# COL216
# Computer Architecture

Processor design -

Pipelining

7th February, 2022

# Outline of this lecture

- Timings of Single cycle, multi-cycle designs
- Increasing performance with pipelining
- Limitations of pipelined approach
  - Hazards
- Handling hazards
  - Removing hazards
  - Reducing effect of hazards

# Single cycle design

# Problems with single cycle design

- Slowest instruction pulls down the clock frequency
- Resource utilization is poor
- There are some instructions which are impossible to be implemented in this manner

# Multi-cycle design

# Features of multi-cycle design

- Actions split into multiple steps
- Registers hold intermediate values
- All instructions need not take same steps
- Clock frequency decided by slowest step
- Resources can be shared across cycles
  - Eliminate adders
  - Use single memory
  - More multiplexing

# Improving the design further

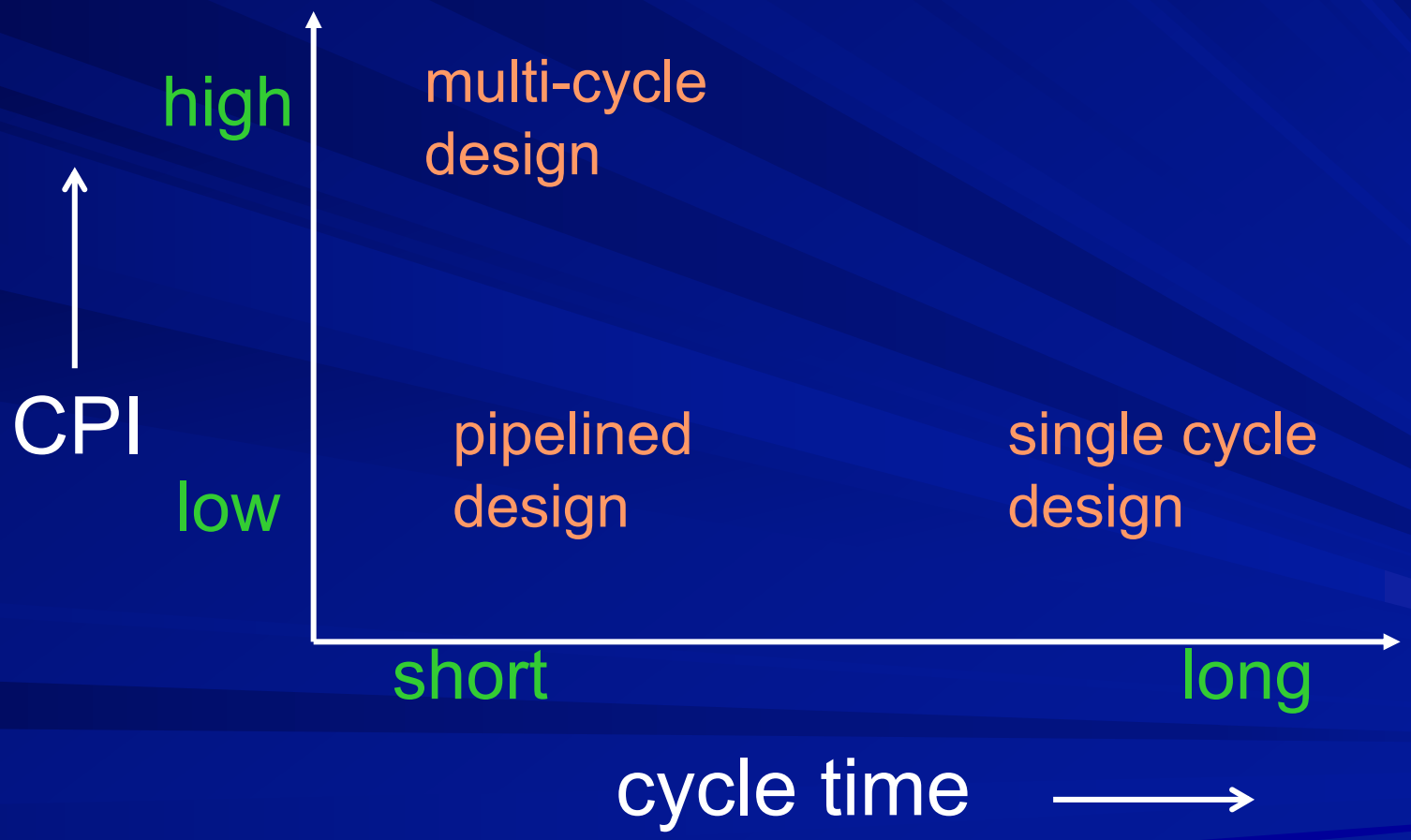
If resource saving is not important,
can the performance be improved further?

# Pipelined design

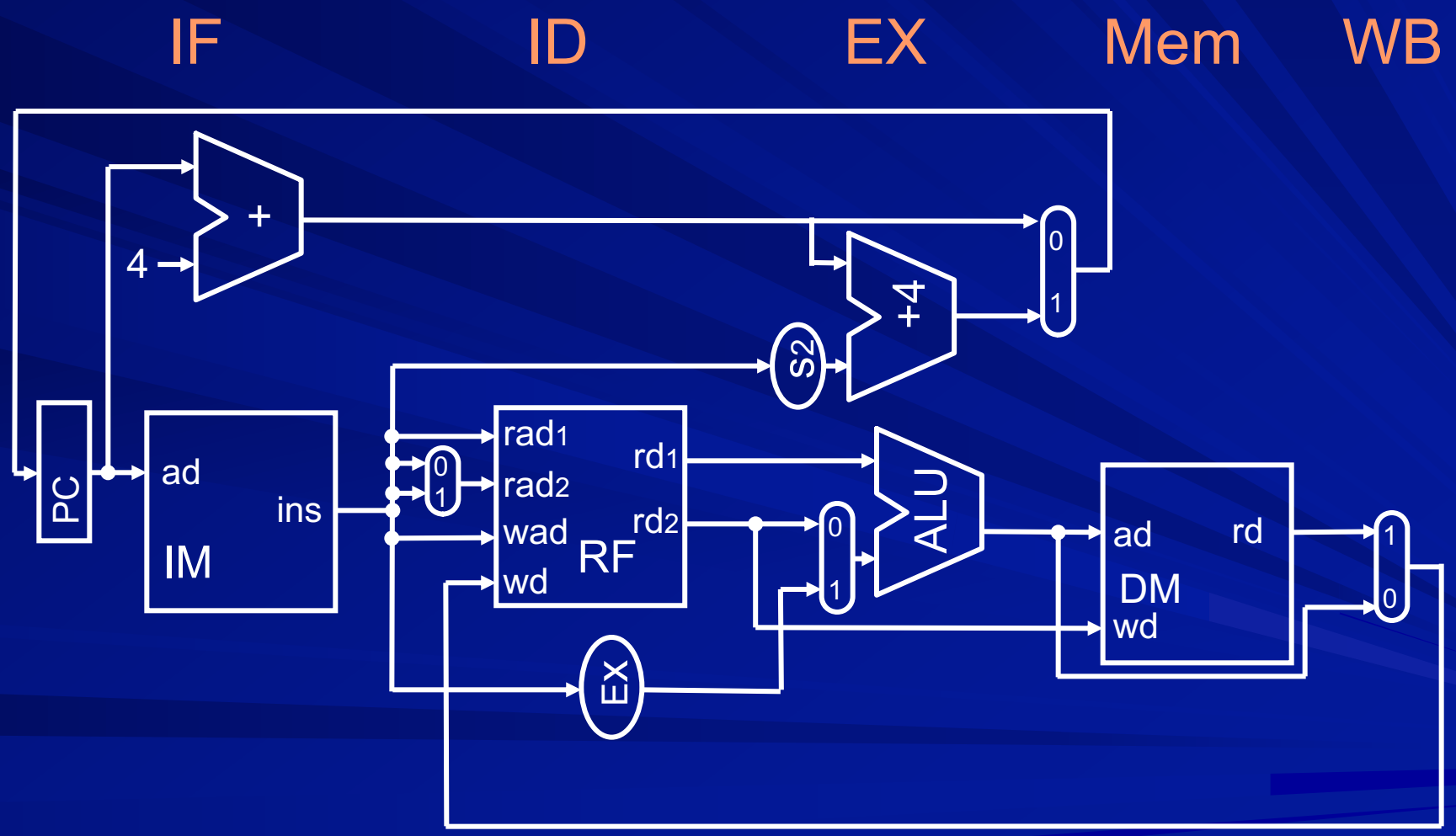
# Resource usage in different designs

- Single cycle design
  - each resource is tied up for the entire duration of the instruction execution
- Multi-cycle design
  - resource utilized in cycle `t` of instruction `I` is available again for cycle `t+1` of instruction `I`
- Pipelined design
  - resource utilized in cycle `t` of instruction `I` is available again for cycle `t` of instruction `I+1`

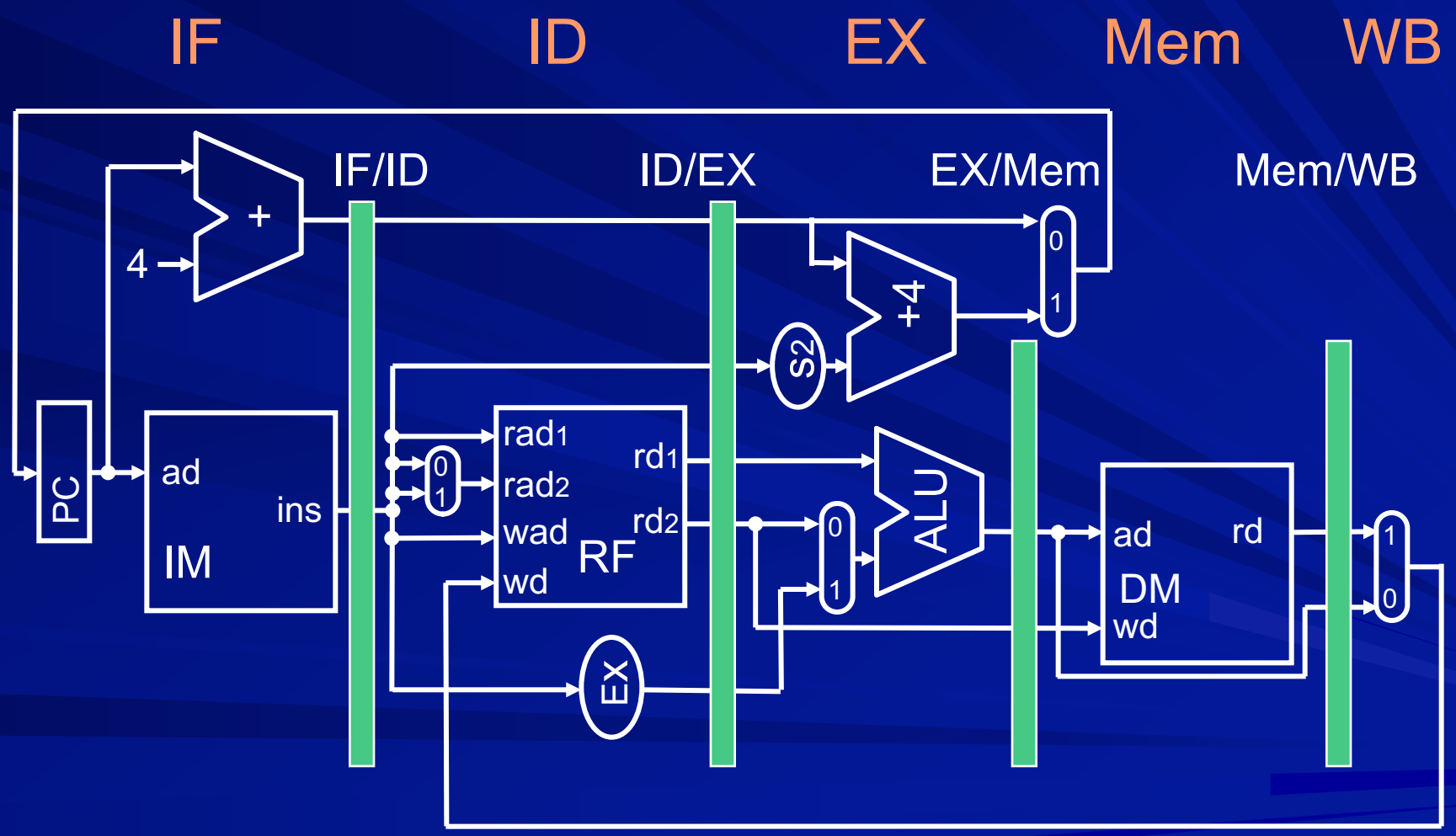
# Performance of different designs



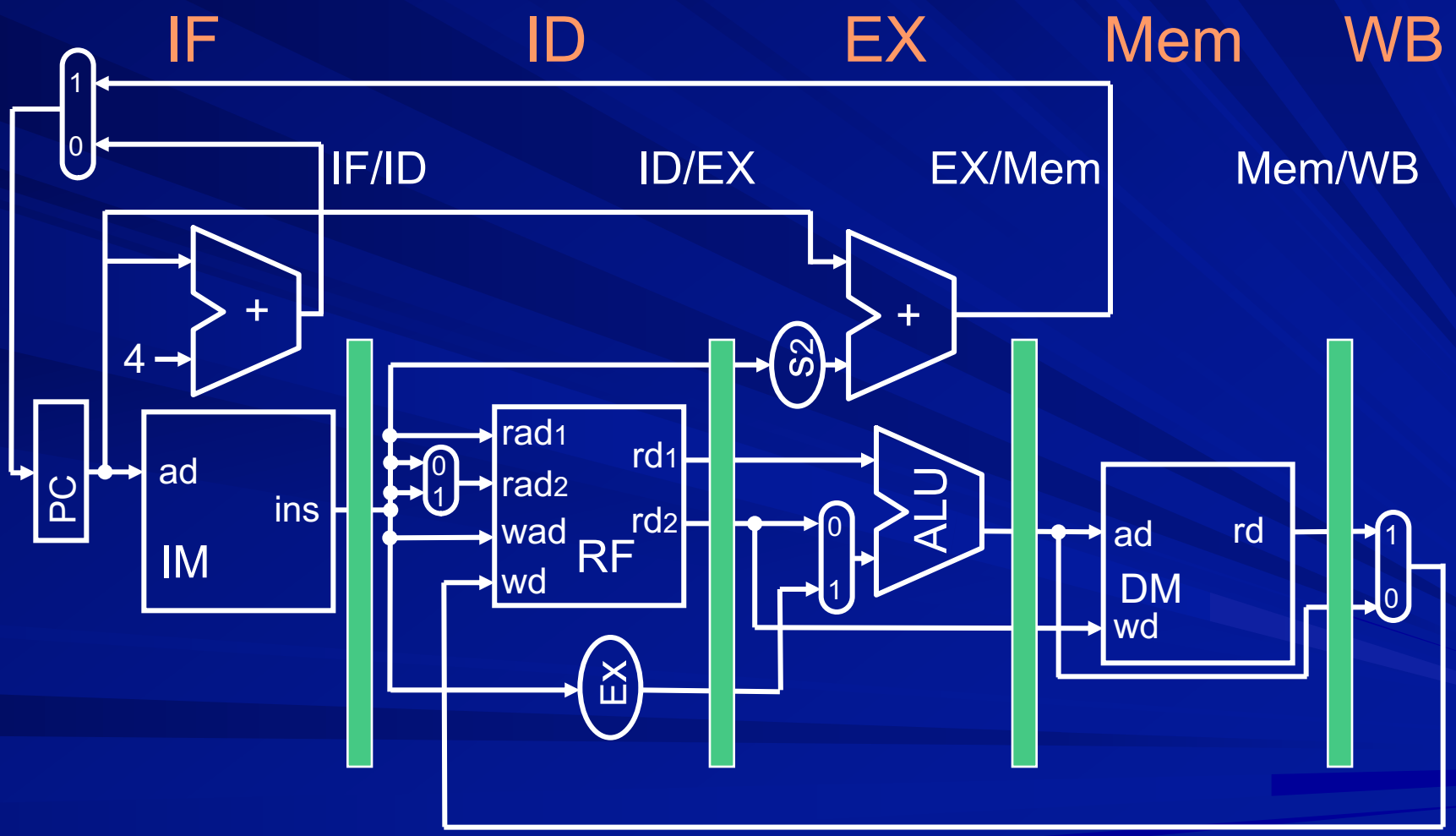
# Single cycle datapath



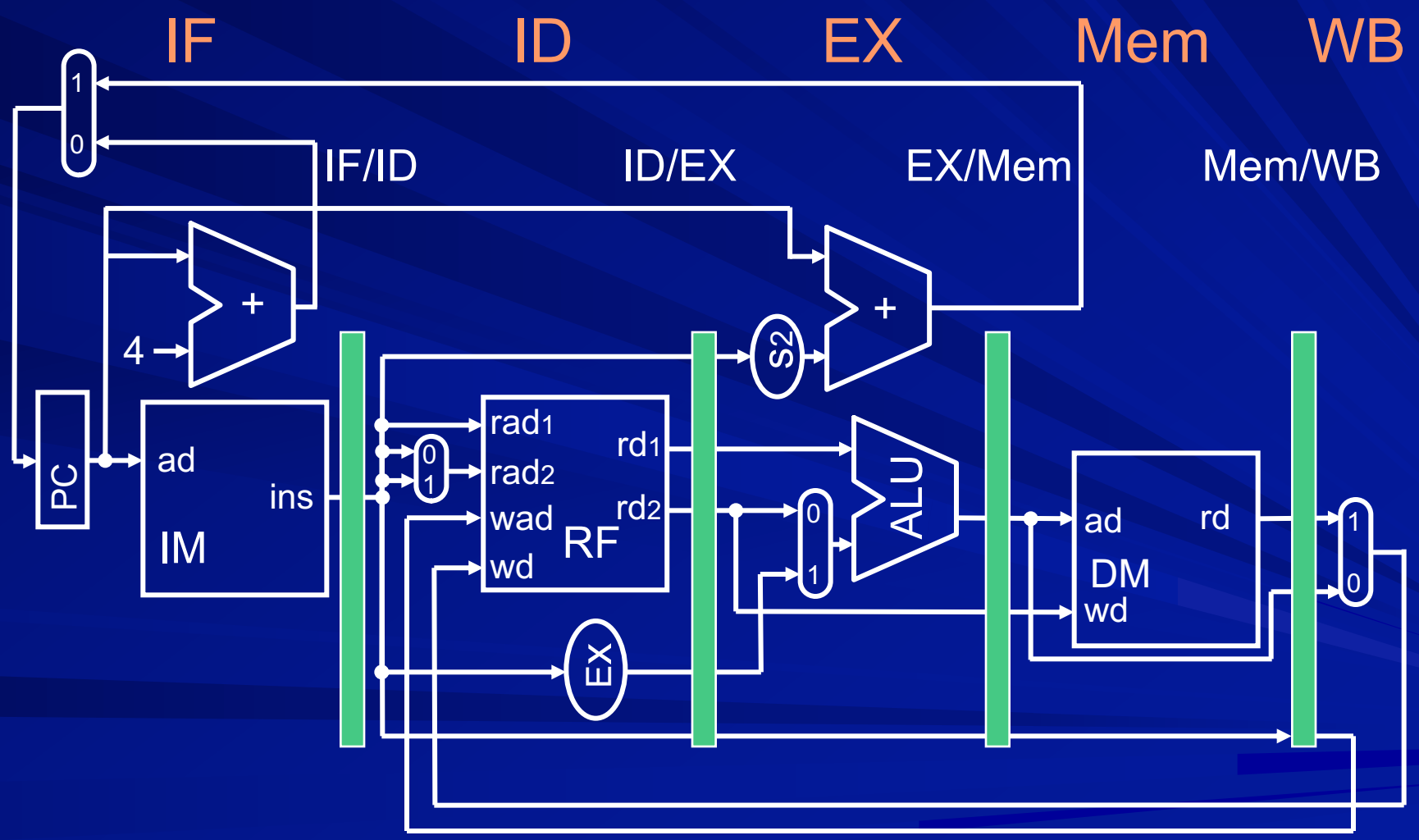
# Pipelined datapath



# Fetch new instruction every cycle



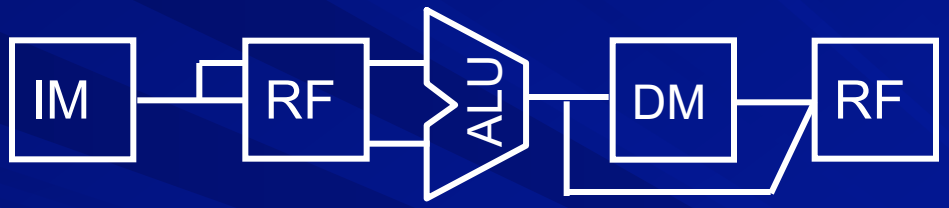
# Correction for WB stage



# Graphical representation

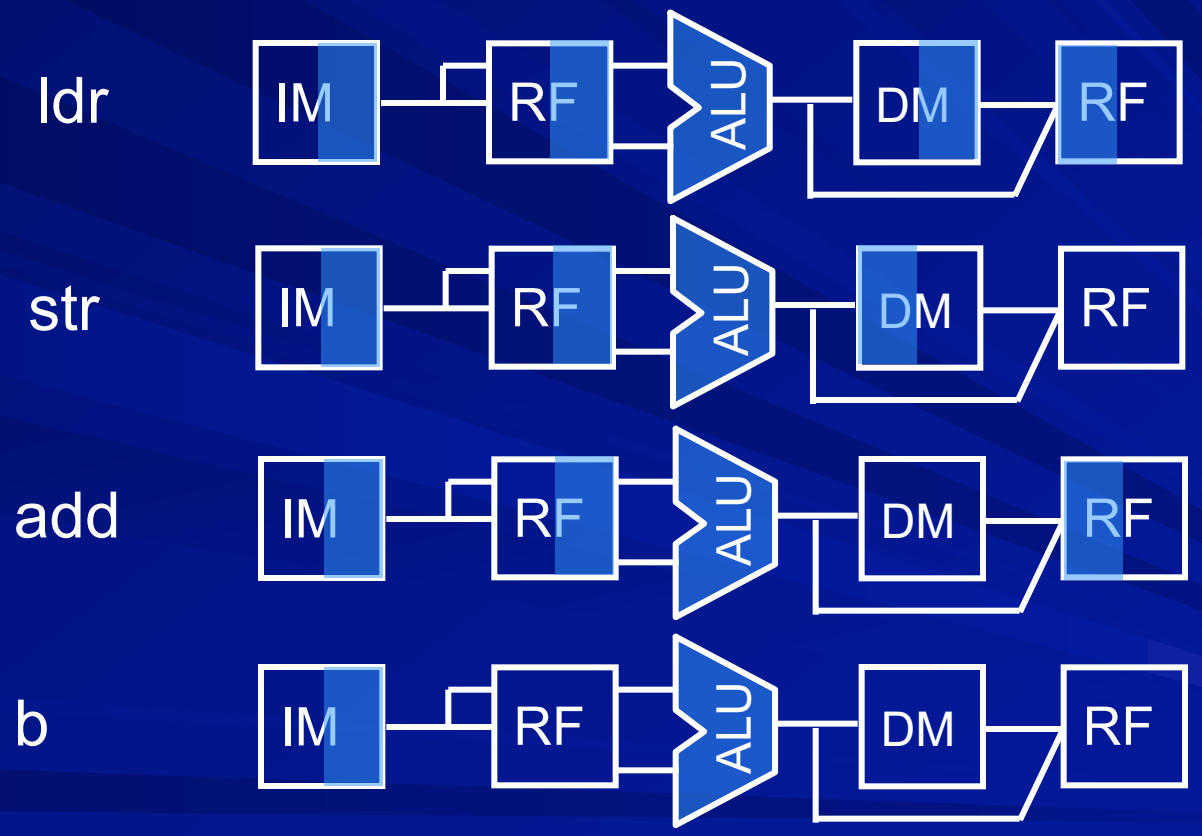
5 stage pipeline

stages




actions

# Usage of stages by instructions



# Representing pipelined execution

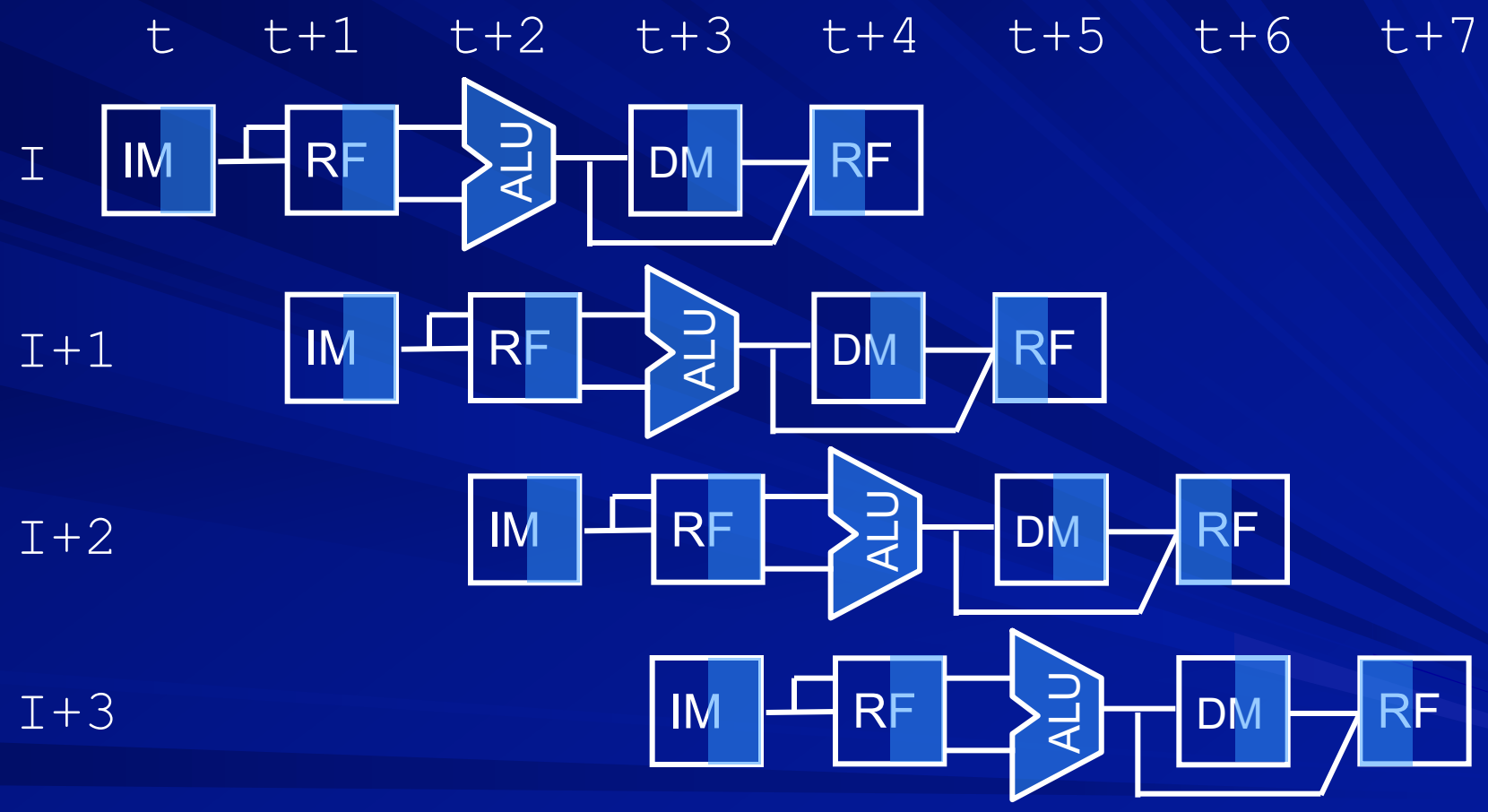
## Representation I

- Horizontal axis: time
- Vertical axis: instructions

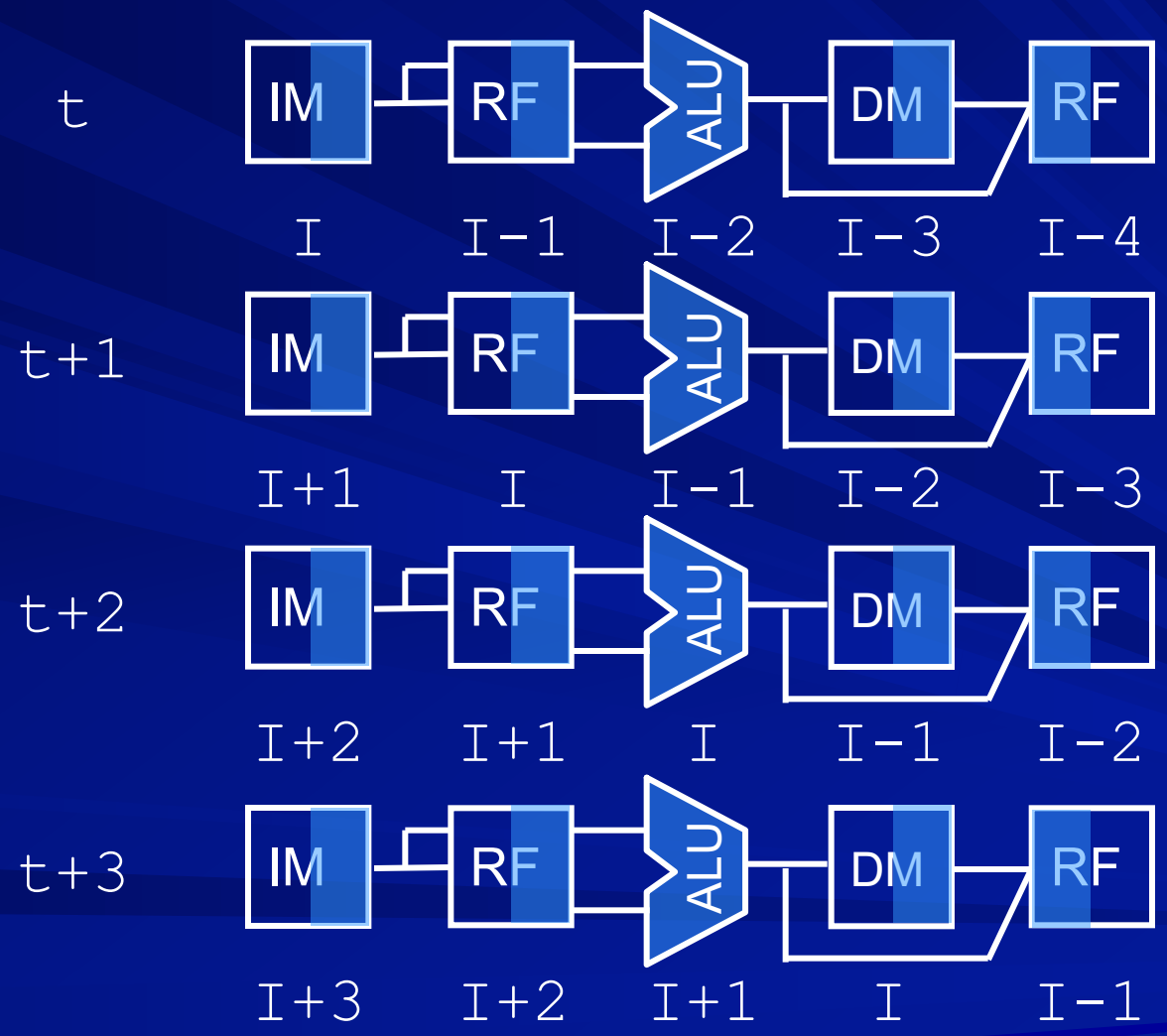
## Representation II

- Horizontal axis: pipeline stages
- Vertical axis: time

# Representation I



# Representation II



# Hurdles in instruction pipelining

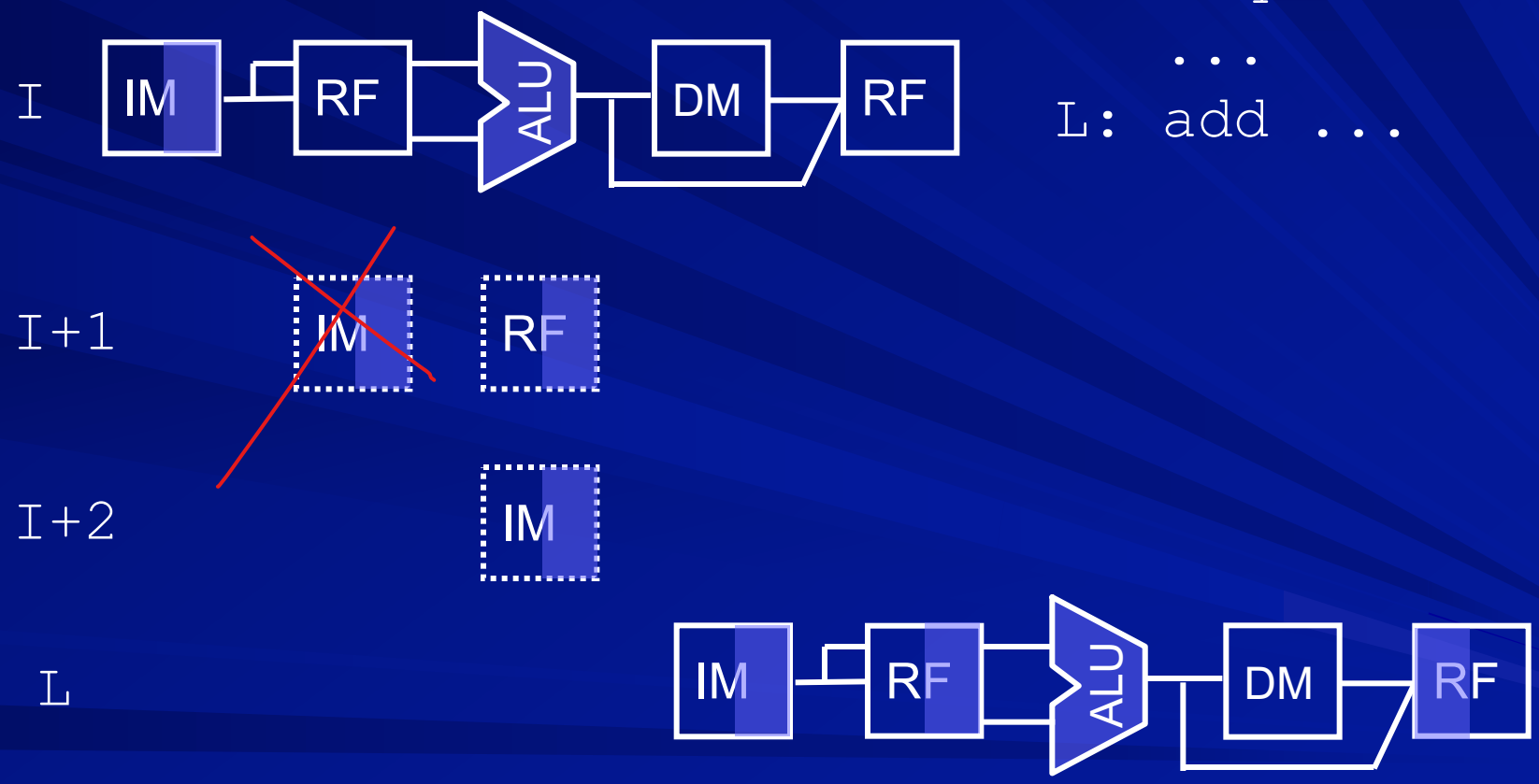
- Structural hazards
  - Resource conflicts - two instruction require same resource in the same cycle
- Data hazards
  - Data dependencies - one instruction needs data which is yet to be produced by another instruction
- Control Hazards
  - Decision about next instruction needs more cycles

