Pipelining: Basic and Intermediate Concepts

Appendix C

Outline

- Introduction
- The major hurdle of pipelining-pipeline hazards
- How is pipelining implemented?
- What Makes Pipelining Hard to Implement?
- Extending the RISC-V Pipeline to Handle Multicycle Operations
- Putting It All together: The MIPS R4000 Pipeline
- Crosscutting Issues

Introduction

- What is Pipelining

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.

Pizza example

Let's look at two ways to make pizzas.

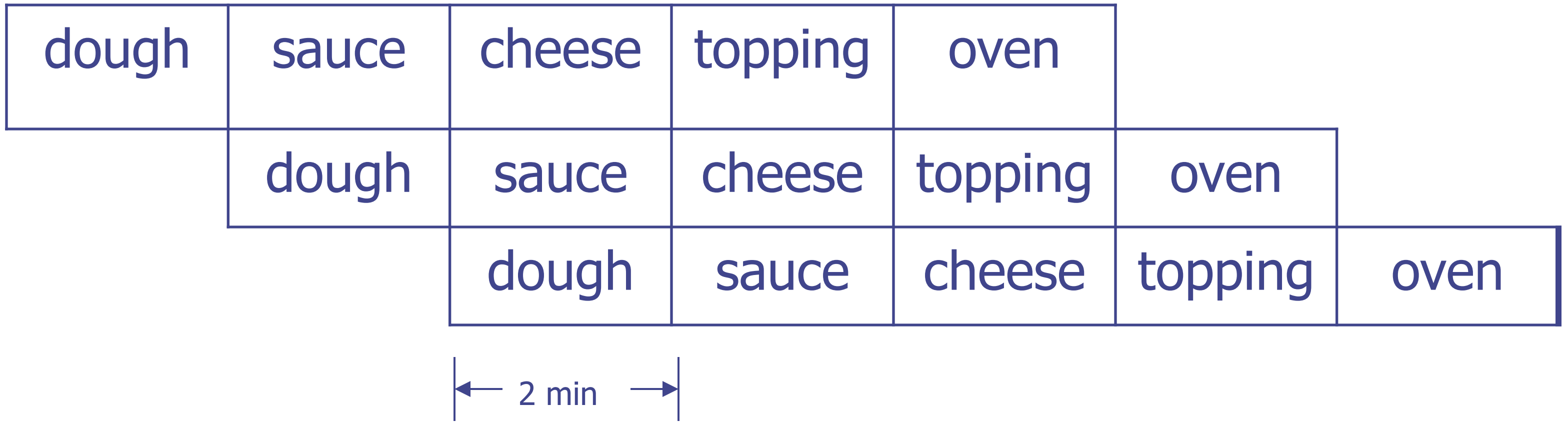
The first away is based on a sequential procedure.

- Toss the dough (2 min)
- Add the sauce (2 min)
- Add the cheese (2 min)
- Add the toppings (2 min)
- Bake in oven (2 min – to help this example!)

10 mins for one pizza
Throughput= 1/10 pizza/min

Since each step in pizza making is distinct, we can start a second pizza once we have tossed the dough for the first.

Therefore, a pizza pipelining schedule can be arranged as shown below.


dough sauce cheese topping oven
dough sauce cheese topping oven
dough sauce cheese topping oven
2 min

2 mins for one pizza
Throughput = 0.5 pizza/min

As shown in the pizza example, a pipeline can be separated into many steps. Each of these steps is called a ***pipe stage*** or pipe segment.

There are 5 pipe stages in the pizza pipeline.

◆The basics of a RISC instruction set

The pipeline can be effectively used to enhance the throughput of the RISC (reduced instruction set computer).

RISC architectures are characterized by the following properties:

(a) All operations on data apply to data in registers and typically change the entire register.
(b) The only operations that affect memory are load and store operations.
(c) The instruction formats are few in number with all instructions typically being one size.

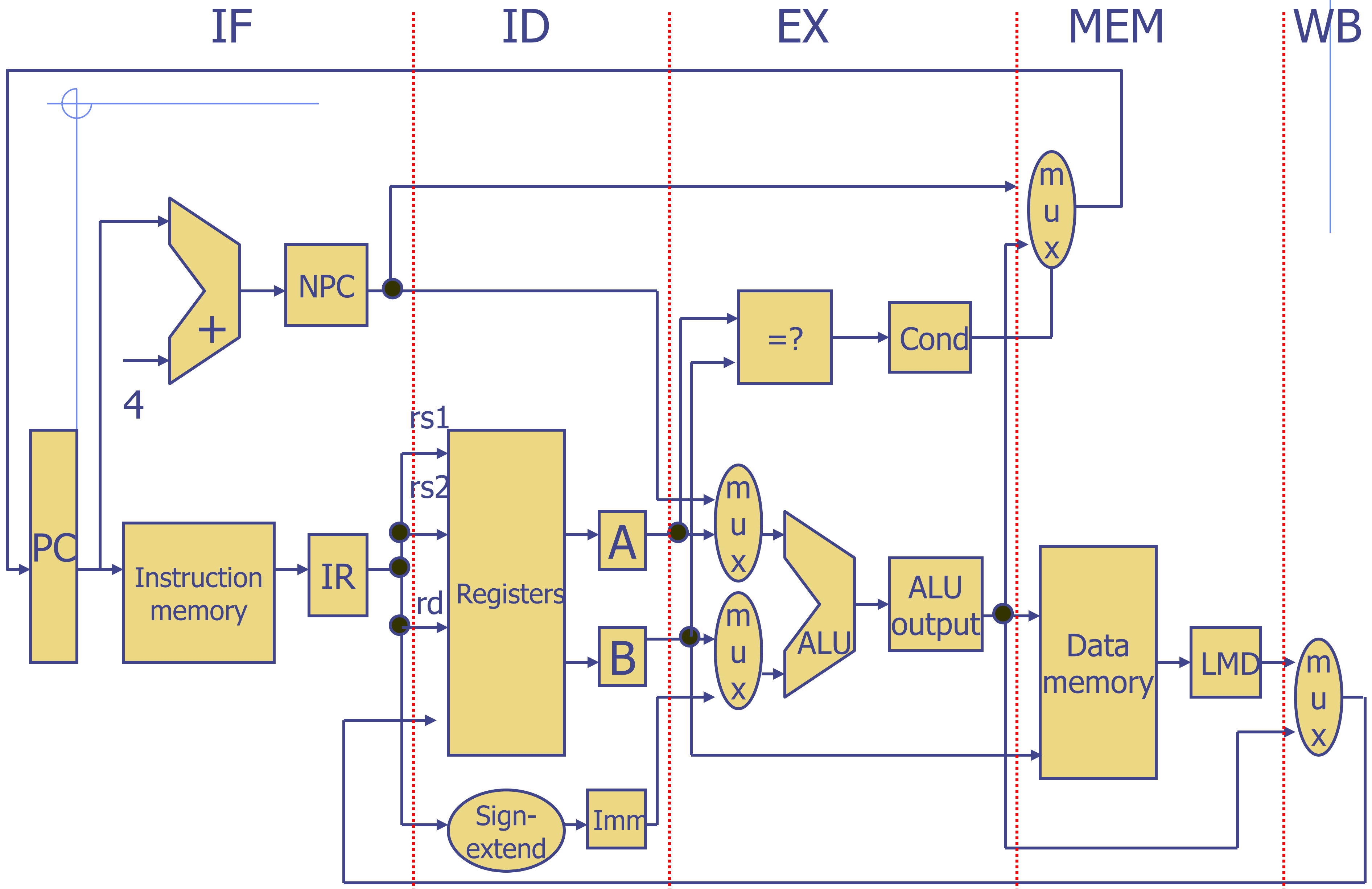
Most RISC architectures have three classes of instructions:

1. ALU instructions
2. Load and store instructions
3. Branches and jumps

◆A simple implementation of a RISC instruction set

Here we consider the implementation of a RISC instruction set without pipelining. In this implementation, every instruction takes at most 5 clock cycles. The 5 cycles are shown below.

1. Instruction fetch cycle (IF),
2. Instruction decode/register fetch cycle (ID),
3. Execution/effective address cycle (EX),
4. Memory access/branch completion cycle (MEM),
5. Write-back cycle (WB).



IF
ID
EX
MEM
WB
+
4
NPC
PC
Instruction memory
IR
rs1
rs2
rd
Registers
A
B
Sign-extend
Imm
m u x
m u x
ALU
=?
Cond
m u x
ALU output
Data memory
LMD
m u x

All the instructions have identical IF

- **IF**

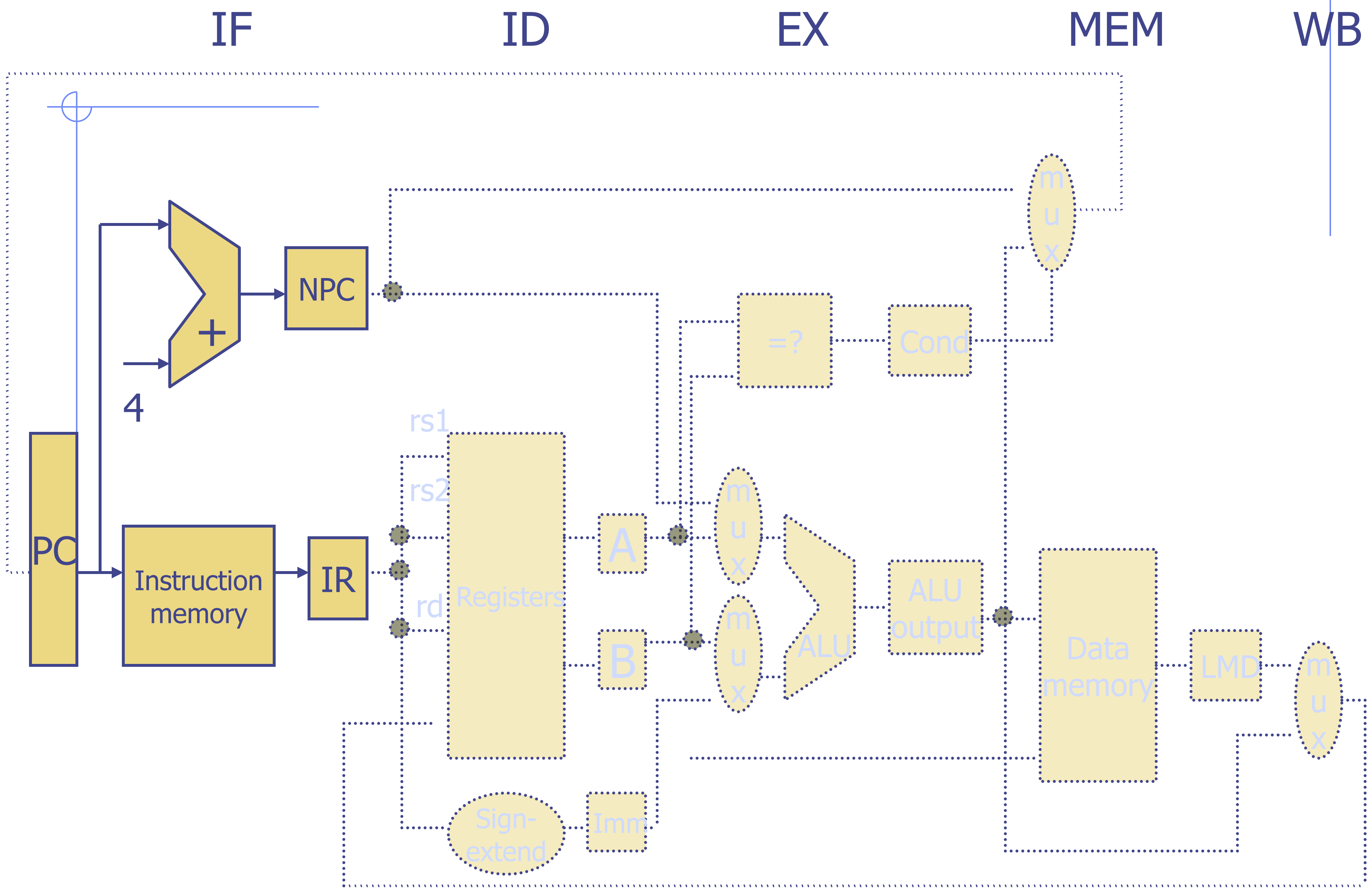
1. Send out the PC and fetch the instruction from memory into the IR.

