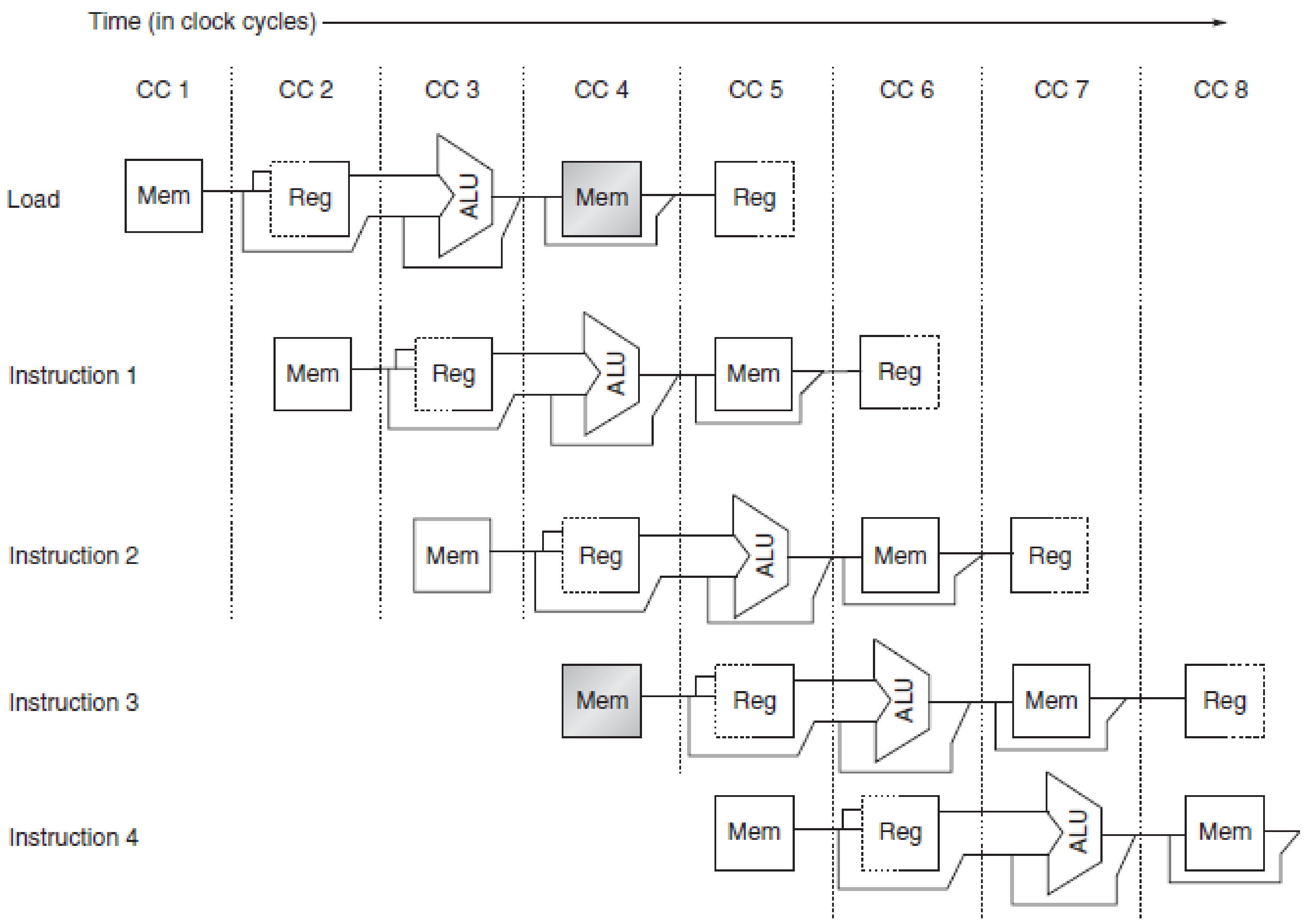
Pipeline Hazards

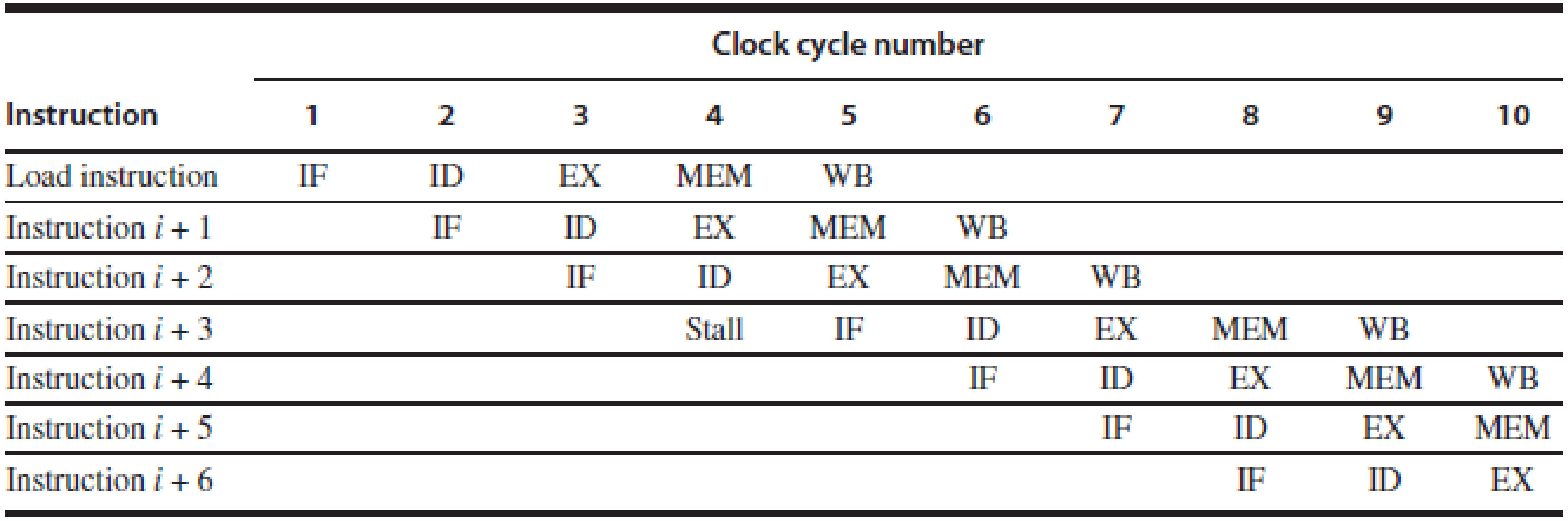
Pipeline Hazards

- Situations prevent the next instruction in the instruction stream from executing during its designated clock cycle – Hazards
- Reduces the performance from the ideal speedup gained by pipelining
- classes of hazards
  - Structural hazards - resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.
  - Data hazards - an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - Control hazards - pipelining of branches and other instructions that change the PC.