# Structural hazards

- No structural hazards in the present design
  - separate instruction and data memories
  - adders for PC increment and offset addition to PC separate from main ALU
  - each instruction uses ALU at most in one cycle
  - one instruction can read from RF while other can write into it in the same cycle

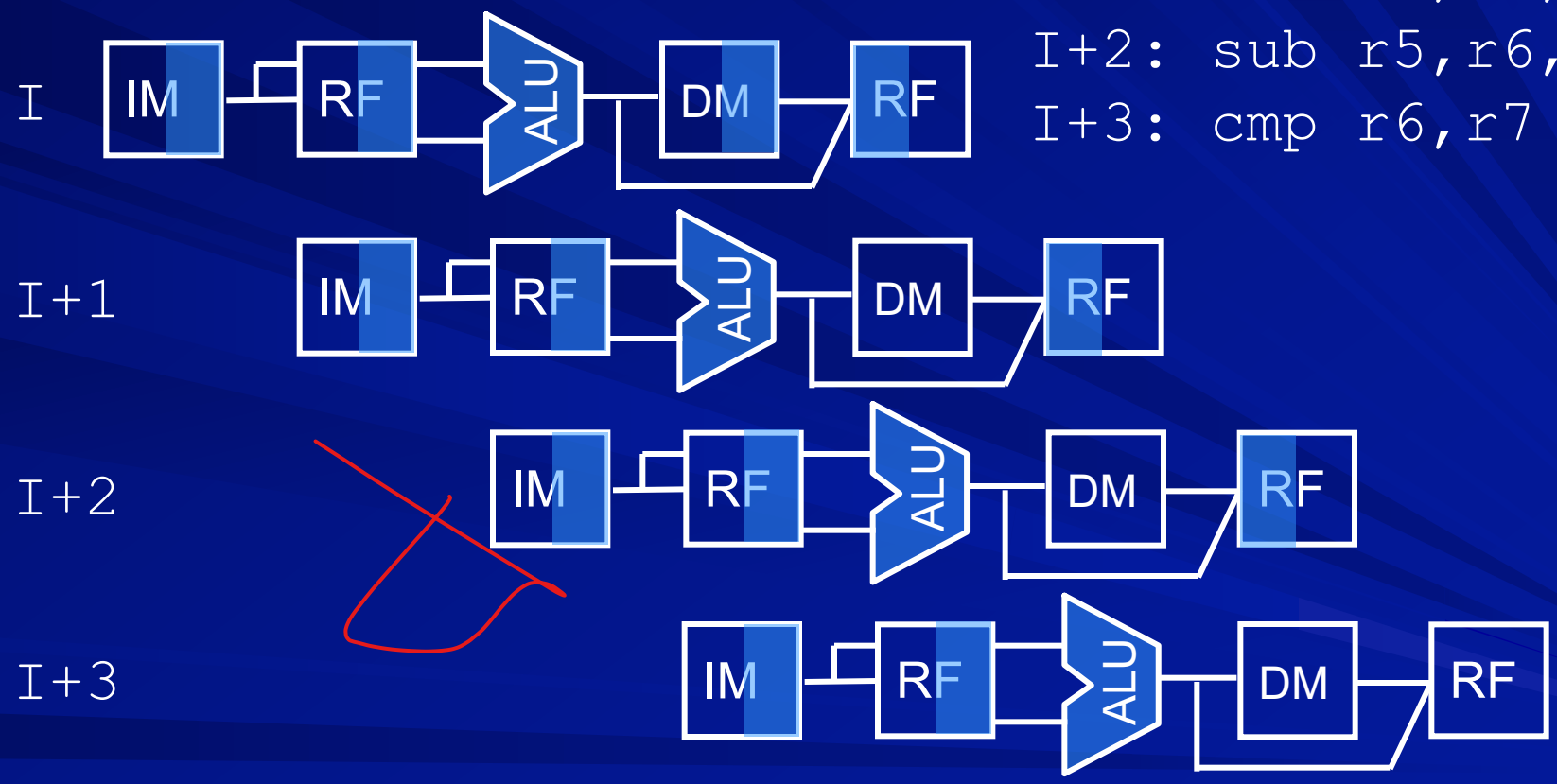
# Stalls due to control hazards

```
I: beq L
   ...
L: add ...
```



# Data hazards

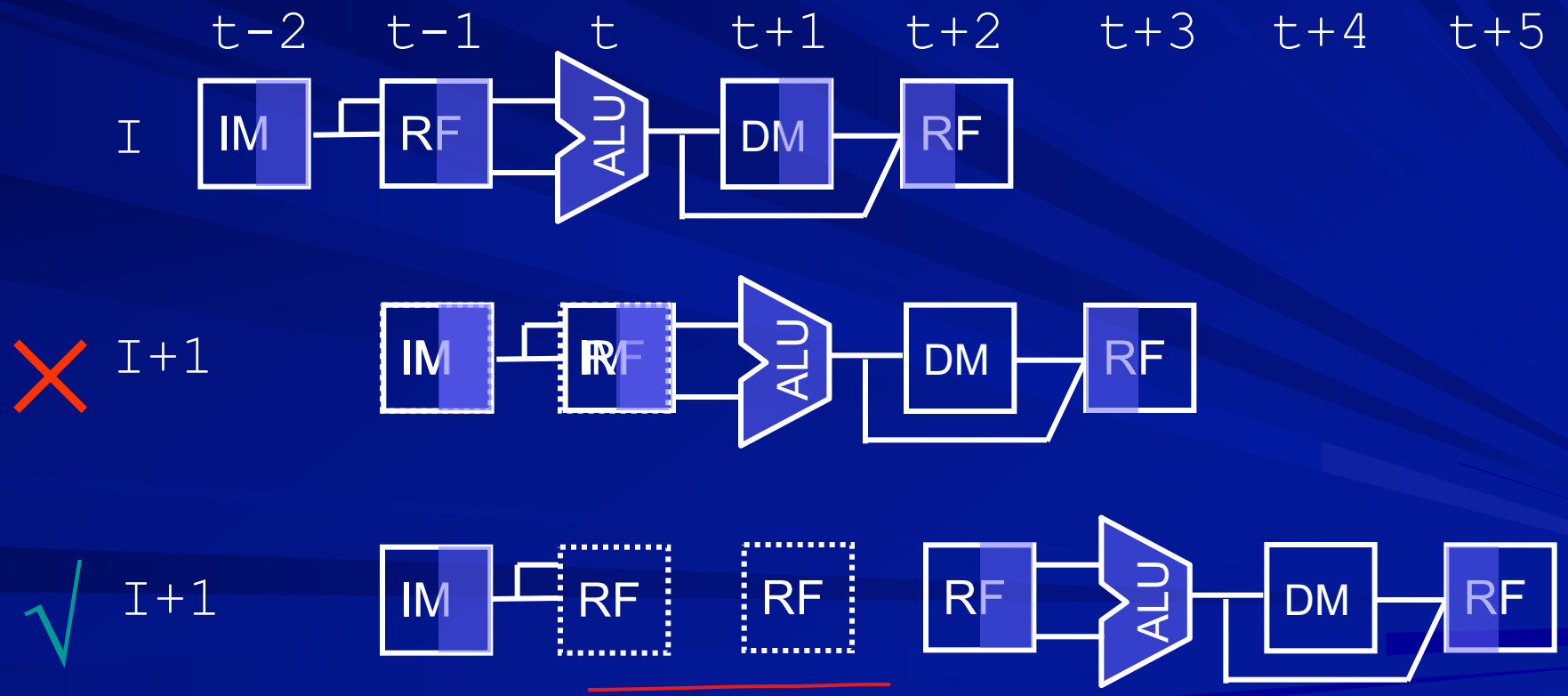
```
I:    ldr r6,...
I+1:  add r4,r6,..
I+2:  sub r5,r6,..
I+3:  cmp r6,r7
```



# Stalls due to data hazards

## instruction view

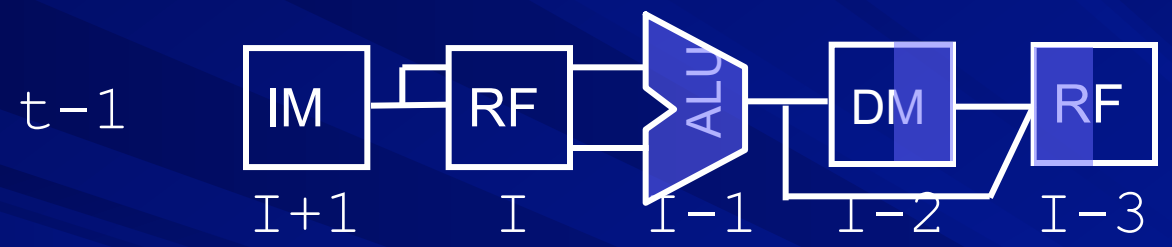
```
I:   ldr r6,...
I+1: add r4,r6,..
```



# Stalls due to data hazards

## stage-wise view

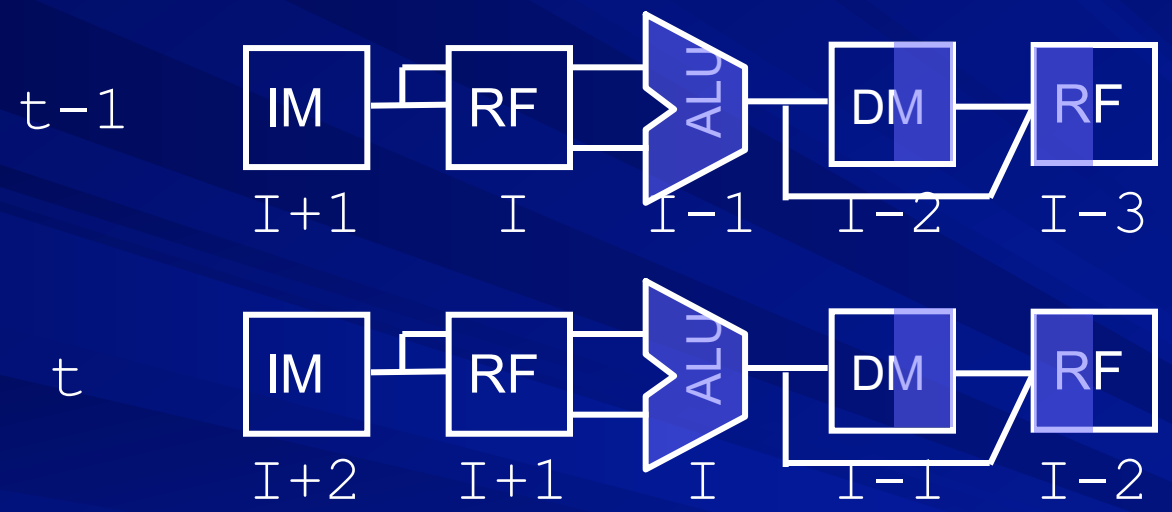
```
I:   ldr r6,...
I+1: add r4,r6,..
```



# Stalls due to data hazards

## stage-wise view

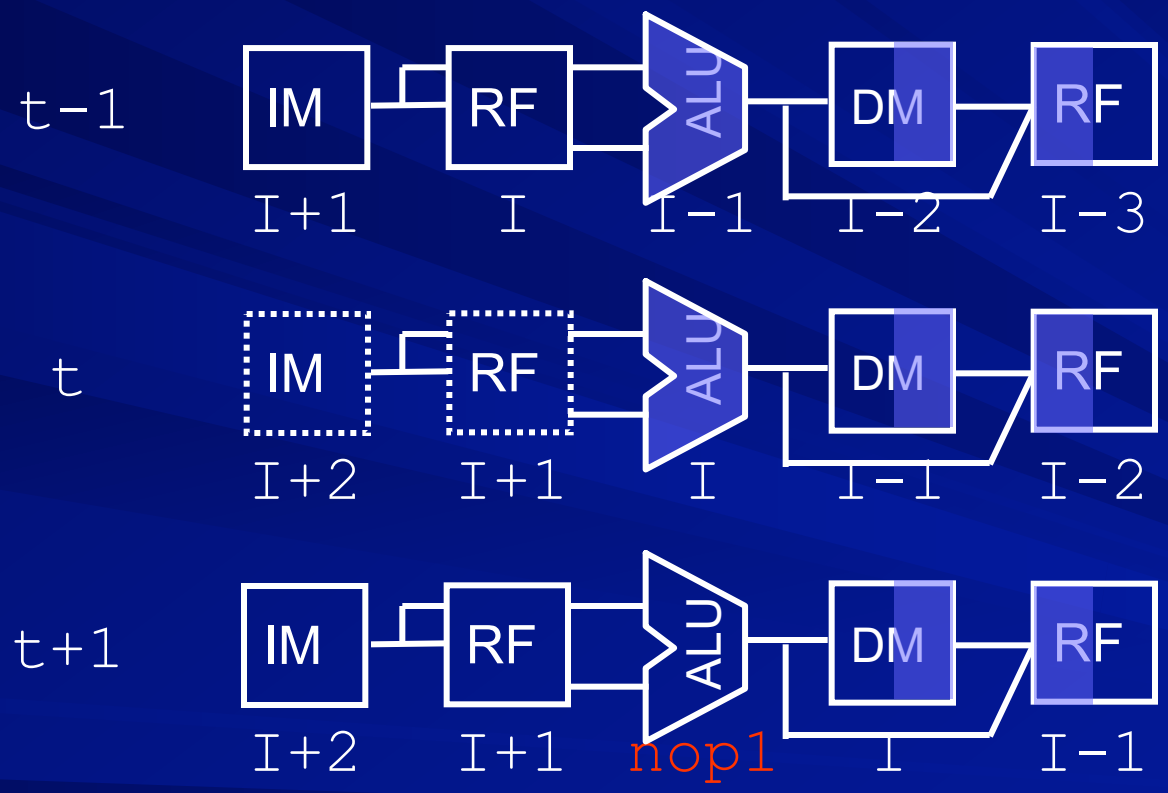
```
I:   ldr r6,...
I+1: add r4,r6,..
```



# Stalls due to data hazards

## stage-wise view

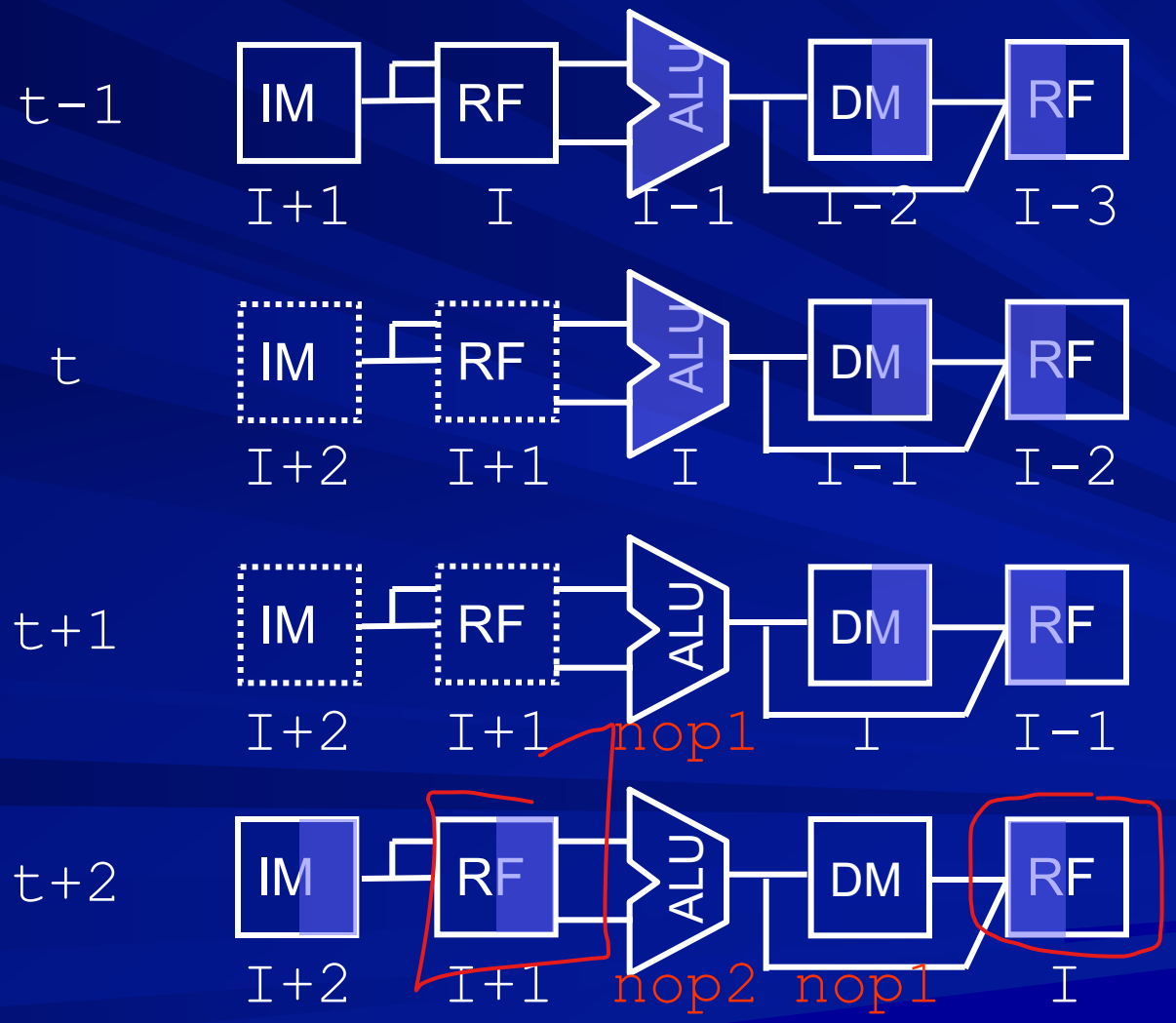
```
I:   ldr r6,...
I+1: add r4,r6,..
```



# Stalls due to data hazards

## stage-wise view

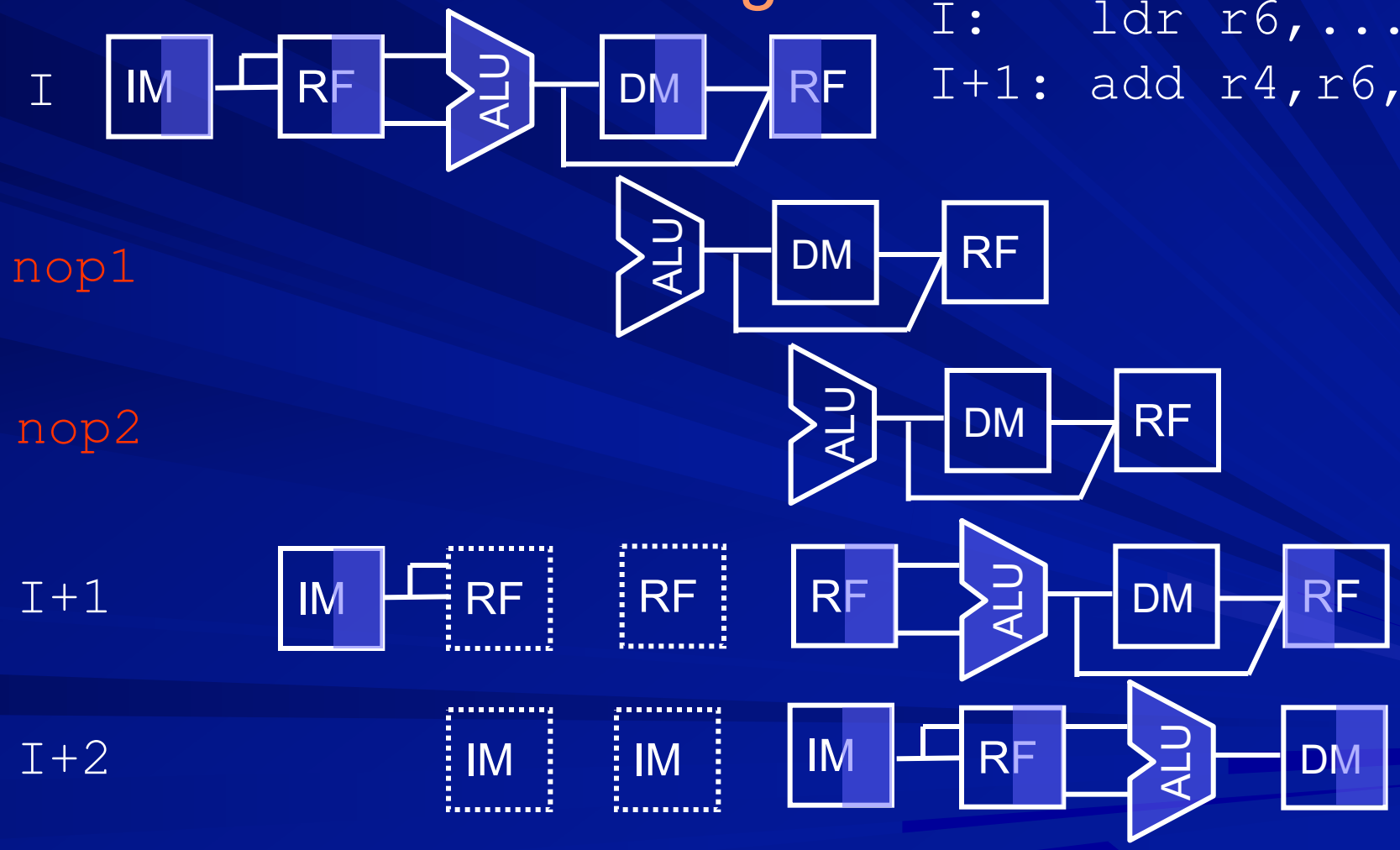
```
I:   ldr r6,...
I+1: add r4,r6,..
```



# Stalls due to data hazards

## instruction view again

```
I:    ldr r6,...
I+1:  add r4,r6,..
```



# Handling hazards

- Data hazards
  - detect instructions with data dependence
  - introduce nop instructions (bubbles) in the pipeline
  - more complex: data forwarding
- Control hazards
  - detect branch instructions
  - flush inline instructions if branching occurs
  - more complex: branch prediction

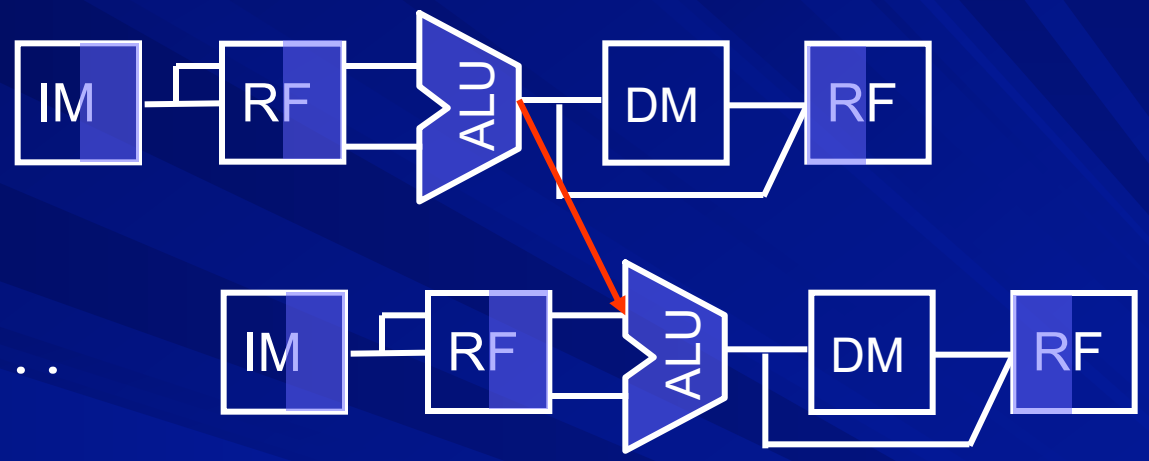
# Are there software solutions?

- Separate dependent instructions by reordering code
- Insert nop instructions in worst case
- Treat branches as delayed branches and insert suitable instructions in delay slots

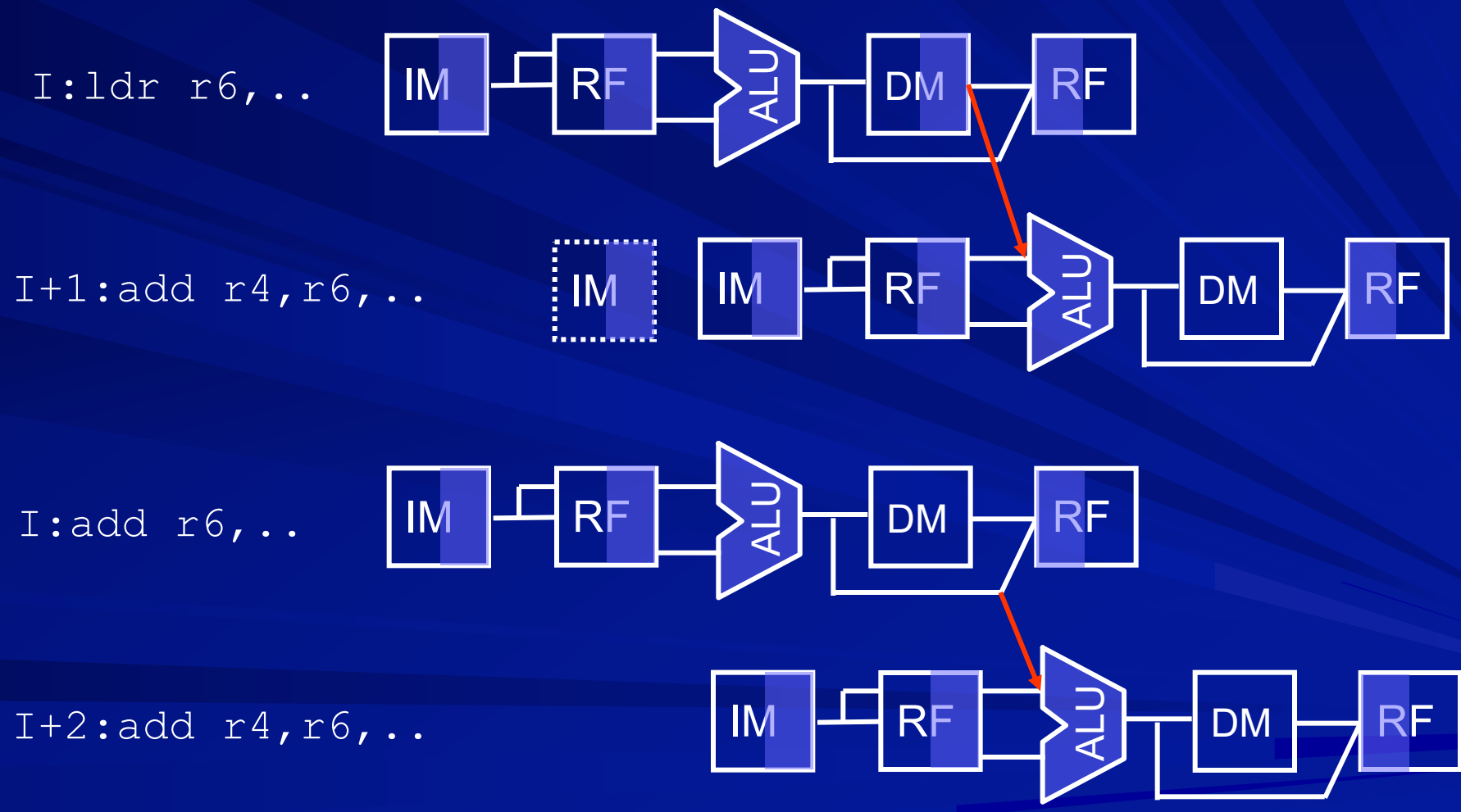
# Data forwarding path P1

I:add r6,..