   IR $\leftarrow$ Mem[PC];

2. Increment the PC by 4 to address the next sequential instruction.

   NPC $\leftarrow$ PC+4;

IF


IF
ID
EX
MEM
WB
+
4
NPC
PC
Instruction memory
IR
rs1
rs2
rd
Registers
A
B
Sign-extend
Imm
m u x
m u x
ALU
=?
Cond
ALU output
m u x
Data memory
LMD
m u x

All the instructions have identical ID

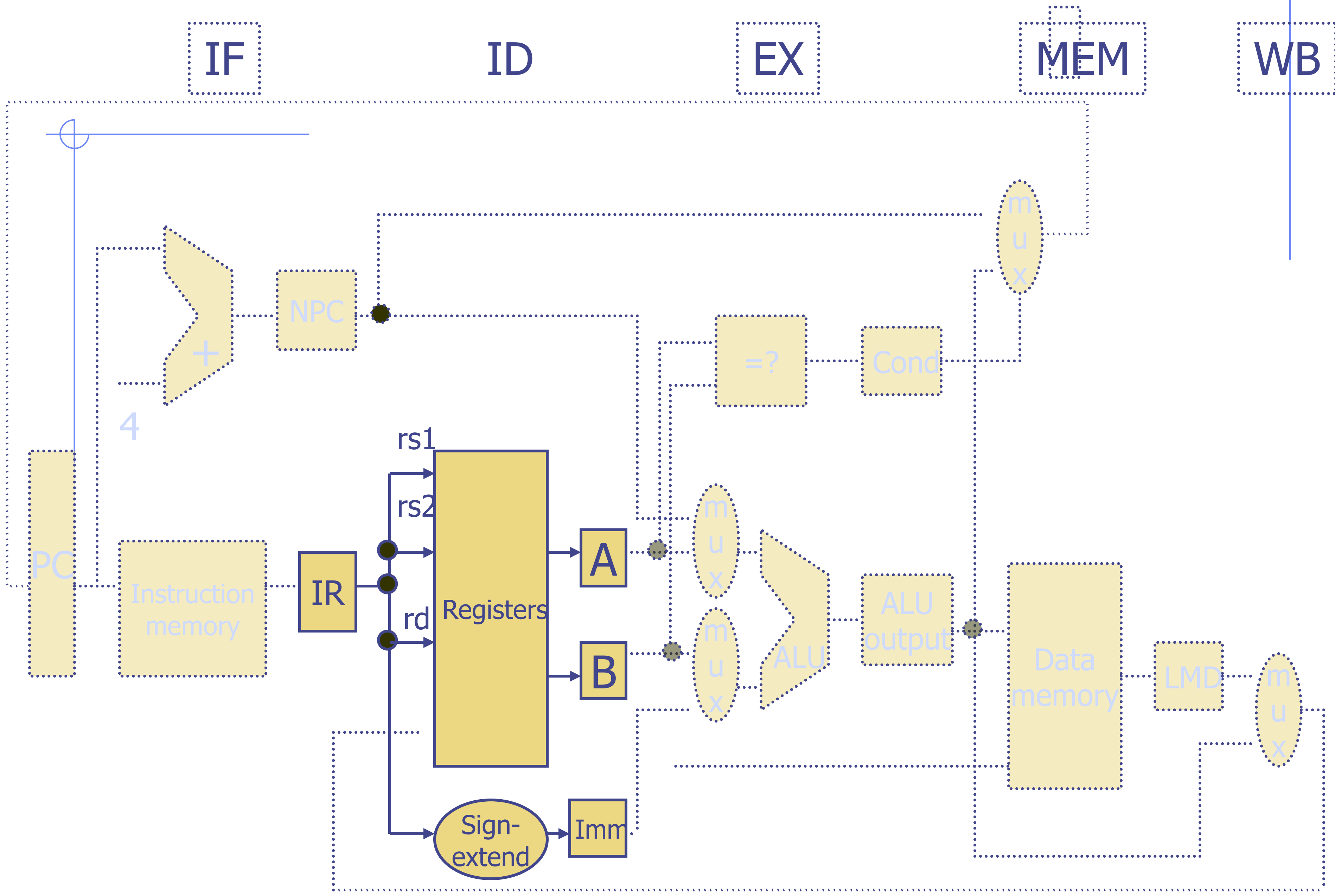
- **ID**

1. Decode the instruction.
2. Access the register file to read the registers. The output of the general purpose registers are read into two temporary registers (A and B) for use in later clock cycles

A ← Regs[rs1];
B ← Regs[rs2];

3. The immediate field of the IR is also sign extended and store into temporary register Imm.

Imm ← sign-extended immediate field of IR;



IF
ID
EX
MEM
WB
NPC
+
4
PC
Instruction memory
IR
rs1
rs2
rd
Registers
A
B
Sign-extend
Imm
=?
Cond
m u x
m u x
m u x
ALU
ALU output
Data memory
LMD
m u x

The EX, MEM and WB stages may be different for different instructions.

Load instruction

- **EX**

  ALUOutput ← A + Imm;

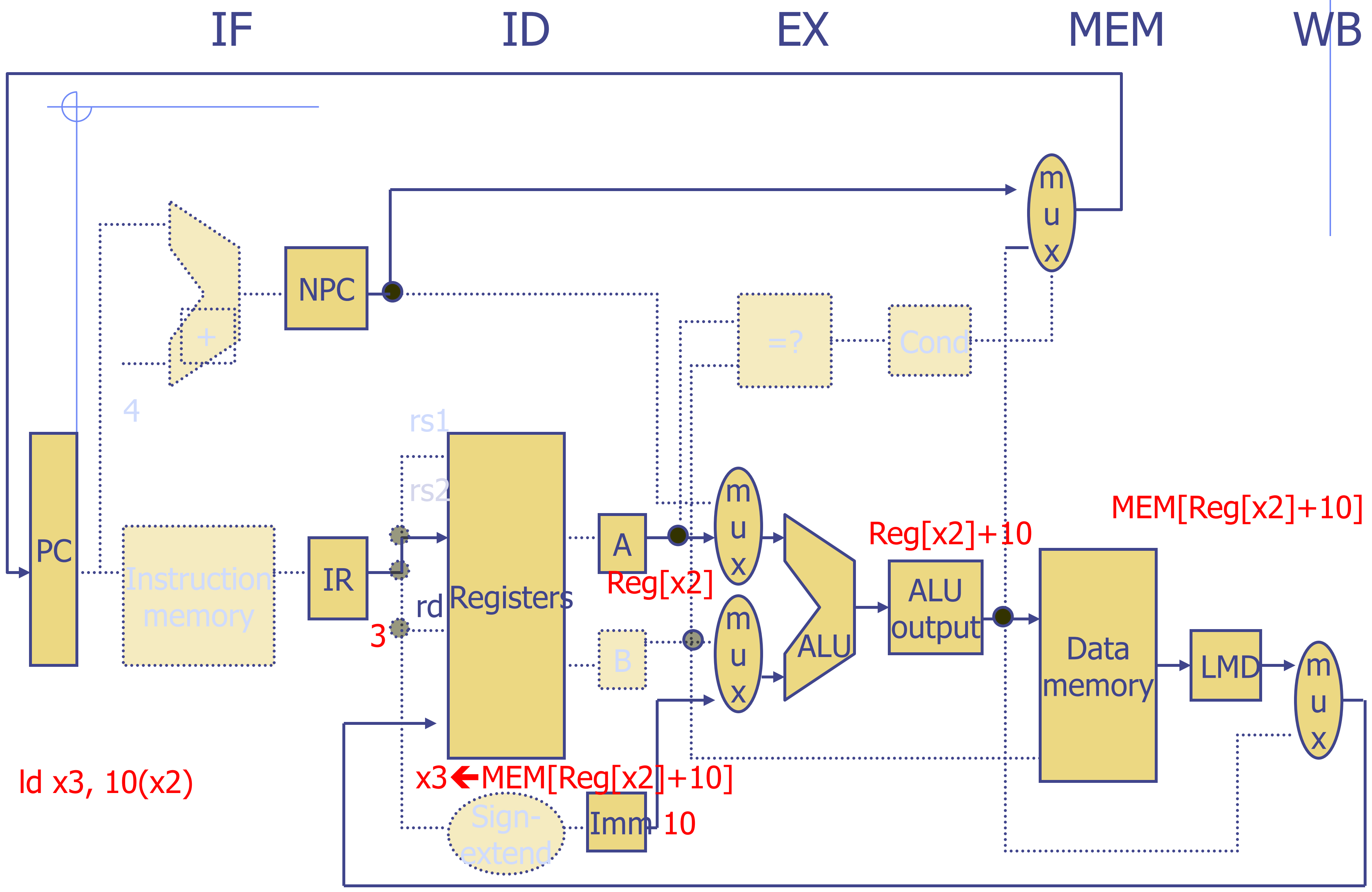
- **MEM**

  LMD ← Mem[ALUOutput];
  PC ← NPC;

- **WB**

  Regs[rd] ← LMD;

Load instruction(EX, MEM and WB)



ld x3, 10(x2)

Store instruction

- EX

  ALUOutput ← A + Imm;

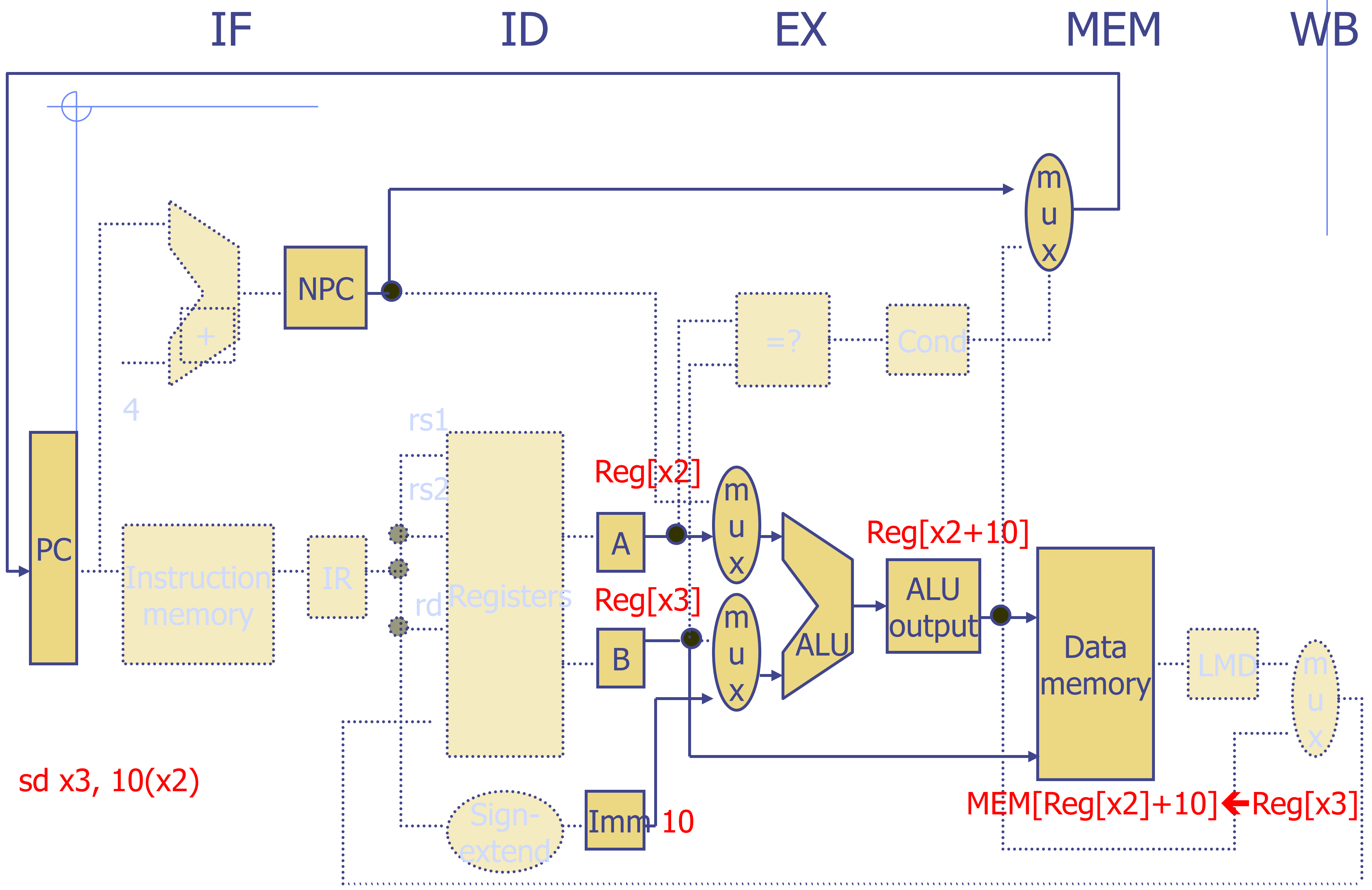
- MEM

  Mem[ALUOutput] ←B;
  PC ← NPC;

- WB

  No Operation

Store instruction(EX, MEM and WB)



sd x3, 10(x2)

Register-Register ALU instruction

- EX

  ALUOutput ← A func B;