Structural Hazards



Structural Hazards (contd)



	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction $i + 1$		IF	ID	EX	MEM	WB				
Instruction $i + 2$			IF	ID	EX	MEM	WB			
Instruction $i + 3$				Stall	IF	ID	EX	MEM	WB	
Instruction $i + 4$						IF	ID	EX	MEM	WB
Instruction $i + 5$							IF	ID	EX	MEM
Instruction $i + 6$								IF	ID	EX

Data Hazards

- Occurs 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.
- Consider the execution of following instructions

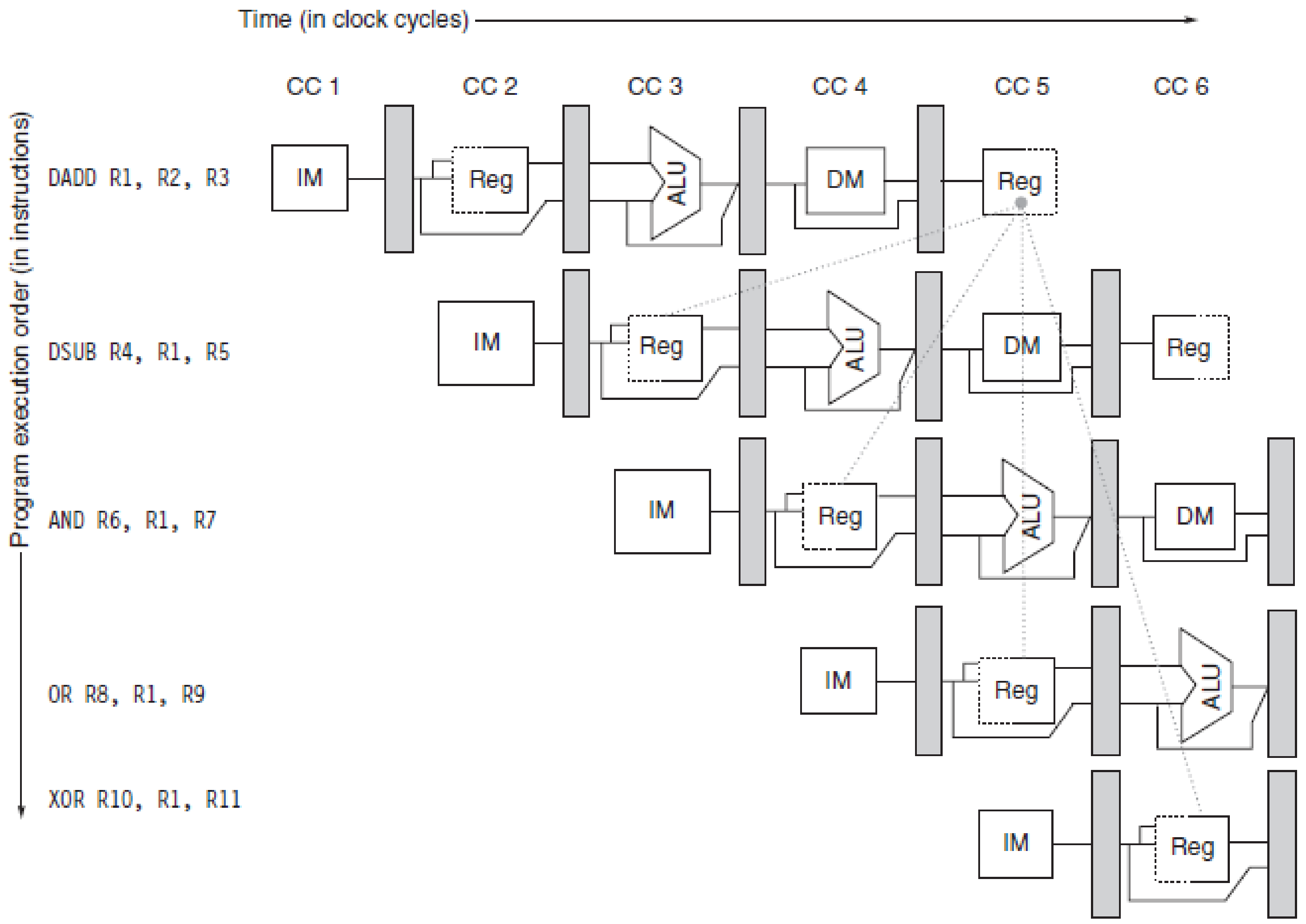
```
DADD R1,R2,R3
DSUB R4,R1,R5
AND  R6,R1,R7
OR   R8,R1,R9
XOR  R10,R1,R11
```

- All instructions after DADD use the result of DADD instruction
- DADD instruction writes value of R1 in the WB pipe stage, but DSUB instruction reads the value during its ID stage.
- DSUB instruction will read the wrong value and try to use it.

