinations: inputs of
  → ALU, Memory, Branch decision
- Forwarding done by controlling MUXs of the inputs at destinations
- Compare registers in EX/MEM or MEM/WB with registers in ID/EX and EX/MEM.
- Ex:

```
LD R1, 45(R2)
DADD ...
DSUB R8, R1, R7
```

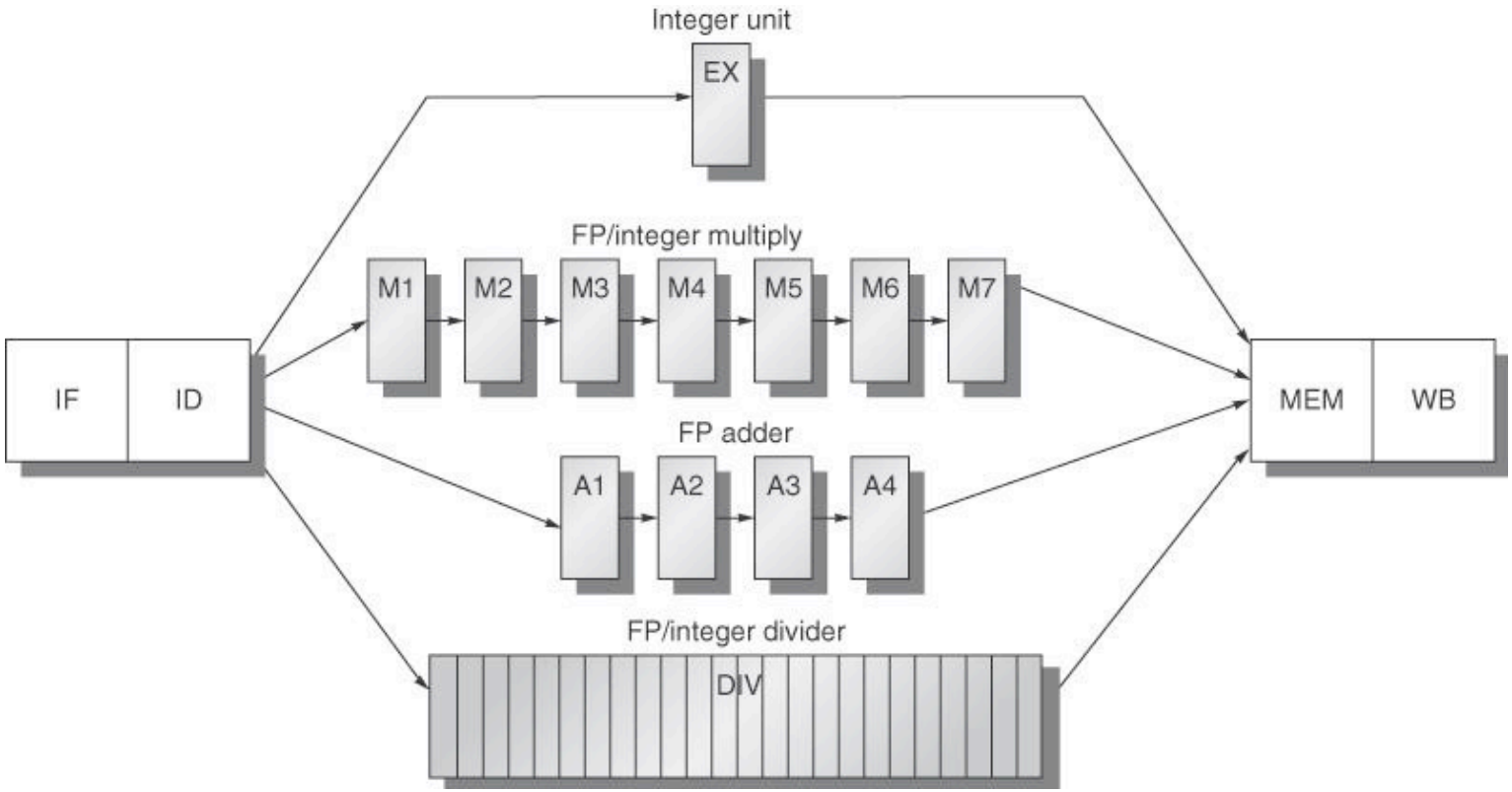
MEM/WB.IR[rt] == ID/EX.IR[rt]
controls an input MUX to ALU.

Extend Pipeline to Handle Long Latency

- Impractical to require all FP operations to complete in 1 cycle
  - → Longer cycle time
  - → Large area overhead
- Long FP operations
  - → take multiple cycles
- Non-pipelined
  - → Stalls caused by long latency FP operations