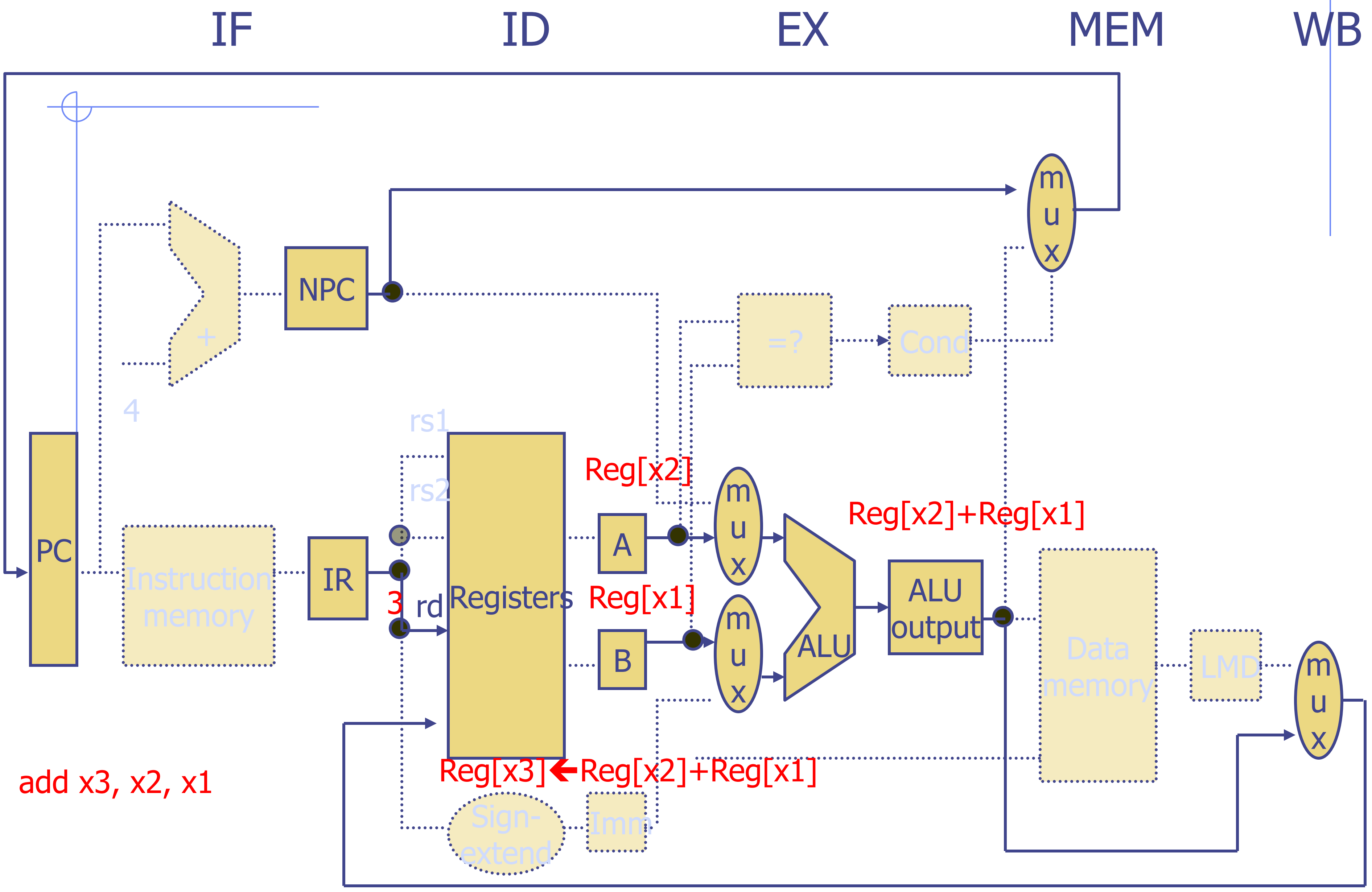
- MEM

  PC ← NPC;

- WB

  Regs[rd] ← ALUOutput;

Register-register ALU (EX, MEM and WB)



add x3, x2, x1

Register-Immediate ALU instruction

- EX

  ALUOutput ← A op Imm;

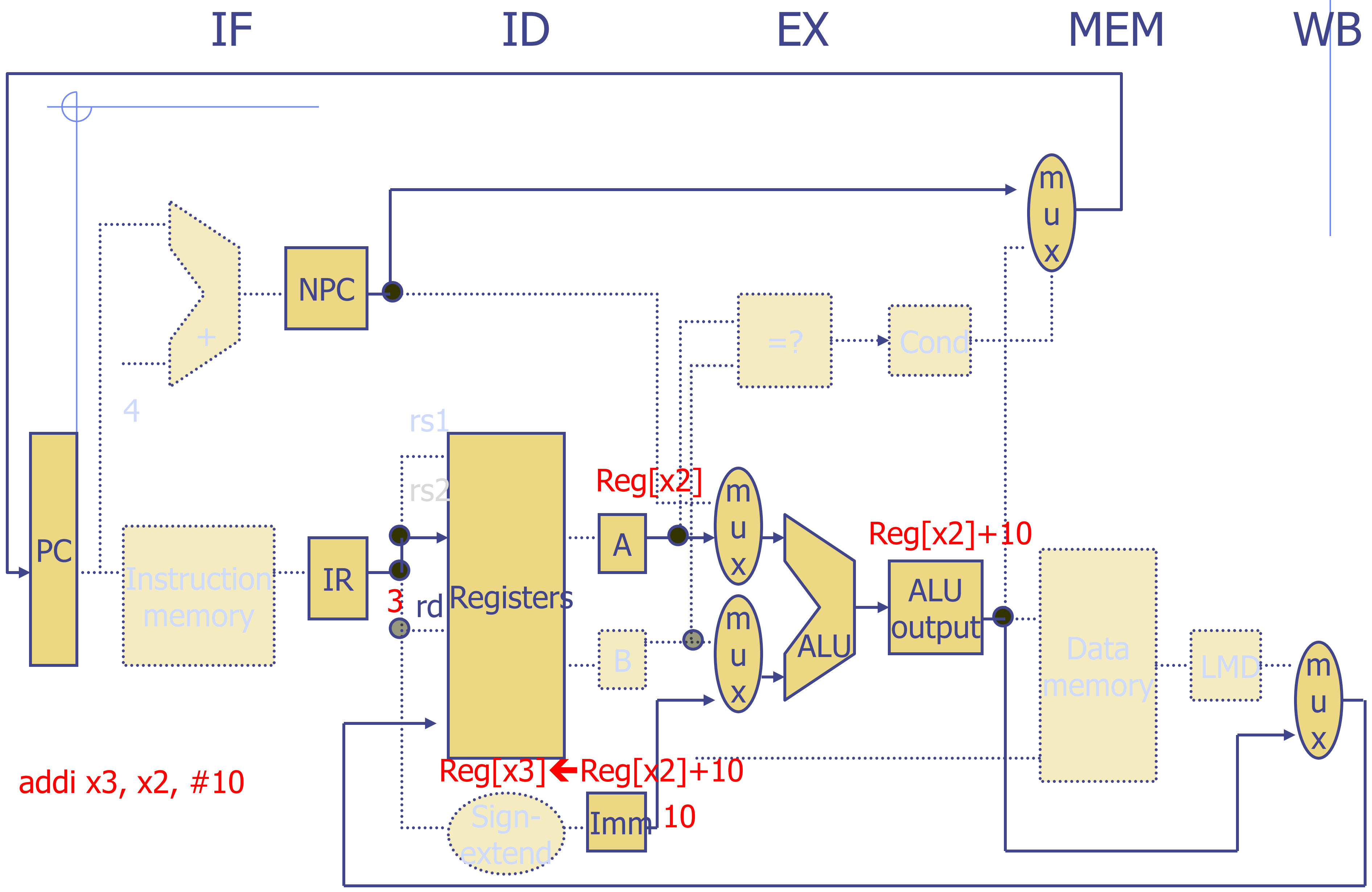
- MEM

  PC ← NPC;

- WB

  Regs[rd] ← ALUOutput;

Register-Immediate ALU (EX, MEM and WB)



addi x3, x2, #10

Branch instruction

- **EX**

  ALUOutput ← NPC + (Imm << 2);
  Cond ← (A==B)

for BEQ

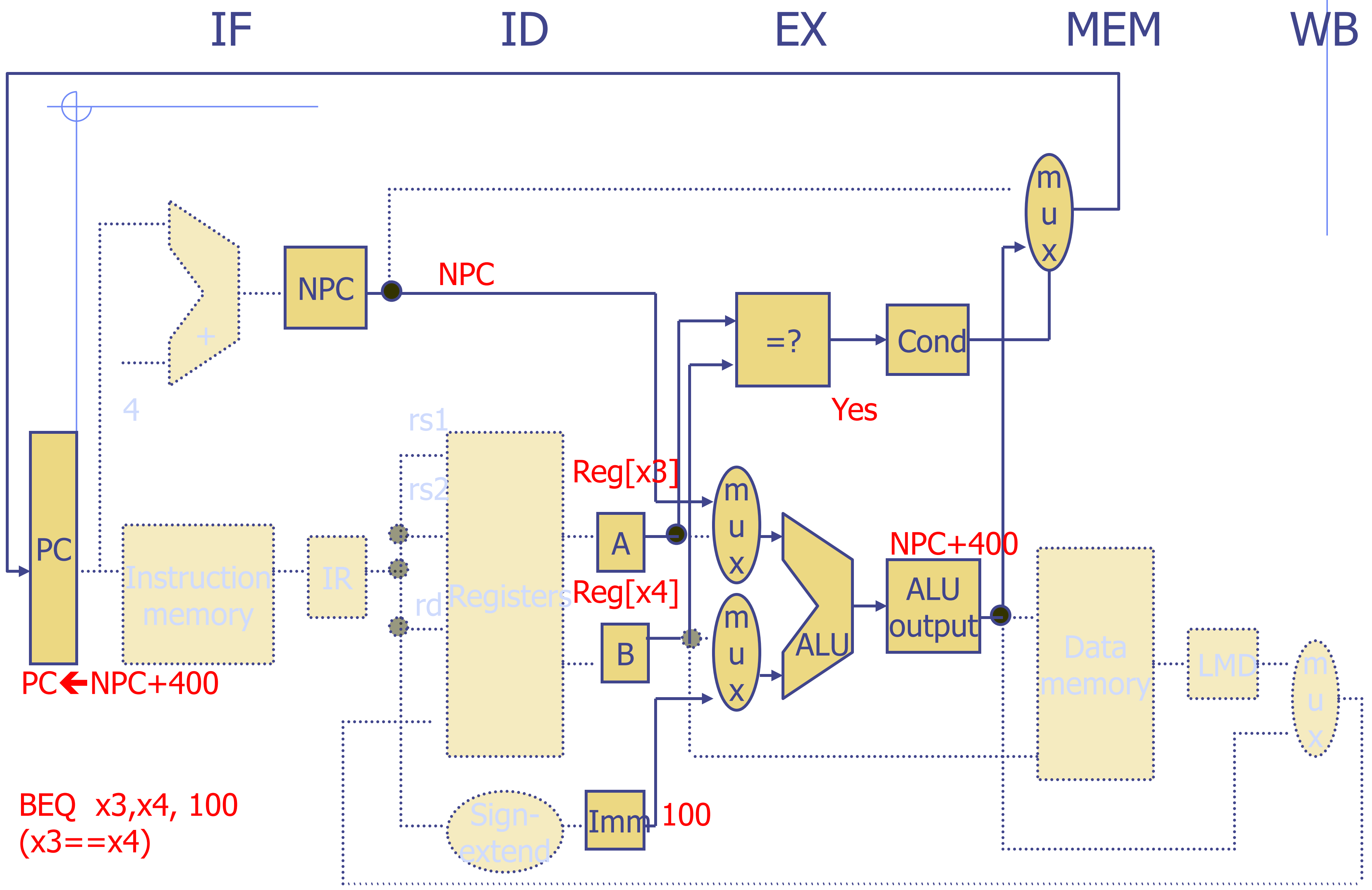
- **MEM**

  If (Cond) PC ← ALUOutput
  else PC ← NPC;

- **WB**

  No operation

Branch (EX, MEM and WB)



BEQ x3,x4, 100
(x3==x4)

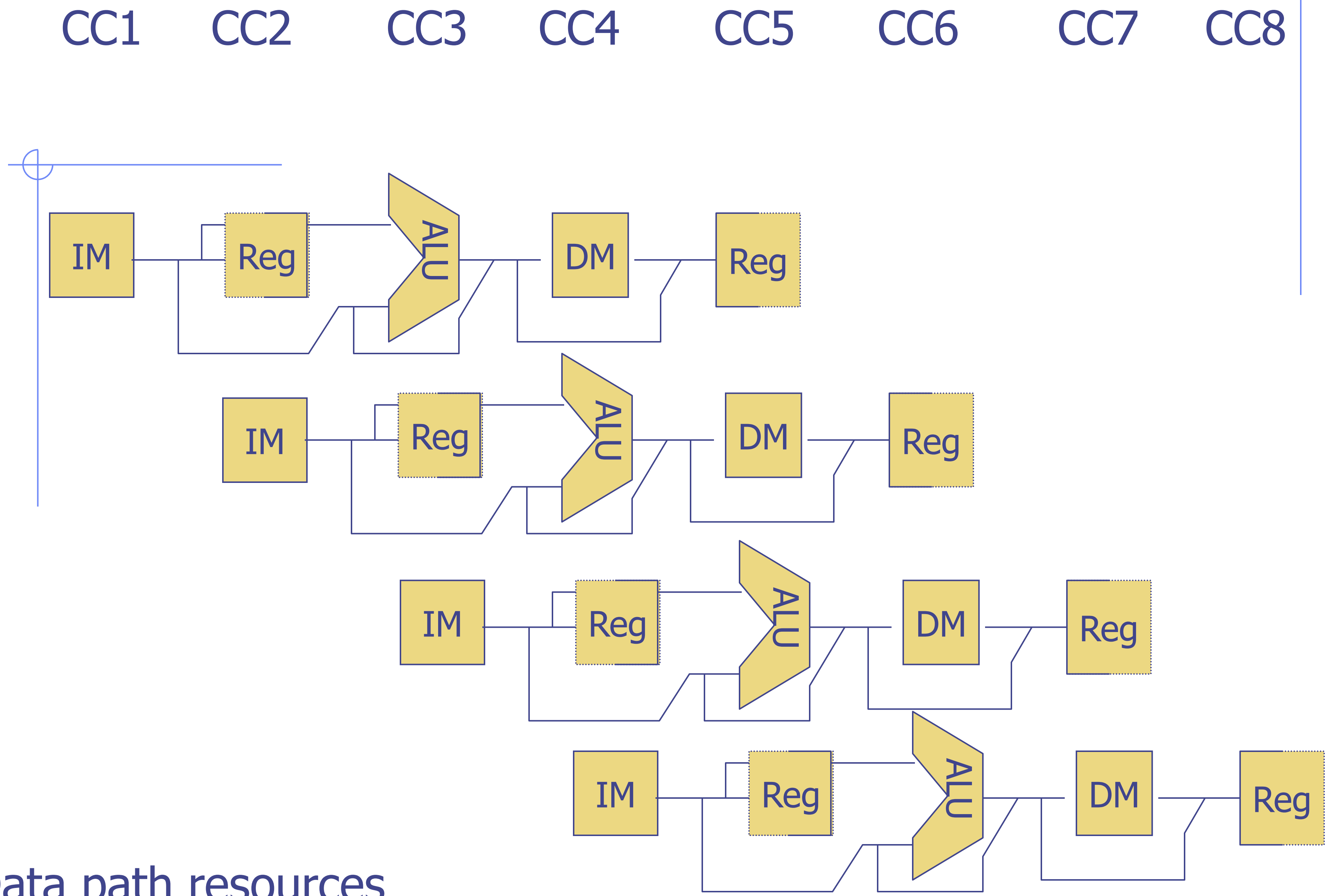
◆The classic five-stage pipeline for a RISC Processor

A simple five-stage implementation of pipeline.