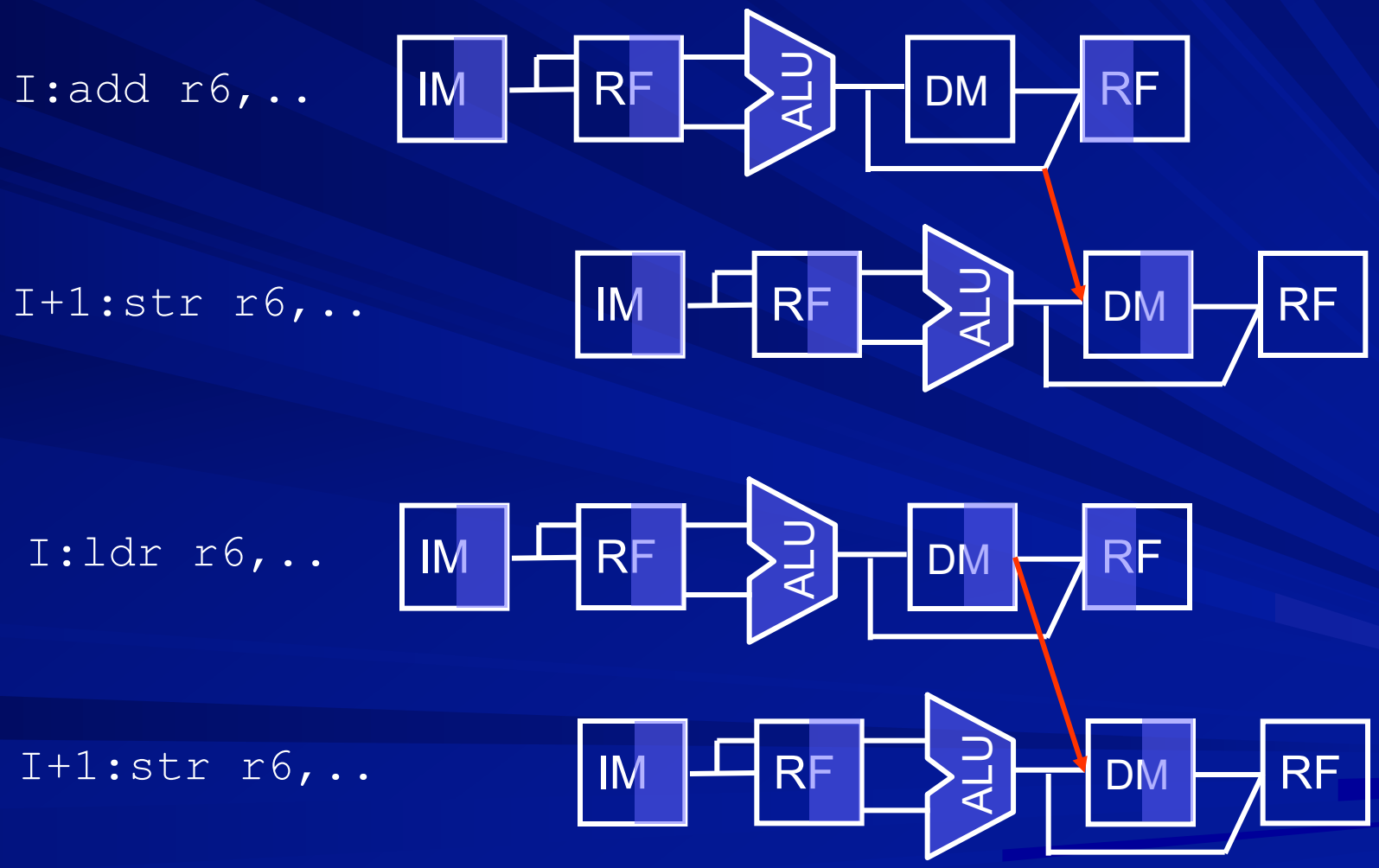
I+1:add r4,r6,..



# Data forwarding path P2



# Data forwarding path P3

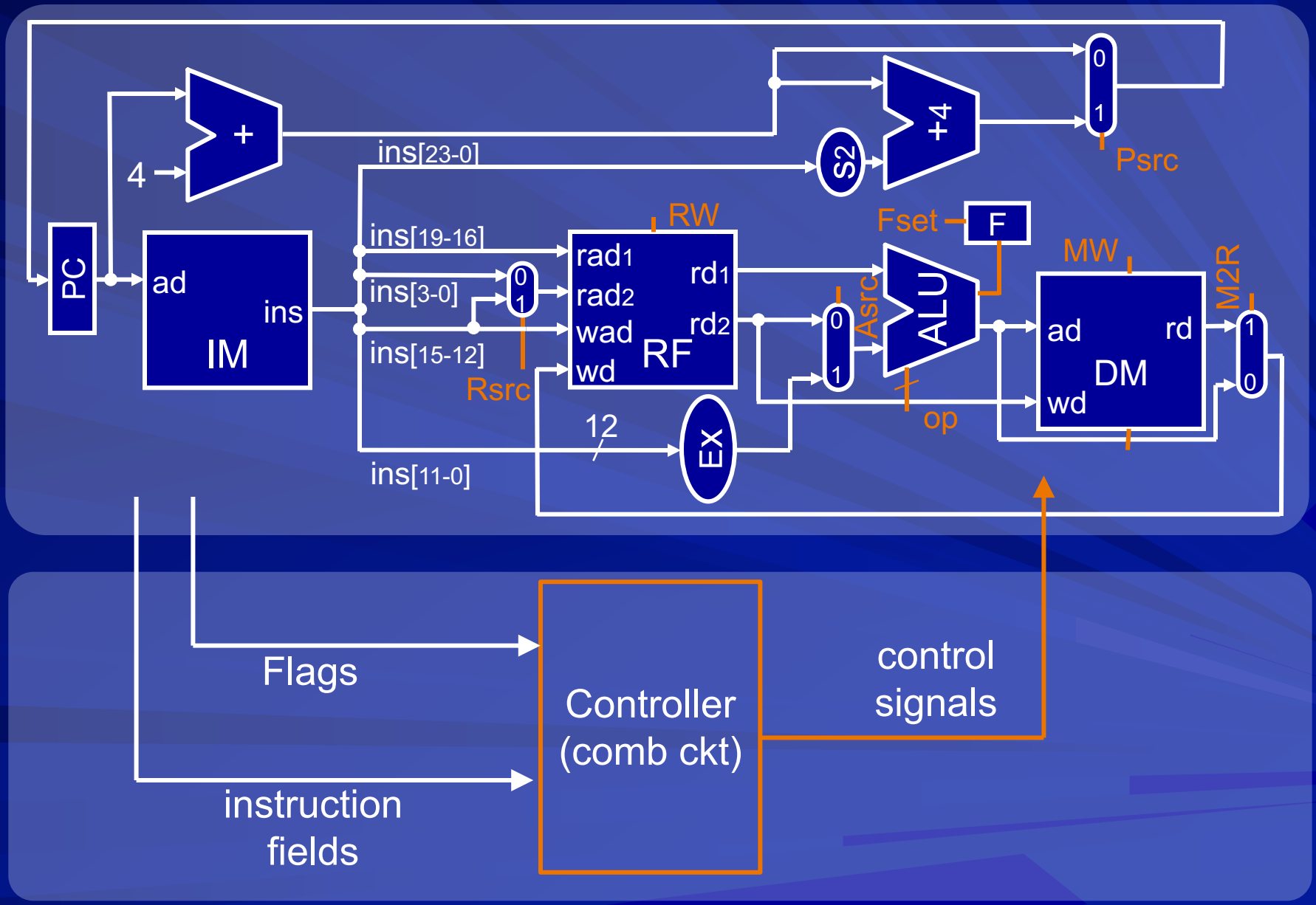


# Thanks

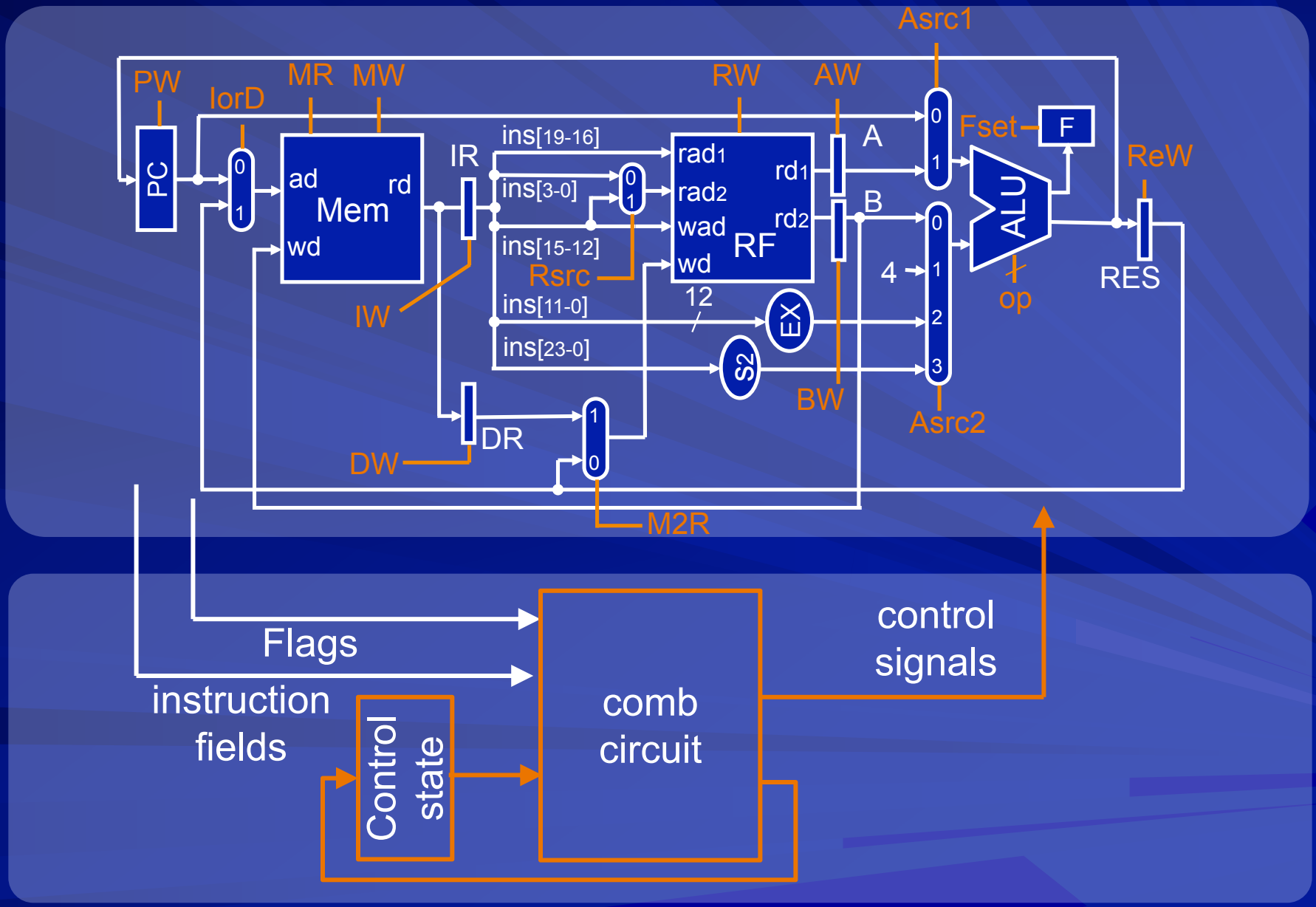
# COL216
# Computer Architecture

Pipelined Processor design –
Controller, Data forwarding
10th February 2022

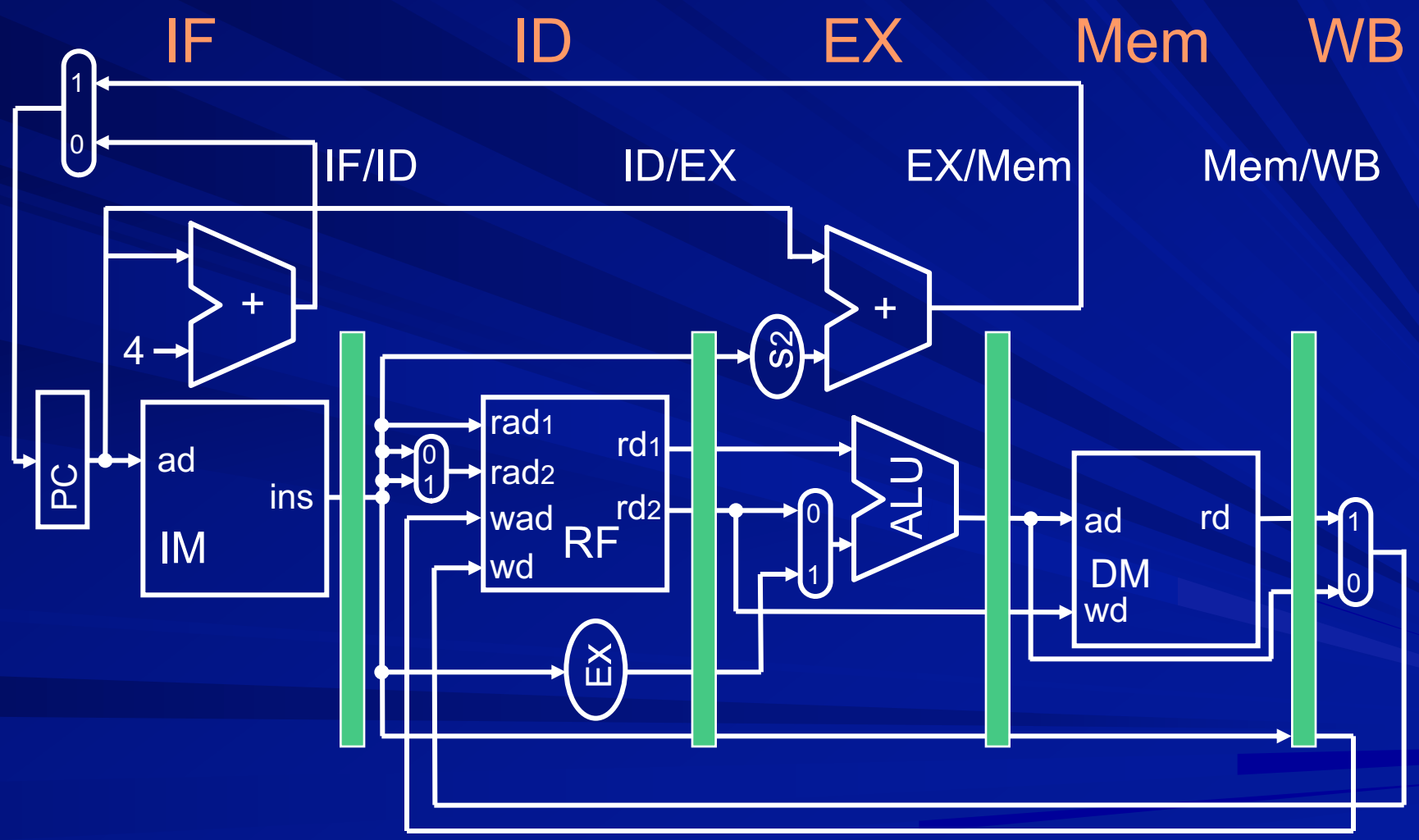
# Single Cycle Datapath + Controller



# Multi Cycle Datapath + Controller



# 5 Stage Pipeline



# Hurdles in instruction pipelining

- Structural hazards
  - Resource conflicts - two instruction require same resource in the same cycle
- Data hazards
  - Data dependencies - one instruction needs data which is yet to be produced by another instruction
- Control Hazards
  - Decision about next instruction needs more cycles

# Handling data hazards

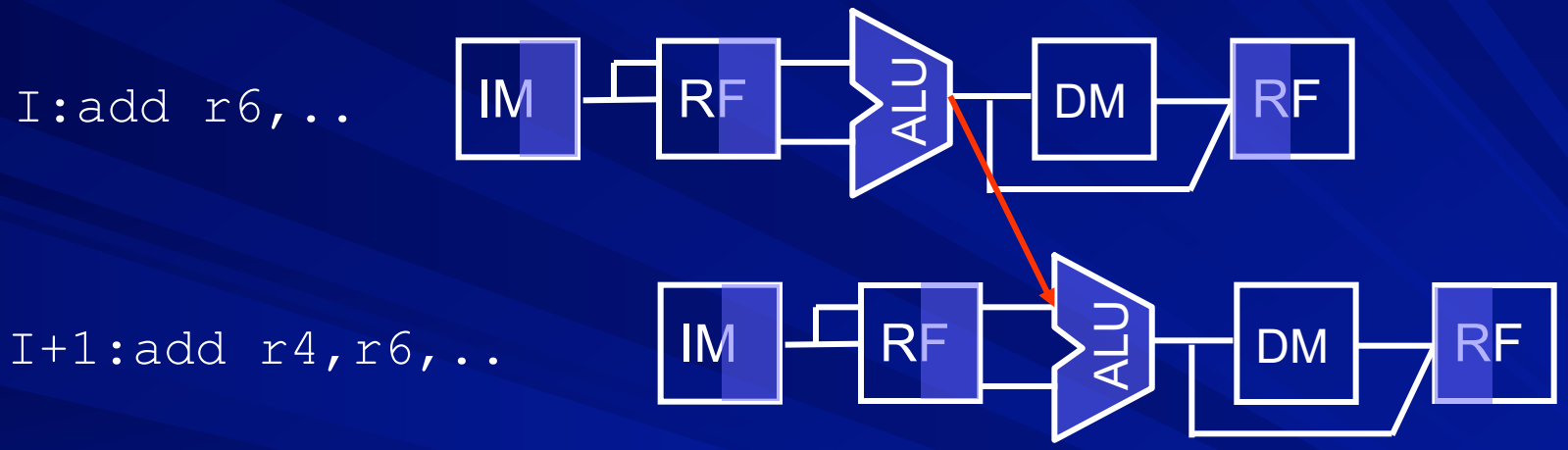
- Assume no data hazards
  - leave it to compiler to remove hazards
- Introduce stalls/bubbles
  - requires hazard detection
    (check data dependence among instructions)
  - compiler may still help in reducing hazards
- Do data forwarding
  - this also requires hazard detection
  - stalls may also be required in some cases

# Detecting data hazard

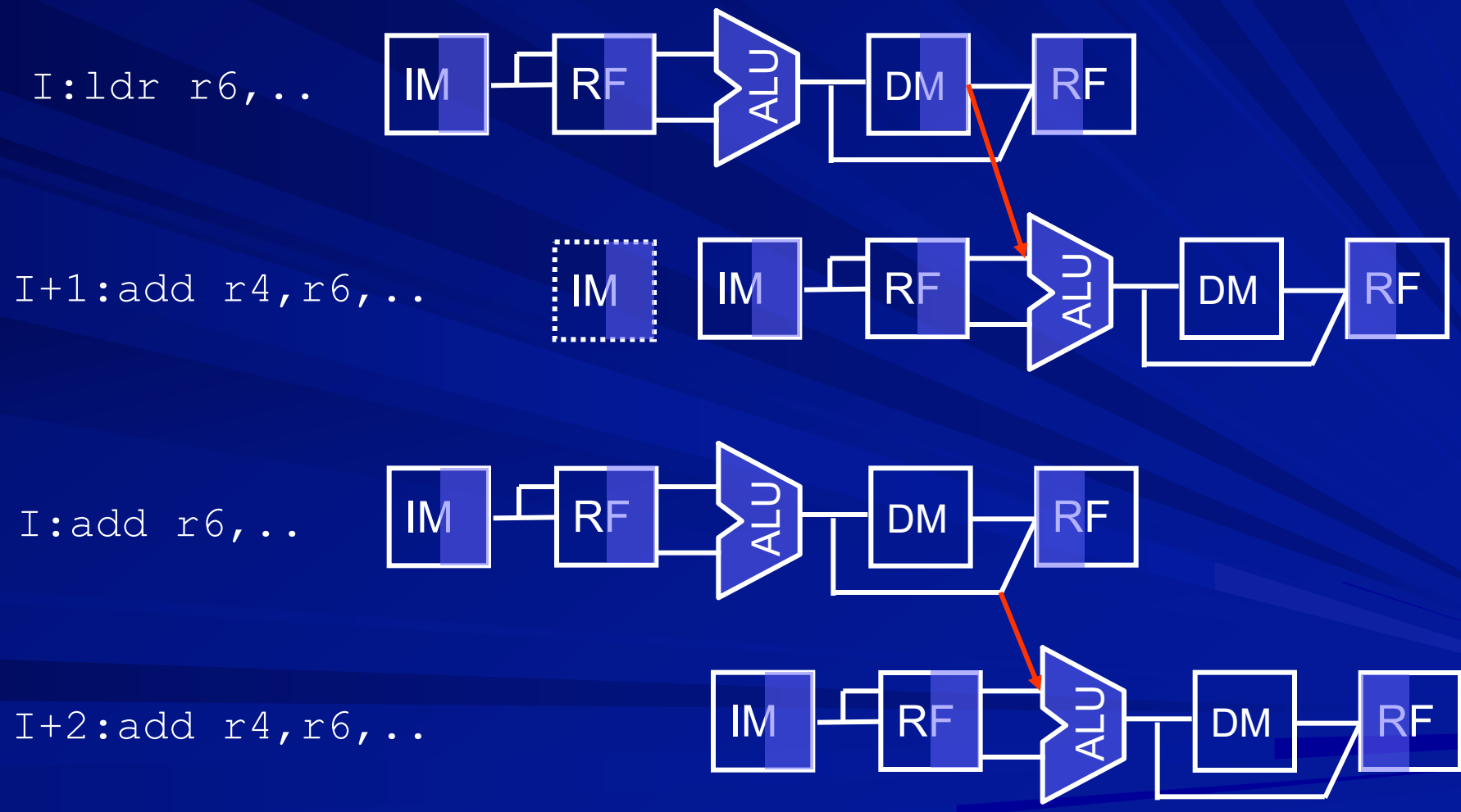
Condition to be checked:

Instruction in RF stage reads from a register in which instruction in ALU stage or DM stage is going to write

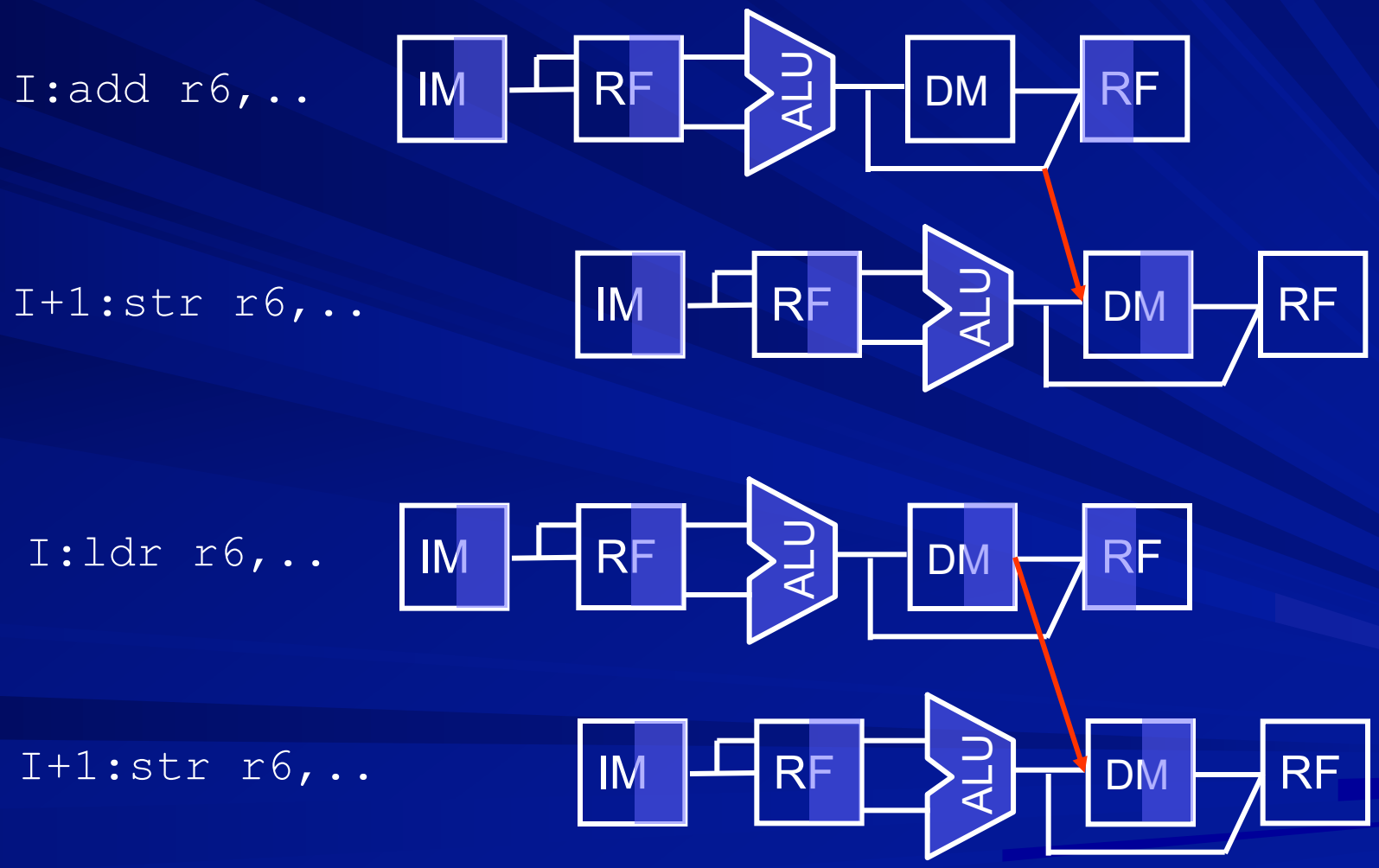
# Data forwarding path P1



# Data forwarding path P2



# Data forwarding path P3



# Data forwarding path list

- P1
  from ALU out to ALU in1/2
  (EX/DM register)
- P2
  from DM/ALU out to ALU in1/2
  (DM/WB register)
- P3
  from DM/ALU out to DM in
  (DM/WB register)

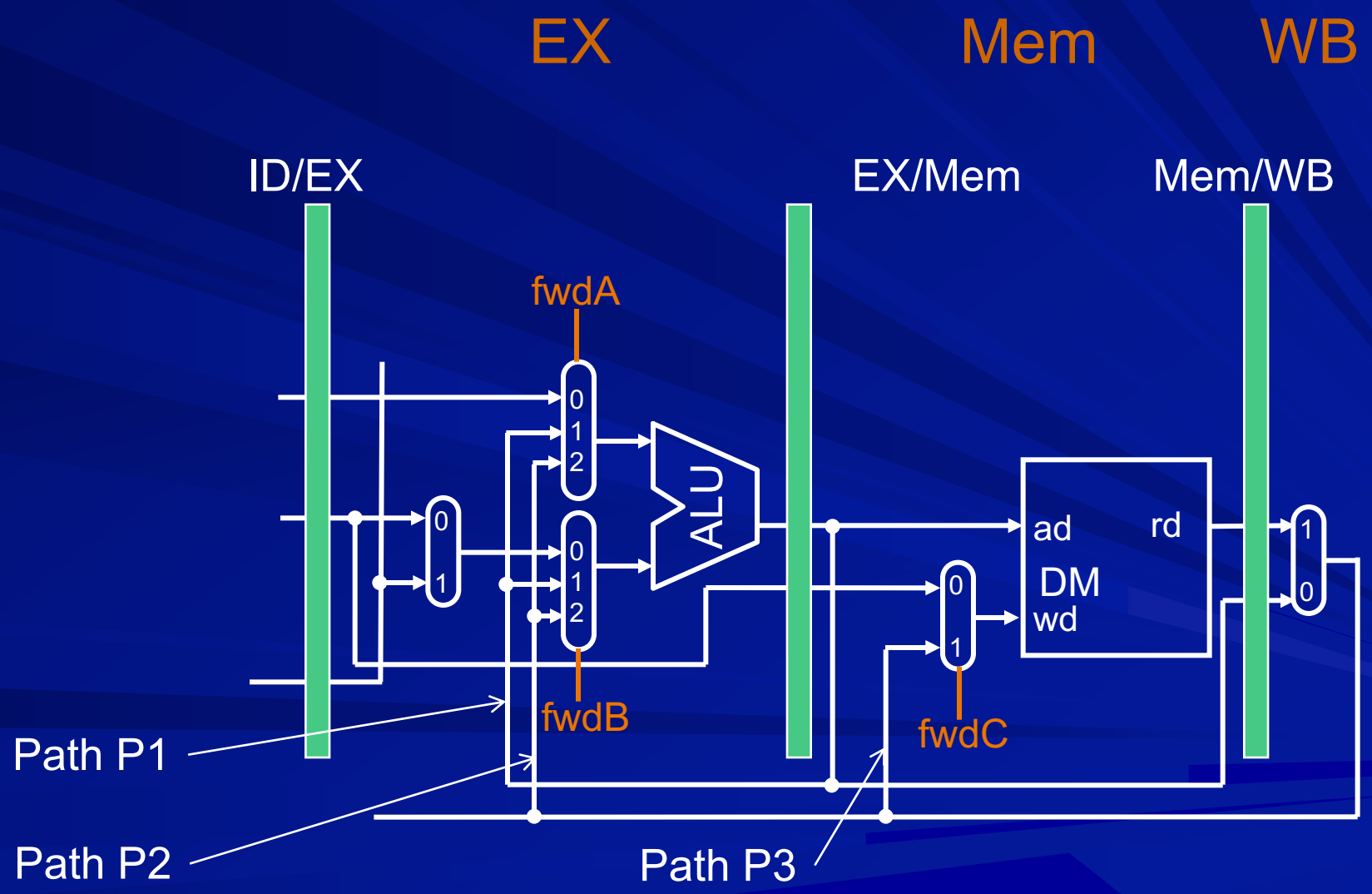
# Dependence check logic

Condition to be checked:

Operand of instruction in RF stage is a register in which instruction in ALU stage or DM stage is going to write

We need to ensure that instruction in RF stage actually reads Rn and/or Rm

# Data forwarding paths



# Executing branch instructions

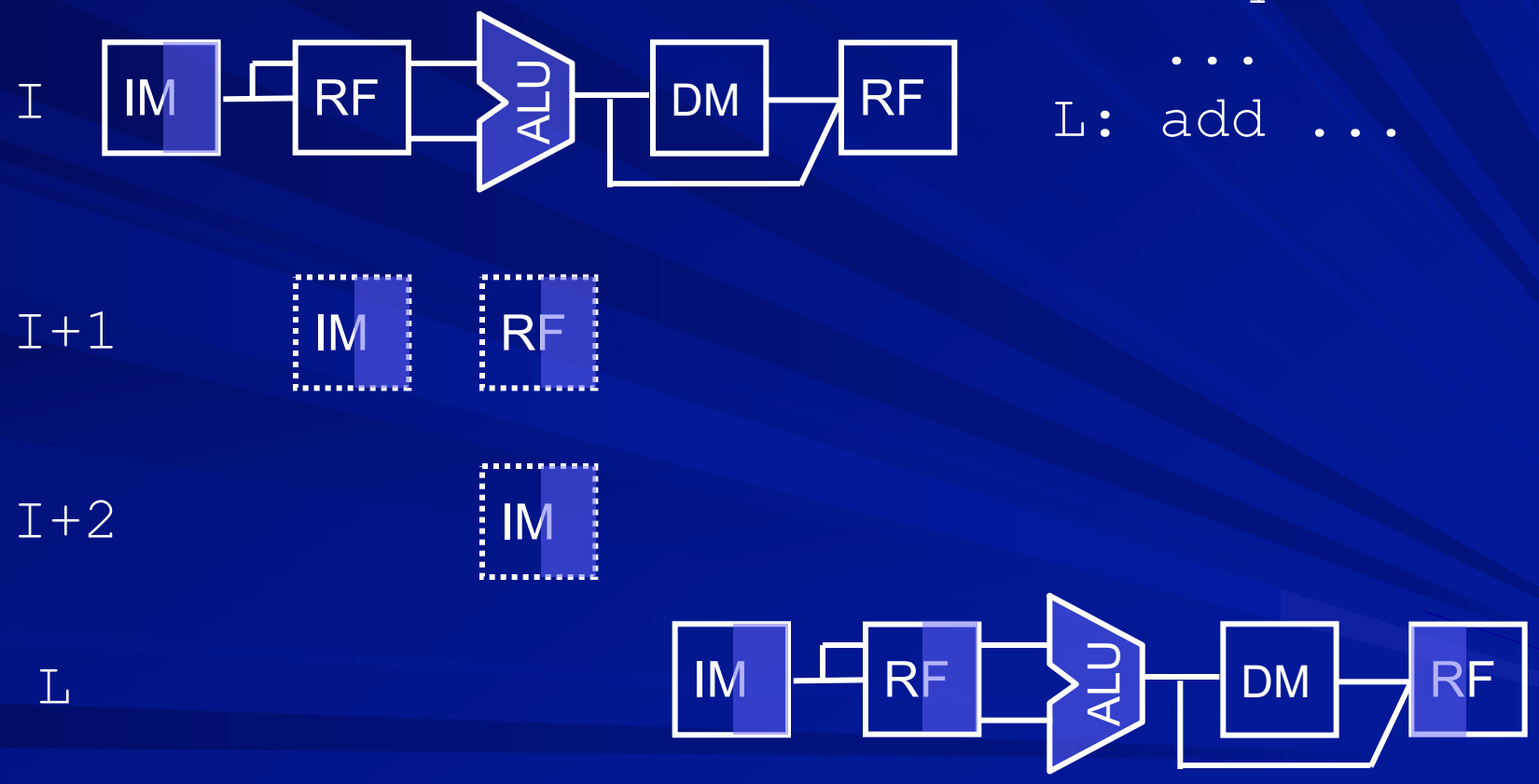
- In which cycle the instruction is found to be a branch instruction?
- In which cycle the branch decision is known?
- In which cycle the target address is computed?