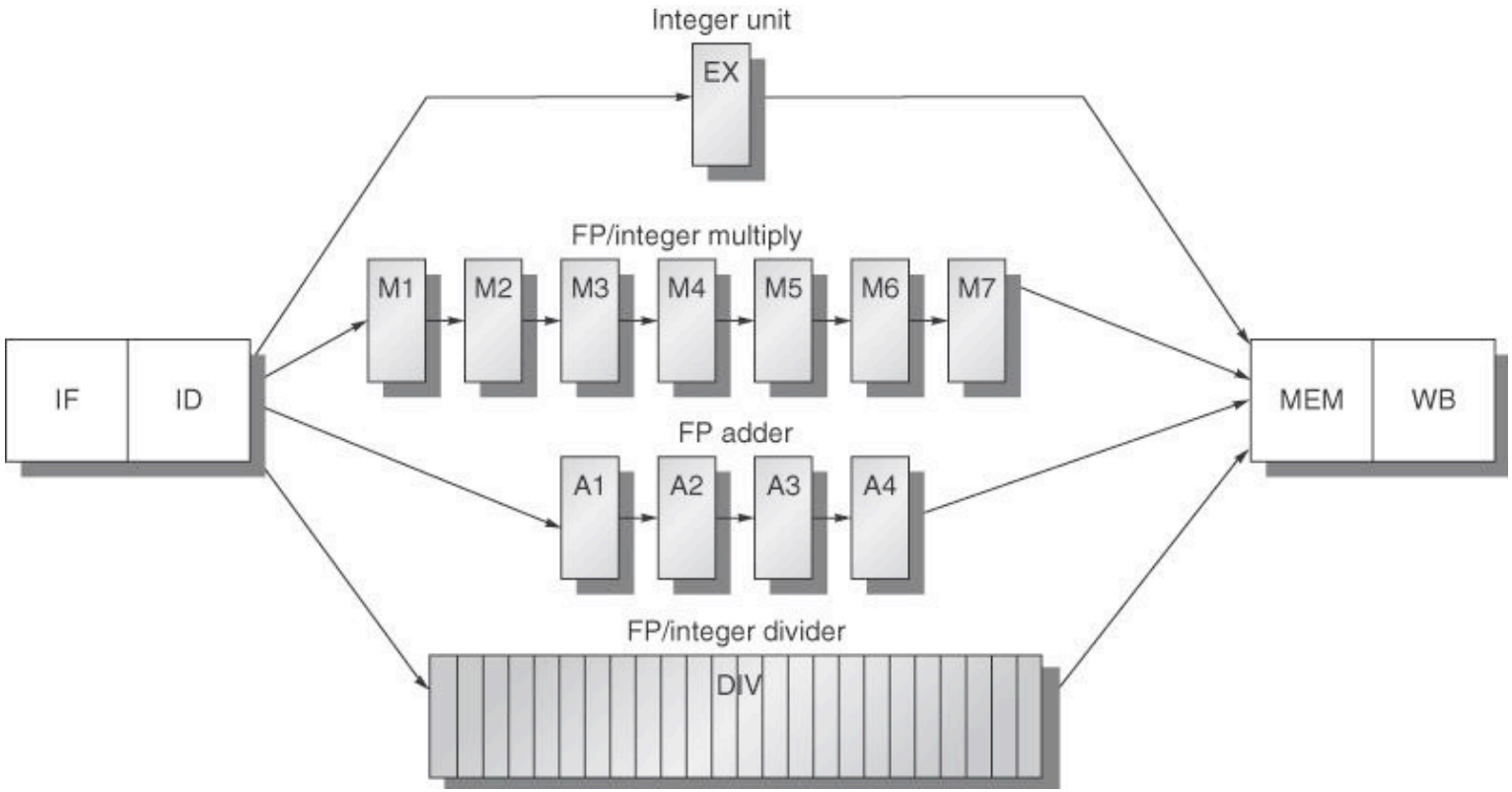


Extend Pipeline to Handle Long Latency



- Pipelining functional units allow multiple instructions to be executed.

Extend Pipeline to Handle Long Latency



Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

Extend Pipeline to Handle Long Latency

- Structural hazards can occur due to long latency
- Simultaneous register writes are possible
  → due to various execution latencies
- WAW hazards are now possible
- Instructions can complete in different orders

	Clock cycle number										
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
`MUL.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				
`ADD.D F2,F4,F6`				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
`L.D F2,0(R2)`							IF	ID	EX	MEM	WB

Extend Pipeline to Handle Long Latency

- Long latency causes more 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**
L.D	F4,0(R2)	IF	ID	EX	MEM	WB												
MUL.D	F0,F4,F6		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADD.D	F2,F0,F8			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
S.D	F2,0(R2)					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

Instruction Level Parallelism

Instruction-Level Parallelism

- Overlap executions of multiple instructions
- When exploiting instruction-level parallelism, goal is to maximize CPI
  - → Pipeline CPI =
    - ➤ Ideal pipeline CPI +
    - ➤ Structural stalls +
    - ➤ Data hazard stalls +
    - ➤ Control stalls
- Two approaches to exploiting ILP
  - → Static scheduling
  - → Dynamic scheduling

Dependences

- Parallelism with **basic block** is limited
  - → Typical size of basic block = 3-6 instructions
  - → Must optimize across branches
- Increase ILP – Loop-Level Parallelism
  - → Unroll loop statically or dynamically
  - → Use SIMD (vector processors and GPUs)
- Challenges: Data dependency
  - → Instruction *j* is data dependent on instruction *i* if

    Instruction *i* produces a result that may be used by instruction *j*
    Instruction *j* is data dependent on instruction *k* and instruction *k* is data dependent on instruction *i*
- Dependent instructions cannot be executed simultaneously

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall
- **Data dependence** conveys:
  - → Possibility of a hazard
  - → Order in which results must be calculated
  - → Upper bound on exploitable instruction level parallelism
- Dependencies that flow through memory locations are difficult to detect
  - → SD R1, 45(R5)
  - → LD R2, 90(R7)

Name Dependence

- Two instructions use the same name but no flow of information
  - → Not a true data dependence, *but is a problem when reordering instructions*
  - → **Antidependence**: instruction j writes a register or memory location that instruction i reads
    - ➤ Initial ordering (i before j) must be preserved
  - → **Output dependence**: instruction i and instruction j write the same register or memory location
    - ➤ Ordering must be preserved
- To resolve, use **renaming** techniques