Instruction Number	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

To design a pipeline system, we first have to make sure that two different operations will not be performed with the same **data path resource** on the same clock cycle.

For example, a single ALU can not be asked to compute an effective address and perform a subtract operation at the same time.



Data path resources
used by the executions of multiple instructions.

In the pipeline, overlapping the execution of multiple instructions introduces relatively few conflicts.

There are three observations on which this fact rests.

(1) We use **separate instruction and data memories,** which we would typically implement with separate **instruction and data caches**. The use of separate caches eliminates a conflict for a single memory that would arise between instruction fetch and data memory access.

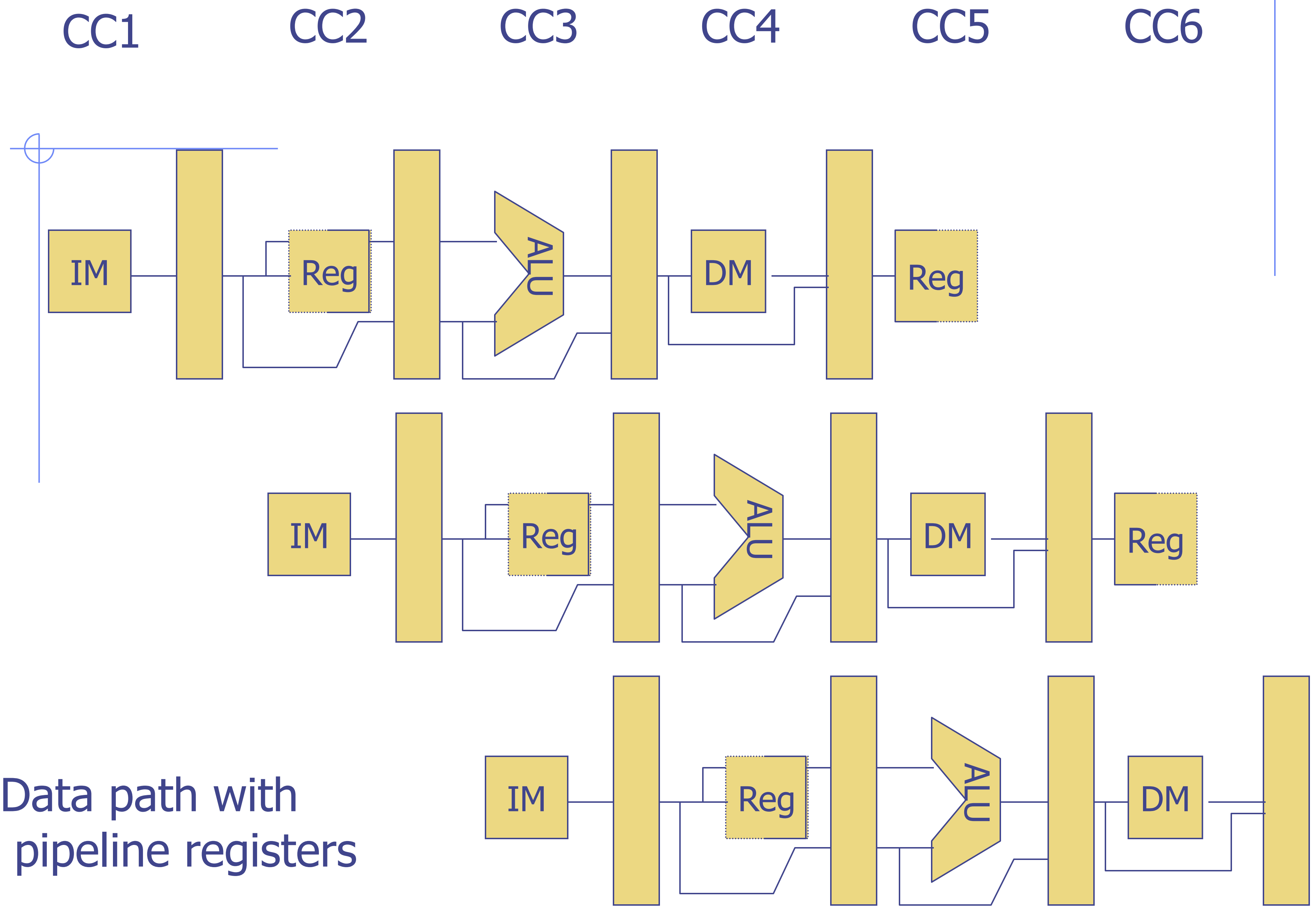
(2) The register file is used in the two stages: one for reading in ID and one for writing in WB. To handle reads and a write to the same register, we perform the **register write in the first half of the clock and the read in the second half**.

(3) To start a new instruction every clock, we must **increment and store PC every clock,** this must be done during the IF stage in preparation for the next instruction. Furthermore, we must also have an adder to compute the potential branch target. One further problem is that a branch does not change the PC until MEM stage. We will solve this problem shortly.

Although it is critical to ensure that instructions in the pipeline do not attempt to use hardware resources at the same time, we must also ensure that instructions in different stages of the pipeline do not interfere with one another. **This separation is done by using pipeline registers between successive stages of the pipeline.**

The pipeline registers also play the key role of carrying intermediate results.

For example, the result of an ALU instruction is computed during EX, but not actually stored until WB; it arrives there by passing through two pipeline registers.



Data path with
pipeline registers

The major hurdle of pipelining-pipeline hazards

There are three classes of pipeline hazards:

1. **Structural hazards** arise from resource conflicts when hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.

2. **Data hazards** occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.

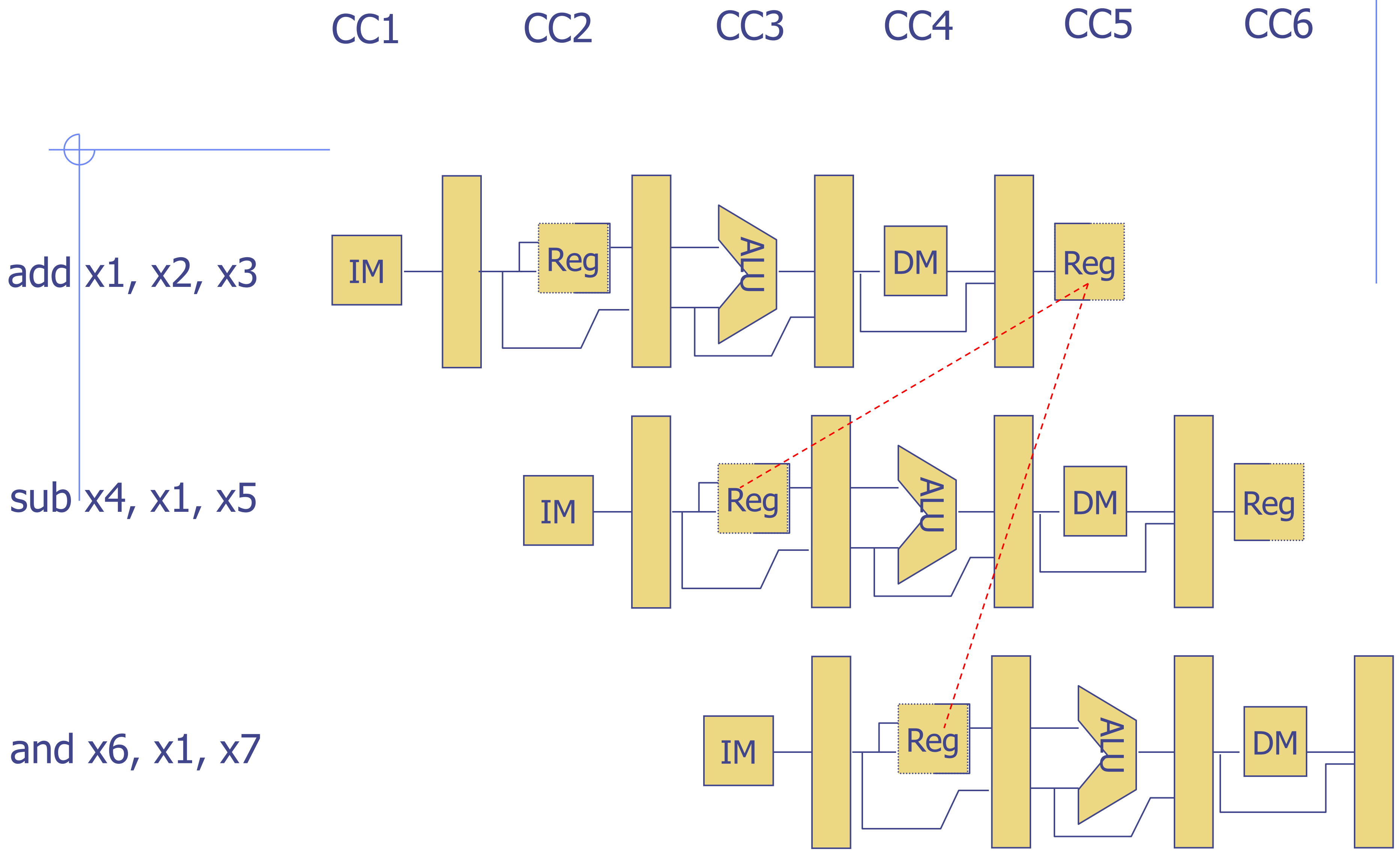
3. **Control hazards** arise from the pipelining of branches and other instructions that change PC.

Data Hazards

Consider the pipelined execution of these instructions:

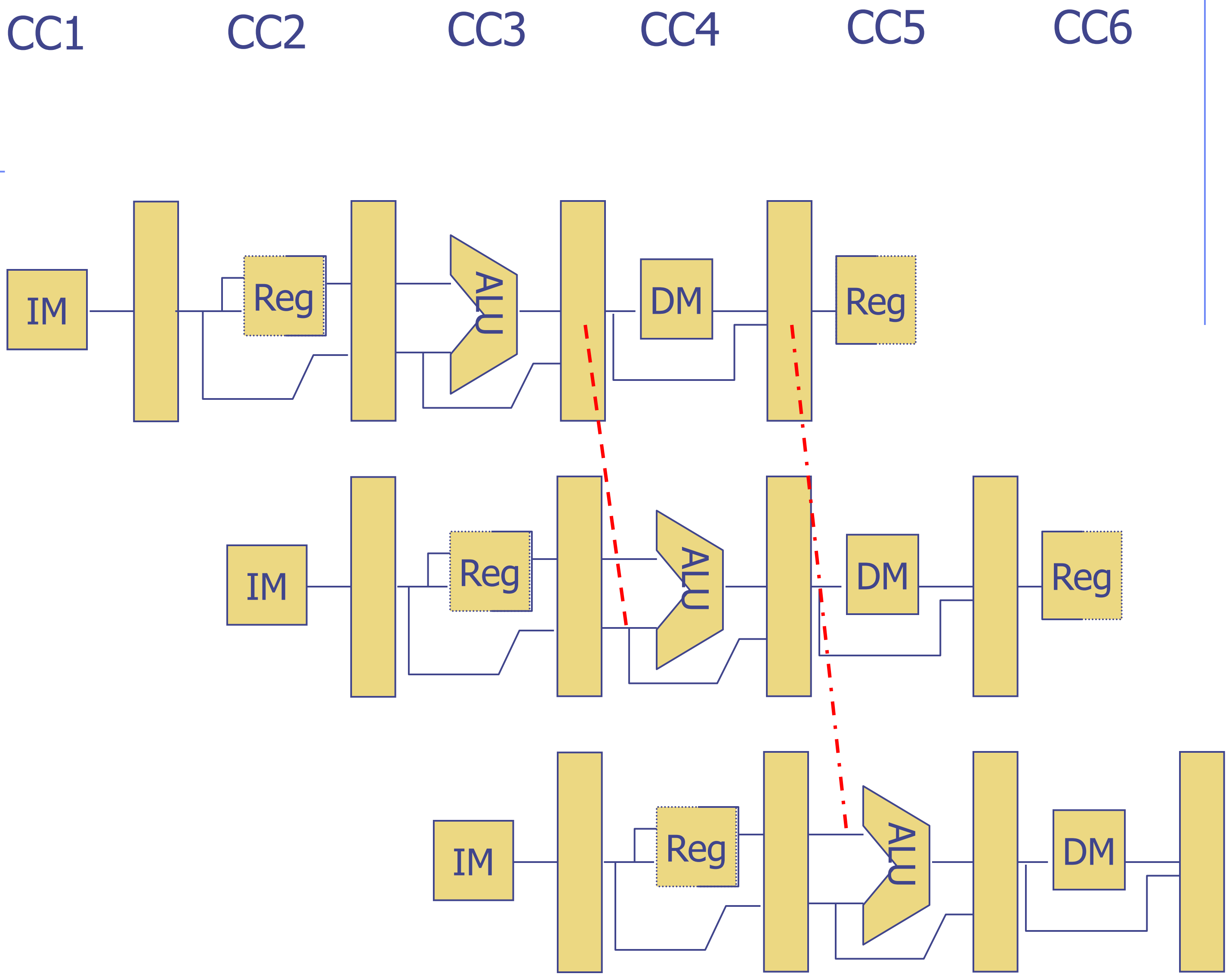
```
add x1, x2, x3
sub x4, x1, x5
and x6, x1, x7
```

The **add** instruction writes the value of x1 in the WB pipe stage, but the **sub** reads the value during its ID stage. This is a typical example of data hazard. In fact, the **and** instruction is also affected by the data hazard.