# On decoding a branch instruction

- Flush the inline instructions
- Freeze (stall) the inline instructions
- Allow the inline instructions to continue

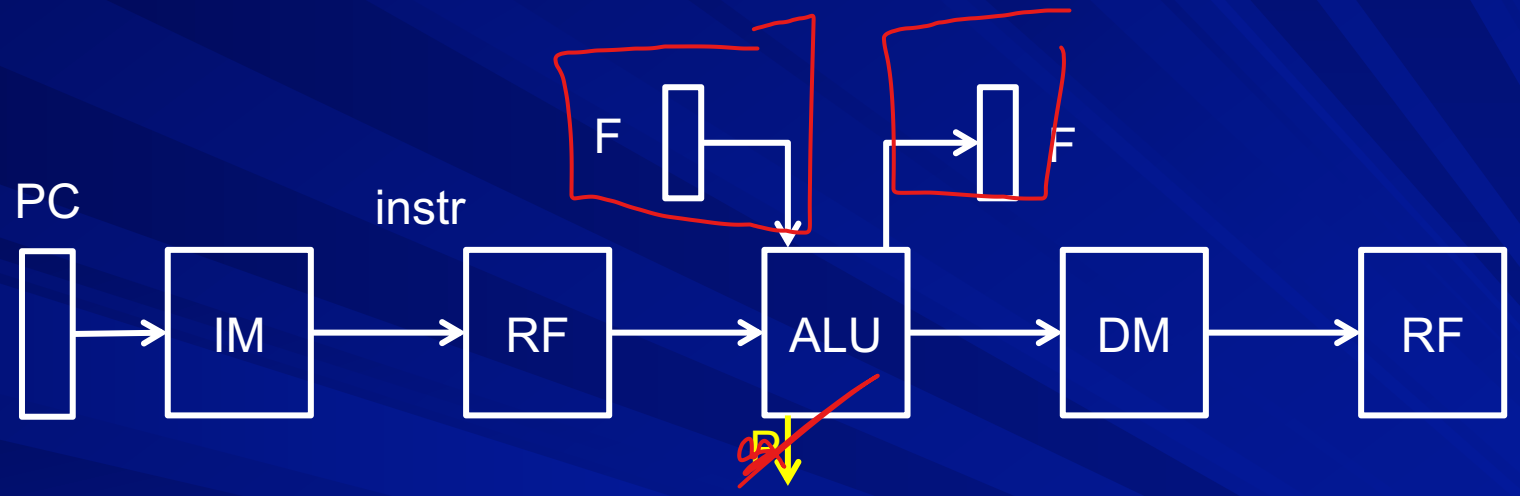
# Stalls due to control hazards

```
I: beq L
   ...
L: add ...
```



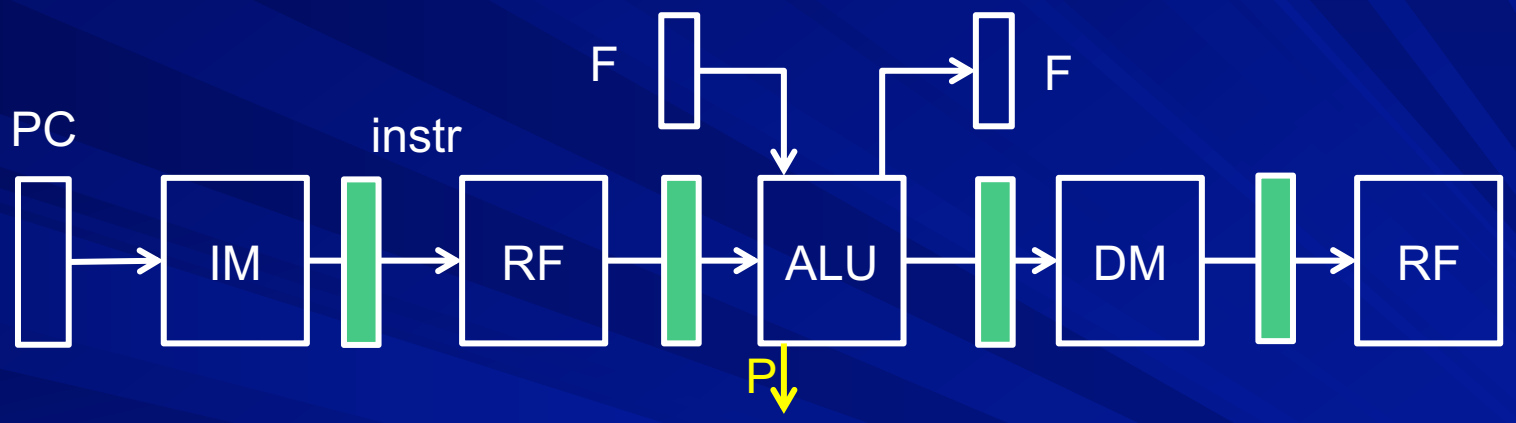
# Controller design

# Single cycle datapath



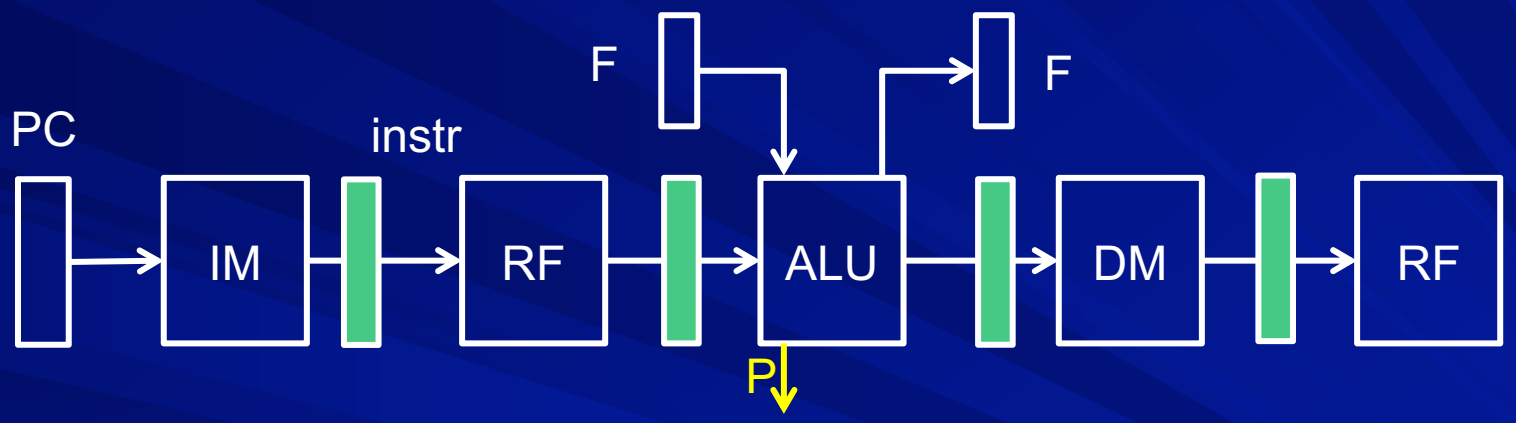
# Multi-cycle datapath

Resource sharing possible across cycles

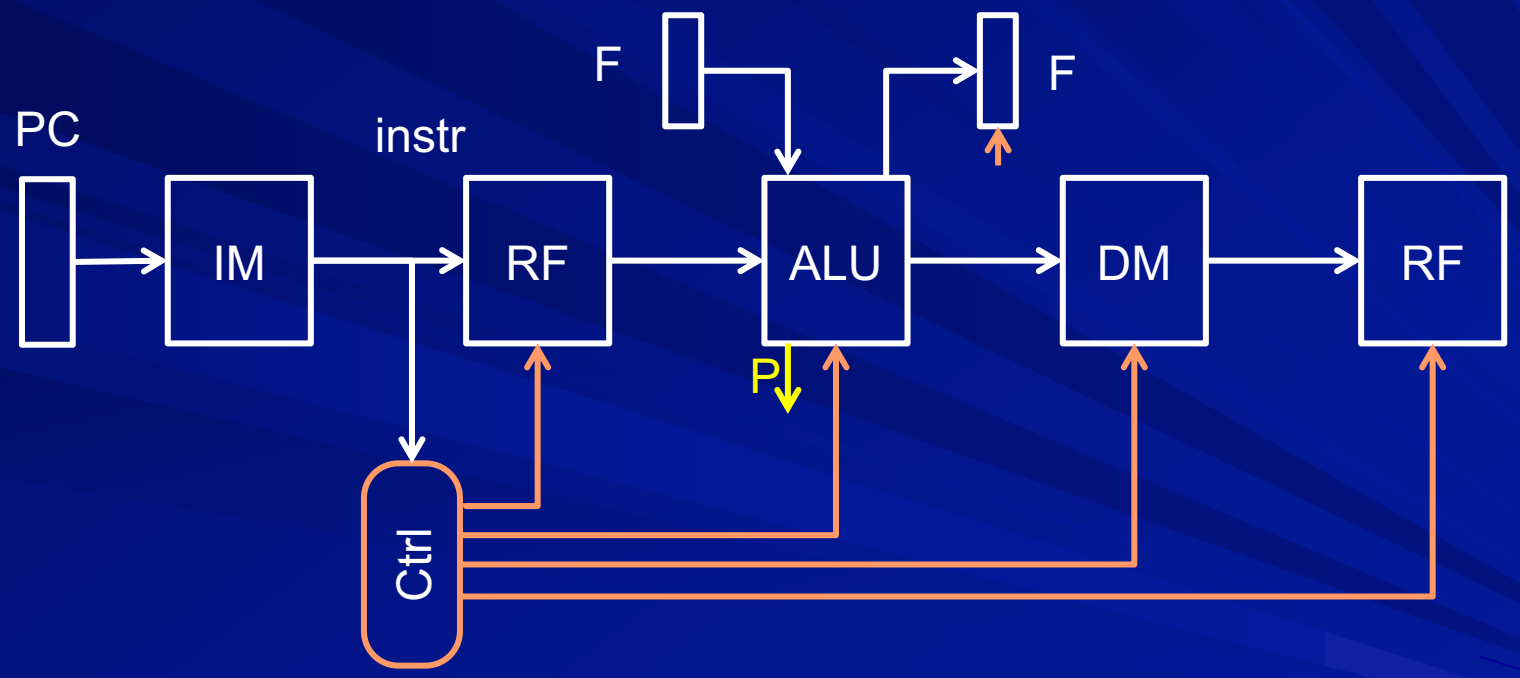


# Pipelined datapath

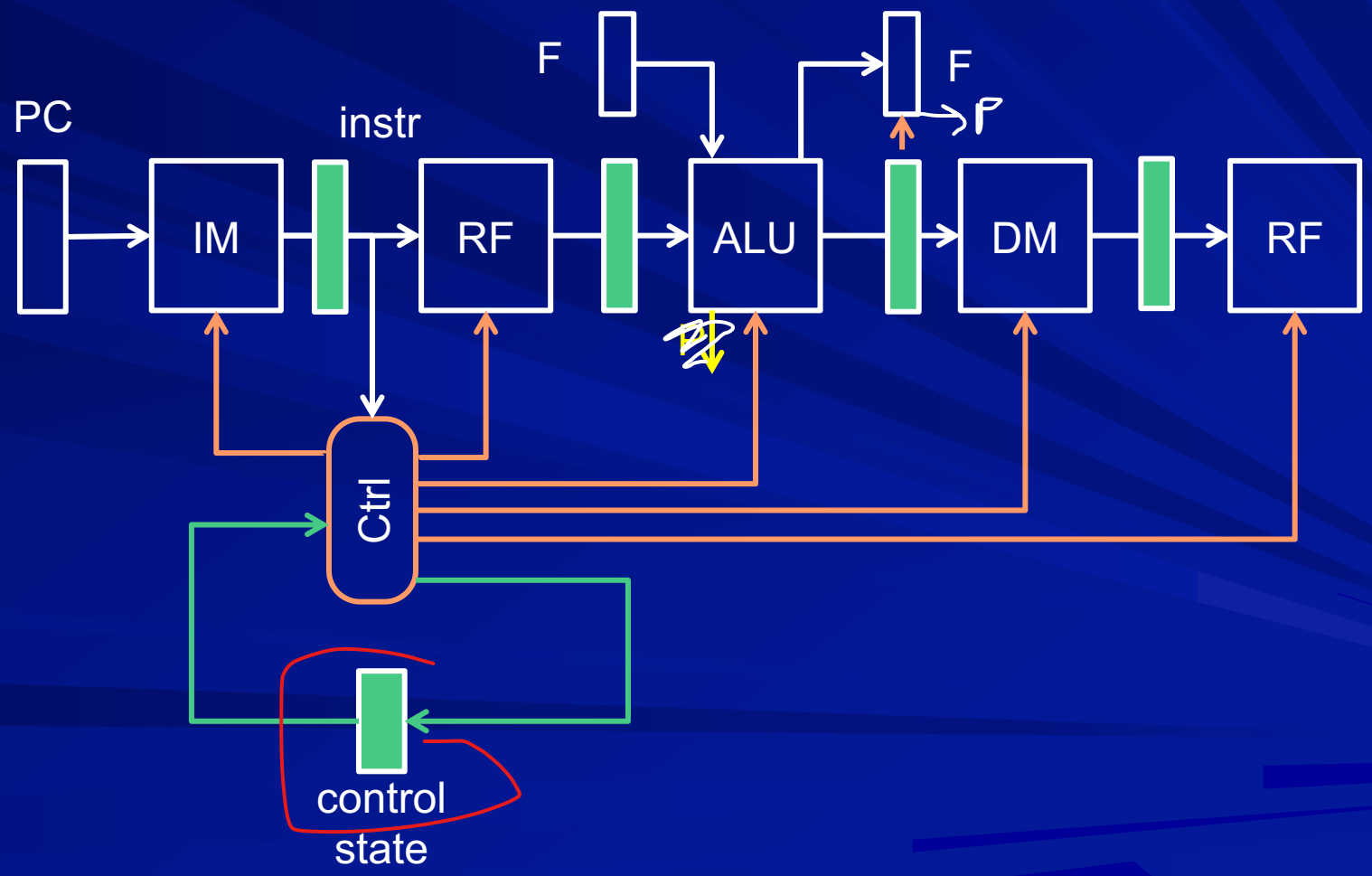
Resource sharing leads to hazards



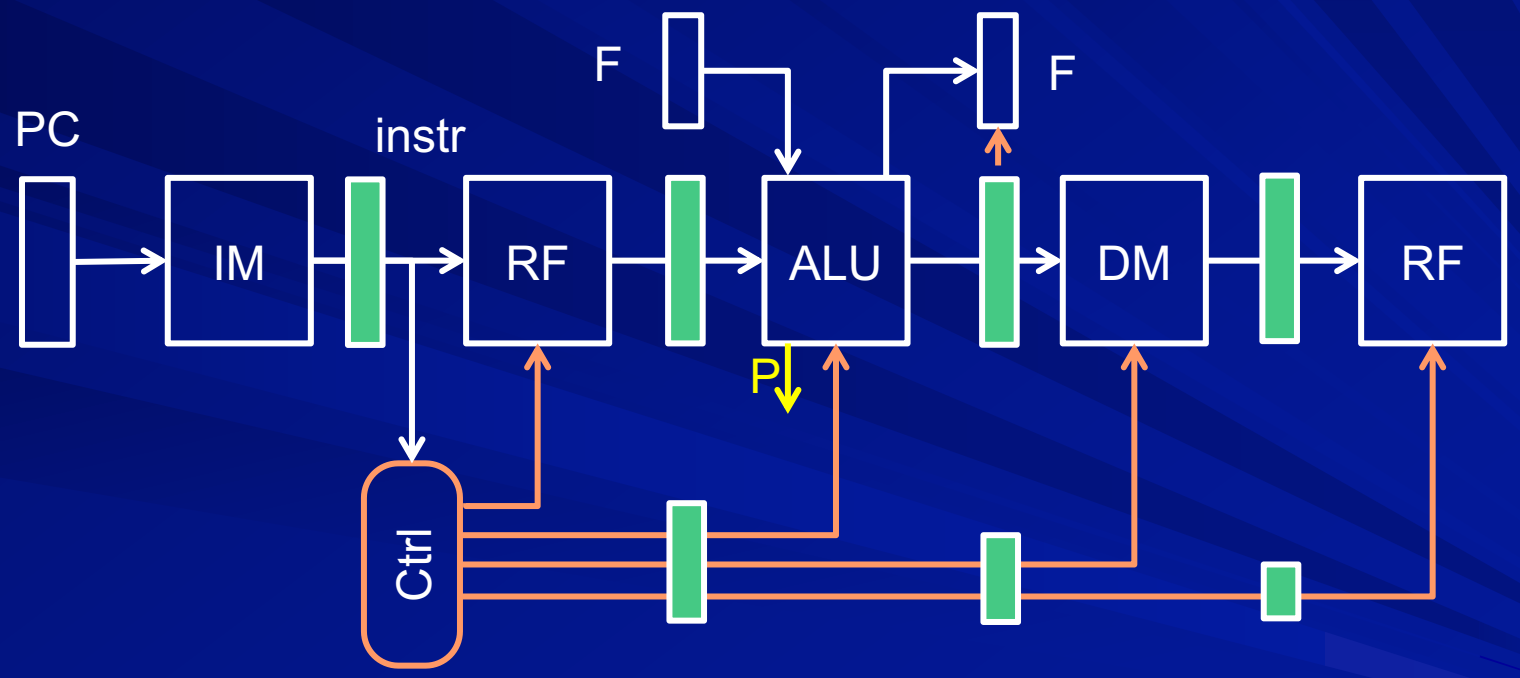
# Controller for single cycle DP



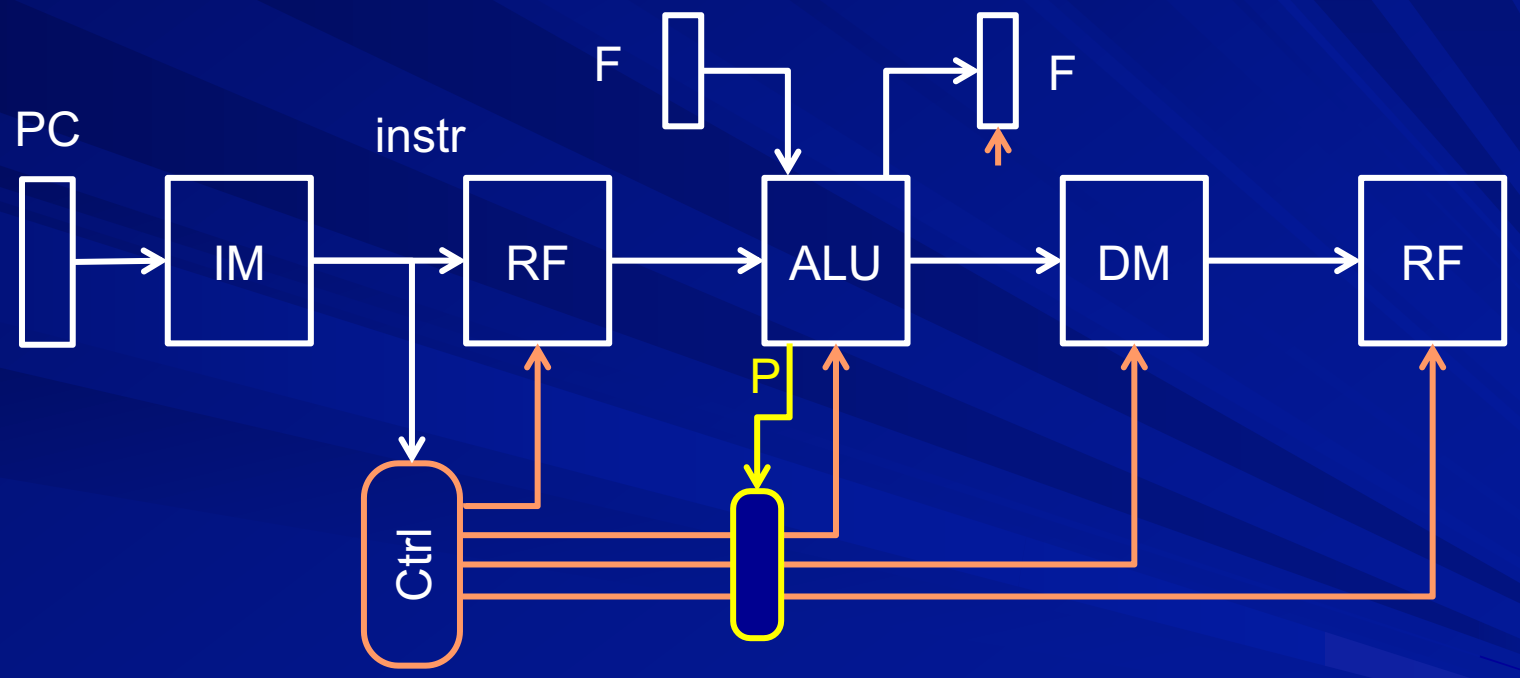
# Controller for multi-cycle DP



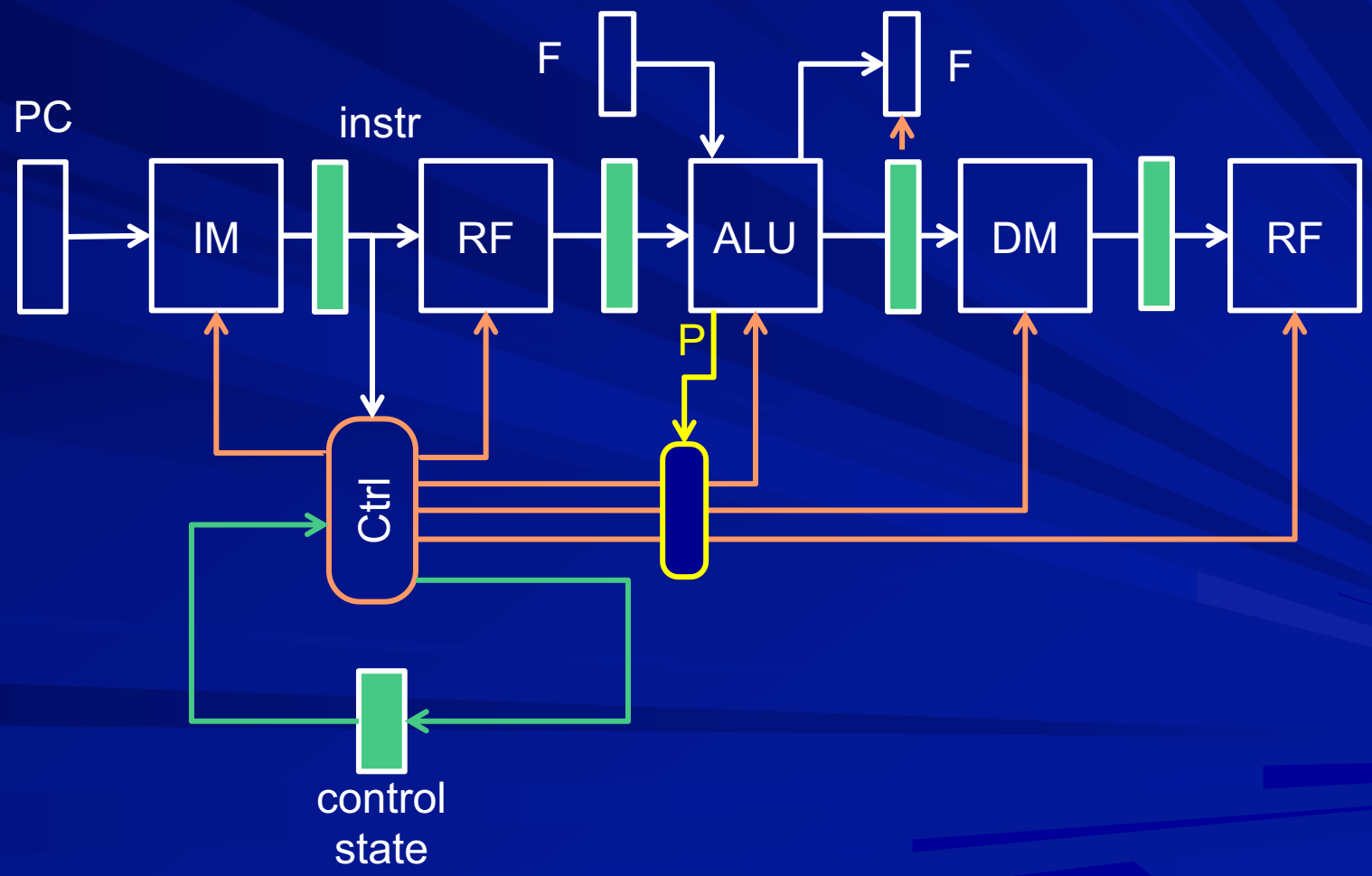
# Controller for pipelined DP



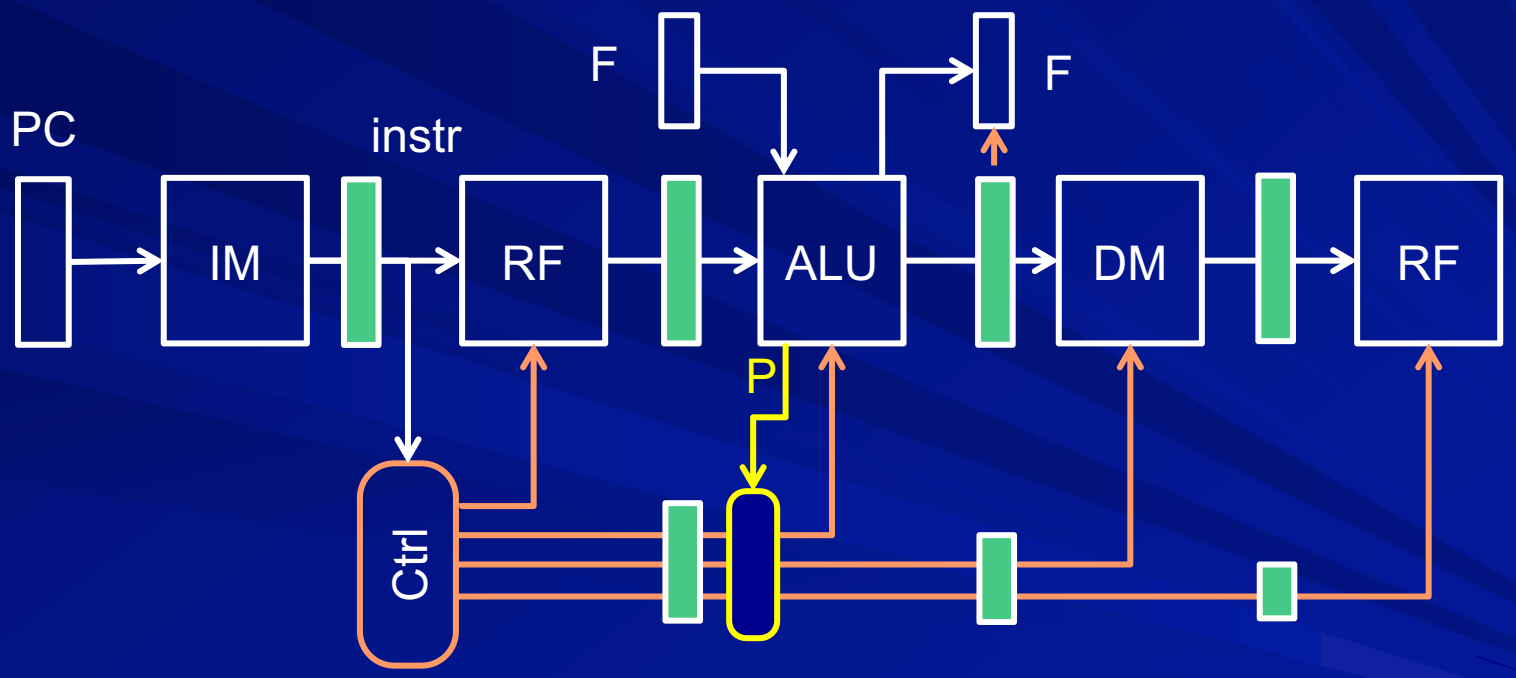
# Controller for single cycle DP



# Controller for multi-cycle DP



# Controller for pipelined DP

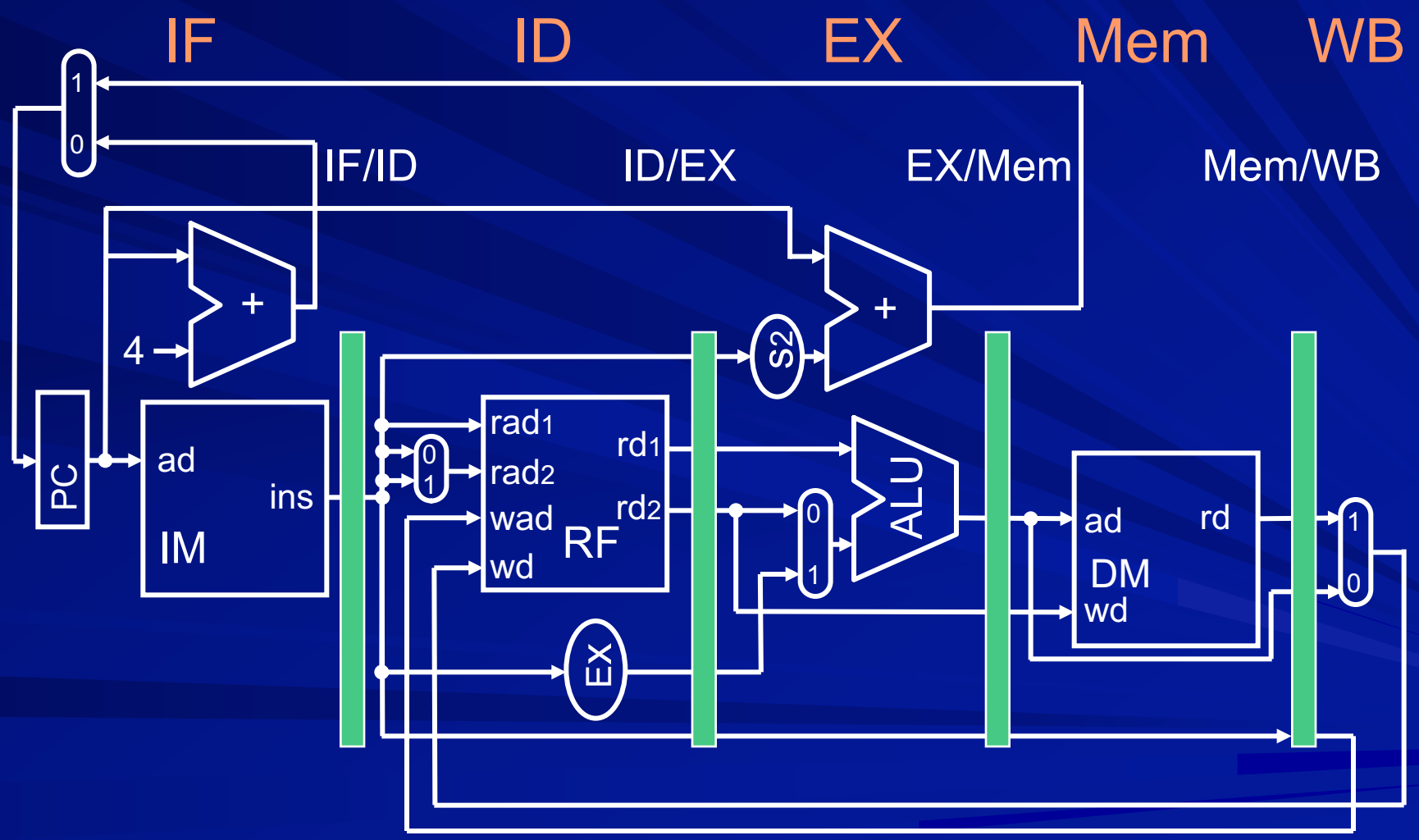


# Thanks

# COL216
# Computer Architecture

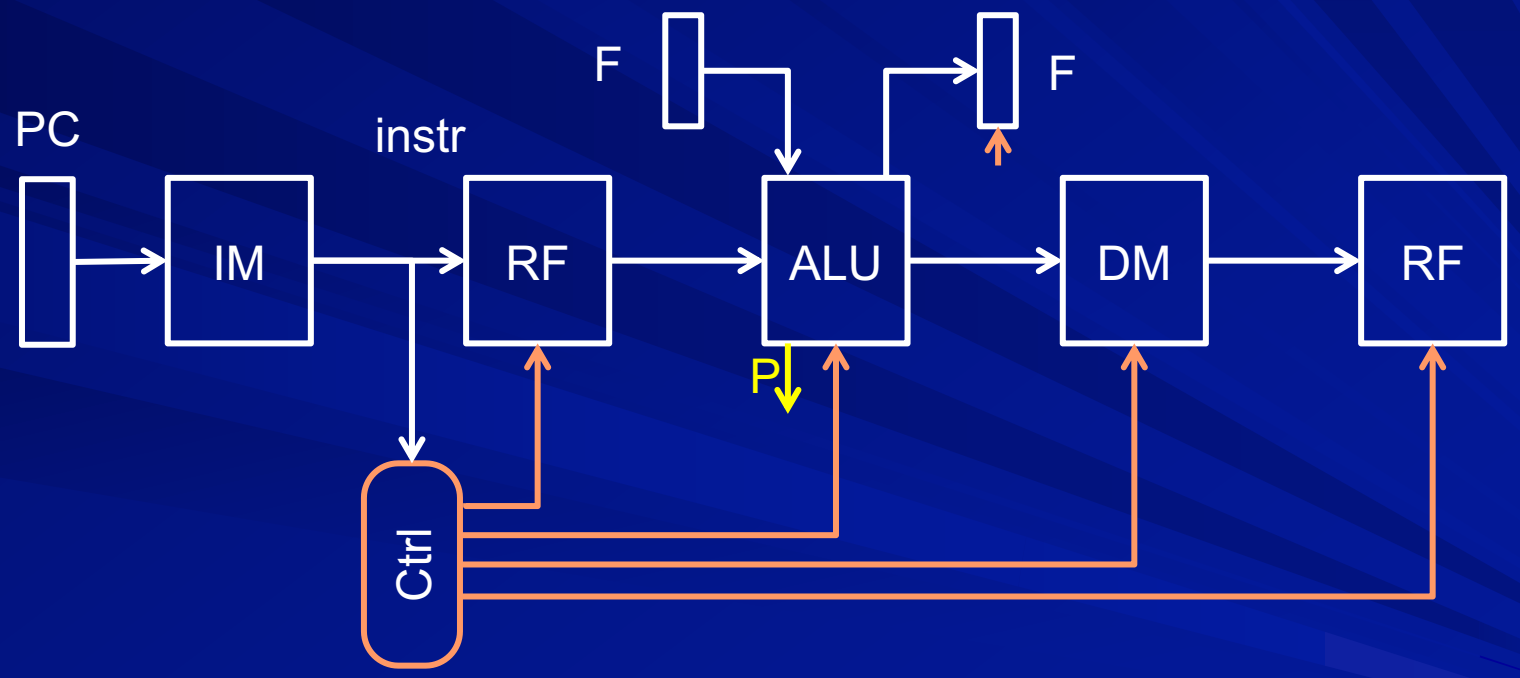
Pipelined Processor design –
Controller, Data forwarding
21st February 2022