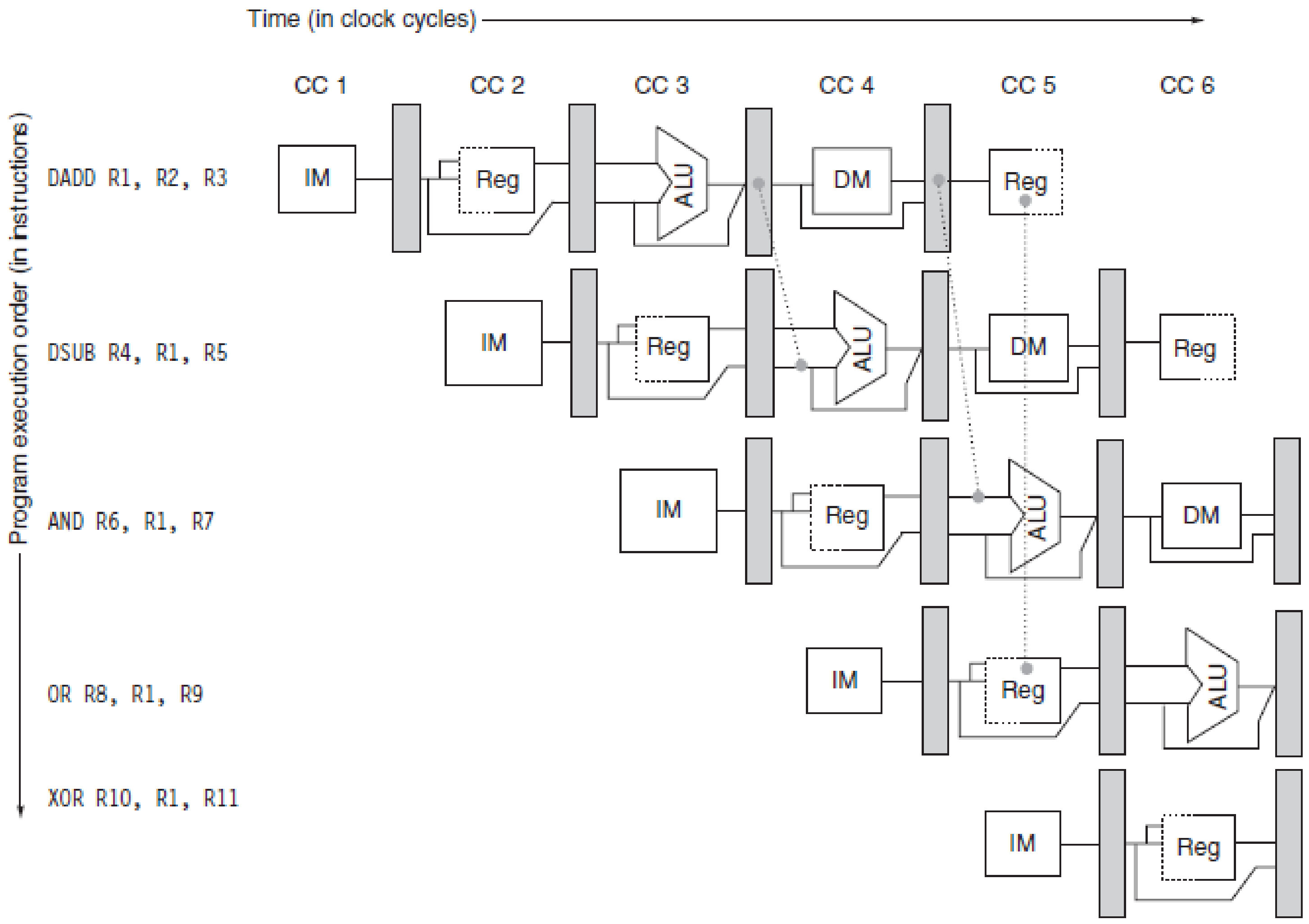
Minimizing Data Hazard Stalls by Forwarding

- The problem can be solved with a simple hardware technique called *forwarding* (also called *bypassing*).
- The key insight in forwarding is that the result is not really needed by the DSUB until after the DADD actually produces it.
- If the result can be moved from the pipeline register where the DADD stores it to where the DSUB needs it, then the need for a stall can be avoided.
- Forwarding can be generalized to include passing a result directly to the functional unit that requires it.

Minimizing Data Hazard Stalls by Forwarding



Minimizing Data Hazard Stalls by Forwarding

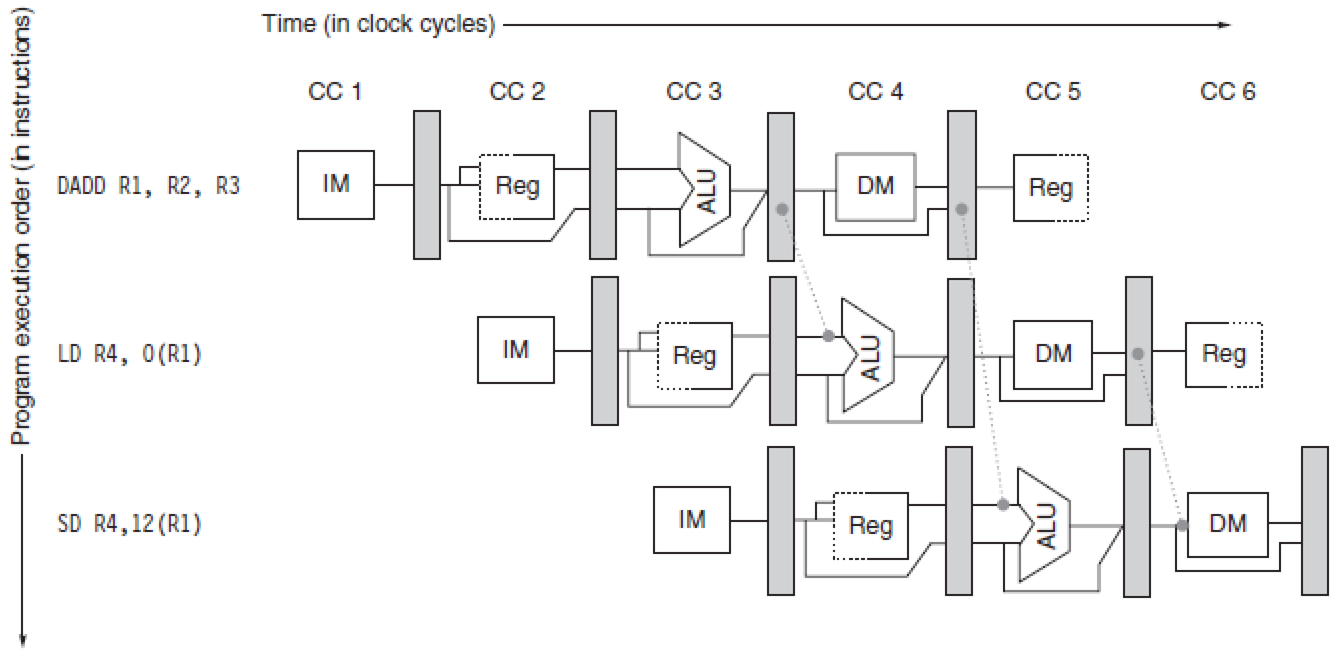


Minimizing Data Hazard Stalls by Forwarding

- Consider the following sequence

```
DADD R1,R2,R3
LD   R4,0(R1)
SD   R4,12(R1)
```

- Result of the load is forwarded from the memory output to the memory input to be stored
- ALU output is forwarded to ALU input for address calculation of both the load and the store