CC1
CC2
CC3
CC4
CC5
CC6
add x1, x2, x3
sub x4, x1, x5
and x6, x1, x7
IM
Reg
ALU
DM
Reg

The data hazard may be solved with a simple hardware technique called **forwarding**.

That is, we forward the result of **add** to **sub** and **and** before it is written into x1.



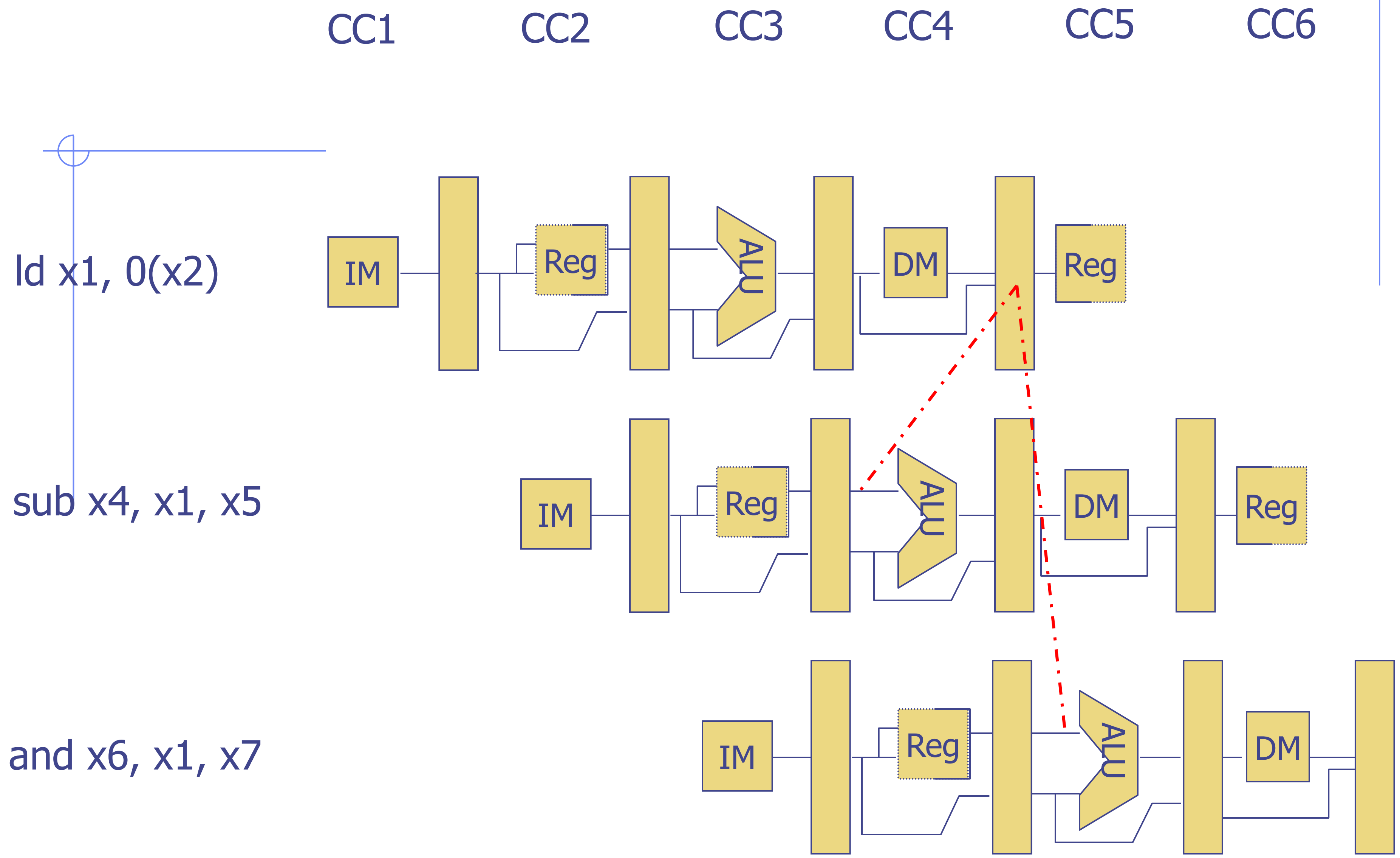
add x1, x2, x3

sub x4, x1, x5

and x6, x1, x7

Not all potential data hazards can be handled by forwarding. Consider the following sequence of instructions:

```
ld  x1,0(x2)
sub x4,x1,x5
and x6,x1,x7
```



CC1
CC2
CC3
CC4
CC5
CC6
ld x1, 0(x2)
IM
Reg
ALU
DM
Reg
sub x4, x1, x5
IM
Reg
ALU
DM
Reg
and x6, x1, x7
IM
Reg
ALU
DM

The **ld** instruction does not have the data until the end of CC4, while the **sub** instruction needs to have the data by the beginning of CC4. Hence, the simple forwarding technique can not eliminate the data hazard problem. Therefore, a **stall** is necessary.

In this case, we need a hardware to detect the hazard and stall the pipeline until the hazard is cleared. The hardware is called the pipeline **interlock**.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
ld x1,0(x2)	IF	ID	EX	MEM	WB				
sub x4,x1,x5		IF	ID	**Stall**	EX	MEM	WB		
and x6,x1,x7			IF	**Stall**	ID	EX	MEM	WB	
or x8,x1,x9				**Stall**	IF	ID	EX	MEM	WB

Data hazards may be classified as one of **three types**, depending on **the order of read and write accesses** in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline.

Consider two instructions i and j, with i occurring before j in program order. The possible data hazards are:

RAW (read after write)- j tries to read a source before i writes it, so j incorrectly gets the old value. In the simple 5-stage pipeline considered here, RAW is the only possible pipeline hazard.

WAW(write after write)- j tries to write an operand before it is written by i. WAW may exist between a short integer pipeline and a long floating-point pipeline.

WAR(write after read)- j tries to write a destination before it is read by i, so i incorrectly get a new value. WAR may exist when instructions are reordered.

Control Hazards

Control hazards can cause a greater performance loss for pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4.

If a branch changes the PC to its target address, it is a **taken branch**; if it falls through, it is **untaken**.

- Flush/stall pipeline

The simplest way to handle the branch is to stall the pipeline until the branch is complete. However, this approach may be too slow.

- Predicted-not-taken/Predicted untaken

A common improvement over branch stalling is to assume that branch will not be taken and thus continue execution down the sequential instruction stream. this scheme is called predicted-not-taken scheme. If the branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target.

A pipeline sequence of the predicted-not-taken scheme when the branch is taken.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i (Branch instruction)	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	**Idle**	**Idle**			
Instruction i+2			IF	ID	**Idle**	**idle**	**Idle**		
Instruction i+3				IF	**Idle**	**Idle**	**Idle**	**Idle**	
Branch target					IF	ID	EX	MEM	WB

One way to improve branch performance is to reduce the cost of the taken branch. If we move the branch execution earlier in the pipeline, then fewer instructions need to be flushed.

Many implementations move the branch execution to the ID stage. We will describe this implementation later.

•Predicted-taken

An alternative scheme is to treat every branch as taken. However, because in our 5-stage pipeline we do not know the target address any earlier than we know the branch outcome, there is no advantage in this approach for this outcome.

- Performance of Branch Schemes

$$\text{Pipeline Speedup} = \frac{\text{Pipeline depth}}{1+\text{Pipeline stall cycles from branches}}$$

$\because$ Pipeline stall cycles from branches
 = Branch frequency $\times$ Branch penalty

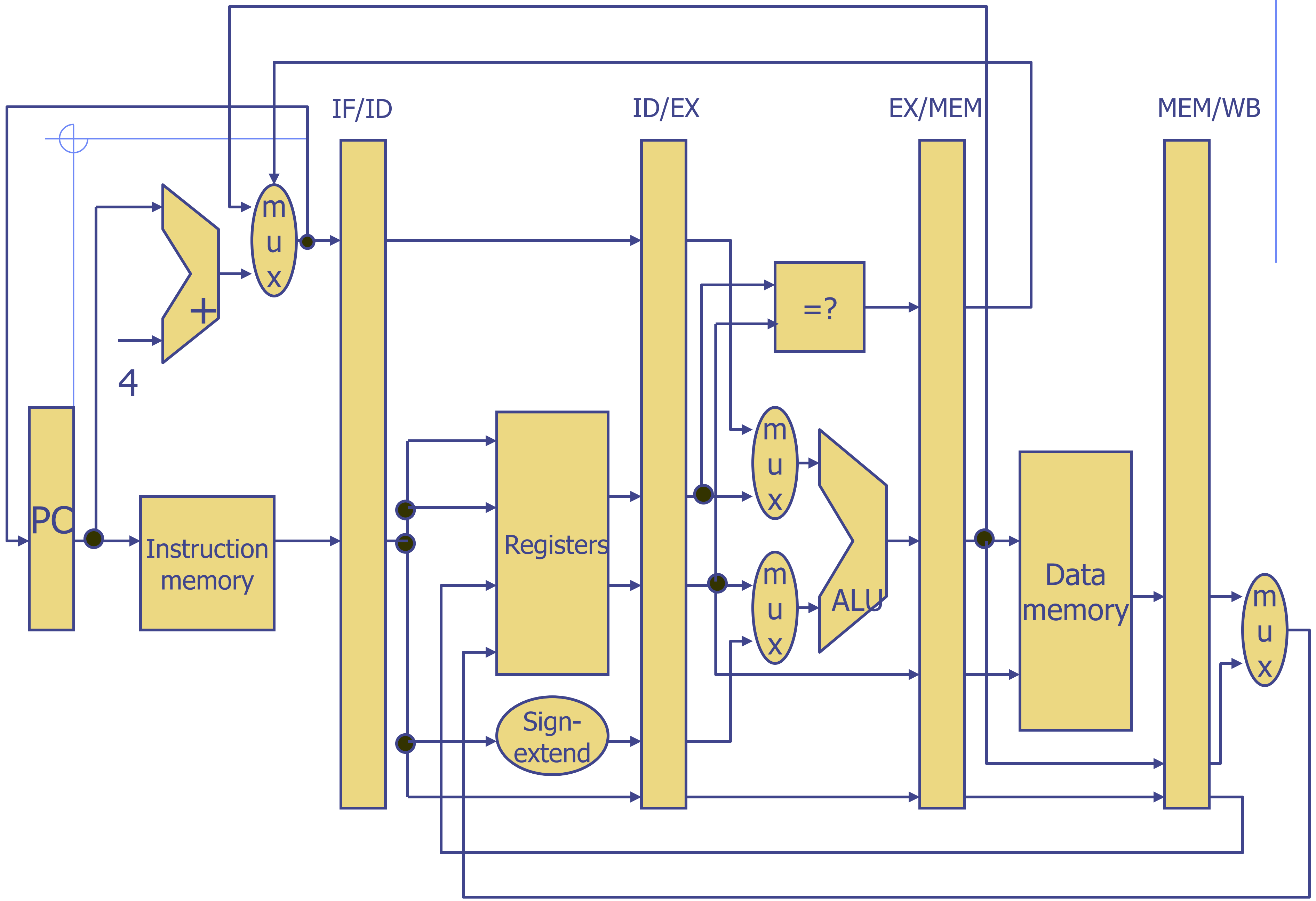
$\therefore$

$$\text{Pipeline Speedup} = \frac{\text{Pipeline depth}}{1+\text{Branch frequency} \times \text{Branch penalty}}$$

How is Pipelining Implemented ?