# 5 Stage Pipeline

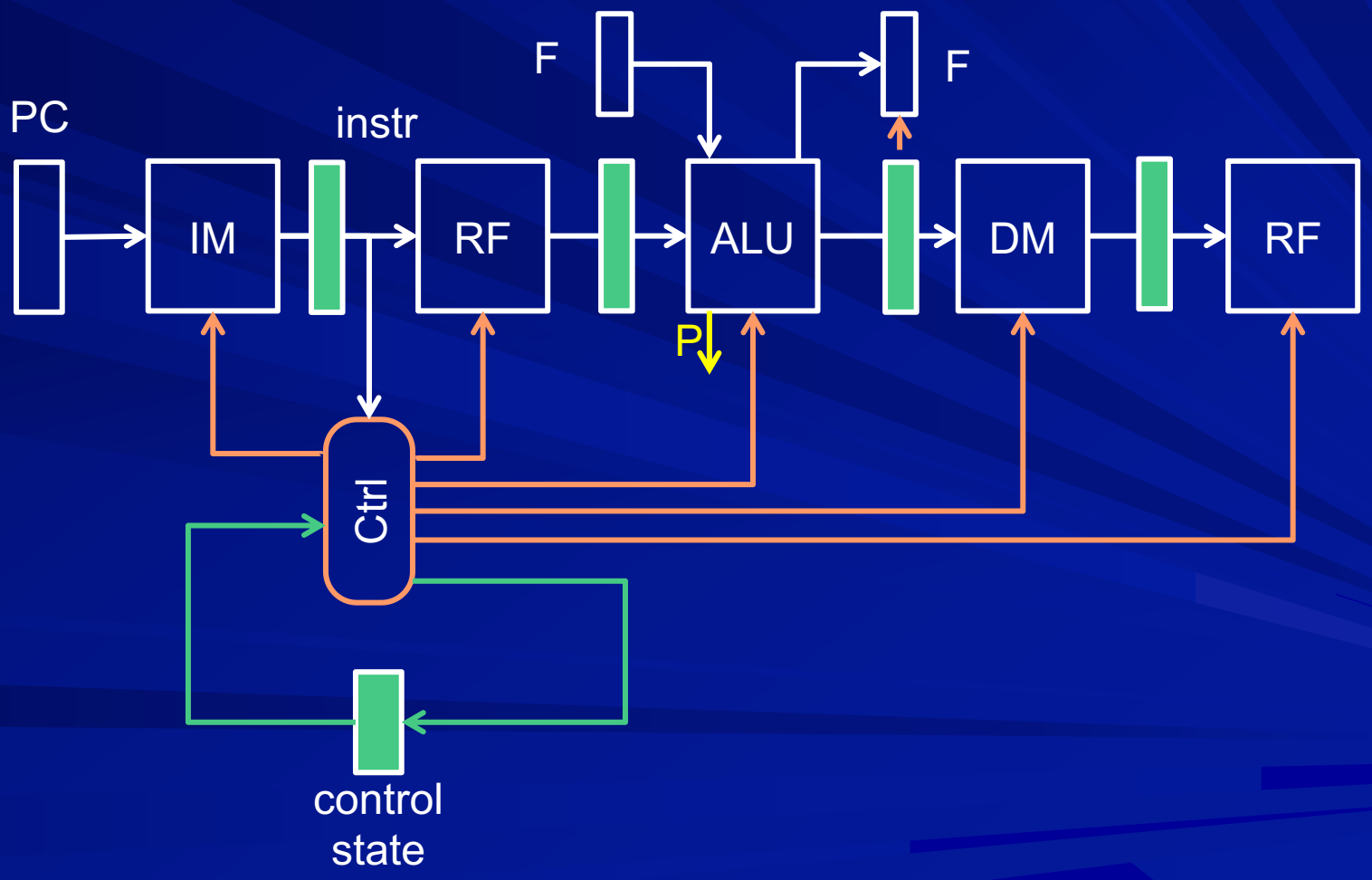


# Controllers for different design styles

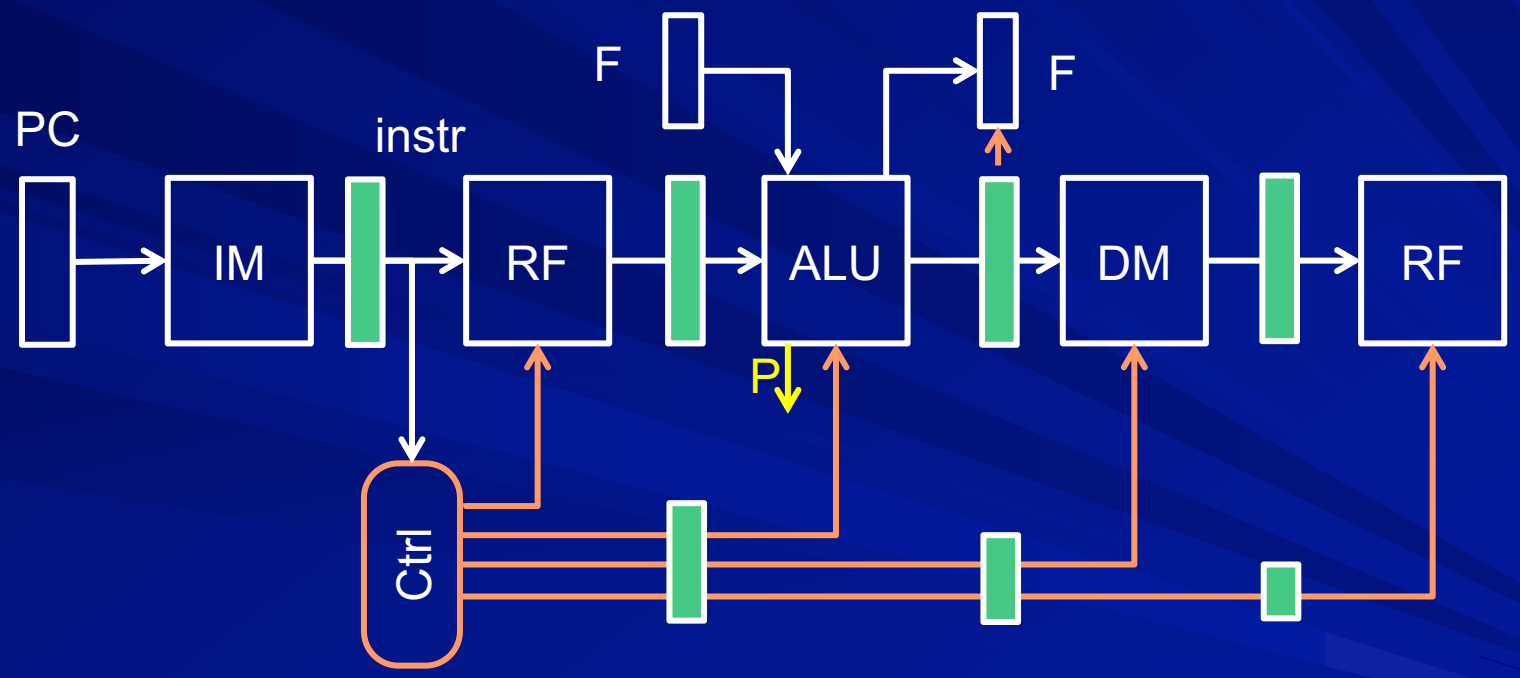
# Controller for single cycle DP



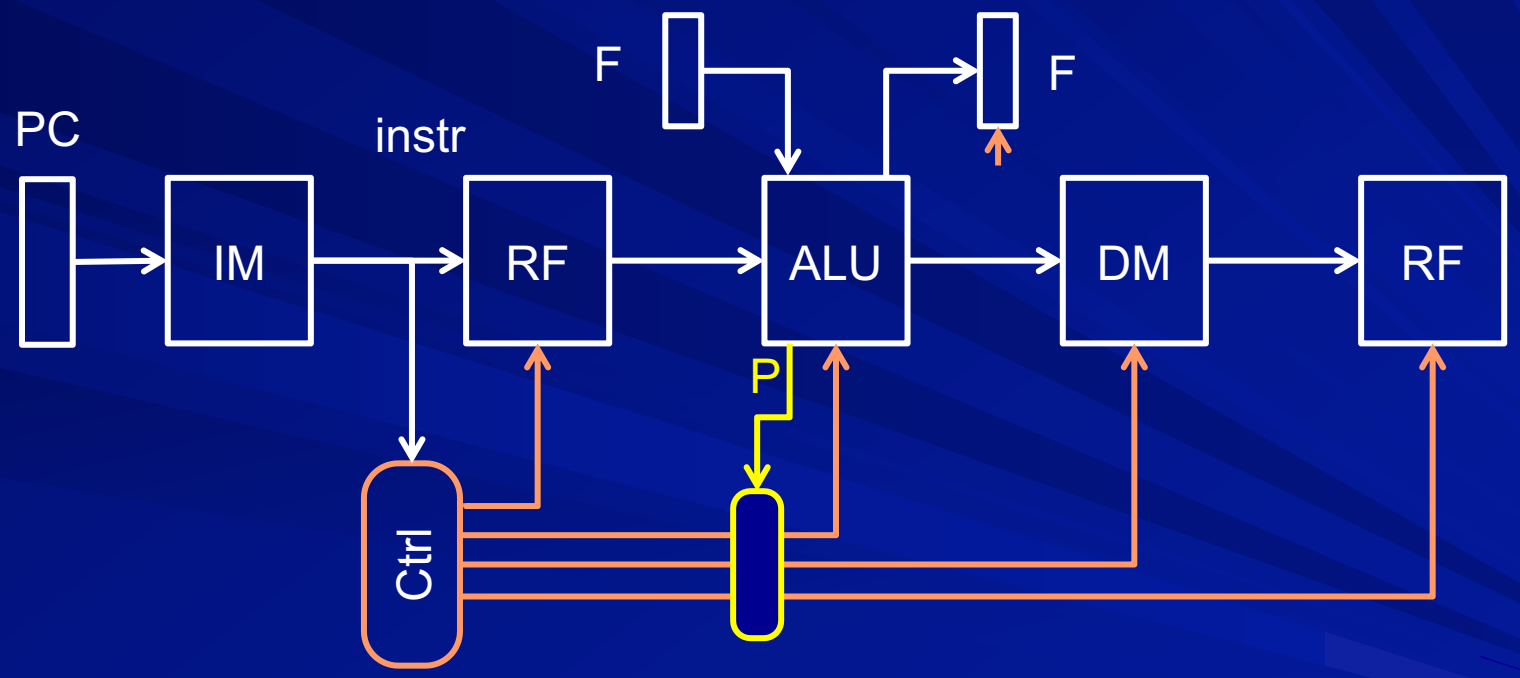
# Controller for multi-cycle DP



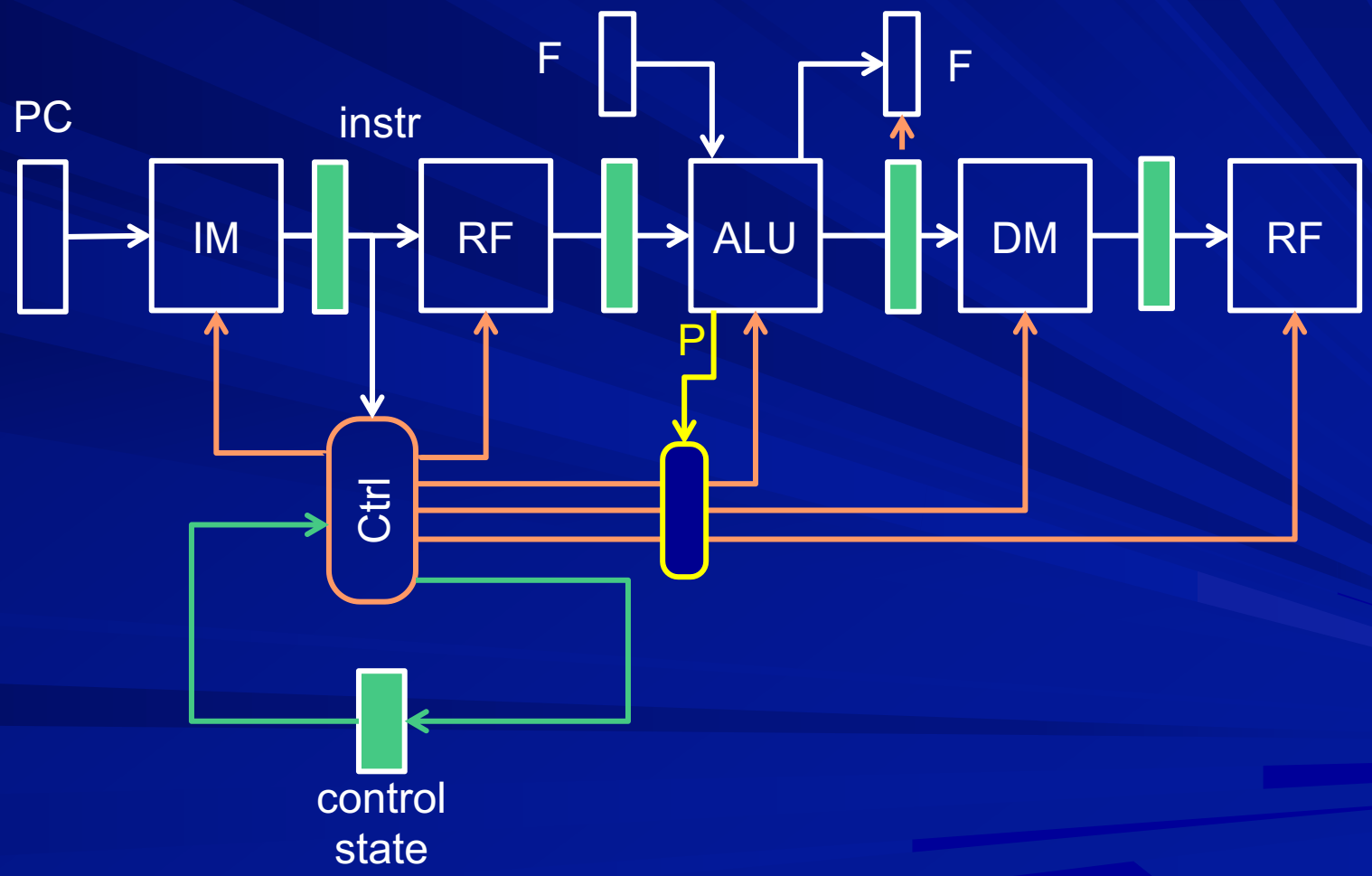
# Controller for pipelined DP



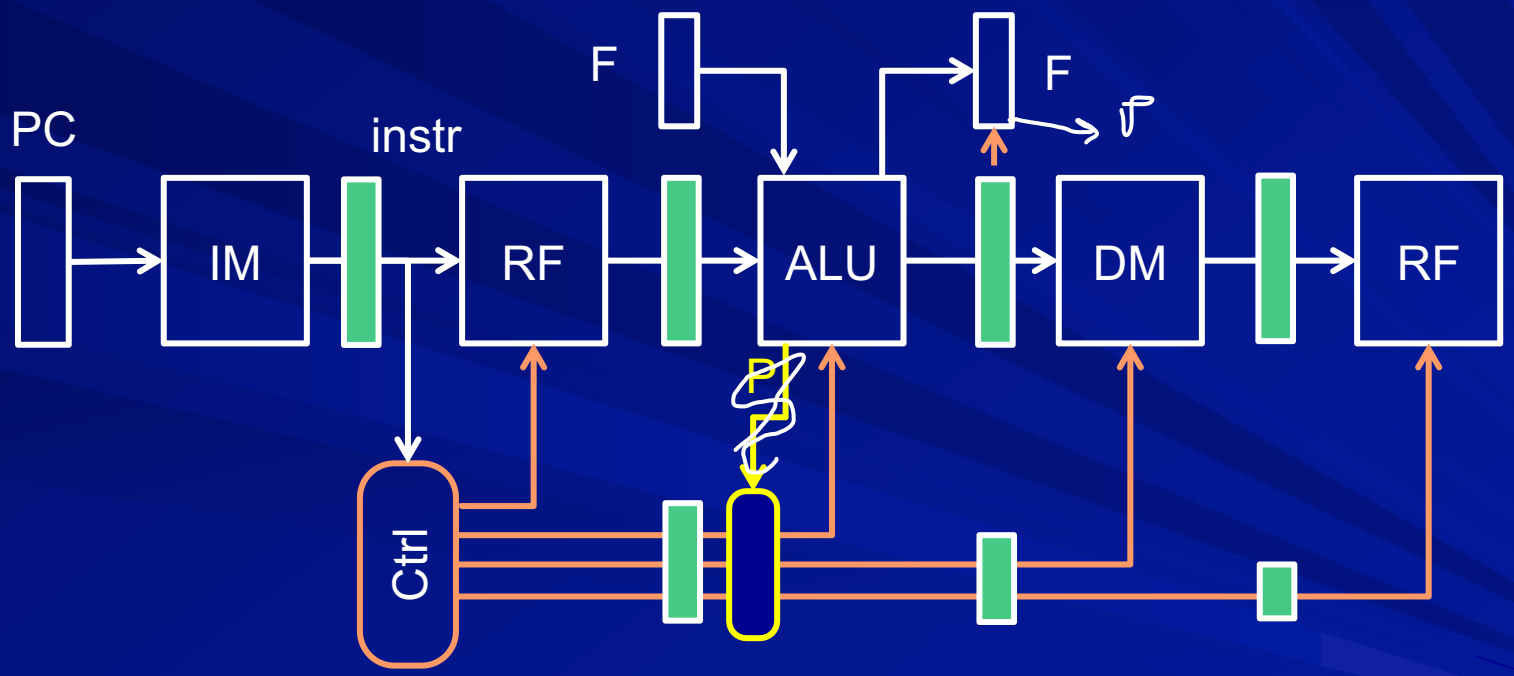
# Controller for single cycle DP



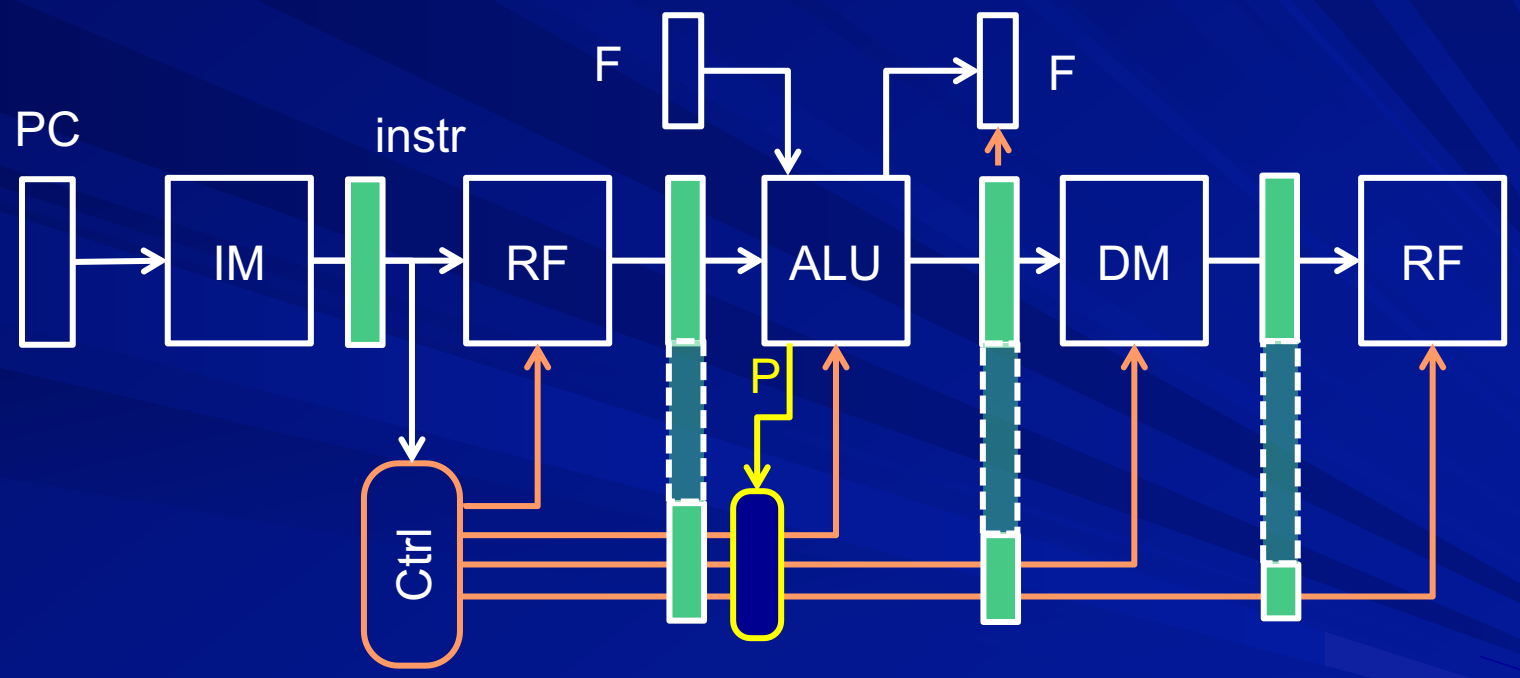
# Controller for multi-cycle DP



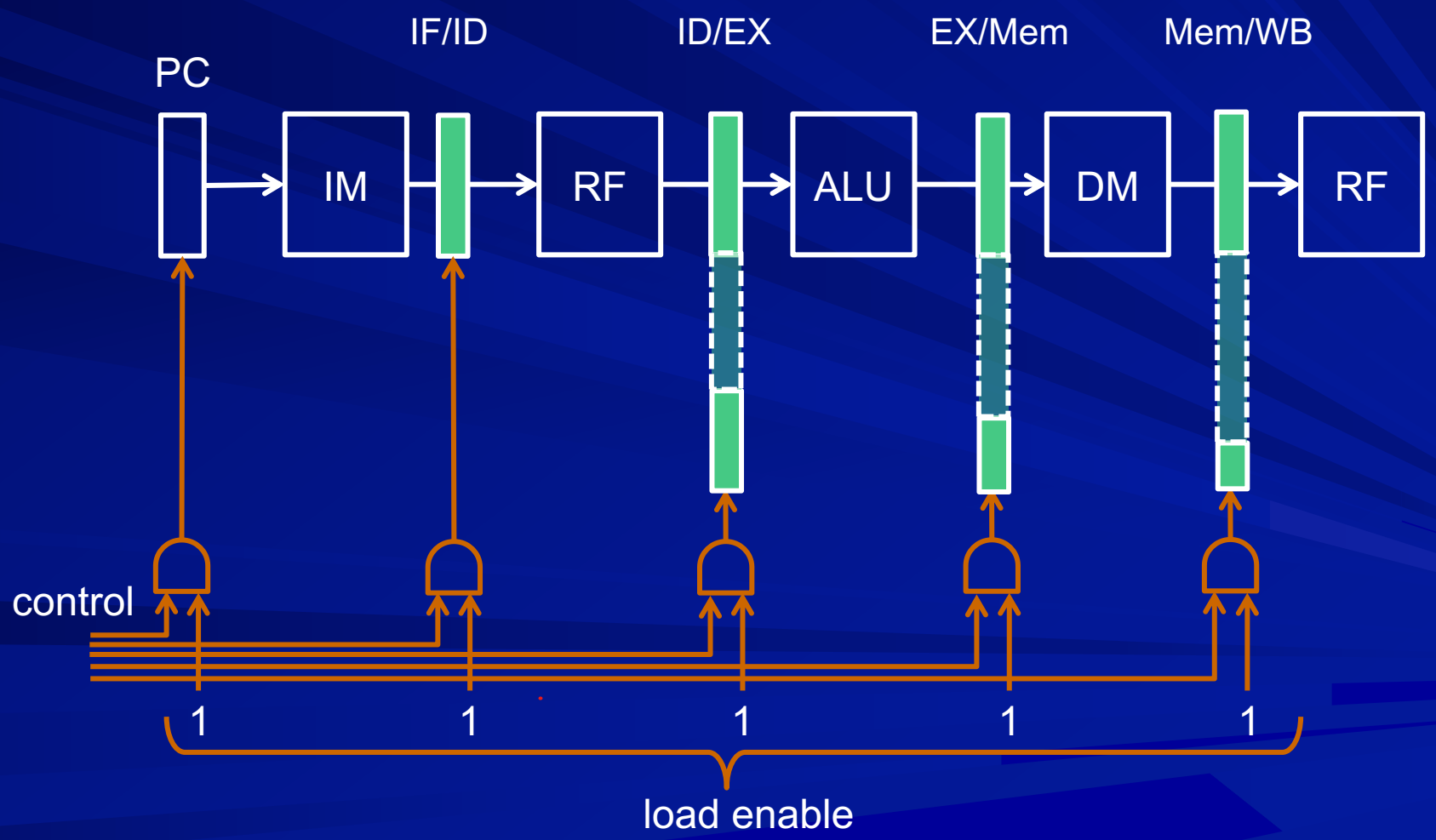
# Controller for pipelined DP



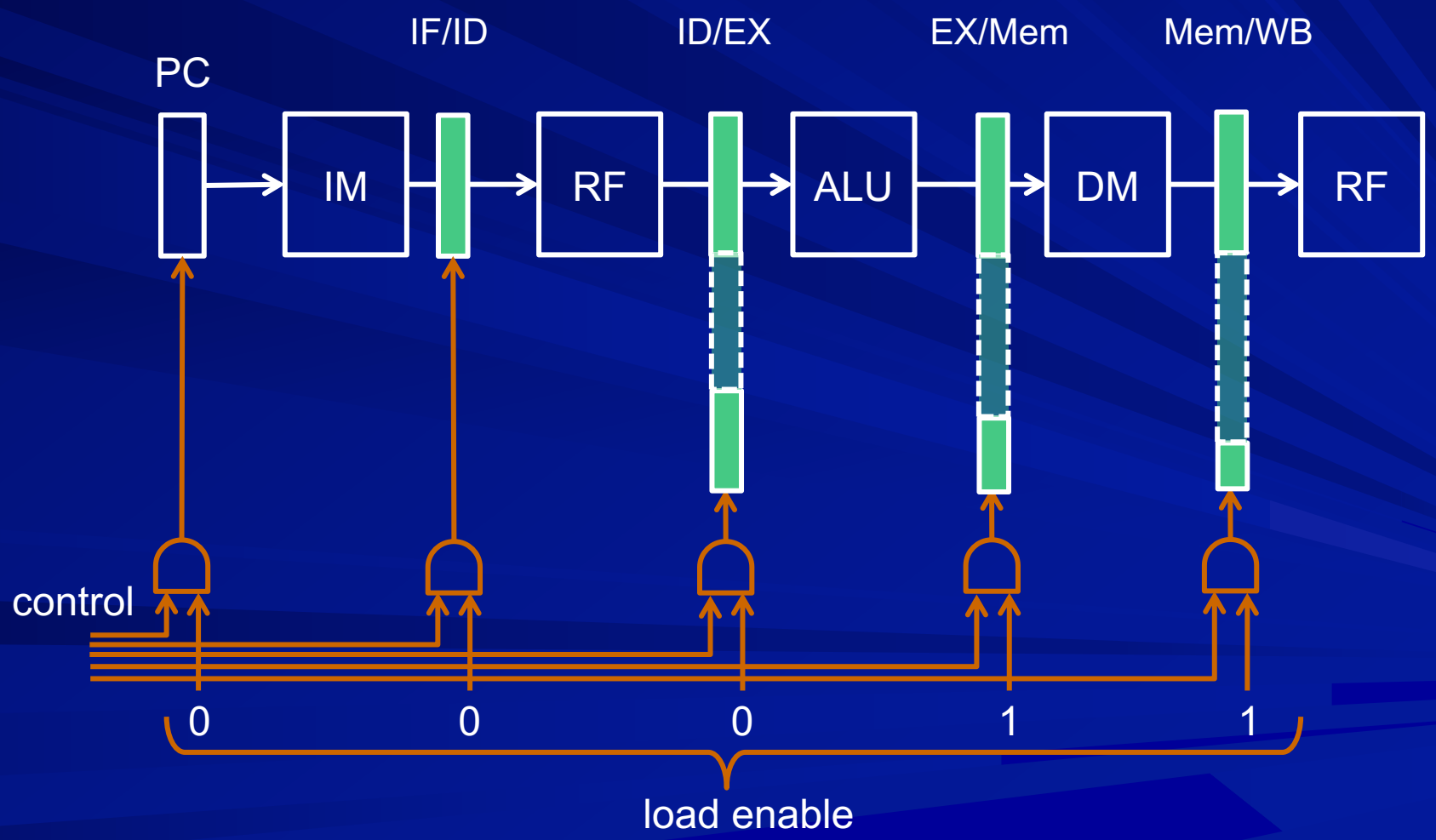
# Extending inter-stage registers



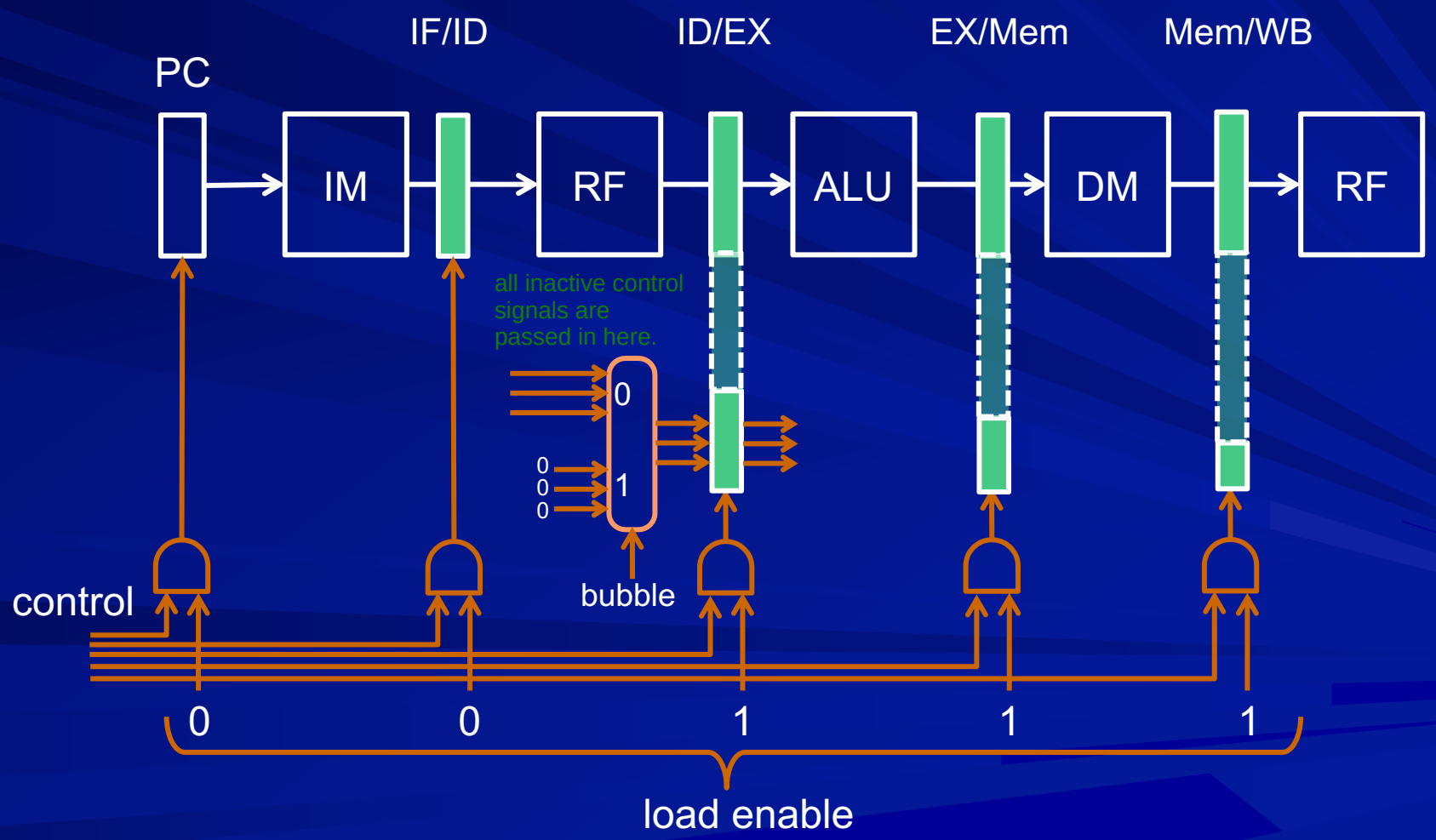
# Controlling inter-stage registers



# Stalling instructions



# Inserting bubbles



# Dependence check logic

Condition to be checked:

