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

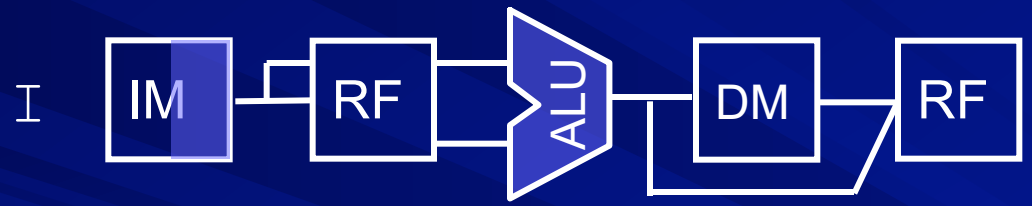
example

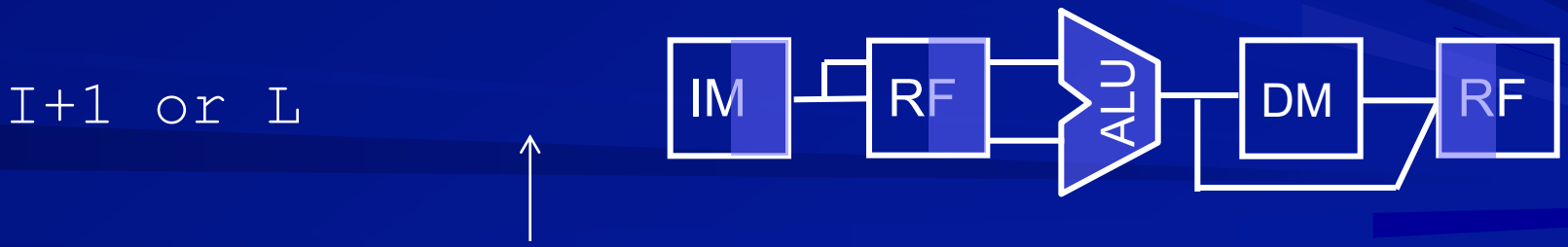
Handling control hazards

- 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