Data Hazards Requiring Stalls

- Not all potential data hazards can be handled by bypassing

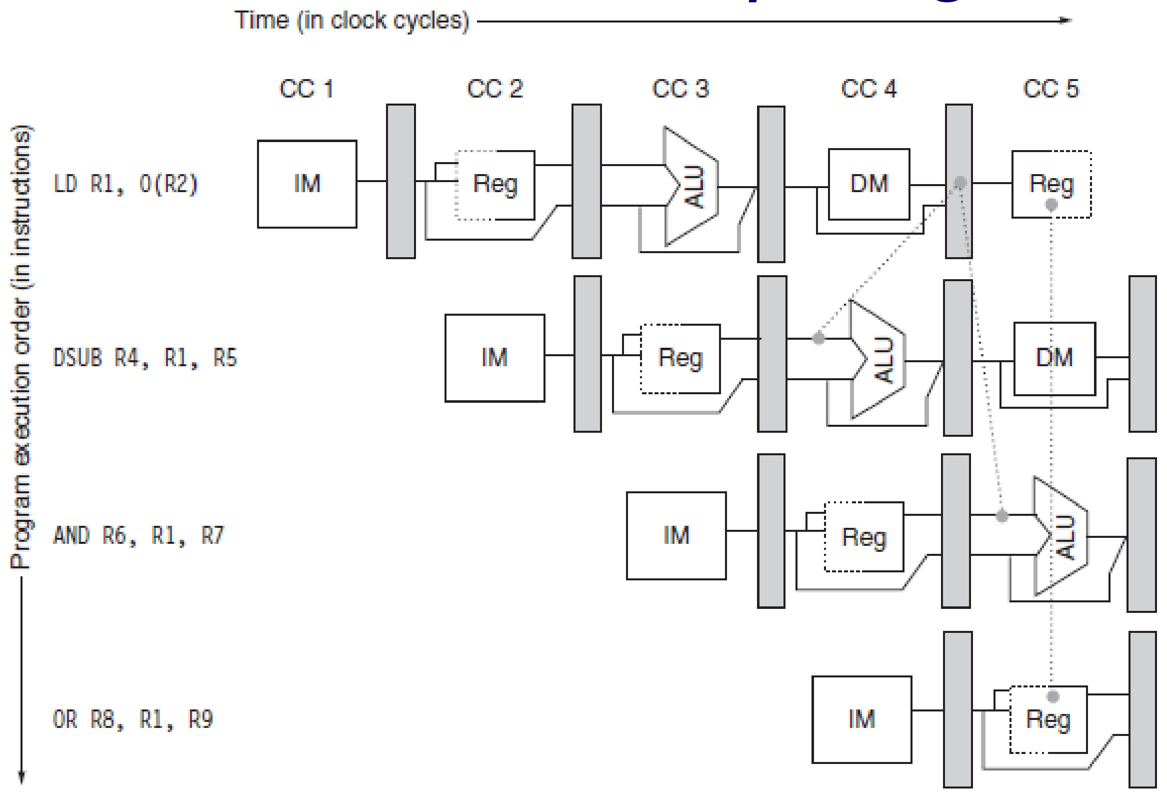
- Consider the following sequence

```
LD   R1,0(R2)
DSUB R4,R1,R5
AND  R6,R1,R7
OR   R8,R1,R9
```

- The LD instruction does not have the data until the end of clock cycle 4 (its MEM cycle), while the DSUB instruction needs to have the data by the beginning of that clock cycle.

- Data hazard from using the result of a load instruction cannot be completely eliminated with simple hardware

Data Hazards Requiring Stalls

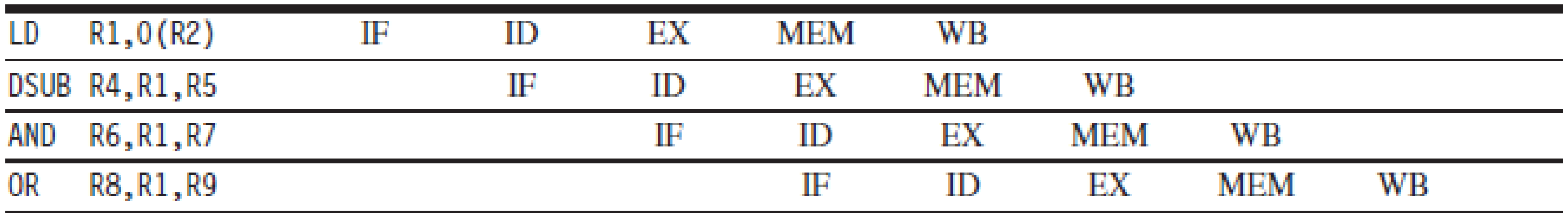


- Load instruction has a delay or latency that cannot be eliminated by forwarding alone.
- Instead, a hardware is required to preserve the correct execution pattern , called a *pipeline interlock*.
- Pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared.

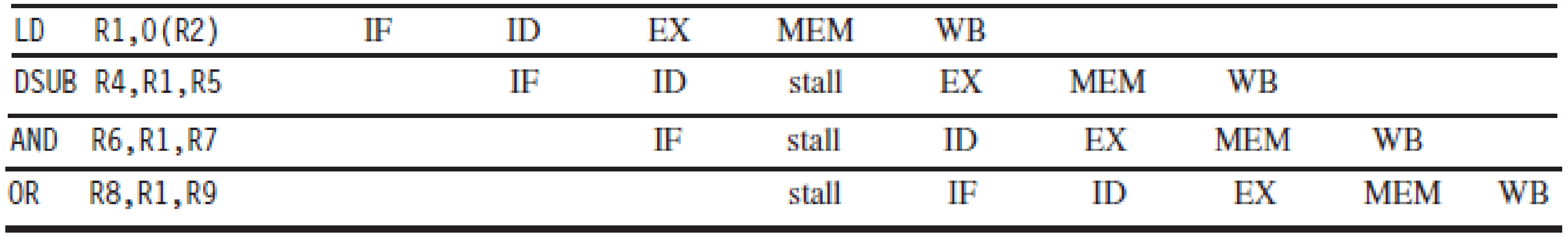
Data Hazards Requiring Stalls

- Interlock stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it.
- Pipeline interlock introduces a stall or bubble like in structural hazard.

```
LD    R1,0(R2)
DSUB R4,R1,R5
AND   R6,R1,R7
OR    R8,R1,R9
```



LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	EX	MEM	WB			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	