- A basic pipeline for RISC-V

The pipeline registers carry both data and control from one pipeline stage to the next. Any value needed on a latter stage must be placed in such a register and copied from one pipeline register to the next.


IF/ID
ID/EX
EX/MEM
MEM/WB
mux
+
4
PC
Instruction memory
Registers
Sign-extend
=?
mux
mux
ALU
Data memory
mux

Events on every pipe stage of the MIPS pipeline are shown below.

IF stage:

```
IF/ID.IR ←Mem[PC];
IF/ID.NPC, PC ←(if ( (EX/MEM.opcode == branch) &
                      EX/MEM.cond){EX/MEM.ALUOutput}
                   else {PC+4});
```

ID stage:

```
ID/EX.A ←Regs[IF/ID.IR[rs1]];
ID/EX.B ←Regs[IF/ID.IR[rs2]];
ID/EX.NPC ← IF/ID.NPC;
ID/EX.IR ← IF/ID.IR;
ID/EX.Imm ← sign-extend(IF/ID.IR[immediate field]);
```

- ALU instruction

EX stage:
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;

MEM stage:
MEM/WB.IR ← EX/MEM.IR;
MEM/WB.ALUOutput ← EX/MEM.ALUOutput;

WB stage:
Regs[MEM/WB.IR[rd]] ←MEM/WB.ALUOutput;

- Load or store instruction

EX stage:
EX/MEM.IR ← ID/EX.IR;
EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm;
EX/MEM.B ← ID/EX.B;

MEM stage:
For load:
MEM/WB.IR ←EX/MEM.IR;
MEM/WB.LMD ←Mem[EX/MEM.ALUOutput];
For store:
Mem[EX/MEM.ALUOutput] ← EX/MEM.B;

WB stage:
For load only:
Regs[MEM/WB.IR[rd]] ← MEM/WB.LMD;

- Branch instruction

EX stage:

EX/MEM.ALUOutput ← ID/EX.NPC + (ID/EX.Imm <<2);
EX/MEM.cond ← (ID/EX.A == ID/EX.B);

Dealing with data hazards

Here we briefly describe the implementation of interlock and forwarding logics for the data hazards.

To implement the interlock, we only consider the **ld** instruction for the demonstration purpose.

A table showing the a variety of situations that we must handle for load.

Situation	Example Code Sequence
No dependence	ld **x1**, 45(x2) add x5, x6, x7 sub x8, x6, x7 or x9, x6, x7
Dependence requiring stall	ld **x1**, 45(x2) add x5, **x1**, x7 sub x8, x6, x7 or x9, x6, x7
Dependence overcome by forwarding	ld **x1**, 45(x2) add x5, x6, x7 sub x8, **x1**, x7 or x9, x6, x7
Dependence with accesses in order	ld **x1**, 45(x2) add x5, x6, x7 sub x8, x6, x7 or x9, **x1**, x7

If there is a RAW hazard with the source instruction being a load, the load instruction will be in the EX stage when an instruction that needs the load data is in the ID stage.

IF	ID	EX	MEM	WB
	Instruction need load data	**load**		

A small table describing all possible hazard situations for load.

It can be directly translated into an interlock implementation.

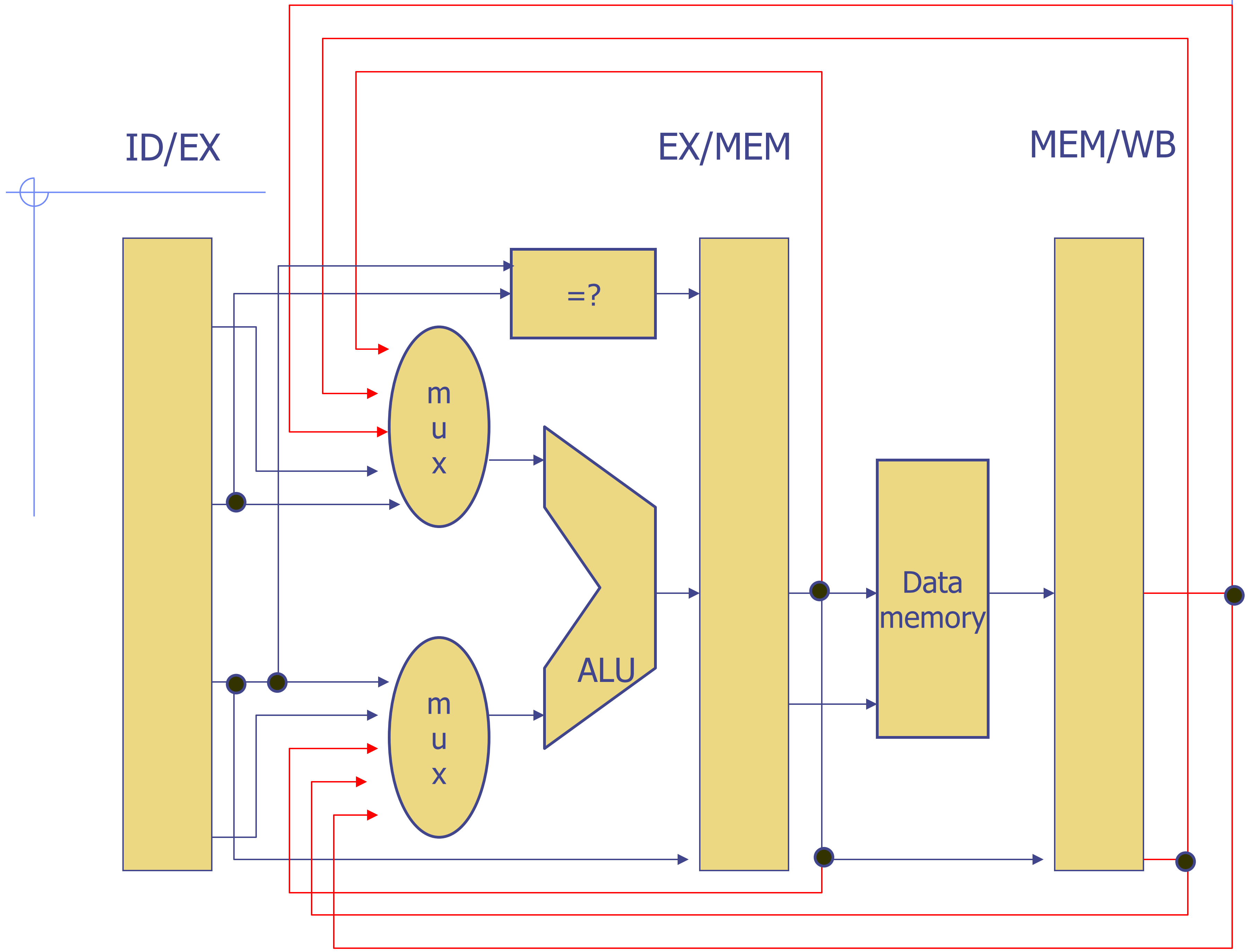
Opcode field of ID/EX (ID/EX.$IR_{0..6}$)	Opcode field of IF/ID (IF/ID.$IR_{0..6}$)	Matching operand fields
Load	Register-register ALU, Load, Store, ALU Immediate, or Branch	ID/EX.IR[rd]== IF/ID.IR[rs1]
Load	Register-Register ALU, Store or branch	ID/EX.IR[rd]== IF/ID.IR[rs2]

Implementing the forwarding logic is similar, although there are more cases to consider. All forwarding happens from the ALU output or data memory output to the ALU input, data memory input, or the equality detection unit.

A table showing all the possible forwarding operations for the instruction currently in the EX.

Source Instruction		Destination Instruction		Condition
Pipeline Register	Opcode	Pipeline Register	Opcode	
EX/MEM	Register-register ALU, ALU Immediate	ID/EX	Register-register ALU, ALU Immediate, Load, Store, Branch	EX/MEM.IR[rd]== ID/EX.IR[rs1]
EX/MEM	Register-register ALU, ALU Immediate	ID/EX	Register-register ALU Store, Branch	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	MEM/WB.IR[rd]== ID/EX.IR[rs1]
MEM/WB	Register-register ALU, ALU Immediate, Load	ID/EX	Register-register ALU Store, Branch	MEM/WB.IR[rd]== ID/EX.IR[rs2]

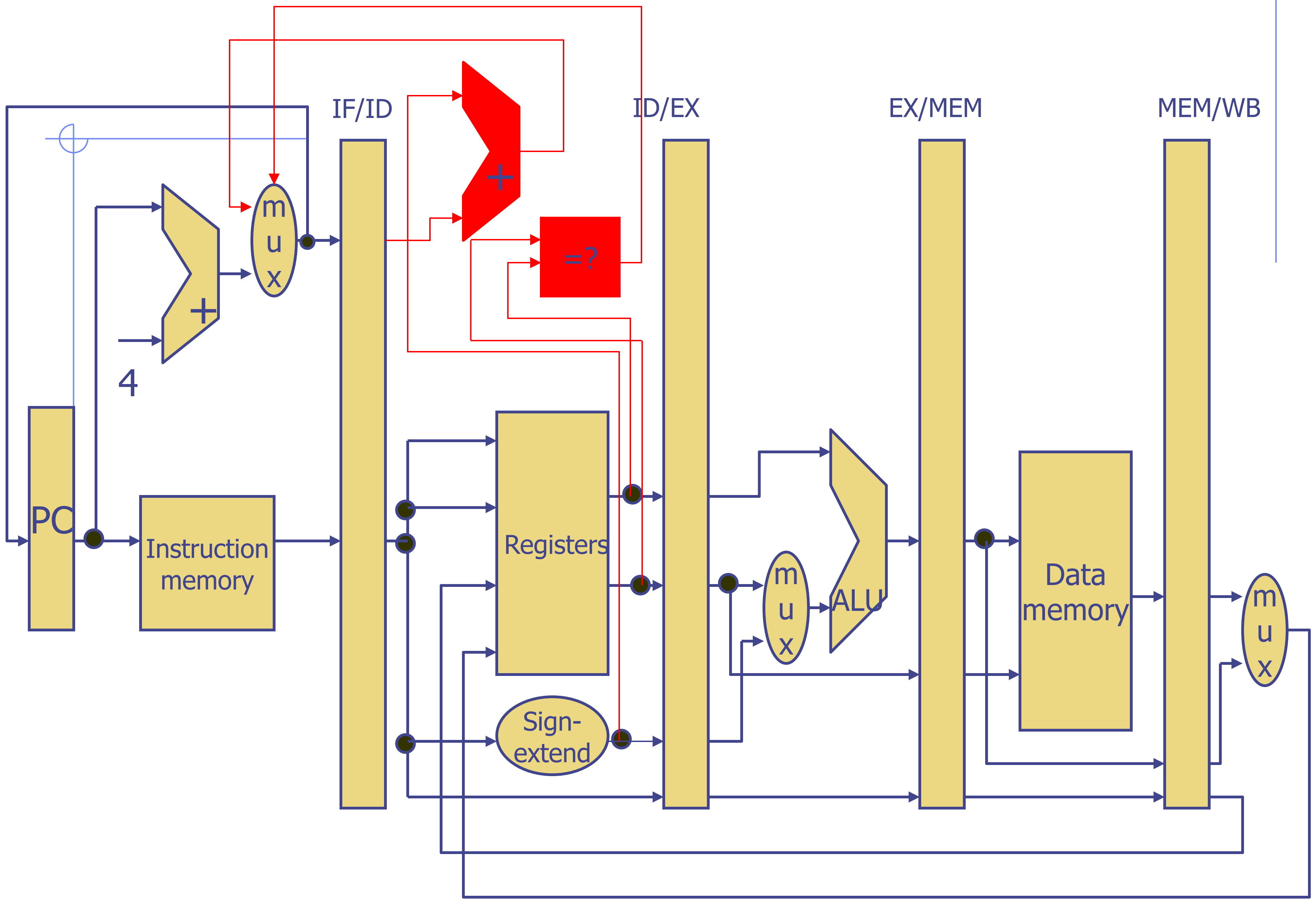
In addition to the comparators and combinational logic that we need to determine when a forwarding path needs to be enabled, we also need to enlarge the multiplexers at the ALU inputs and add the connections from the pipeline registers that are used to forward the results.


ID/EX
EX/MEM
MEM/WB
=?
m
u
x
ALU
Data
memory
m
u
x

Dealing with control hazards

To reduce the branch penalty, it is desired to compute the branch target early. Here we describe the implementation of the pipeline where target address is computed during the ID stage.

To accomplish this goal, we need an additional adder because the main ALU is not usable until EX.


IF/ID
ID/EX
EX/MEM
MEM/WB
PC
4
+
m u x
+
=?
Instruction memory
Registers
Sign-extend
m u x
ALU
Data memory
m u x

What Makes Pipelining hard to Implement?

- Dealing with Exceptions

We use the term *exception* to cover the following events:

1. I/O device request
2. Invoking an operating system service from a user program
3. Tracing instruction
4. Breakpoint
5. Integer arithmetic overflow/ FP arithmetic anomaly
6. **Page fault** (not in main memory)
7. Misaligned memory accesses
8. Memory protection violation
9. Using an undefined or unimplemented instruction
10. Hardware malfunction/ Power failure

If a pipeline provides the ability for the processor to handle the exception, save the state, and restart without affecting the execution of the program, the pipeline or processor is said to be restartable.

While early supercomputers and microcomputers often lacked this property, almost all processors today support it, at least for the integer pipeline, because it is needed to implement virtual memory.