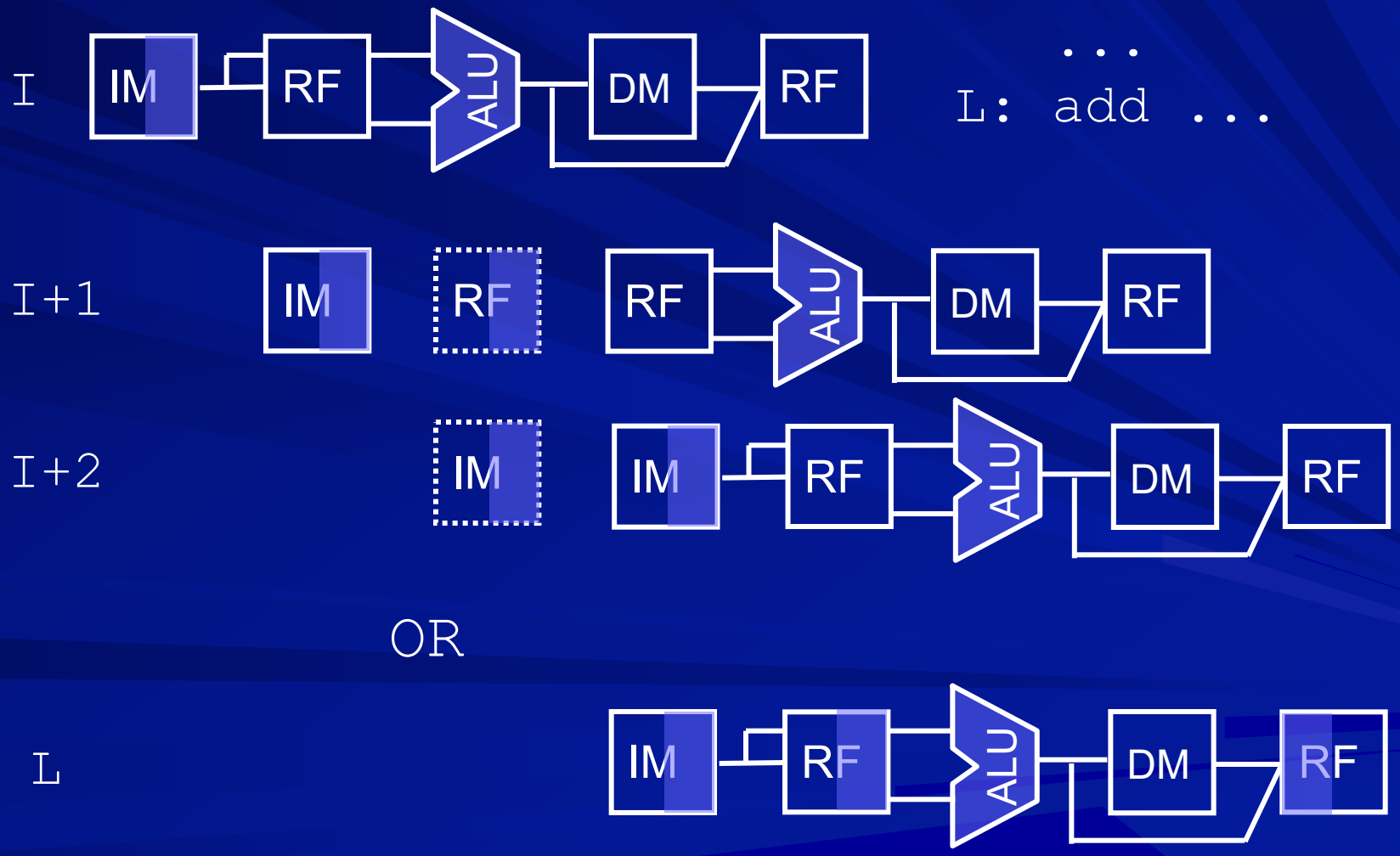
```
I: beq L
   ...
L: add ...
```





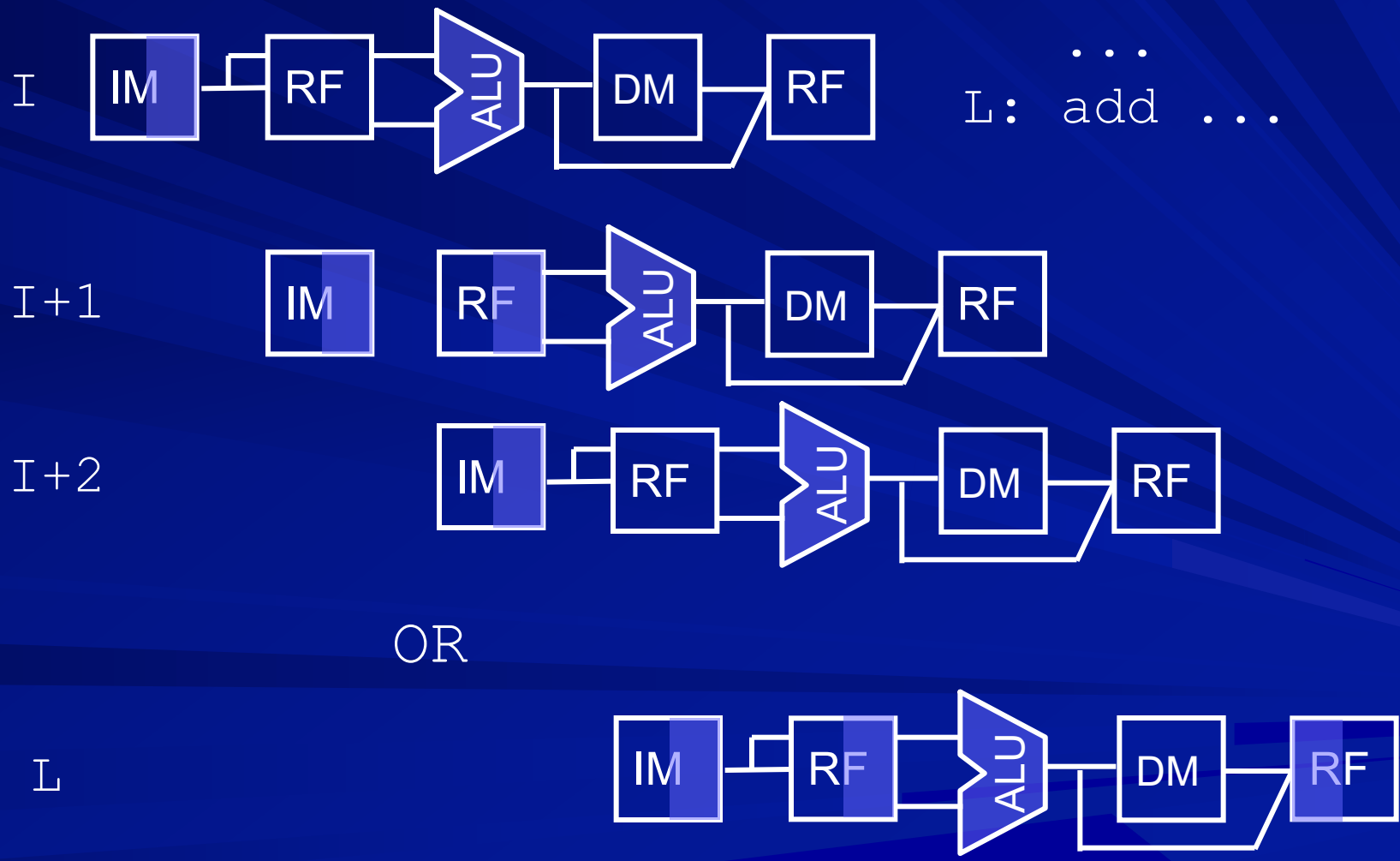
Freeze inline instructions

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