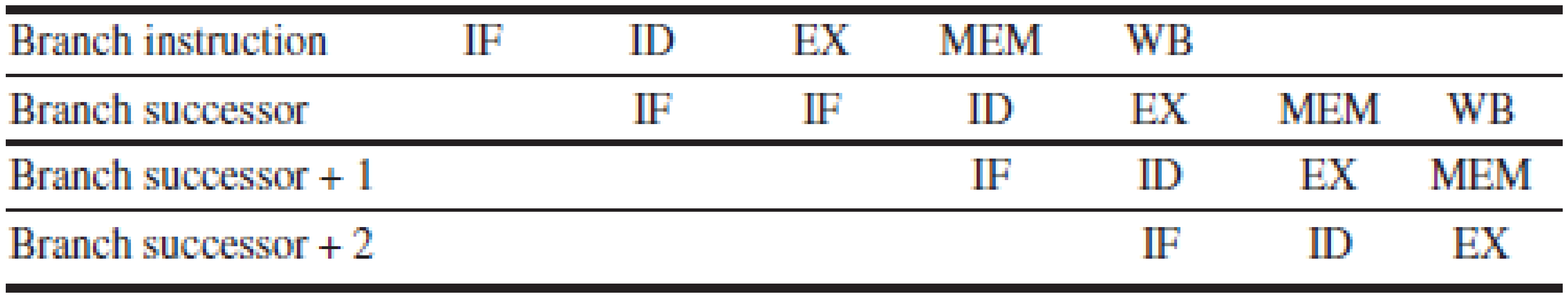
LD	R1,0(R2)	IF	ID	EX	MEM	WB				
DSUB	R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND	R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR	R8,R1,R9				stall	IF	ID	EX	MEM	WB

Branch Hazards

- *Control hazards* cause a greater performance loss for MIPS 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 a branch execution does not change the PC, it is *not taken*, or *untaken.*
- If instruction i is a taken branch, then the PC is normally not changed until the end of ID.
- Best way to handle branches is to redo the fetch of the instruction following a branch, once detected during ID.
- The first IF cycle is essentially a stall, because it never performs useful work.
- One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency.

Branch Hazards

- Instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known.
- If the branch is not taken, the second IF for branch successor is redundant.



Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

Reducing Pipeline Branch Penalties

- *1)* The simplest scheme to handle branches is to *freeze* or *flush* the pipeline, holding or deleting any instructions after the branch until the branch destination is known.

- In this case, Branch penalty is fixed and cannot be reduced by software

- *2)* A higher-performance, and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed.

- Processor state is not changed until the branch outcome is definitely known.

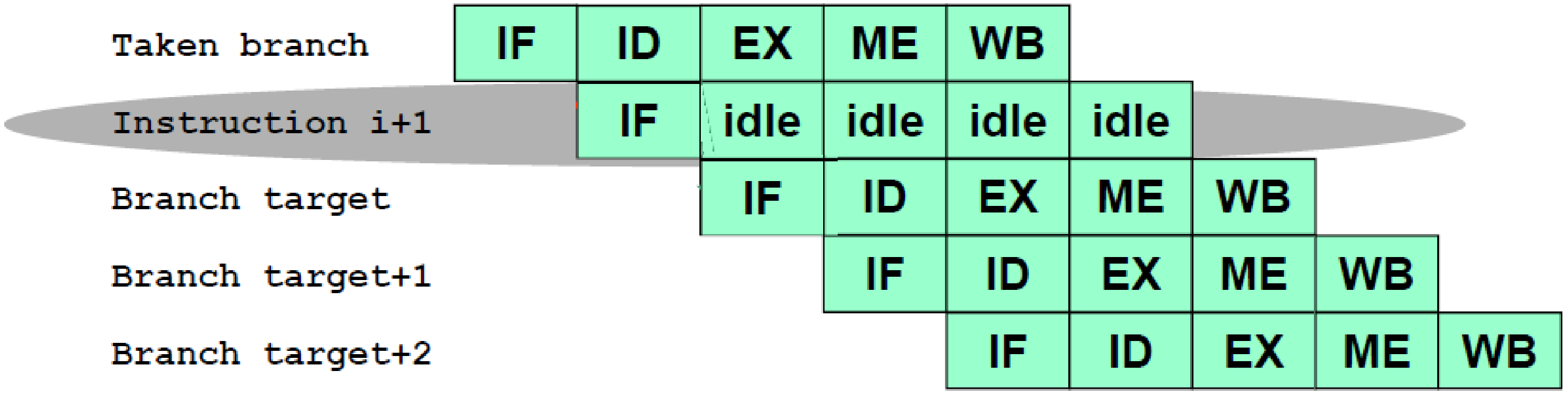
Reducing Pipeline Branch Penalties

- In the simple five-stage pipeline, this *predicted-not-taken* or *predicted untaken* scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction.
- The pipeline looks as if nothing out of the ordinary is happening.
- If the branch is taken, fetched instruction is turned into a no-op and restart the fetch at the target address.

Untaken branch	IF	ID	EX	ME	WB				
Instruction i+1		IF	ID	EX	ME	WB			
Instruction i+2			IF	ID	EX	ME	WB		
Instruction i+3				IF	ID	EX	ME	WB	
Instruction i+4					IF	ID	EX	ME	WB

Reducing Pipeline Branch Penalties

- *3)* An alternative scheme is to treat every branch as taken.
- As soon as the branch is decoded and the target address is computed, assume the branch to be taken and begin fetching and executing at the target.
- In five-stage pipeline, the target address is not known earlier - there is no advantage in this approach



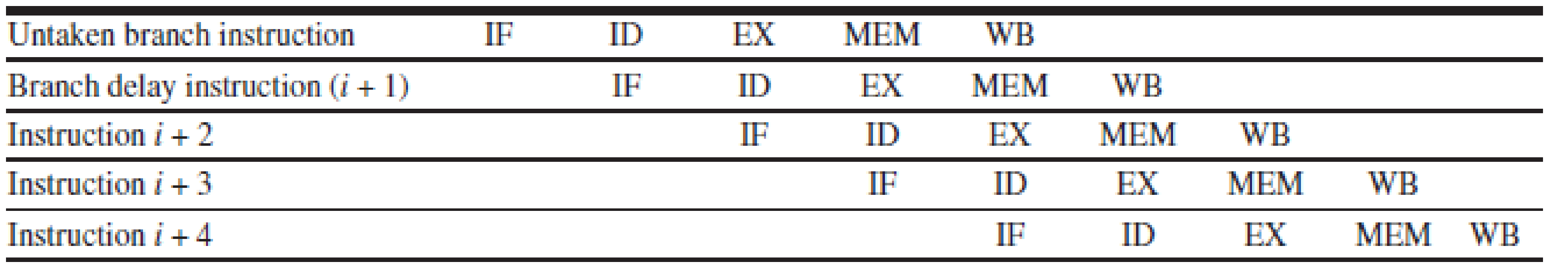
Taken branch	IF	ID	EX	ME	WB				
Instruction i+1		IF	idle	idle	idle	idle			
Branch target			IF	ID	EX	ME	WB		
Branch target+1				IF	ID	EX	ME	WB	
Branch target+2					IF	ID	EX	ME	WB