If the pipeline can be stopped so that the instructions just before the faulting instructions are completed and those after it can be restarted from scratch, the pipeline is said to have **precise exceptions**.

•Exceptions in RISC-V

The RISC-V pipeline stage and the exceptions might occur in each stage.

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 instruction fetch; misaligned memory access; memory protection violation
WB	None

With pipelining, multiple exceptions may occur in the same clock cycle because there are multiple instructions in execution.

For example, consider the following instruction sequence:

ld	IF	ID	EX	MEM	WB	
add		IF	ID	EX	MEM	WB

This pair of instruction can cause a data page fault (**ld** in MEM) and an arithmetic exception (**add** in EX) at the same time. We can solve this problem by dealing with the data page fault and then restart the execution.

The second exception will reoccur, and when the second exception occurs, it can be handled independently.

Exceptions can also occur out of order. The **ld** can get a data page fault in MEM, and **add** can get an instruction page fault in IF. The instruction page fault will occur first, even though it is caused by a later instruction.

Since we are implementing precise exceptions, the pipeline is required to handle the exception caused by the **ld** instruction first. Therefore, **the pipeline can not handle an exception when it occurs in time**.

To solve this problem, the hardware posts all exceptions caused by a given instruction in a status vector associated with that instruction.

The exception status vector is carried along as the instruction goes down the pipeline. Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off.

When an instruction enters WB, the exception status vector is checked. If any exceptions are posted, they are handled at that time.

Extending the RISC-V Pipeline to Handle Multicycle Operations

Here we extend our RISC-V pipeline to handle floating-point operations.

It is impractical to require that all RISC-V floating-point operations complete in 1 clock cycle. Doing so would mean accepting a slow clock.

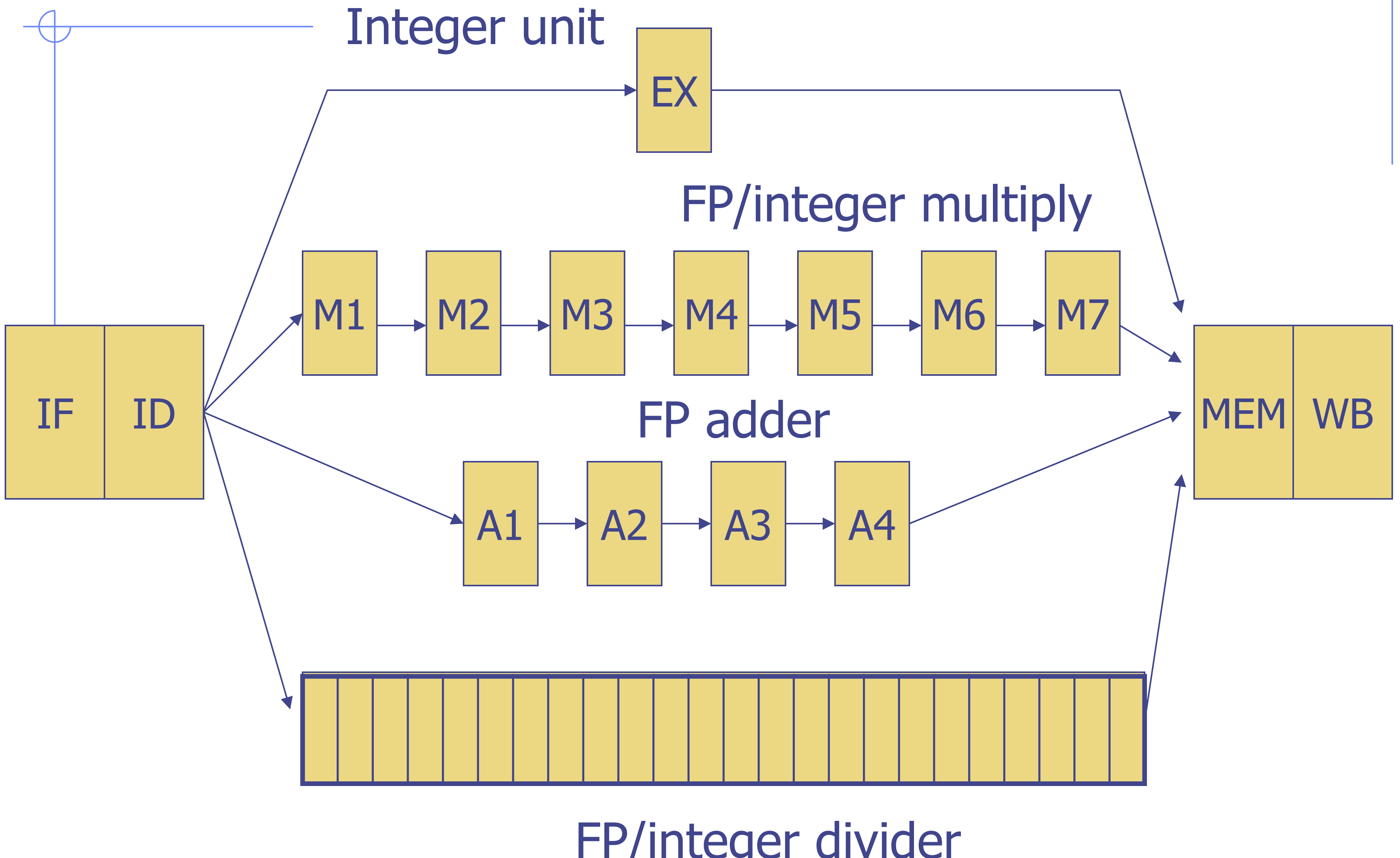
An alternative is to allow the floating-point instructions having the same pipeline as integer instructions with two important changes:

1. EX can be completed in more than one cycles,
2. There may be multiple floating point functional units.

There are four functional units in the RISC-V implementation with floating point.

1. The main integer unit that handles loads, stores, integer ALU operations and branches.
2. FP and integer multiplier.
3. FP adder that handles FP add, subtract and conversion
4. FP and integer divider

A pipeline supporting FP operations.



The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete.

This pipeline structure has a number of problems.

1. Because the divide unit is not fully pipelined, structural hazards can occur. These will need to be detected and **issuing** instructions will need to be installed.

2. Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1.

3. WAW hazards are possible, since instructions no longer reach WB in order. Note that WAR hazards are not possible, since the register reads always occur in ID.

.

4. Instructions can complete in a different order than they were issued, causing problems with exceptions.

5. Because of longer latency of operations, stalls for RAW hazards will be more frequent.

A typical code sequence where the stalls arising from RAW hazards.

Instruction	Clock Cycle Number																
	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 f0,f4,f6		**IF**	**ID**	**Stall**	**M1**	**M2**	**M3**	**M4**	**M5**	**M6**	**M7**	**MEM**	**WB**				
fadd f2,f0,f8			**IF**	**Stall**	**ID**	**Stall**	**Stall**	**Stall**	**Stall**	**Stall**	**Stall**	**A1**	**A2**	**A3**	**A4**	**MEM**	
fsd f2,0(x2)					**IF**	**Stall**	**Stall**	**Stall**	**Stall**	**Stall**	**Stall**	**ID**	**EX**	**Stall**	**Stall**	**Stall**	**MEM**

An example where three instructions want to perform a write back to the FP register file simultaneously.

Instruction	Clock Cycle Number										
	1	2	3	4	5	6	7	8	9	10	11
fmul 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 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

We can view the write-back contention problem as a structural hazard. One way to implement the interlock for this hazard is to track the use of the write port in the ID stage and to stall an instruction before it issues. Tracking the use of the write port can be done with a shift register that indicates when already-issued instructions will use the register file.

Another problem is the WAW hazard. The **fld** instruction shown below has a destination of f2.

It then creates a WAW hazard, because it writes f2 one cycle earlier then the **fadd**.

Instruction	Clock Cycle Number										
	1	2	3	4	5	6	7	8	9	10	11
fmul 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 f2,f4,f6				IF	ID	A1	A2	A3	A4	MEM	WB
…					IF	ID	EX	MEM	WB		
fld f2,0(x2)						IF	ID	EX	MEM	WB	

Maintaining Precise Exceptions

Another problem caused by these long-running instructions can be illustrated with the following sequence of code:

```
DIV.D  F0,F2,F4
ADD.D F10,F10,F8
SUB.D F12,F12,F14
```

In this case, we can expect ADD.D and SUB.D to complete before the DIV.D completes. This is called **out-of-order completion**. Suppose that the SUB.D causes a floating-point arithmetic exception at a point where the ADD.D has completed but the DIV.D has not. The result will be an imprecise exception.

There are 4 approaches to deal with out-of-order completion.

1. Ignore the problem and settle for the imprecise exception.
2. Buffer the results of an operation until all the operations that were issued earlier are complete.
3. Allow the exceptions to become somewhat imprecise. However, keep enough information so that the trap-handling routines can create a precise sequence for the exception.
4. Allow the instruction issue to continue only if it is certain that all instructions before the issuing instruction will complete without causing an exception.

Putting It All together: The MIPS R4000 Pipeline

The MIPS R4000 implements MIPS64 but uses a deeper pipeline than our 5-stage design. This deeper pipeline allows it to achieve higher clock rates by decomposing the 5-stage integer pipeline into eight stages.

The function of each stage is as follows:

IF: First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.

IS: Second half of instruction fetch, complete instruction cache access.

RF: Instruction decode and register fetch, hazard checking, and also instruction cache hit detection.

EX: Execution, which includes effective address calculation, ALU operation, branch target computation and condition evaluation.

DF: Data fetch, first half of data cache access.

DS: Second half of data fetch, completion of data cache access.

TC: Tag check, determine whether the data cache access hit.

WB: Write back for loads and register-register operations.

In this pipeline, a load instruction followed by an immediate use results in a 2-cycle stall.

Instruction	Clock Cycle Number										
	1	2	3	4	5	6	7	8	9	10	11
ld x1,10(x7)	IF	IS	RF	EX	DF	DS	TC	WB			
add x2,x1,x8		IF	IS	RF	Stall	Stall	EX	DF	DS	TC	WB
sub x3,x1,x9			IF	IS	Stall	Stall	RF	EX	DF	DS	TC
or x4,x1,x10				IF	Stall	Stall	IS	RF	EX	DF	DS

Crosscutting issues

- Dynamically Scheduled Pipelines

All the techniques discussed in this appendix so far use in-order instruction issue, which means that if an instruction is stalled in the pipeline, no later instructions can proceed. With in-order issue, if two instructions have a hazard between them, the pipeline will stall, even if there are **later** instructions that are **independent** and would not stall.

The dynamic scheduling techniques can be used to solve this problem by **rearranging** the instruction execution using hardware. The technique will do out-of-order execution, which implies out-of-order completion.

It is important to observe that, the WAR hazards, which did not exist in the MIPS floating and integer pipelines, may arise when instruction execute out of order.

Consider the following code sequence:

```
DIV.D   F0,F2,F4
ADD.D  F10,F0,F8
SUB.D  F8,F8,F14
```

If the pipeline executes the SUB.D before the ADD.D, a WAR hazard will occur.

To implement out-of-order execution, we must split the ID pipe stage into two stages:

1. Issue
2. Read Operands

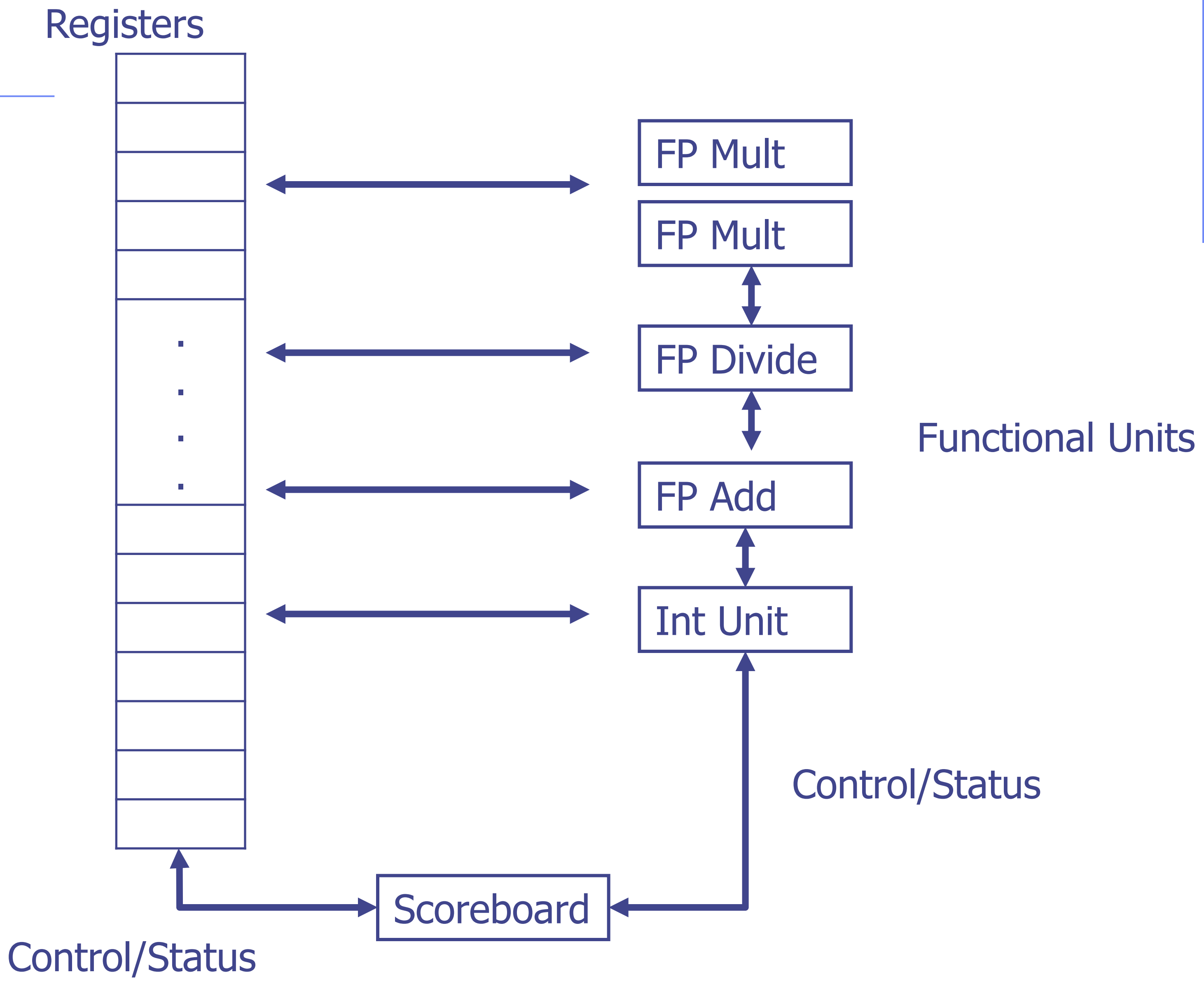
All instructions pass through the issue stage in order; however, they can be stalled or **bypass** each other in the second stage (read operands) and thus enter the execution out-of-order.