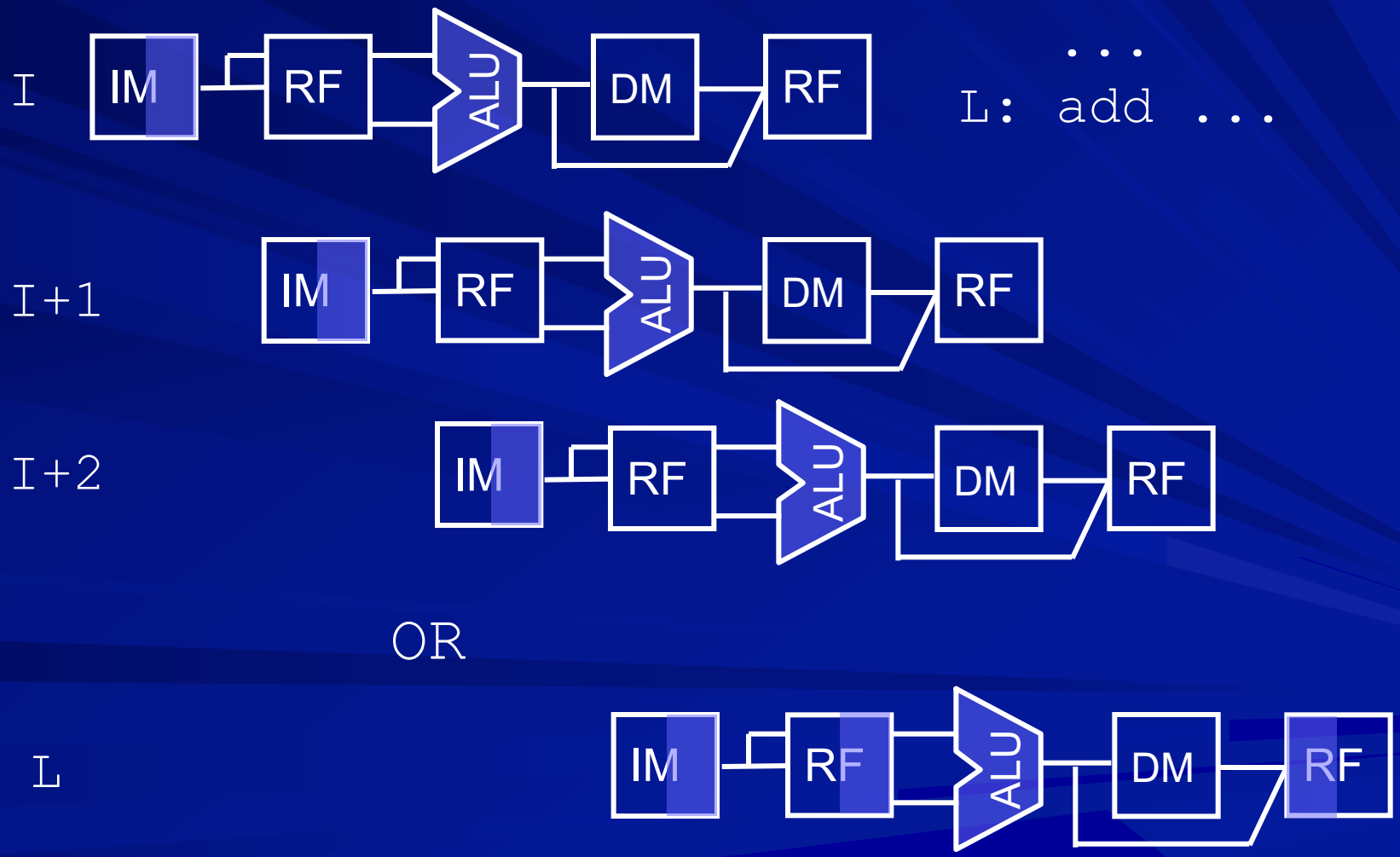
Allow inline instructions

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



Delayed branch

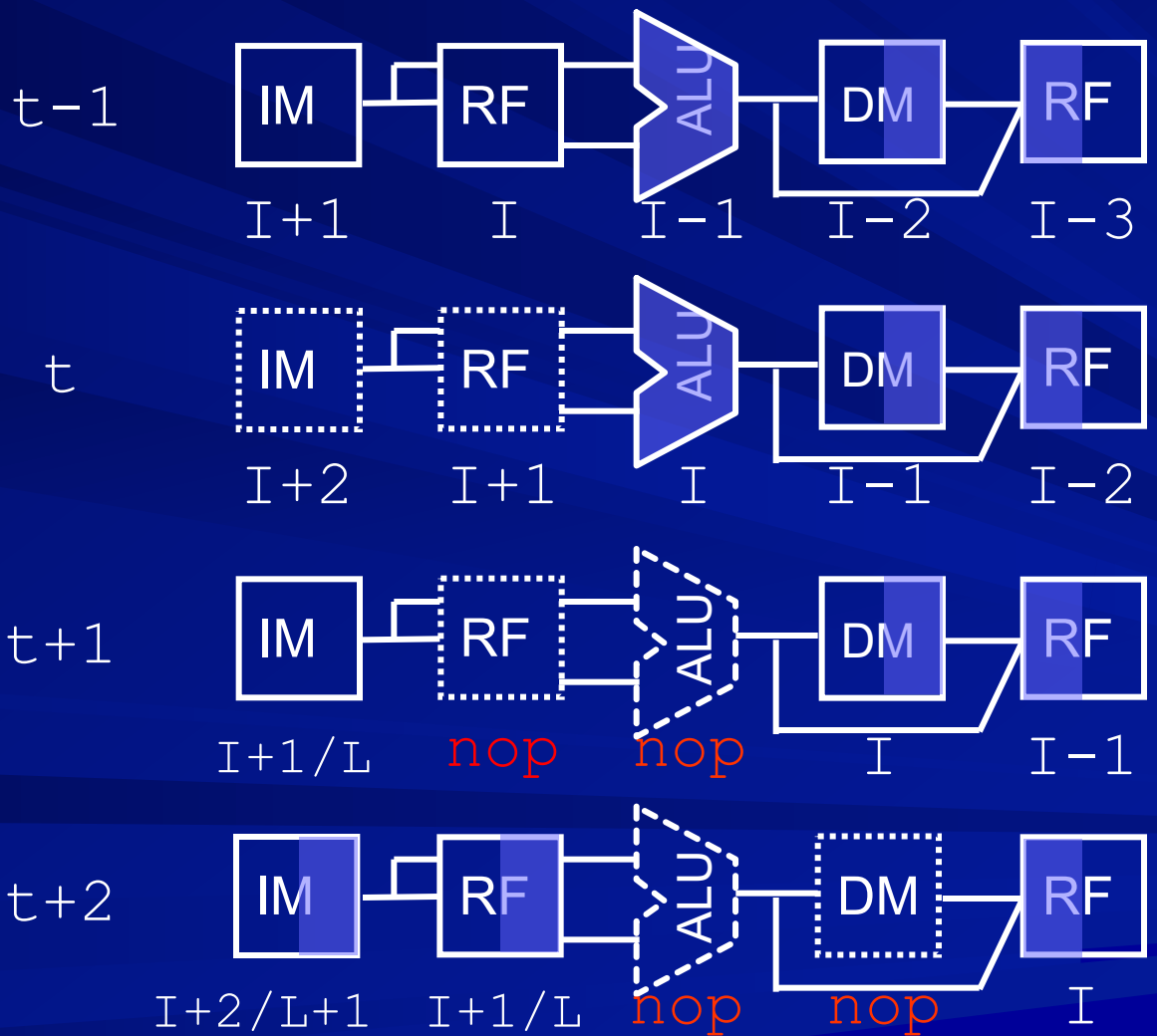