Operand of instruction in RF stage is a register in which instruction in ALU stage or DM stage is going to write

ID/EX.RW = 1 and

also depends on whether instruction has only one operand like mov or not

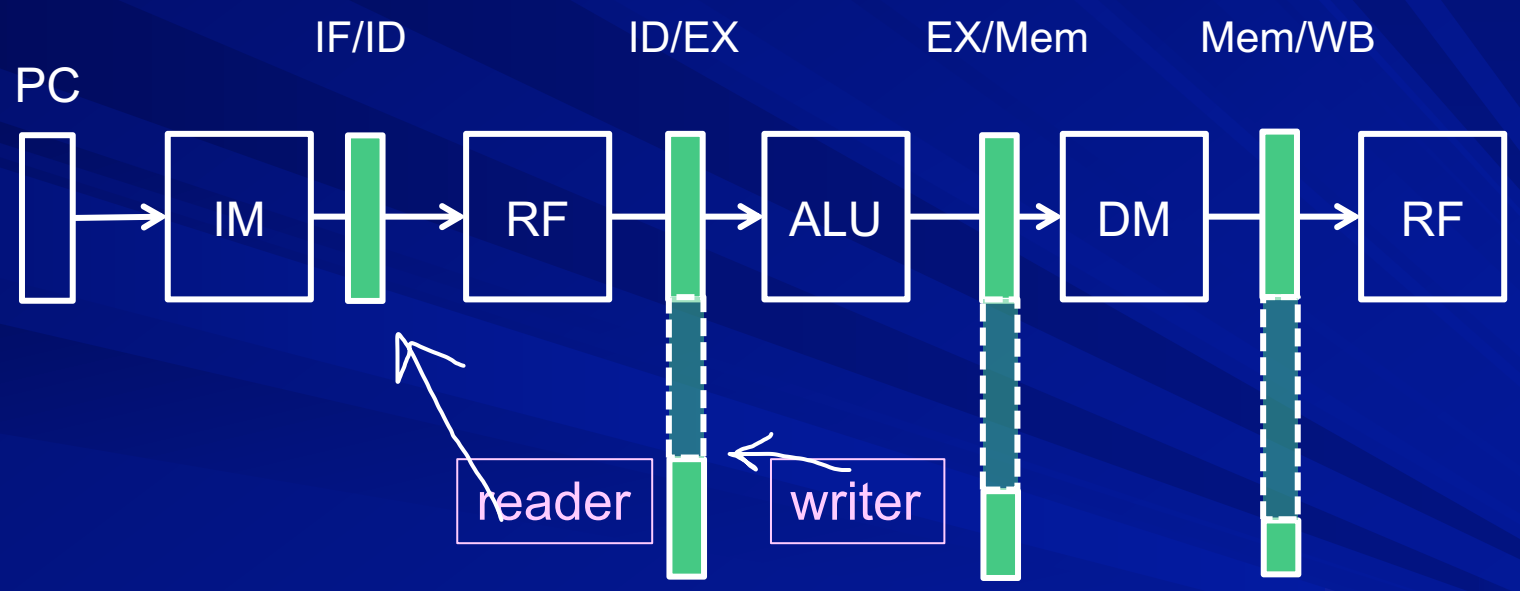
(IF/ID.Rn=ID/EX.Rd or IF/ID.Rm=ID/EX.Rd)

EX/Mem.RW = 1 and

(IF/ID.Rn=EX/Mem.Rd or IF/ID.Rm=EX/Mem.Rd)

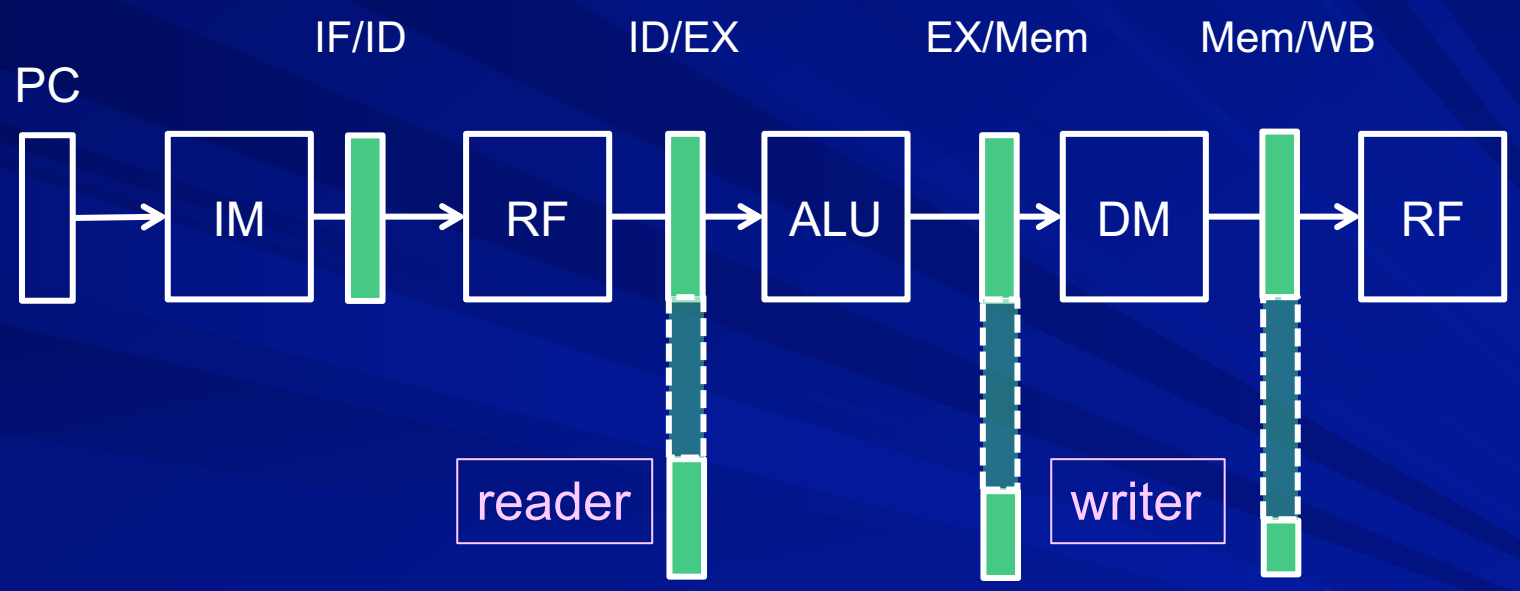
We need to ensure that instruction in RF stage actually reads Rn and/or Rm (not taken care here)

# Dependence check



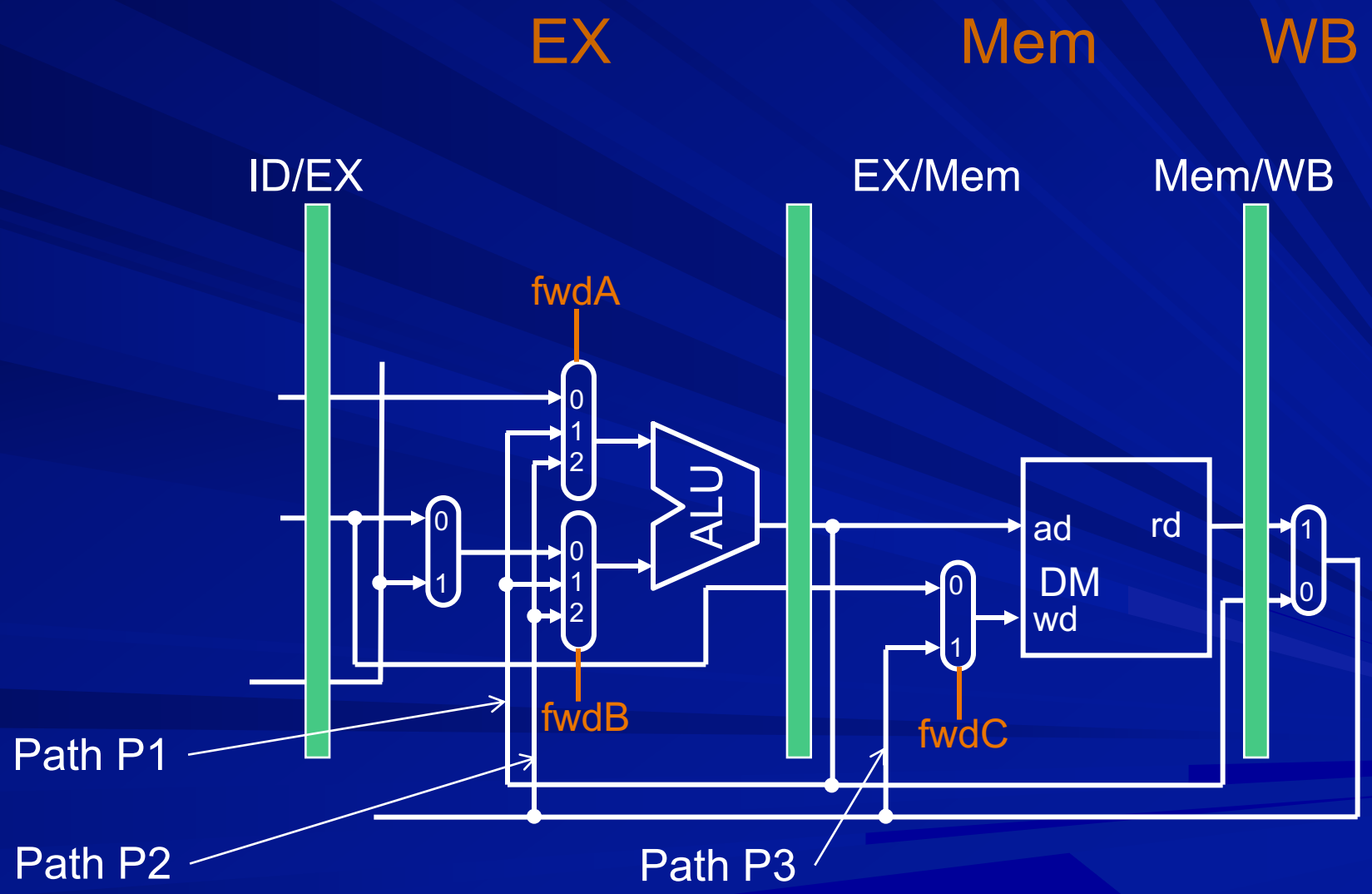

ID/EX.RW = 1 and
(IF/ID.Rn = ID/EX.Rd or
IF/ID.Rm = ID/EX.Rd)

# Dependence check




EX/Mem.RW = 1 and
(IF/ID.Rn = EX/Mem.Rd or
IF/ID.Rm = EX/Mem.Rd)

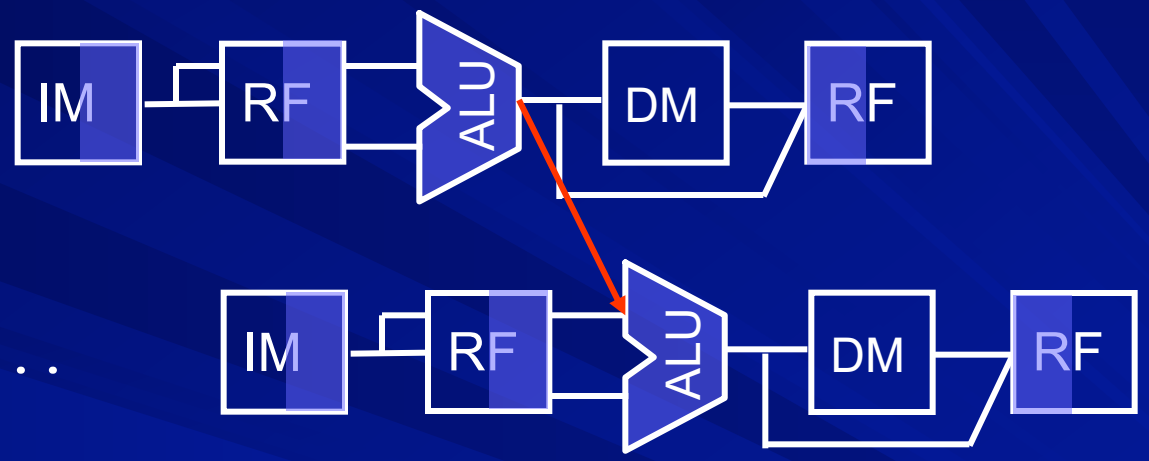
# Data forwarding paths



# Data forwarding path P1

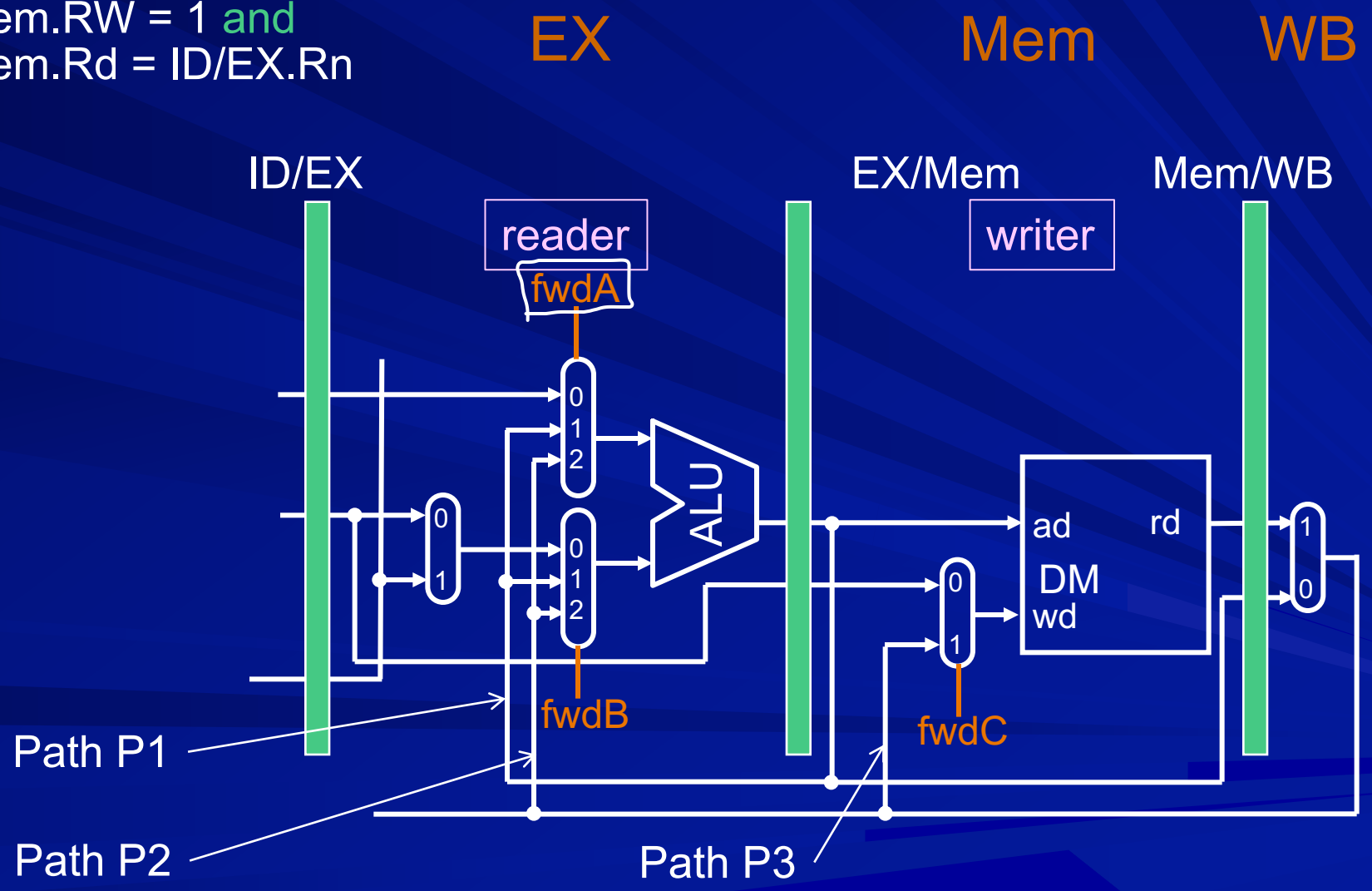
I:add r6,..

I+1:add r4,r6,..