Reducing Pipeline Branch Penalties

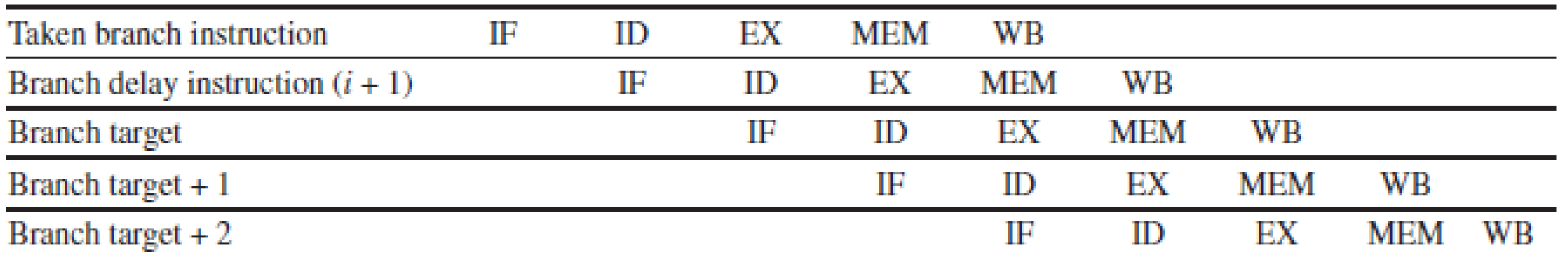
- *4)* *Delayed branch* - technique heavily used in early RISC processors and works well in five-stage pipeline
- execution cycle with branch delay of one

```
branch instruction
sequential successor₁
branch target if taken
```

- sequential successor is in the *branch delay slot and is* executed whether or not the branch is taken.



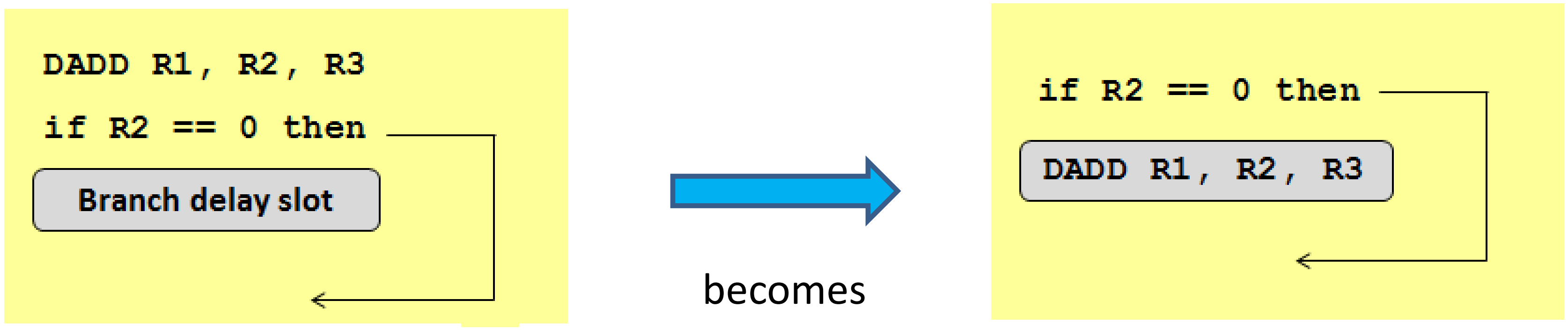
Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB



Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Scheduling the delay slots

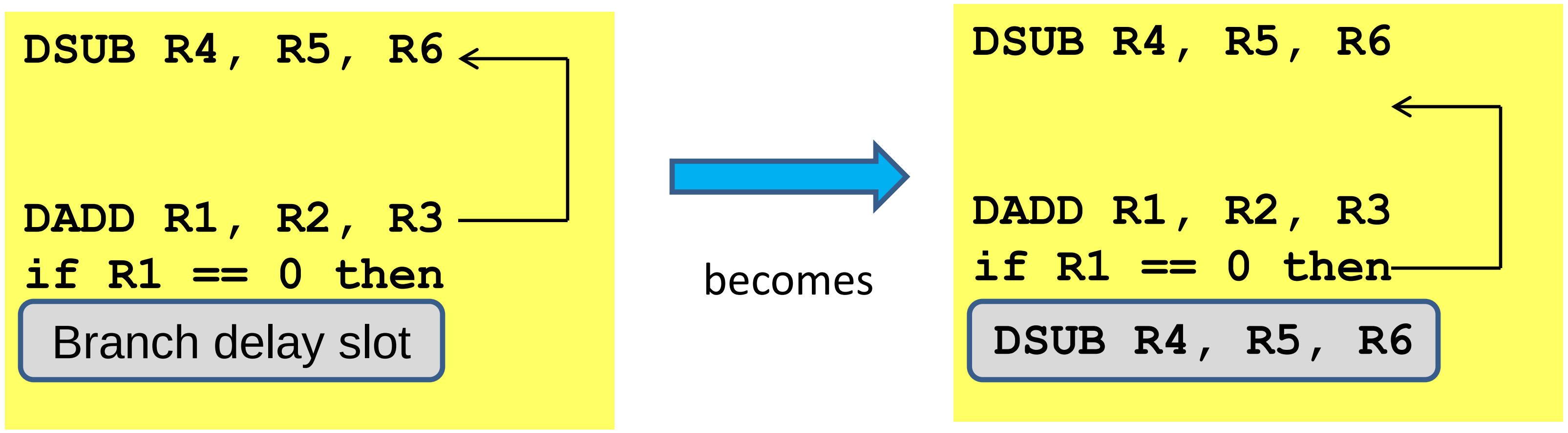
i) From before – scheduled with an independent instruction from before the branch



Scheduling the delay slots

i) From before – scheduled with an independent instruction from before the branch

ii) From target – preferred when the branch is taken with high probability, such as a loop branch.