```
I: beq L
   ...
L: add ...
```



Stalls due to control hazards

flush: stage-wise view

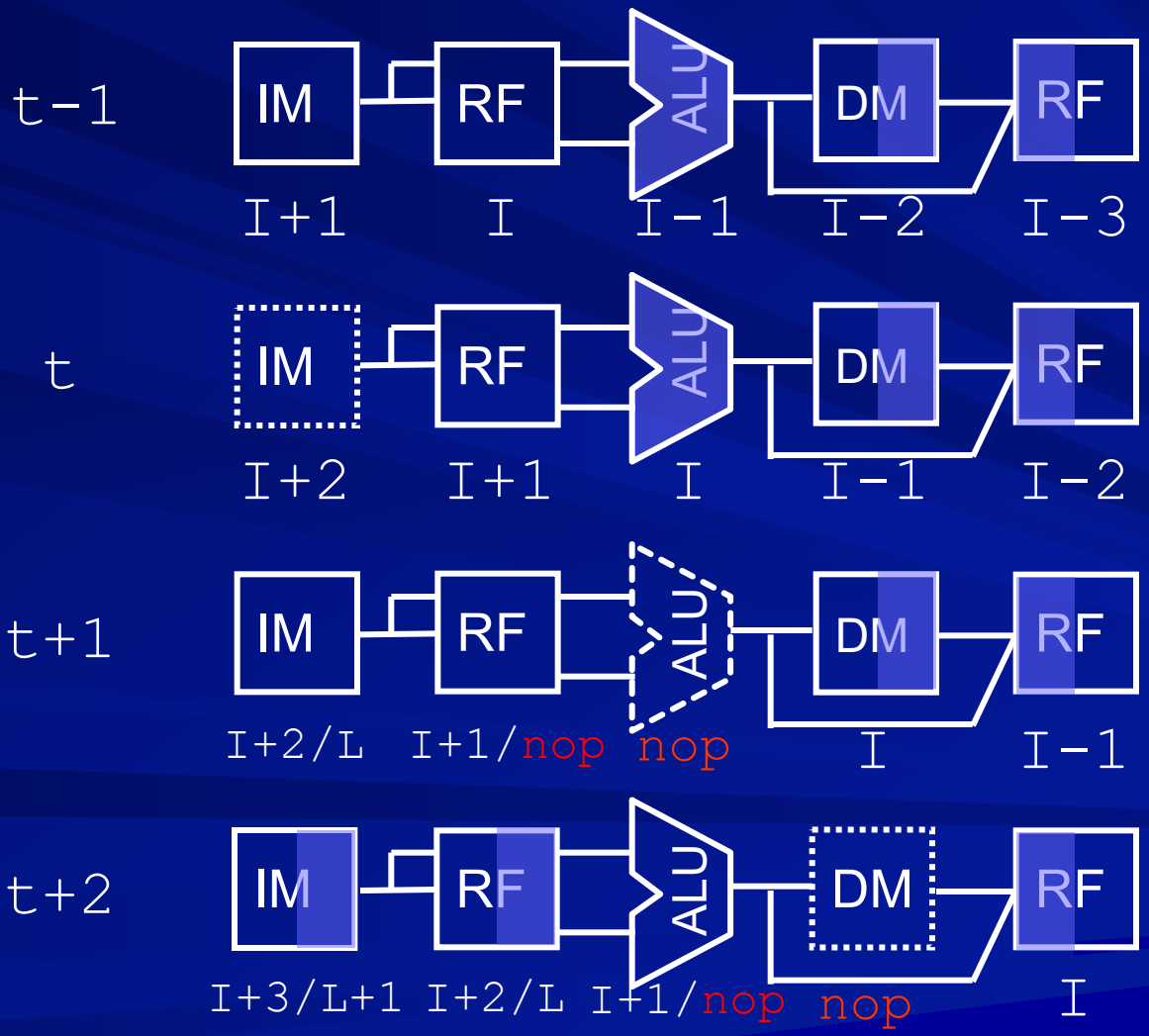
```
I:   beq L
```



Stalls due to control hazards

freeze: stage-wise view