Scoreboarding is a technique for allowing instructions to execute out-of-order when there are sufficient resources and no data dependence.

The goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible. Thus, when the next instruction to execute is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instructions.

Taking advantage of out-of-order execution requires multiple instructions to be in their EX stage simultaneously. This can be achieved with multiple function units.

Here we assume that there are 2 multipliers, 1 adder, 1 divide unit, and 1 integer unit.



The scoreboarding technique can be separated into four stages, which replaces the ID, EX, MEM and WB, respectively.

Issue→ If a **functional unit** for the instruction is **free** and no other active instruction has the **same destination register**, the scoreboard issues the instruction to the functional unit and updates its internal data structure.
(structural and WAW hazards can be avoided)

Read operands → The scoreboard monitors the availability of the source operands. A source operand is available if no earlier issued active instruction is going to write it. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution.
(RAW hazard can be avoided).

Execution→ the functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed the execution.

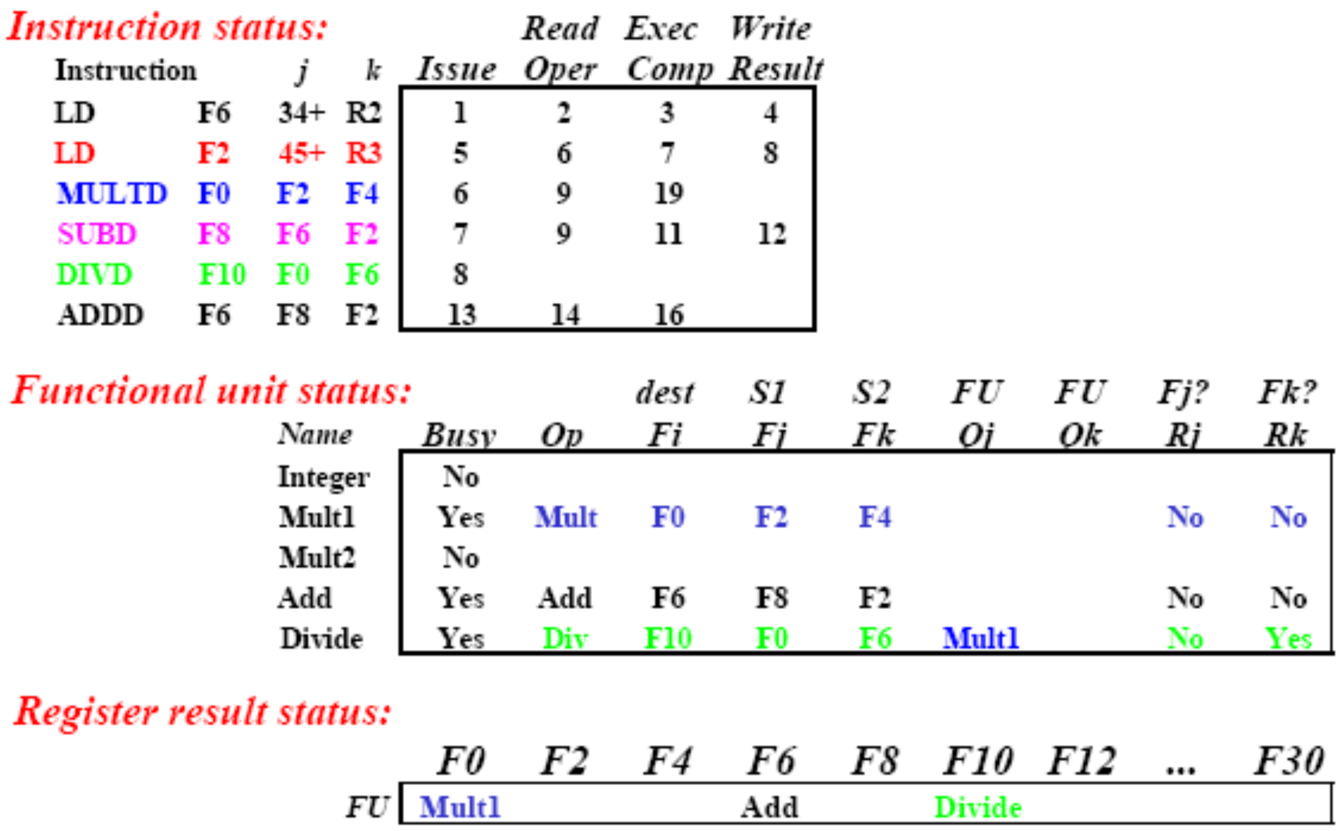
Write result →Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards and stalls the completing instruction, if necessary.

There are three parts to the scoreboard:

1. Instruction status: indicates which of the four steps the instruction is in.
2. Functional unit status: indicates the state of functional unit (FU). There are nine fields for each FU:

Busy → indicates whether the unit is busy or not.
Op → operation to perform in the unit.
F_i → destination register
F_j , F_k → source registers
Q_j , Q_k → functional units producing source registers F_j , F_k
R_j , R_k → flags indicating when F_j , F_k are ready for and are not yet read. Set to No after operands are read.

3. Register result status: indicates which functional unit will write each register, if an active instruction has the register as its destination. This field is set to blank whenever there are no pending instructions that will write that register.



Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

			dest	S1	S2	FU	FU	Fj?	Fk?
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
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:

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1			Add		Divide			

Components of the scoreboard

Example:

Assume the following EX cycle latencies for the floating-point functional units: Add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. Consider the code segment shown below.

```
L.D F6,34(R2)
L.D F2,45(R3)
MUL.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
```

Show what the status table look like for the execution of each instruction.

Scoreboard Example

Instruction status:

Instruction		j	k	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2				
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
	FU									

Scoreboard Example: Cycle 1

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1			
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock
1

	F0	F2	F4	F6	F8	F10	F12	...	F30
FU				Integer					

Scoreboard Example: Cycle 2

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2		
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F6		R2				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
2	*FU*				Integer					

- Issue 2nd LD?

Scoreboard Example: Cycle 3

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F6		R2				No
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU				Integer					

- Issue MULT?

Scoreboard Example: Cycle 4

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3				
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

				dest	S1	S2	FU	FU	Fj?	Fk?
Time	Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU									

Scoreboard Example: Cycle 5

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5			
MULTD	F0	F2	F4				
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				Yes
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU		Integer							

Scoreboard Example: Cycle 6

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6		
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	Yes	Load	F2		R3				Yes
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6	*FU*	Mult1	Integer							

Scoreboard Example: Cycle 7

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				No
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	Integer			Add				

- Read multiply operands?

Scoreboard Example: Cycle 8a (1st half of cycle)

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	Yes	Load	F2		R3				No
	Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
8	FU	Mult1	Integer			Add	Divide			

Scoreboard Example: Cycle 8b (2nd half of cycle)

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6			
SUBD	F8	F6	F2	7			
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	*FU*	Mult1				Add	Divide			

Scoreboard Example: Cycle 9

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9		
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Note Remaining

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
10	Mult1	Yes	Mult	F0	F2	F4			Yes	Yes
	Mult2	No								
2	Add	Yes	Sub	F8	F6	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
9	FU	Mult1				Add	Divide			

- Read operands for MULT & SUB? Issue ADDD?

Scoreboard Example: Cycle 10

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9		
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
9	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
1	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	*FU*	Mult1				Add	Divide			

Scoreboard Example: Cycle 11

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
8	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
0	Add	Yes	Sub	F8	F6	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	*FU*	Mult1				Add	Divide			

Scoreboard Example: Cycle 12

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2				

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
7	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	No								
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
12	FU	Mult1					Divide			

- Read operands for DIVD?

Scoreboard Example: Cycle 13

Instruction status:

Instruction		j	k	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13			

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
6	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
13	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 14

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14		

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
5	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
2	Add	Yes	Add	F6	F8	F2			Yes	Yes
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
14	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 15

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14		

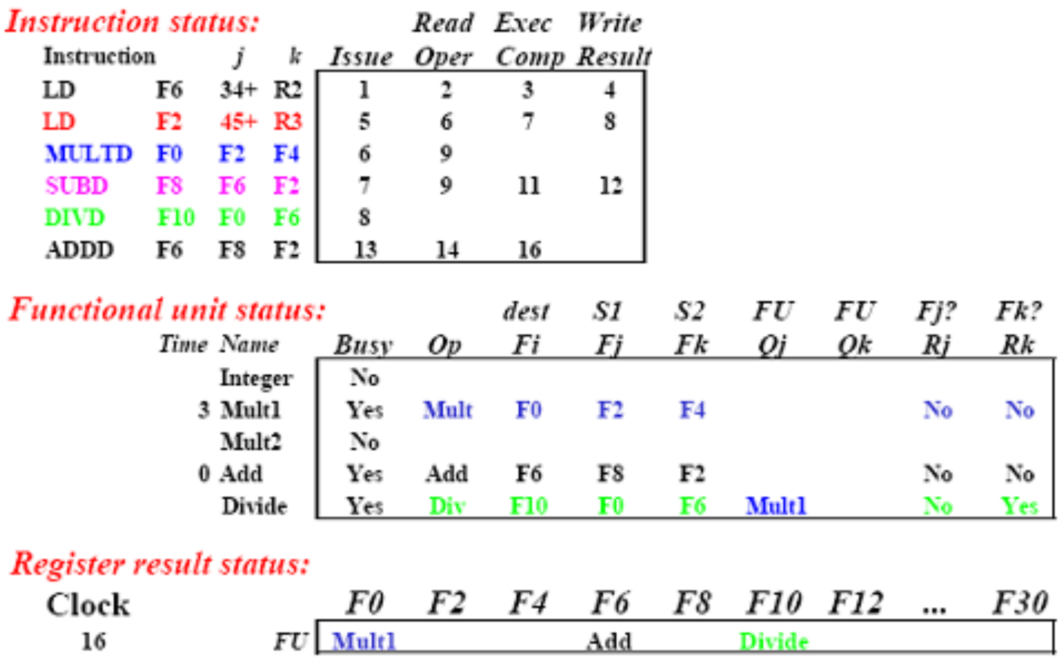
Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
4	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
1	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1			Add		Divide			

Scoreboard Example: Cycle 16



Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
3	Mult1	Yes	Mult	F0	F2	F4			No	No
	Mult2	No								
0	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 17

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

WAR Hazard!

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
2	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:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
17	*FU*	Mult1			Add		Divide			

- Why not write result of ADD???

Scoreboard Example: Cycle 18

Instruction status:

Instruction		j	k	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9		
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
1	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:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
18	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 19

Instruction status:

Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	*Name*	*Busy*	*Op*	*dest Fi*	*S1 Fj*	*S2 Fk*	*FU Qj*	*FU Qk*	*Fj? Rj*	*Fk? Rk*
	Integer	No								
0	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:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
19	*FU*	Mult1			Add		Divide			

Scoreboard Example: Cycle 20

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8			
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
20	*FU*				Add		Divide			

Scoreboard Example: Cycle 21

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	Yes	Add	F6	F8	F2			No	No
	Divide	Yes	Div	F10	F0	F6			Yes	Yes

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
21	FU				Add		Divide			

- WAR Hazard is now gone...

Scoreboard Example: Cycle 22

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21		
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
39	Divide	Yes	Div	F10	F0	F6			No	No

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
22	FU						Divide			

Scoreboard Example: Cycle 61

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
0	Divide	Yes	Div	F10	F0	F6			No	No

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
61	FU						Divide			

Review: Scoreboard Example: Cycle 62

Instruction status:

Instruction		j	k	Issue	Read Oper	Exec Comp	Write Result
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
62	FU									

- In-order issue; out-of-order execute & commit