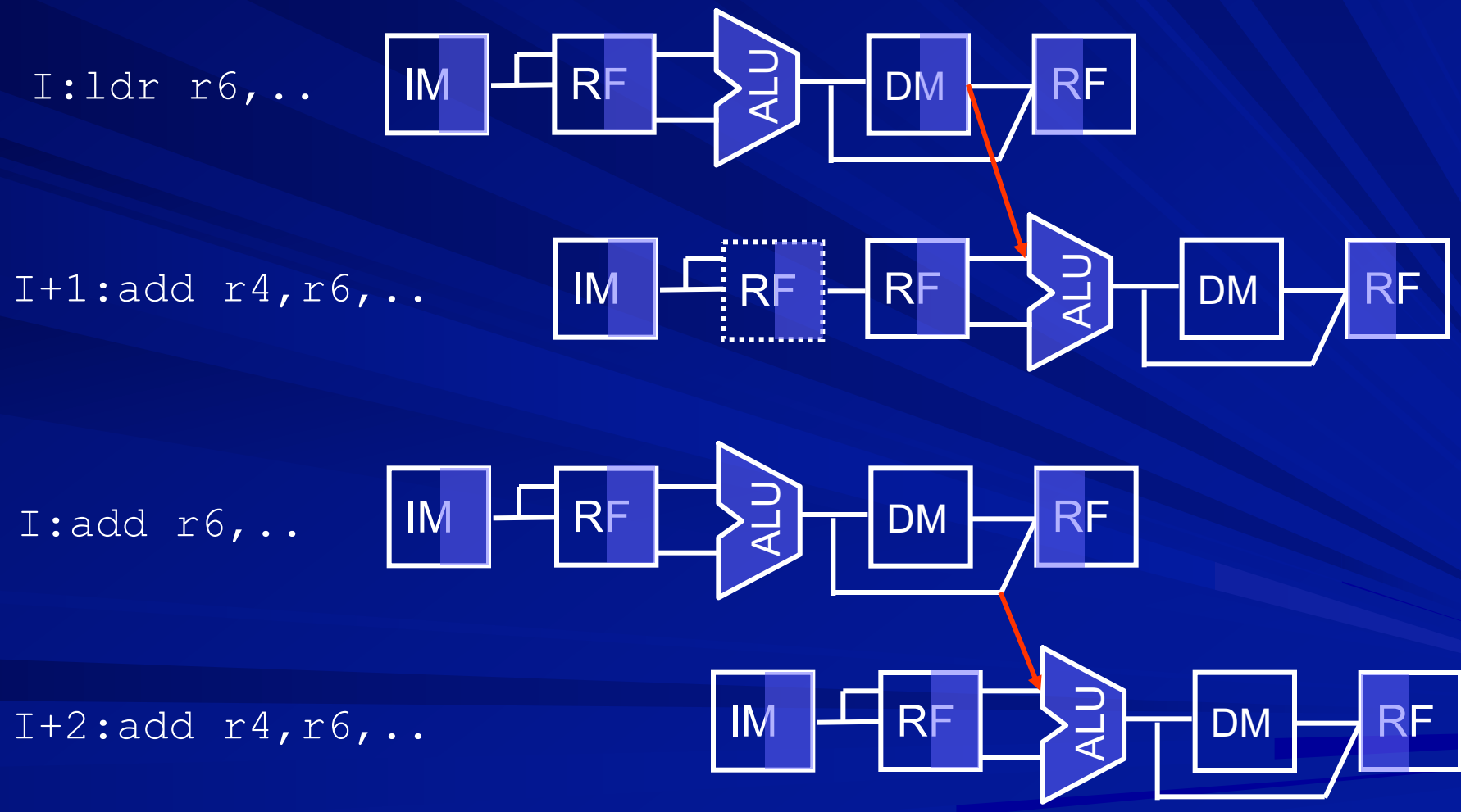


# Control for forwarding path P1

fwdA = 1 if
EX/Mem.RW = 1 and
EX/Mem.Rd = ID/EX.Rn

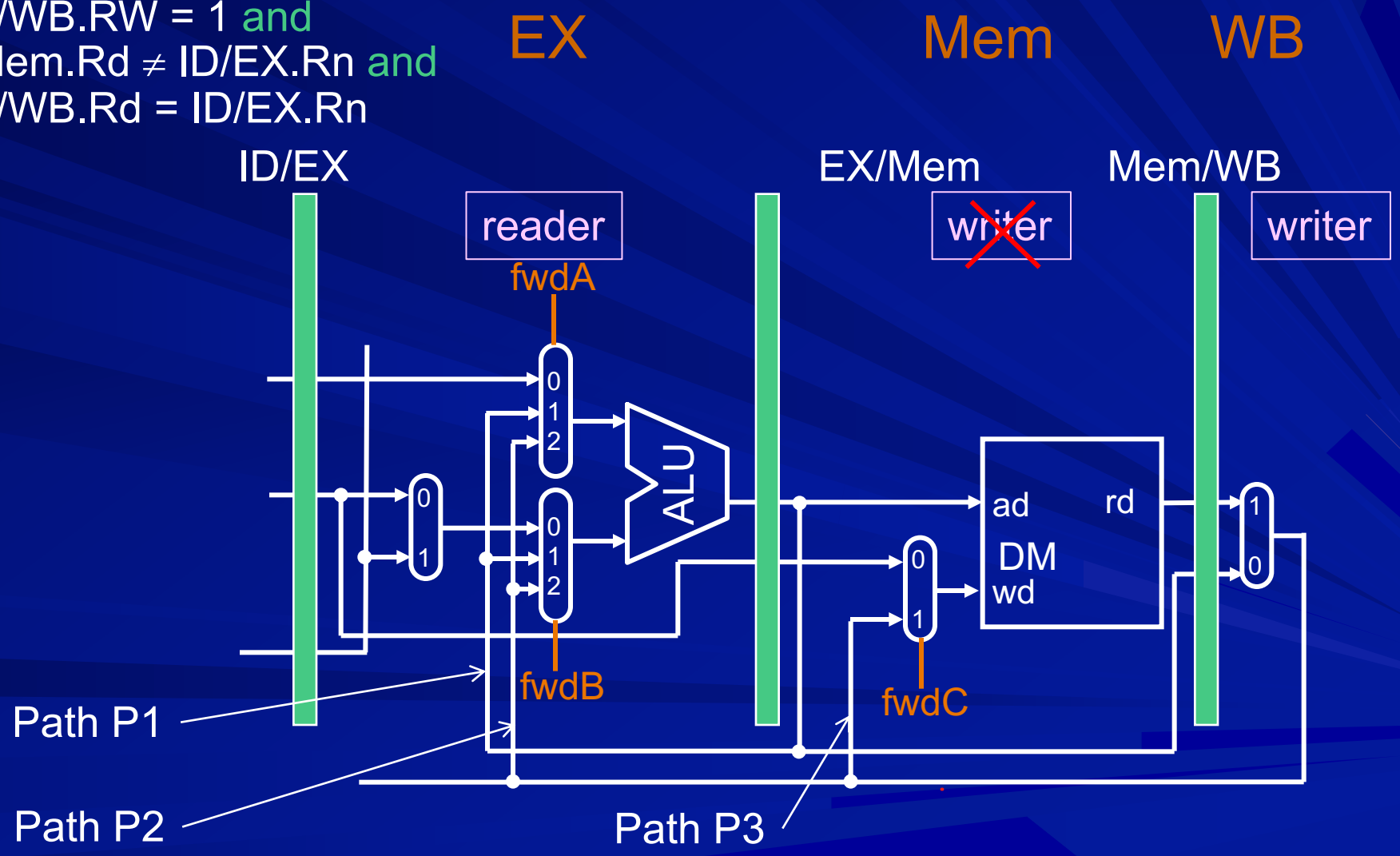


# Data forwarding path P2

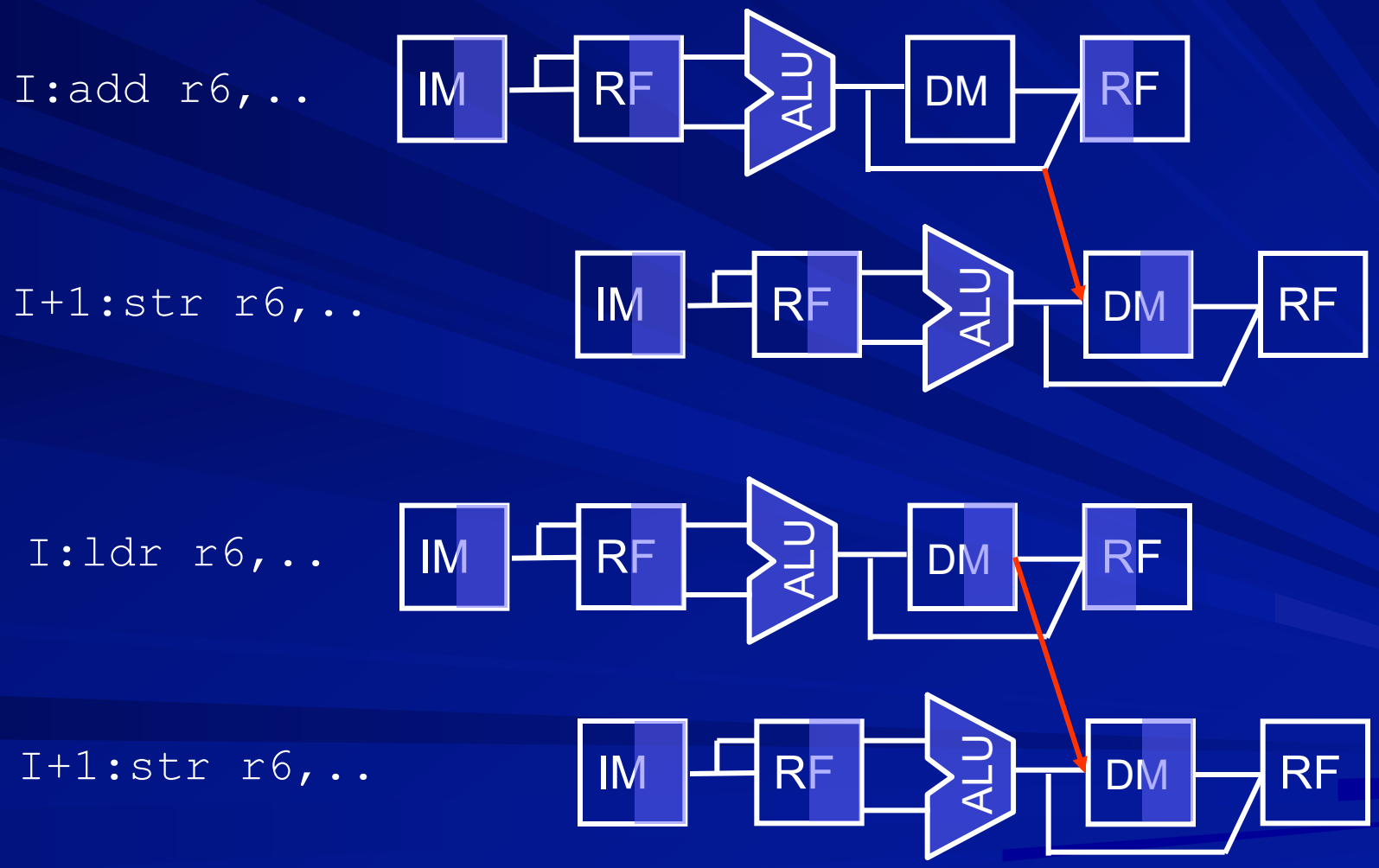


# Control for forwarding path P2

fwdA = 2 if
Mem/WB.RW = 1 and
EX/Mem.Rd ≠ ID/EX.Rn and
Mem/WB.Rd = ID/EX.Rn

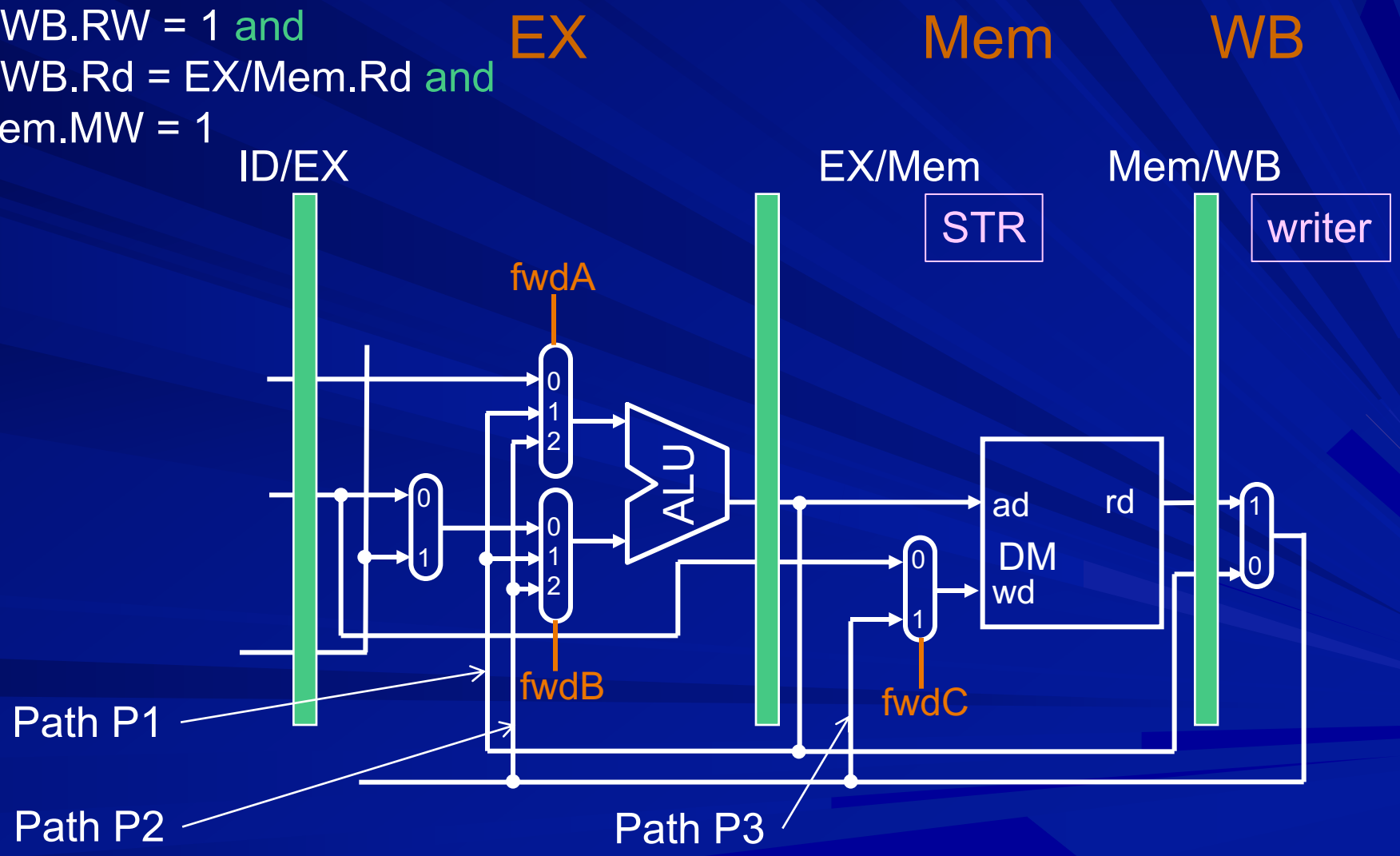


# Data forwarding path P3



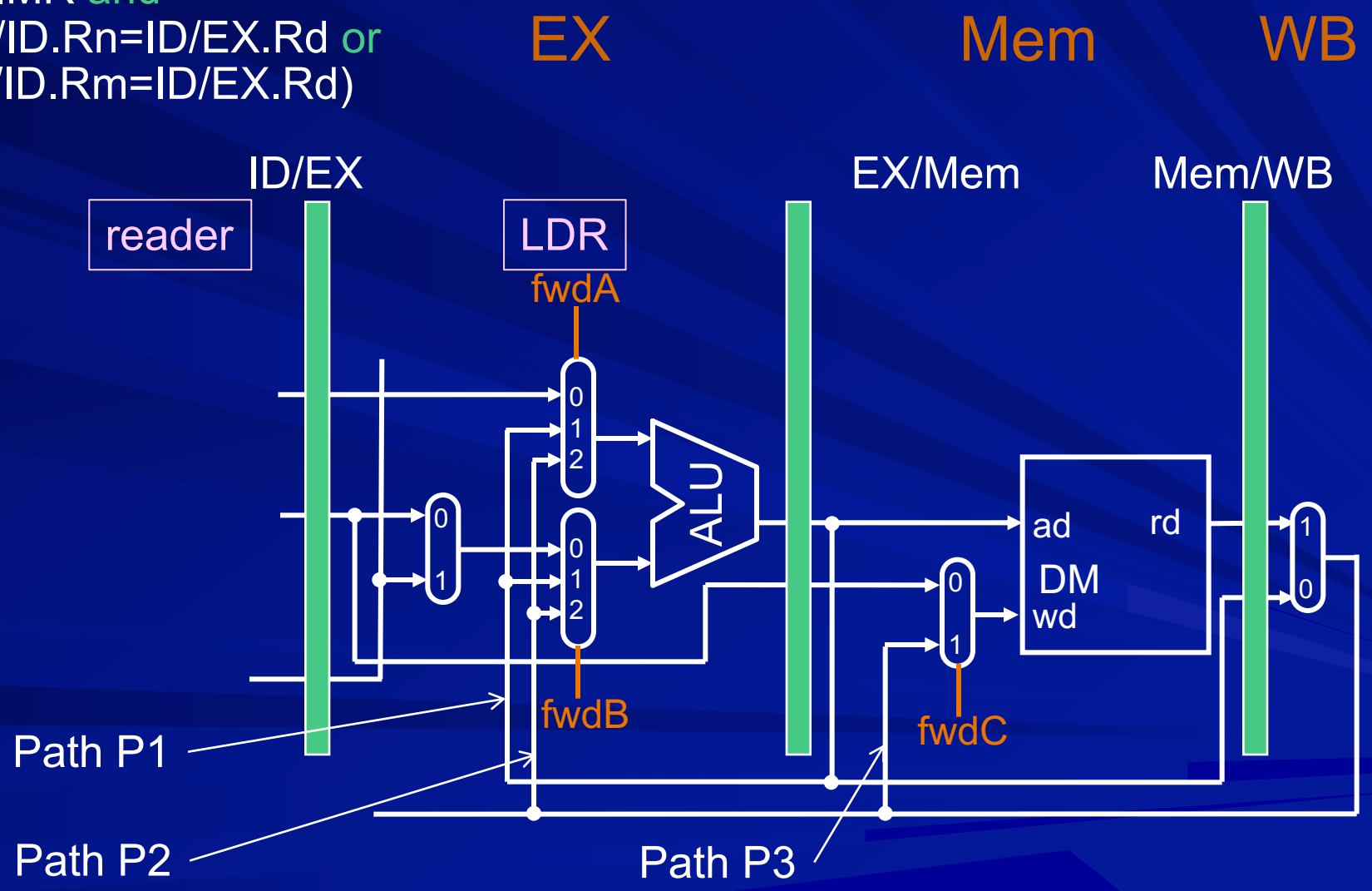
# Control for forwarding path P3

fwdC = 1 if
Mem/WB.RW = 1 and
Mem/WB.Rd = EX/Mem.Rd and
EX/Mem.MW = 1



# Stalling with data forwarding

ID/EX.MR and
(IF/ID.Rn=ID/EX.Rd or
IF/ID.Rm=ID/EX.Rd)



# Thanks

# COL216
# Computer Architecture

## Pipelined processor : Handling hazards

24th February, 2022

# Executing branch instructions

- In which cycle the instruction is found to be a branch instruction? 2
- In which cycle the branch decision is known? 2 or 3
- In which cycle the target address is computed? 3

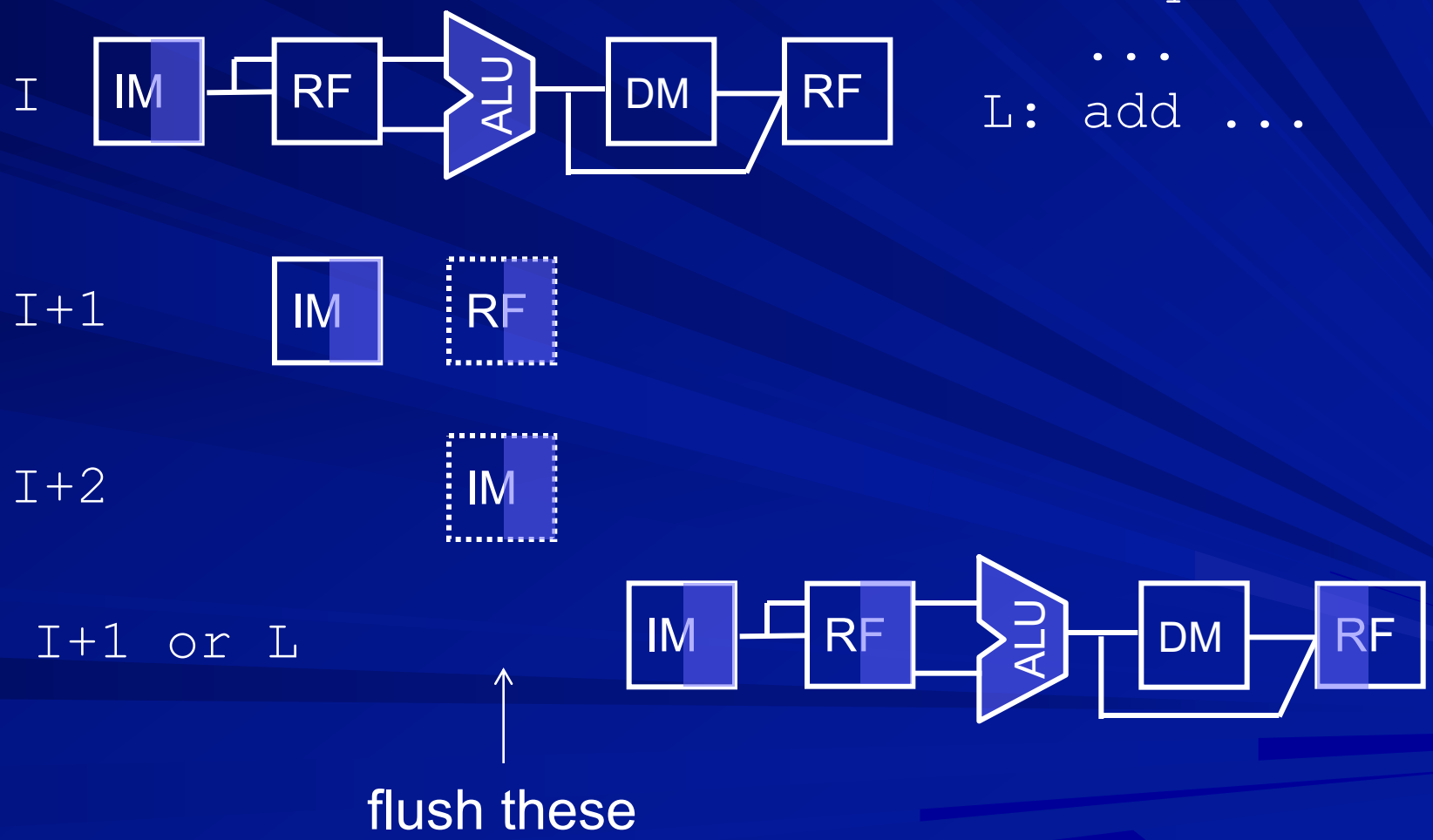
example

# Handling control hazards

- Flush the inline instructions
- Freeze (stall) the inline instructions
- Allow the inline instructions to continue
- Delayed branch
- Predict the branch decision
- Predict the target address

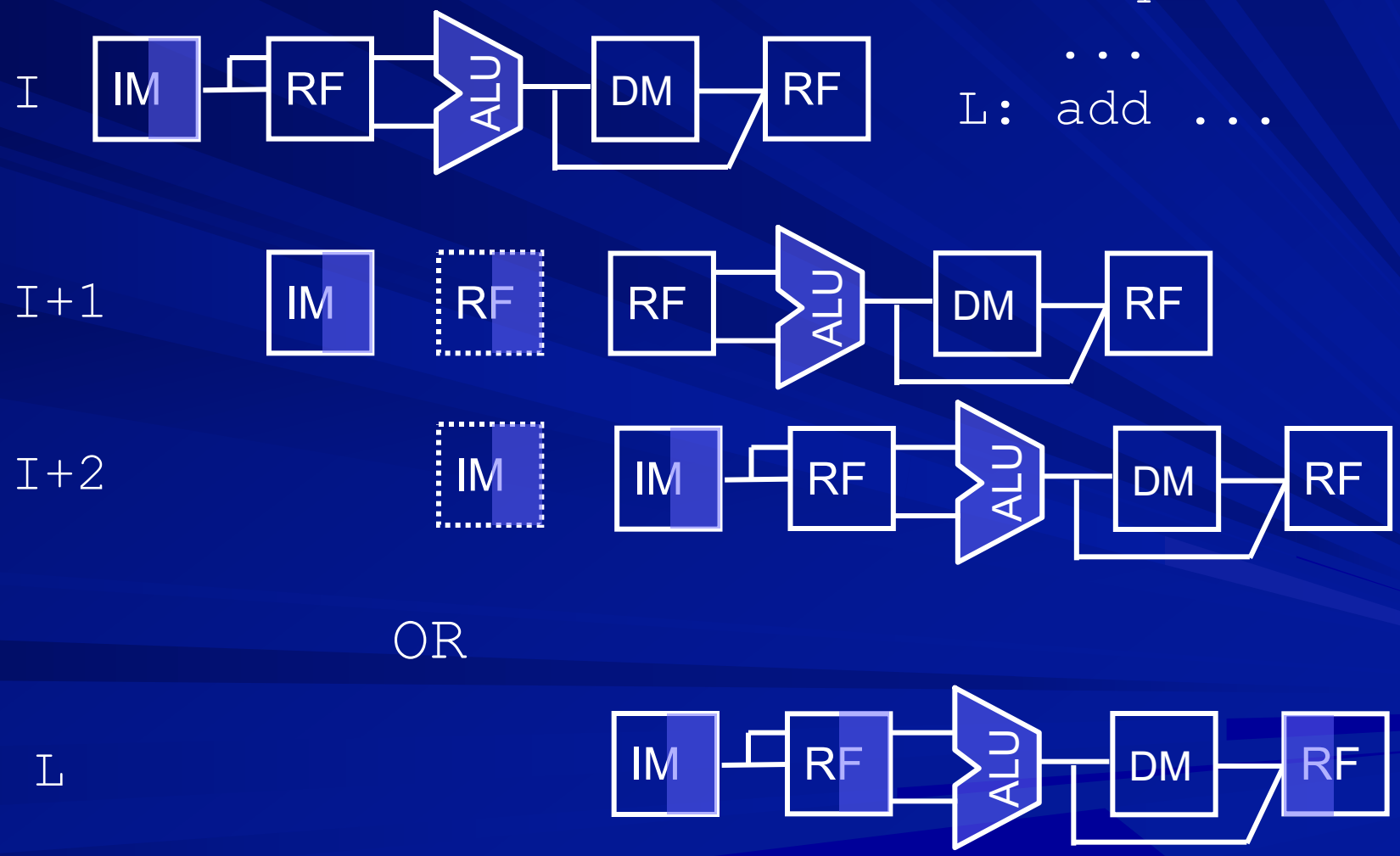
# Flush inline instructions

```
I: beq L
   ...
L: add ...
```



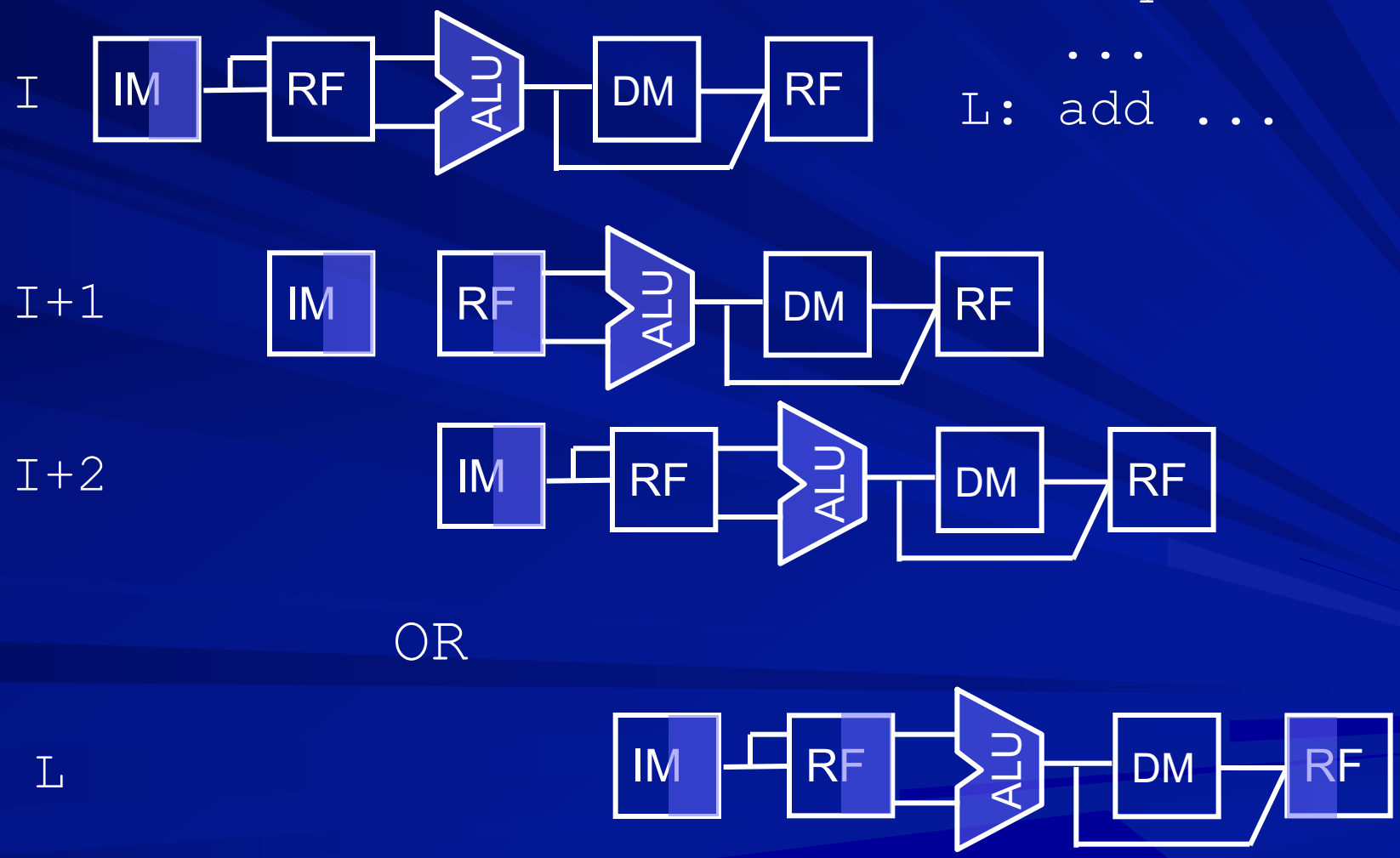
# Freeze inline instructions

```
I: beq L
   ...
L: add ...
```