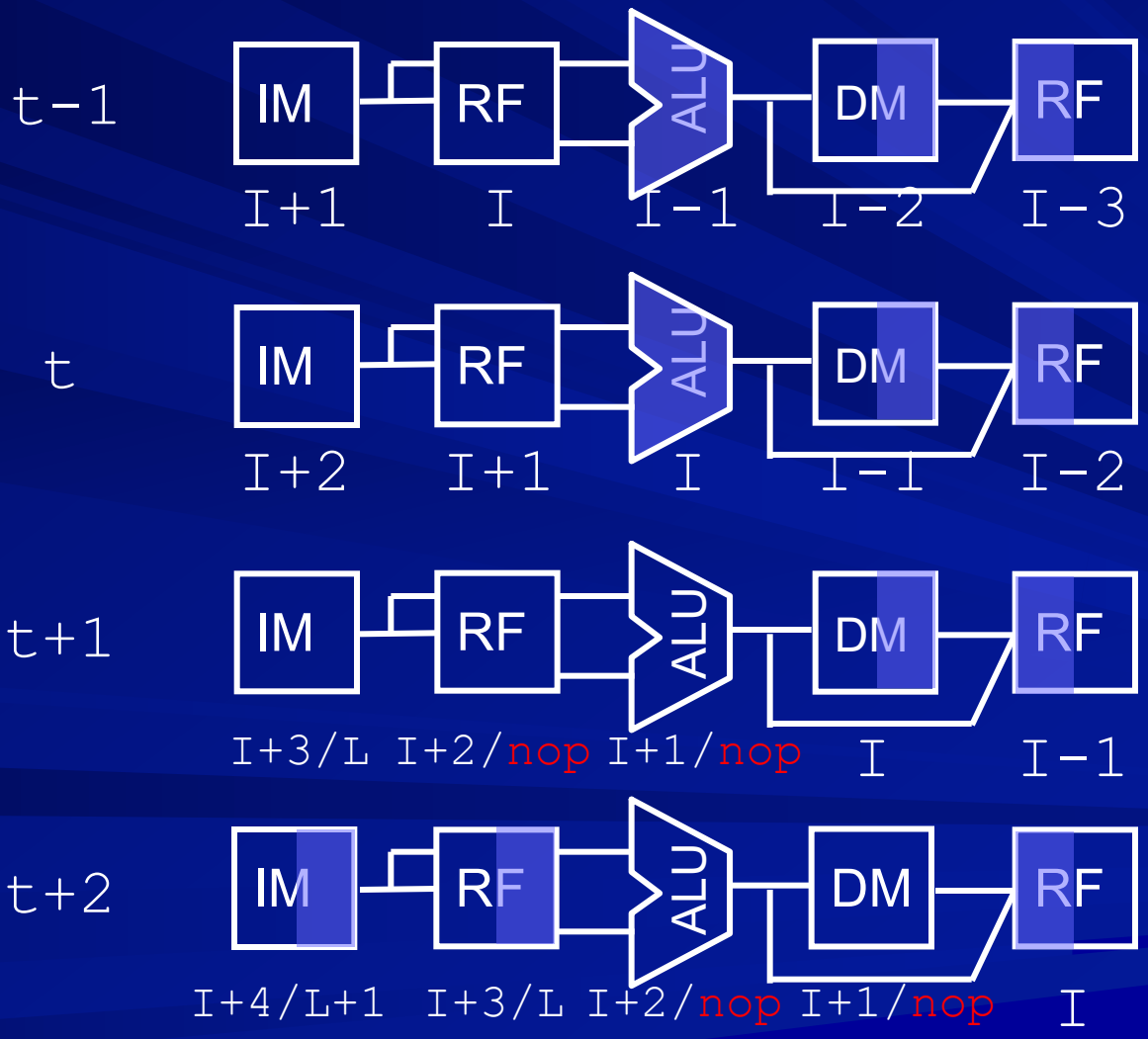
```
I:   beq L
```



Stalls due to control hazards

continue: stage-wise view

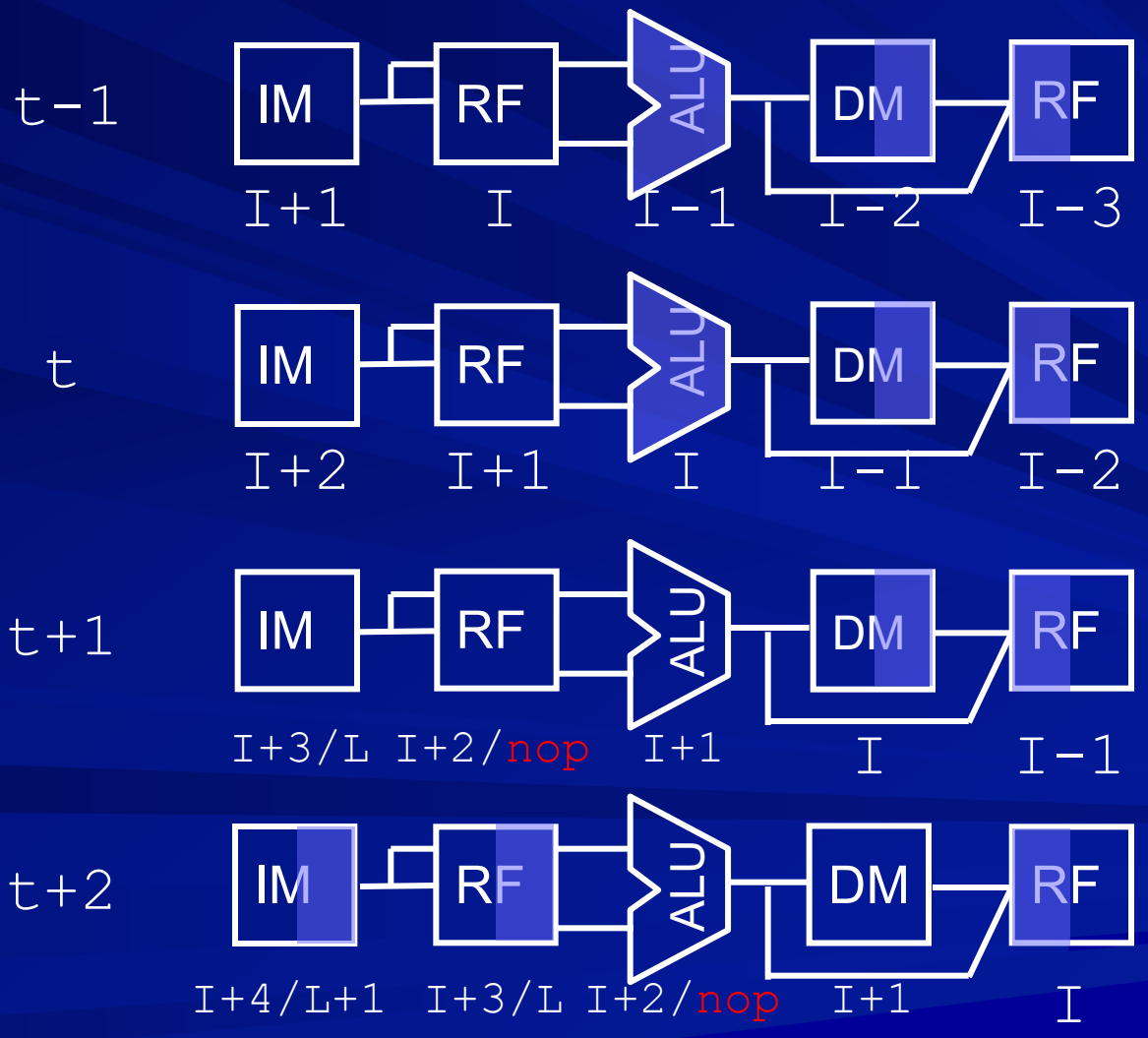
```
I:   beq L
```



Stalls due to control hazards