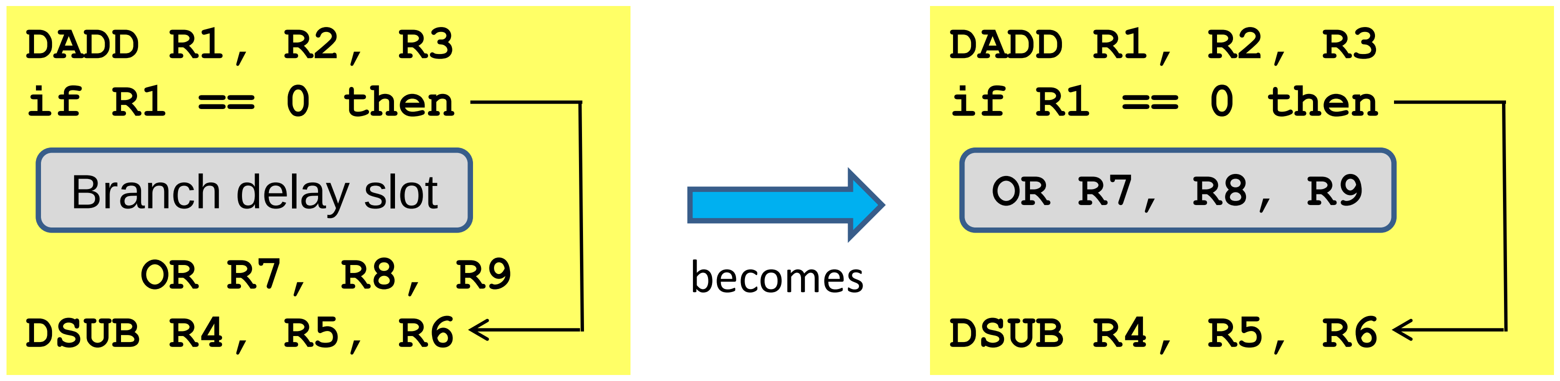


Scheduling the delay slots

i) From before – scheduled with an independent instruction from before the branch

ii) From target – preferred when the branch is taken with high probability, such as a loop branch.

iii) the branch may be scheduled from the not-taken fall-through



```
DADD R1, R2, R3
if R1 == 0 then
  Branch delay slot
    OR R7, R8, R9
DSUB R4, R5, R6
```

becomes

```
DADD R1, R2, R3
if R1 == 0 then
  OR R7, R8, R9

DSUB R4, R5, R6
```

Scheduling the delay slots

- Limitations on delayed-branch scheduling arise from:
  - restrictions on the instructions that are scheduled into the delay slots
  - ability to predict at compile time whether a branch is likely to be taken or not
- To improve the ability of the compiler to fill branch delay slots - *canceling* or *nullifying* branch

- When the branch behaves as predicted, the instruction in the branch delay slot is executed as it would normally be with a delayed branch.

- When the branch is incorrectly predicted, the instruction in the branch delay slot is simply turned into a no-op.

Performance of Branch Schemes

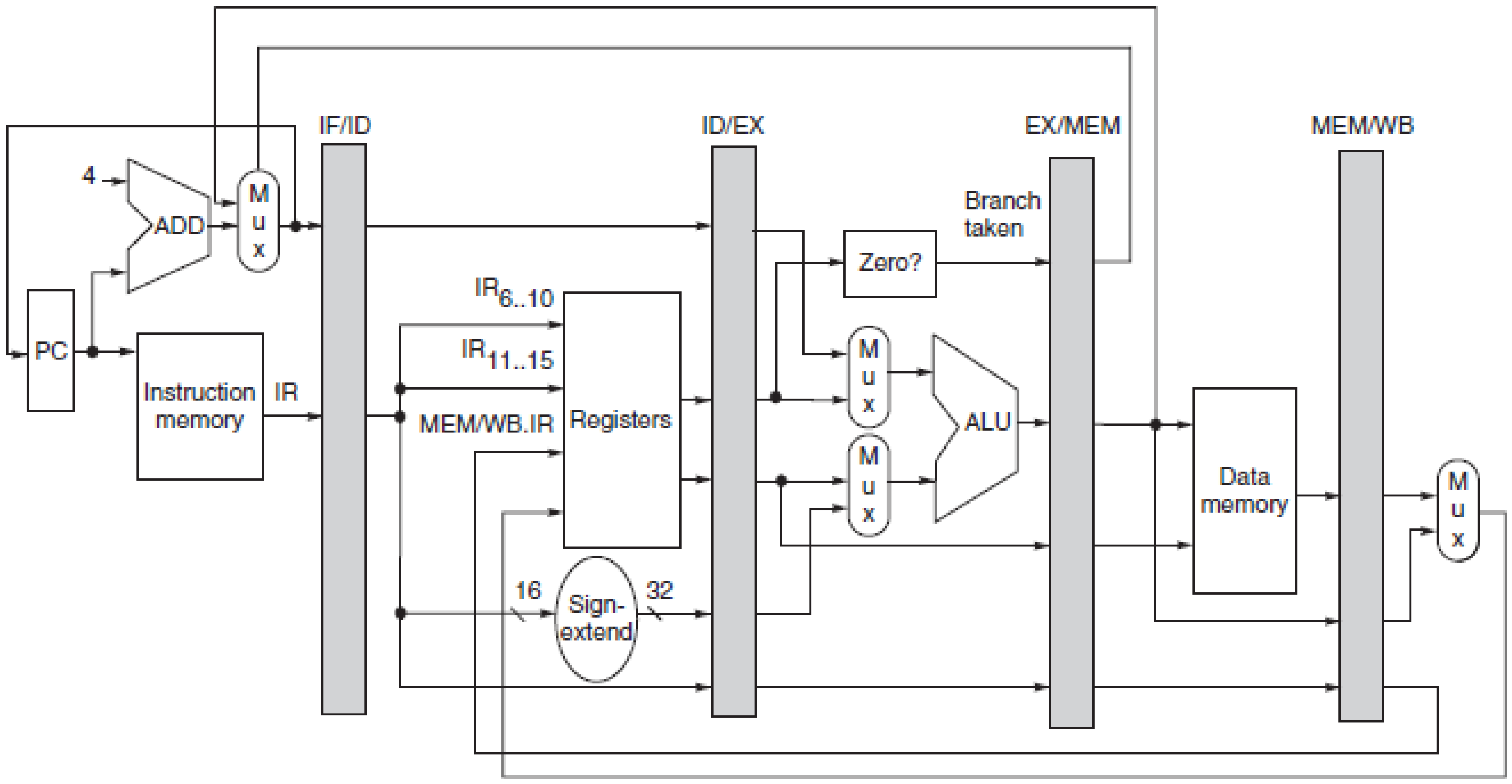
- Effective pipeline speedup with branch penalties, assuming an ideal CPI of 1, is