# Allow inline instructions

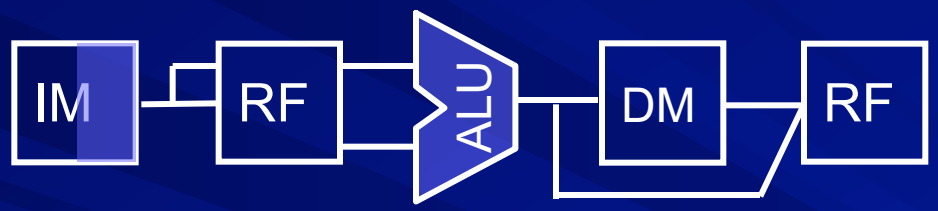
```
I: beq L
   ...
L: add ...
```



# Delayed branch

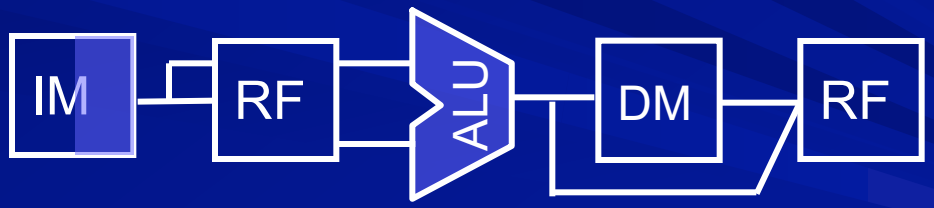
```
I: beq L
   ...
L: add ...
```

I



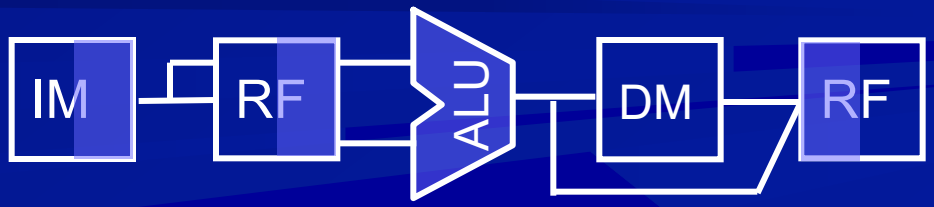

I+1


I+2




OR

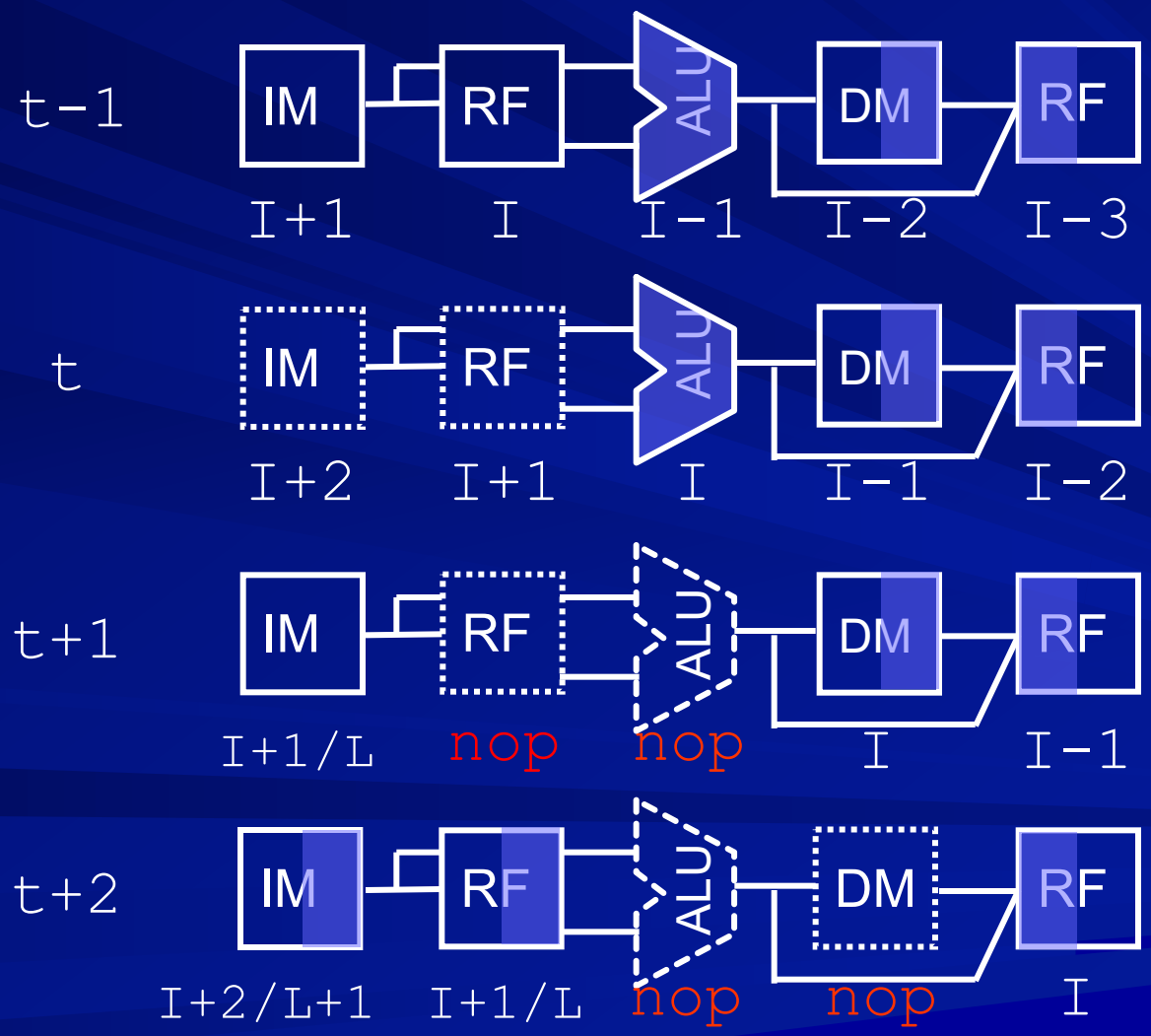
L



# Stalls due to control hazards

## flush: stage-wise view

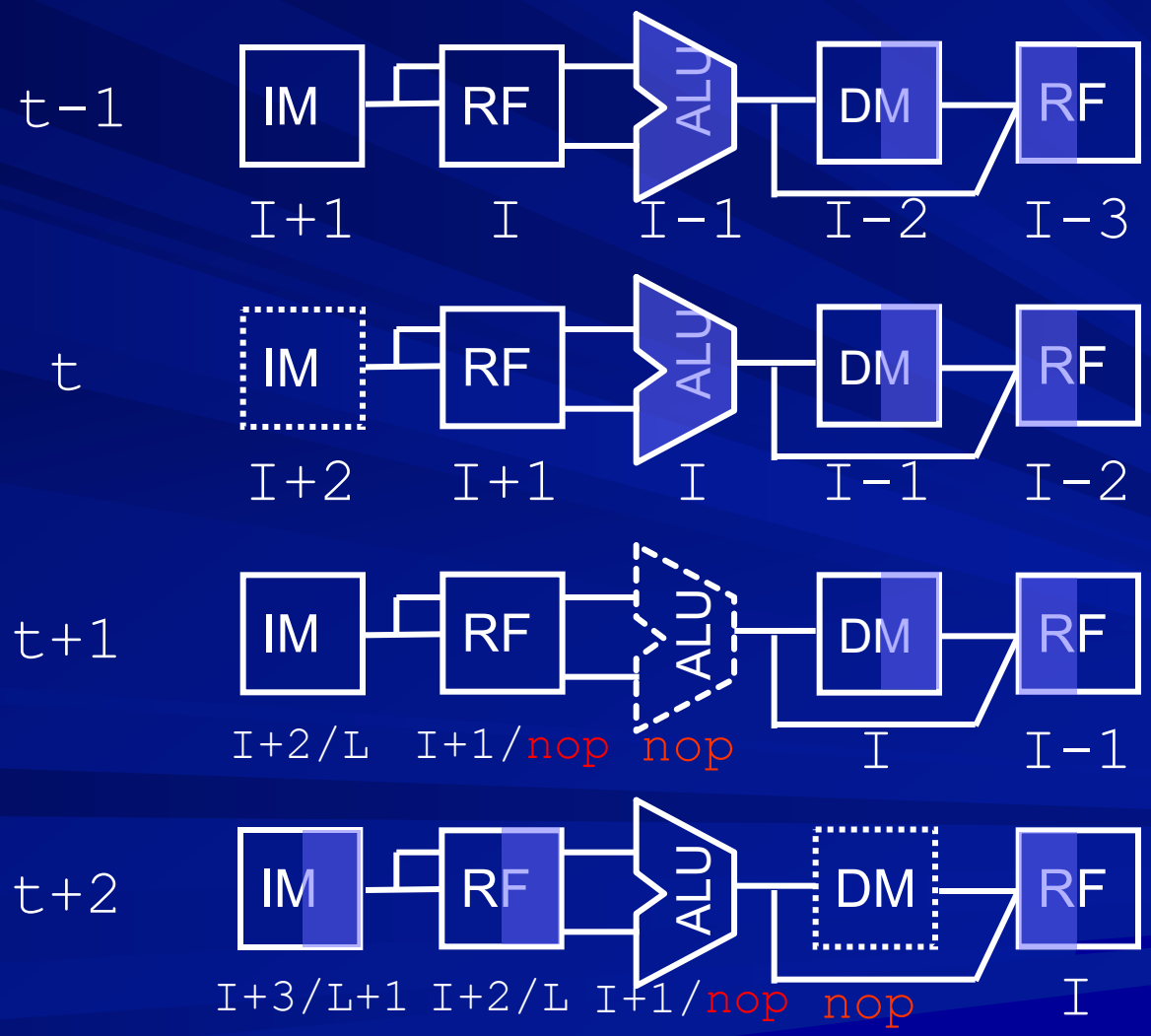
```
I:    beq L
```



# Stalls due to control hazards

## freeze: stage-wise view

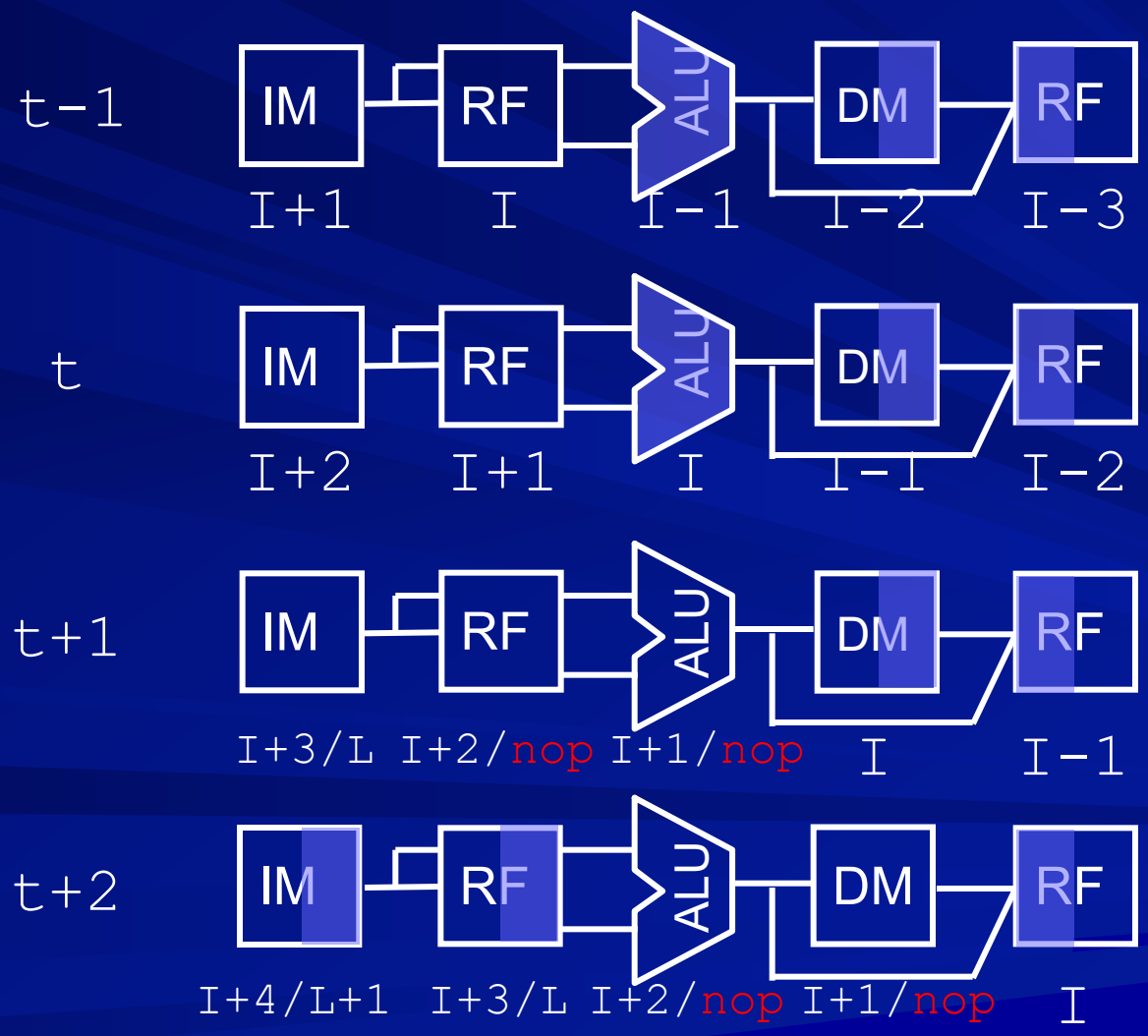
```
I:    beq L
```



# Stalls due to control hazards

## continue: stage-wise view

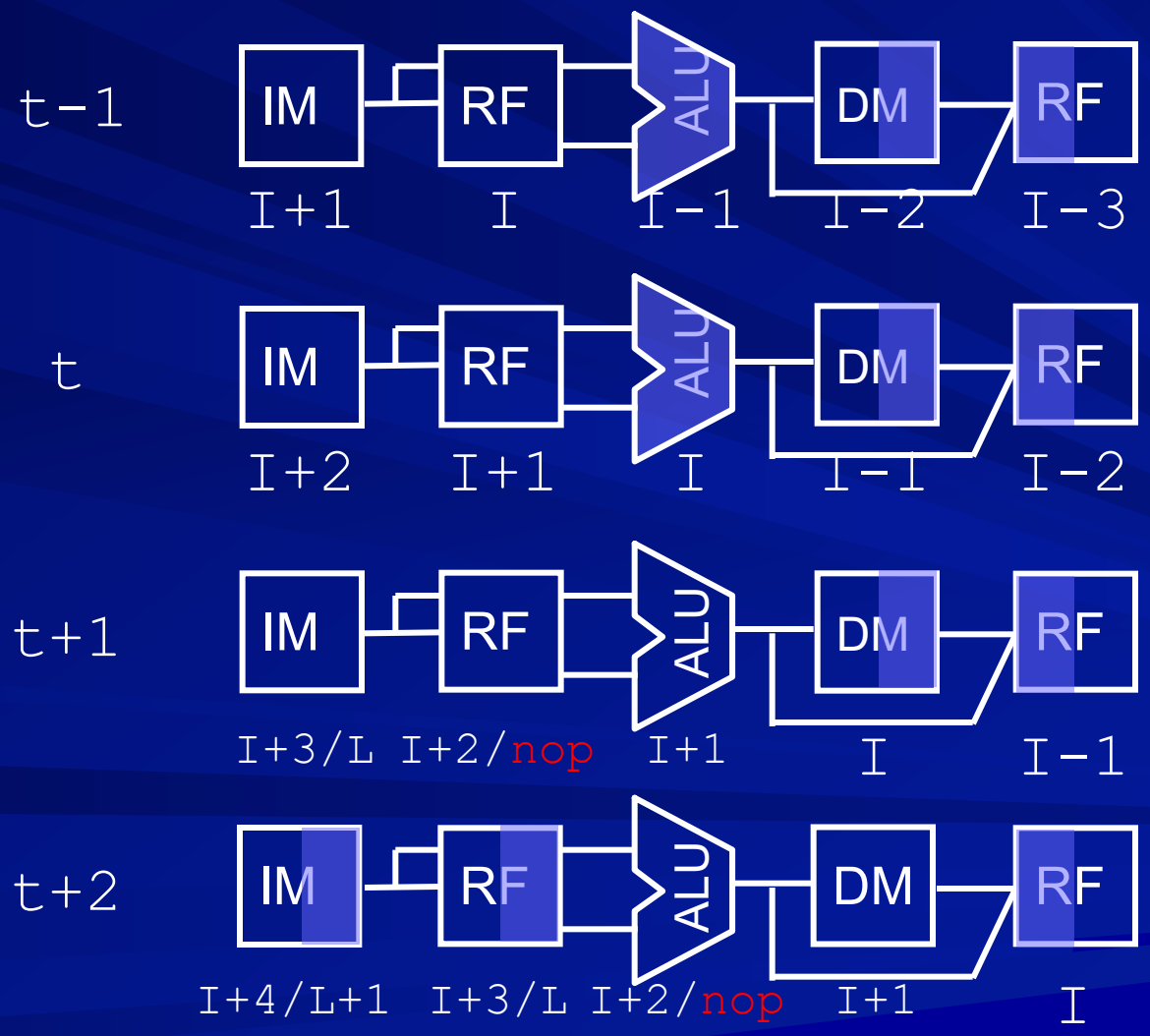
```
I:    beq L
```



# Stalls due to control hazards

## delayed branch: stage-wise view

```
I:    beq L
```




delayed branch misses 1 cycle in case of branch taken and misses 0 cycles in case of branch not taken

# Branch Prediction

- Treat conditional branches as unconditional branches / NOP
- Undo if necessary

Strategies:

- Static
- Dynamic

# Branch Prediction

- Fixed
  - *always predict inline*
- Static
  - *predict on the basis of instruction type, target address or profiling information*
- Dynamic
  - *predict based on recent history*

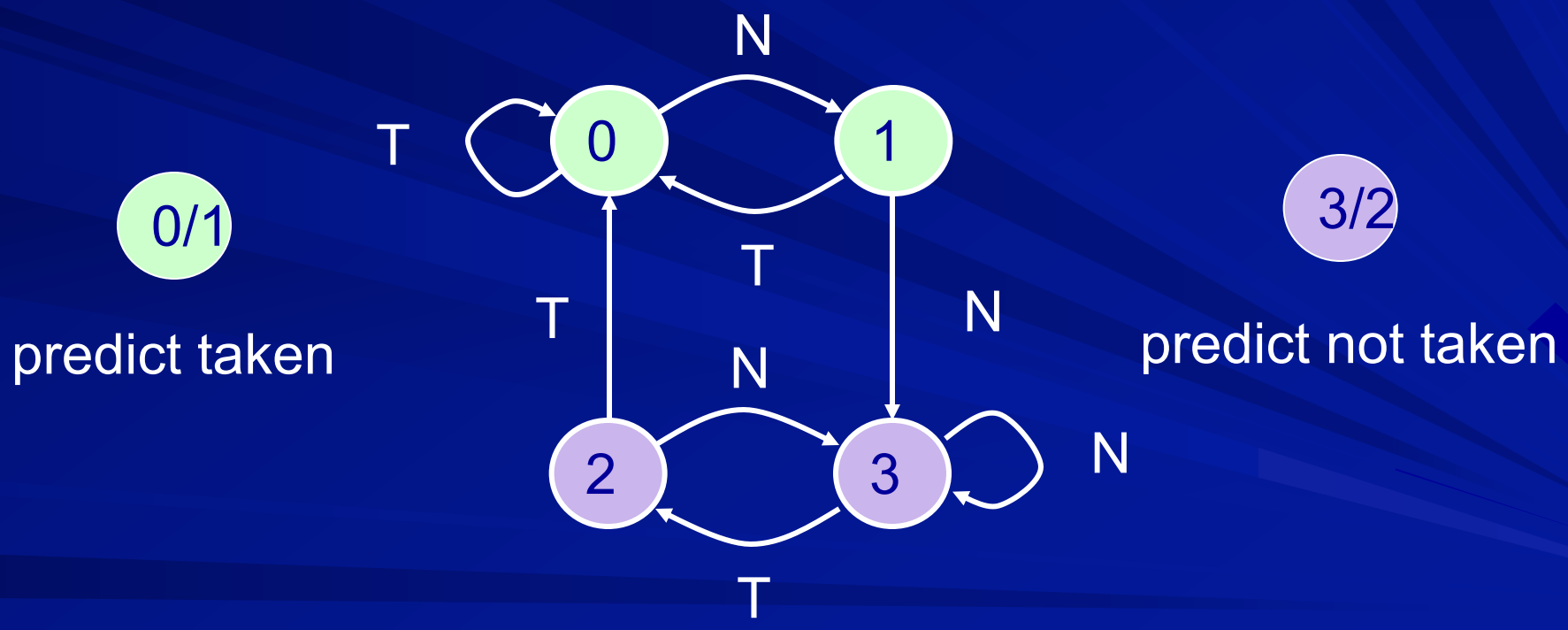
# Dynamic Branch Prediction - basic idea

Predict based on the history of previous branch

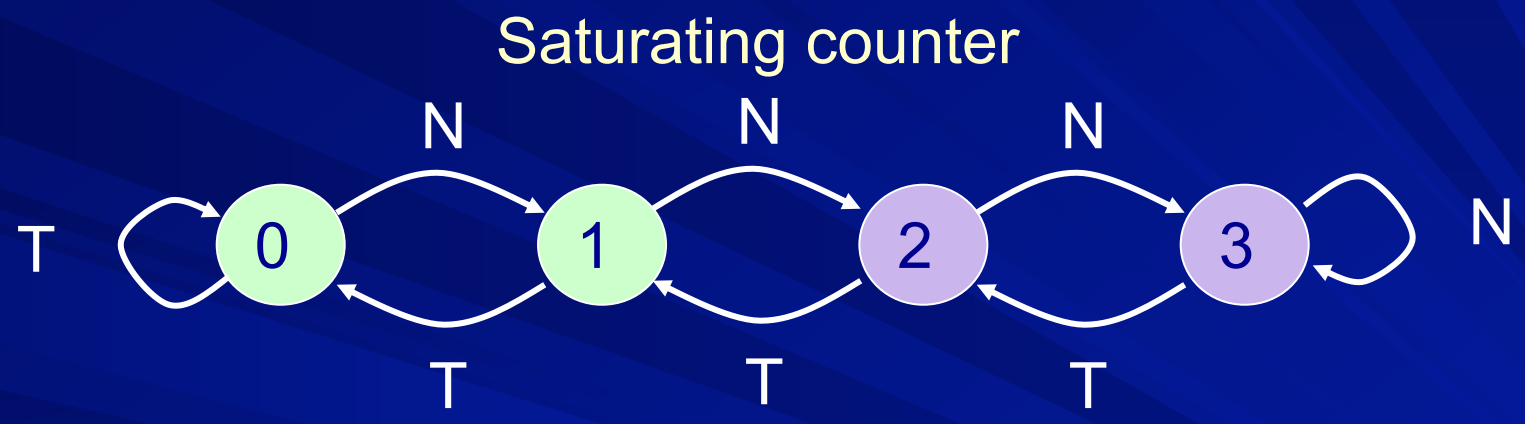
```
loop: xxx
      xxx
      xxx
      xxx
      b<cond> loop
```

2 mispredictions for every occurrence of the loop

# Dynamic Branch Prediction - A 2-bit prediction scheme



# Another 2-bit prediction scheme





predict taken

predict not taken

# Dynamic information about branch

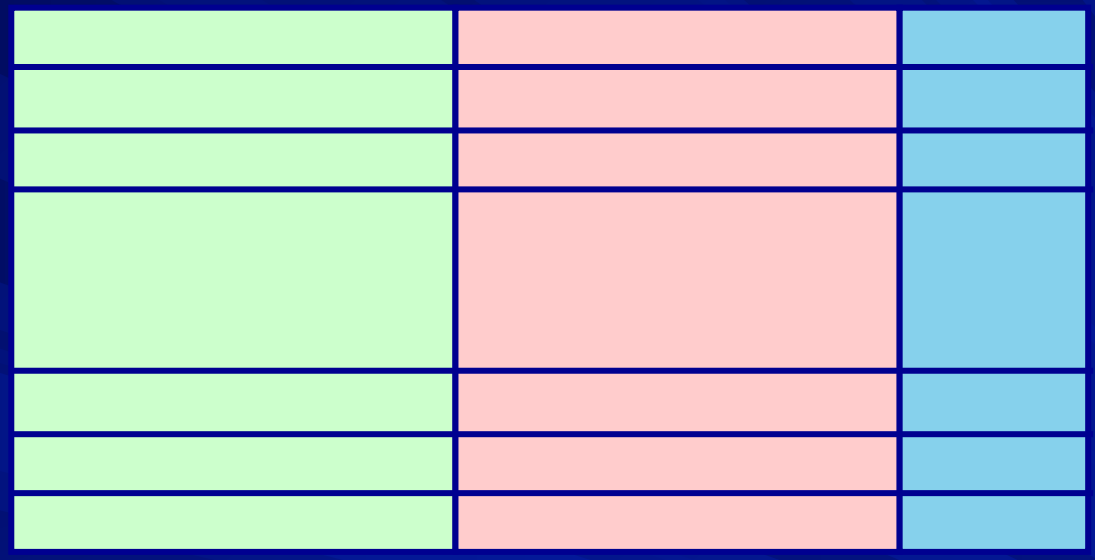
- Previous branch decisions
- Previous target address or target instruction
- Stored in
  - Cache
  - Separate buffer

    Branch History Table (BHT)

    Branch Target Buffer (BTB)

    Target Instruction Buffer (TIB)

# Branch Target Buffer




- hit ⇒ explicit prediction using prediction bits

  prediction → go target (use target info)
  prediction → go inline (ignore target info)

- miss ⇒ go inline

# Accessing BTB

- In which cycle do you access BTB?
- Before checking condition?
- Before address computation?
- Just after decoding?
- Along with instruction fetch? yes

# Correlation between branches

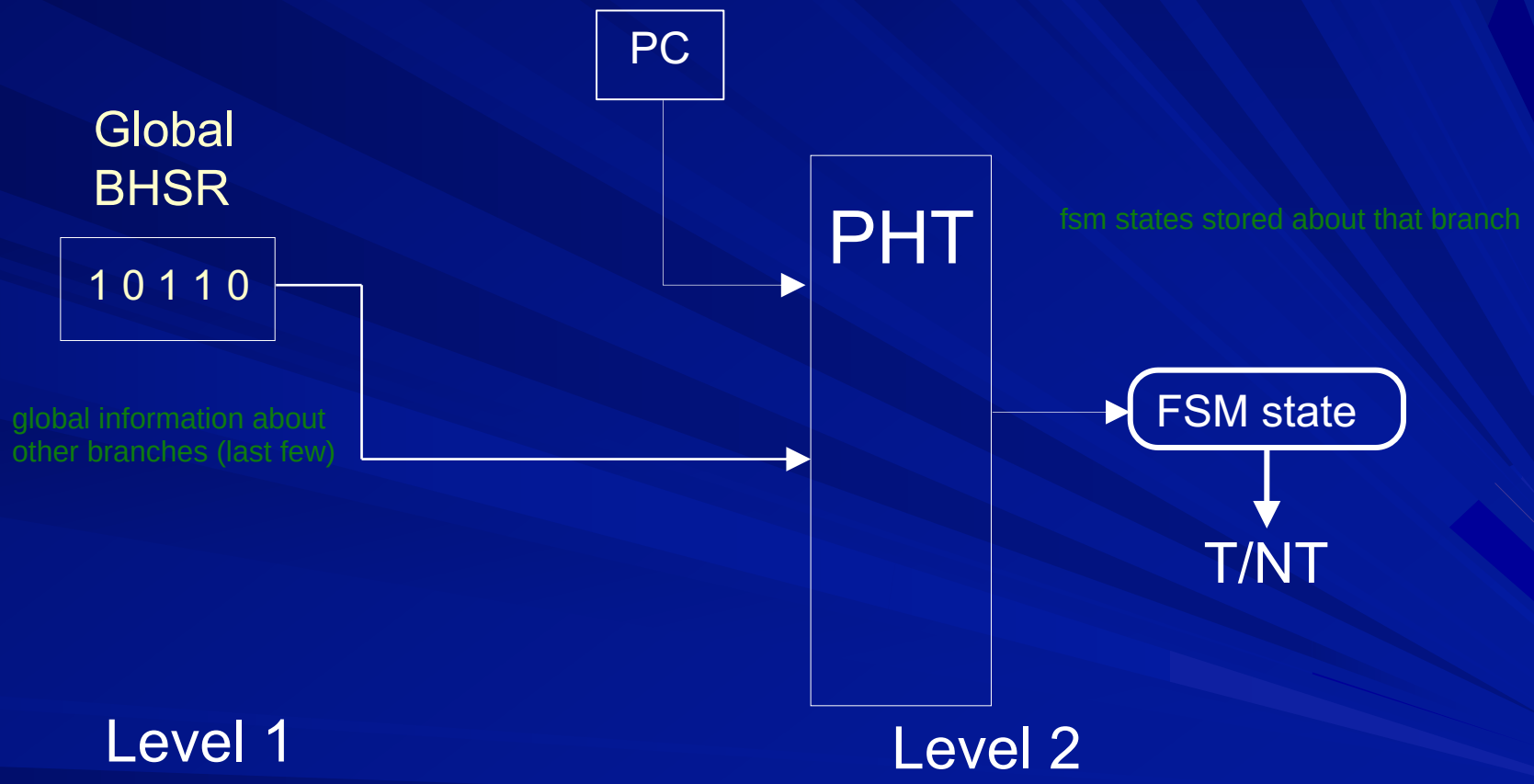
```
B1: if (x)
        ...
B2: if (y)
        ...
    z = x && y
B3: if (z)
        ...
```

- B3 can be predicted with 100% accuracy based on the outcomes of B1 and B2

# Two-Level Branch Predictors

- Level 1
  - Branch History Shift Register (BHSR) - last *n* occurrences
  - Captures patterned behavior of groups of branches
- Level 2
  - Pattern History Table (PHT) - states of predictor FSMs
  - Captures behavior of individual branches

# A two-level branch predictor




Bits from PC and BHSR are combined to index PHT

# Optimizing programs for pipeline

- Instruction reordering
- Moving instructions across branches
- Delayed branches
- Predication

# Running program on pipeline

```
        mov   r1, #1
        mov   r2, #100
L1:     str   r1, [r2, #0]
        add   r2, r2, #4
        add   r1, r1, #1
        cmp   r1, #11
        bne   L1
        mov   r3, #0
        mov   r2, #100
L2:     sub   r1, r1, #1
        cmp   r1, #0
        beq   Over
        ldr   r4, [r2, #0]
        add   r3, r3, r4
        add   r2, r2, #4
        b     L2
Over:
```

# Running program on pipeline

# Without stalls

| | | | | | |
|---|---|---|---|---|---|
| | mov | r1, #1 | 1 | | |
| | mov | r2, #100 | 2 | | |
| L1: | str | r1, [r2, #0] | 3 | 8 | .. 48 |
| | add | r2, r2, #4 | 4 | | |
| | add | r1, r1, #1 | 5 | | |
| | cmp | r1, #11 | 6 | | |
| | bne | L1 | 7 | 12 | .. 52 |
| | mov | r3, #0 | 53 | | |
| | mov | r2, #100 | 54 | | |
| L2: | sub | r1, r1, #1 | 55 | 62 | .. 118 |
| | cmp | r1, #0 | 56 | | |
| | beq | Over | 57 | | |
| | ldr | r4, [r2, #0] | 58 | | |
| | add | r3, r3, r4 | 59 | | |
| | add | r2, r2, #4 | 60 | | |
| | b | L2 | 61 | 68 | ..124 |
| Over: | | | 128 | | |

# Running program on pipeline

# With stalls

| | | | | |
|---|---|---|---|---|
| | mov | r1, #1 | 1 | |
| | mov | r2, #100 | 2 | |
| L1: | str | r1, [r2, #0] | 5 | 14 .. 86 |
| | add | r2, r2, #4 | 6 | |
| | add | r1, r1, #1 | 7 | |
| | cmp | r1, #11 | 10 | |
| | bne | L1 | 11 | 20 .. 92 |
| | mov | r3, #0 | 95 | |
| | mov | r2, #100 | 96 | |
| L2: | sub | r1, r1, #1 | 97 | 112 .. 232 |
| | cmp | r1, #0 | 100 | |
| | beq | Over | 101 | |
| | ldr | r4, [r2, #0] | 104 | |
| | add | r3, r3, r4 | 107 | |
| | add | r2, r2, #4 | 108 | |
| | b | L2 | 109 | 124..244 |
| Over: | | | 254 | |

# Execute inline instructions after branch

| | | | | |
|---|---|---|---|---|
| | mov | r1, #1 | 1 | |
| | mov | r2, #100 | 2 | |
| L1: | str | r1, [r2, #0] | 5 | 14 .. 86 |
| | add | r2, r2, #4 | 6 | |
| | add | r1, r1, #1 | 7 | |
| | cmp | r1, #11 | 10 | |
| | bne | L1 | 11 | 20 .. 92 |
| | mov | r3, #0 | 95 | |
| | mov | r2, #100 | 96 | |
| L2: | sub | r1, r1, #1 | 97 | 110 .. 214 |
| | cmp | r1, #0 | 100 | |
| | beq | Over | 101 | |
| | ldr | r4, [r2, #0] | 102 | |
| | add | r3, r3, r4 | 105 | |
| | add | r2, r2, #4 | 106 | |
| | b | L2 | 107 | 120..224 |
| Over: | | | 234 | |

Note: this benefit is applicable to instruction “bne L1” as well, but not considered in the calculations shown here.

# Reorder instructions

```
        mov  r1, #1          1
        mov  r2, #100        2
L1:     str  r1, [r2, #0]    5        14  .. 86
        add  r2, r2, #4      6
        add  r1, r1, #1      7
        cmp  r1, #11         10
        bne  L1              11       20  .. 92
        mov  r3, #0          95
        mov  r2, #100        96
L2:     sub  r1, r1, #1      97       110 .. 214
        cmp  r1, #0          100
        beq  Over            101
        ldr  r4, [r2, #0]    102
        add  r3, r3, r4      105
        add  r2, r2, #4      106
        b    L2              107      120..224
Over:                        234
```

# After reordering

```
        mov   r1, #1          1
        mov   r2, #100        2
L1:     str   r1, [r2, #0]    5       13  .. 77
        add   r1, r1, #1      6
        add   r2, r2, #4      7
        cmp   r1, #11         9
        bne   L1              10      18  .. 82
        mov   r3, #0          85
        mov   r2, #100        86
L2:     sub   r1, r1, #1      87      99  .. 195
        cmp   r1, #0          90
        beq   Over            91
        ldr   r4, [r2, #0]    92
        add   r2, r2, #4      93
        add   r3, r3, r4      95
        b     L2              96      108..204
Over:                         214
```

# Further reordering

```
        mov  r1, #1          1
        mov  r2, #100        2
L1:     str  r1, [r2, #0]    5        13  .. 77
        add  r1, r1, #1      6
        add  r2, r2, #4      7
        cmp  r1, #11         9
        bne  L1              10       18  .. 82
        mov  r3, #0          85
        mov  r2, #100        86
L2:     sub  r1, r1, #1      87       99  .. 195
        cmp  r1, #0          90
        beq  Over            91
        ldr  r4, [r2, #0]    92
        add  r2, r2, #4      93
        add  r3, r3, r4      95
        b    L2              96       108..204
Over:                        214
```

# After reordering

```
        mov  r1, #1          1
        mov  r2, #100        2
L1:     str  r1, [r2, #0]    5        13  .. 77
        add  r1, r1, #1      6
        add  r2, r2, #4      7
        cmp  r1, #11         9
        bne  L1              10       18  .. 82
        sub  r1, r1, #1      85
        mov  r3, #0          86
        mov  r2, #100        87
L2:     cmp  r1, #0          88       97  .. 169
        beq  Over            89
        ldr  r4, [r2, #0]    90
        add  r2, r2, #4      91
        sub  r1, r1, #1      92
        add  r3, r3, r4      93
        b    L2              94       103..175
Over:                        182
```

# Delayed branch

```
        mov  r1, #1          1
        mov  r2, #100        2
L1:     str  r1, [r2, #0]    5        13  .. 77
        add  r1, r1, #1      6
        add  r2, r2, #4      7
        cmp  r1, #11         9
        bne  L1              10       18  .. 82
        sub  r1, r1, #1      85
        mov  r3, #0          86
        mov  r2, #100        87
L2:     cmp  r1, #0          88       97  .. 169
        beq  Over            89
        ldr  r4, [r2, #0]    90
        add  r2, r2, #4      91
        sub  r1, r1, #1      92
        add  r3, r3, r4      93
        b    L2              94       103..175
Over:                        182
```

# Delayed branch

Instructions in delay slots

```
        mov  r1, #1          1
        mov  r2, #100        2
        str  r1, [r2, #0]    5
L1:     add  r1, r1, #1      6      13  .. 69
        add  r2, r2, #4      7
        cmp  r1, #11         9
        bne  L1              10
        str  r1, [r2, #0]    11     18  .. 74
        sub  r1, r1, #1      75
        mov  r3, #0          76
        mov  r2, #100        77
        cmp  r1, #0          78
L2:     beq  Over            79     87  .. 151
        ldr  r4, [r2, #0]    80
        add  r2, r2, #4      81
        sub  r1, r1, #1      82
        add  r3, r3, r4      83
        b    L2              84
        cmp  r1, #0          85     93  ..157
Over:                        162
```

# Summary

- Original code without stalls 128
- Original code with stalls 254
- Execute inline instr after branch 234
- After first re-ordering 214
- After second reordering 182
- Using delayed branch 162

# Branch elimination example

(find max element in array)

```
func: str   lr, [sp, #-4]!   1   1
      cmp r2, #0             2   2
      bne L1                 3   3
      b   Ret
L1:   ldr   r3, [r1]         6   6
      cmp r0, r3             9   9
      blo L2                 10  10
      b   L3                     13
L2:   mov r0, r3             13
L3:   add   r1, r1, #4       14  16
      sub   r2, r2, #1       15  17
      bl    func             16  18
Ret:  ldr   pc, [sp], #4
```

data dependence

branches taken

# Branch elimination example

```
func:  str   lr, [sp, #-4]!
       cmp r2, #0
       bne  L1
       b    Ret
L1:    ldr  r3, [r1]
       cmp r0, r3
       blo  L2
       b    L3
L2:    mov r0, r3
L3:    add  r1, r1, #4
       sub  r2, r2, #1
       bl   func
Ret:   ldr  pc, [sp], #4
```

```
beq Ret
```

```
L2:   movlo r0, r3
```

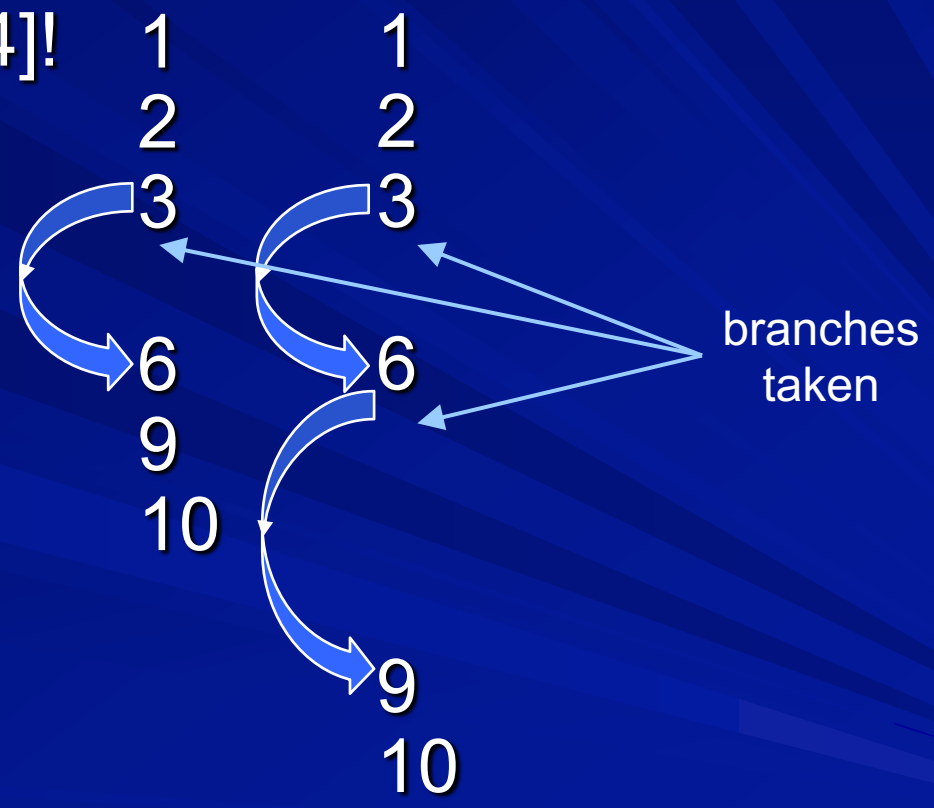
# Reduced branches

```
func:  str     lr, [sp, #-4]!    1
       cmp     r2, #0            2
       beq     Ret               3
L1:    ldr     r3, [r1]          6
       cmp     r0, r3            9
L2:    movlo   r0, r3            10
L3:    add     r1, r1, #4        11
       sub     r2, r2, #1        12
       bl      func              13
Ret:   ldr     pc, [sp], #4
```

# Another example

(GCD)

```
func:  str   lr, [sp, #-4]!
       cmp   r0, r1
       bne   L1
       b     Ret
L1:    bhs   L2
       sub   r1, r1, r0
       bl    func
       b     Ret
L2:    sub   r0, r0, r1
       bl    func
Ret:   ldr   pc, [sp], #4
```



# Another example

```
func:  str   lr, [sp, #-4]!
       cmp   r0, r1
       bne   L1
       b     Ret
L1:    bhs   L2
       sub   r1, r1, r0
       bl    func
       b     Ret
L2:    sub   r0, r0, r1
       bl    func
Ret:   ldr   pc, [sp], #4
```

```
beq Ret
```

```
       sublo r1, r1, r0
L2:    subhs r0, r0, r1
       bl func
```

# Reduced branches

```
func:  str     lr, [sp, #-4]!     1
       cmp     r0, r1             2
       beq     Ret                3
       sublo   r1, r1, r0         6
L2:    subhs   r0, r0, r1         9
       bl      func               10
Ret:   ldr     pc, [sp], #4
```

data dependence

# Moving “sublo r1,r1,ro” up

```
func:  str     lr, [sp, #-4]!    1
       cmp     r0, r1            2
       sublo   r1, r1, r0        3
       beq     Ret               4
L2:    subhs   r0, r0, r1        7
       bl      func              8
Ret:   ldr     pc, [sp], #4
```

# Put “sublo r1,r1,ro” in delay slot

```
func:  str     lr, [sp, #-4]!    1
       cmp     r0, r1            2
       beq     Ret               3
       sublo   r1, r1, r0        4
L2:    subhs   r0, r0, r1        7
       bl      func              8
Ret:   ldr     pc, [sp], #4
```

# THANKS