delayed branch: stage-wise view

```
I:   beq L
```



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*

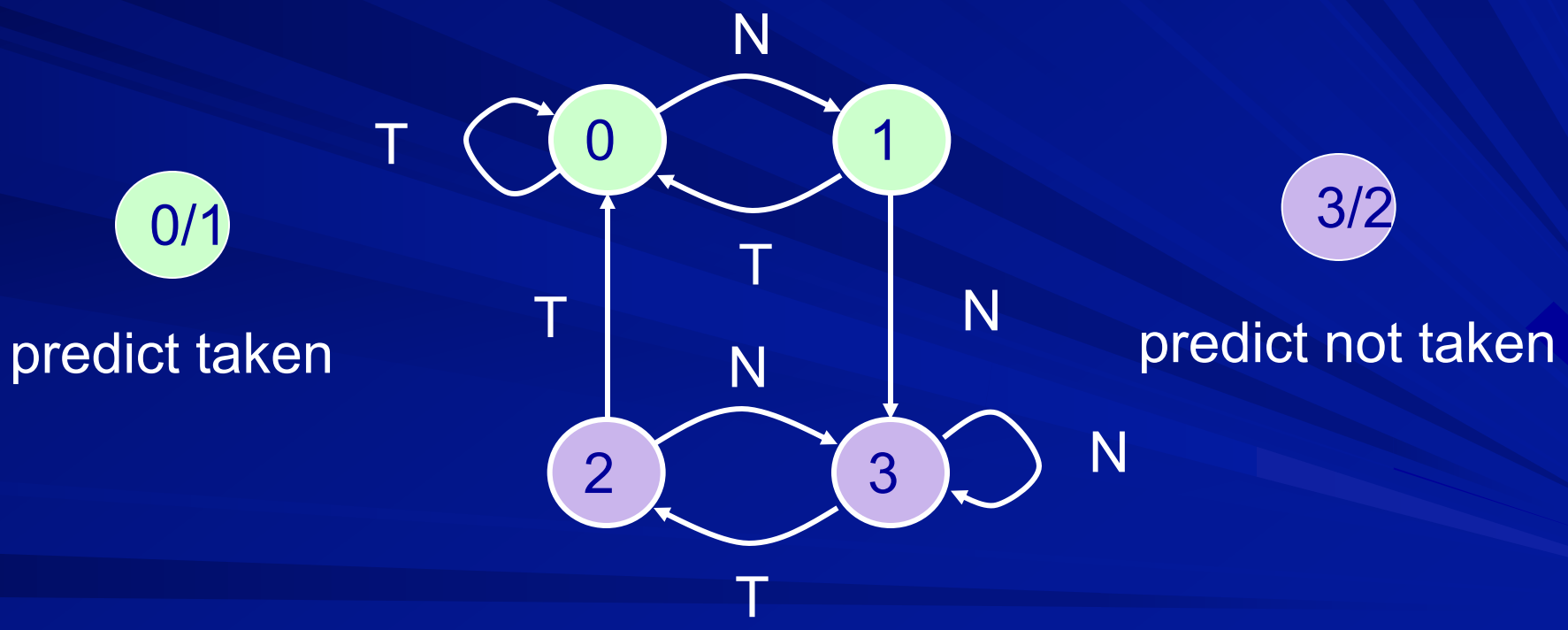
Dynamic Branch Prediction - basic idea