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

$$Pipeline\ stall\ cycles\ from\ branches = Branch\ frequency\ x\ Branch\ penalty$$

$$Pipeline\ speedup = \frac{Pipeline\ depth}{1 + Branch\ frequency\ x\ Branch\ penalty}$$

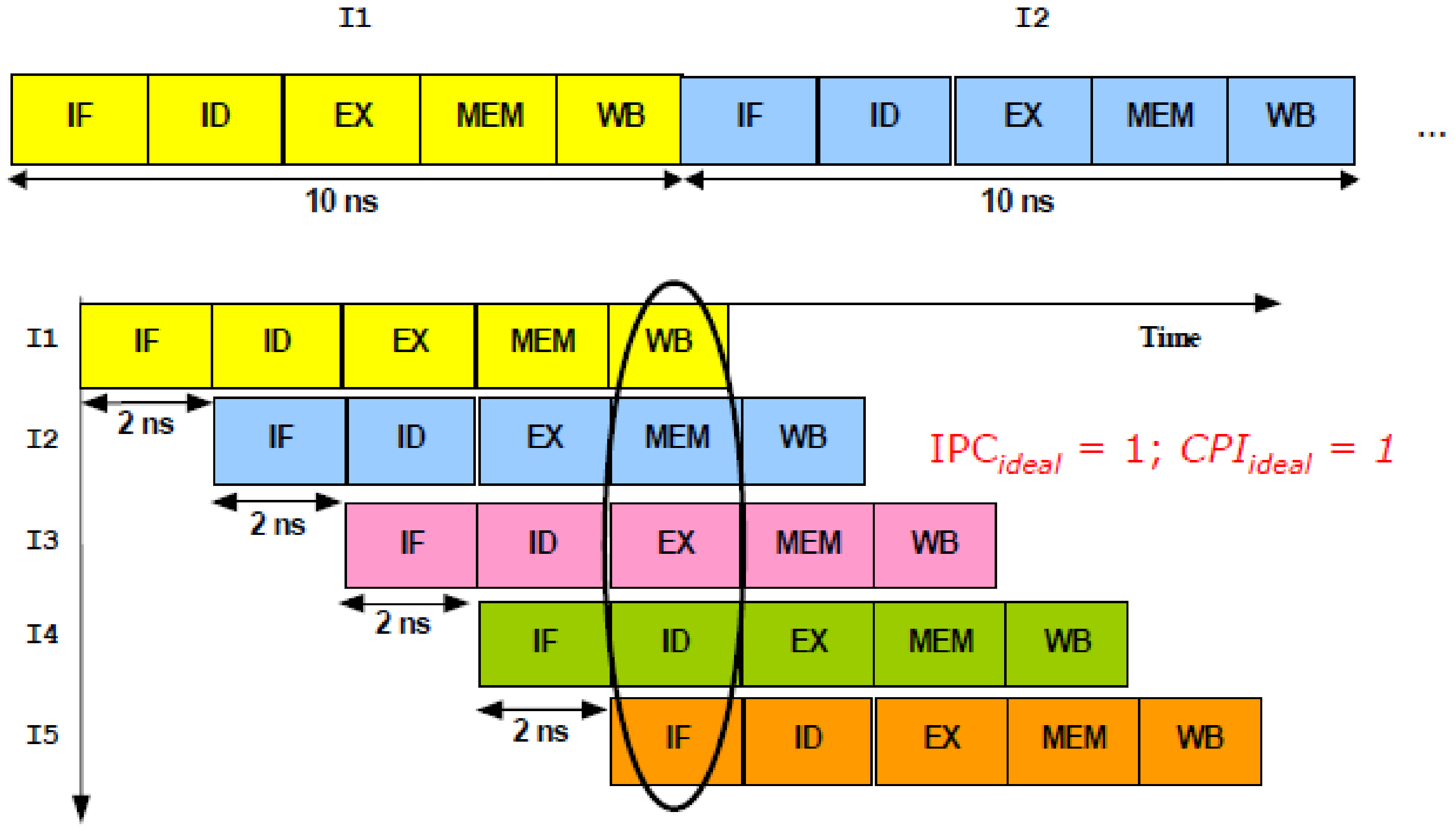
MIPS Pipeline Implementation



Instruction Level Parallelism

Instruction Level Parallelism

- Instructions evaluated in parallel
- Sequential vs. Pipelining Execution



Instruction Level Parallelism

- Two approaches to exploit ILP
  - Hardware based - dynamic
  - Software (compiler) based - static
- In a pipelined machine, actual CPI is derived as:

$$\mathrm{CPI}_{\mathrm{Pipeline}} = \mathrm{CPI}_{\mathrm{Ideal}} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}$$

- Reduction of any right-hand term reduces $\boldsymbol{CPI_{pipeline}}$ *to* $\boldsymbol{CPI_{ideal}}$ or alternatively increase the Instructions Per Clock

$$\mathbf{IPC = 1 / CPI}$$

- Best case: the max throughput would be to complete 1 Instruction Per Clock:

$$IPC_{\mathrm{ideal}} = 1;\ CPI_{\mathrm{ideal}} = 1$$

Instruction Level Parallelism

- For typical MIPS programs, the average dynamic branch frequency is often between 15% and 25%.

- Since these instructions are likely to depend upon one another, the amount of overlap that can be exploited within a basic block is likely to be less than the average basic block size.

- To obtain substantial performance enhancements, ILP must be exploited across multiple basic blocks

- Best way to increase ILP is to exploit parallelism among iterations of a loop - loop-level parallelism.

Instruction Level Parallelism

- Example – Loop level parallelism

```
for (i=1; i<=1000; i++)
     x[i] = x[i] + y[i];
```

- Computation in each iteration is independent of the previous iterations and the loop is thus parallel

- Different techniques to convert loop-level parallelism into Instruction level parallelism.