Predict based on the history of previous branch

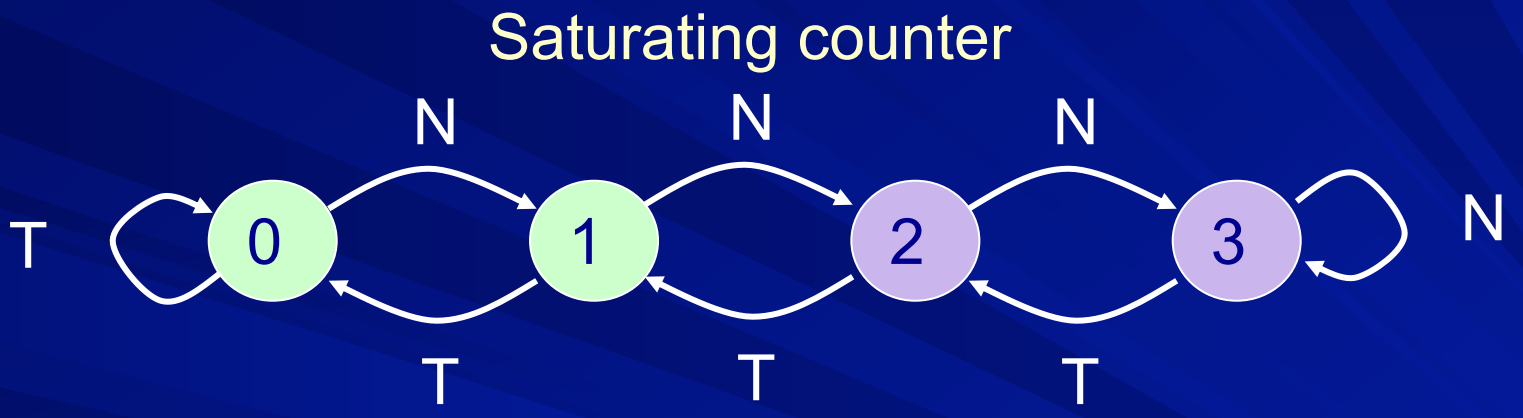
```
loop: xxx
      xxx
      xxx
      xxx
      b<cond> loop
```

2 mispredictions for every occurrence of the loop

Dynamic Branch Prediction - A 2-bit prediction scheme



Another 2-bit prediction scheme



0/1

predict taken

3/2

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?

Correlation between branches

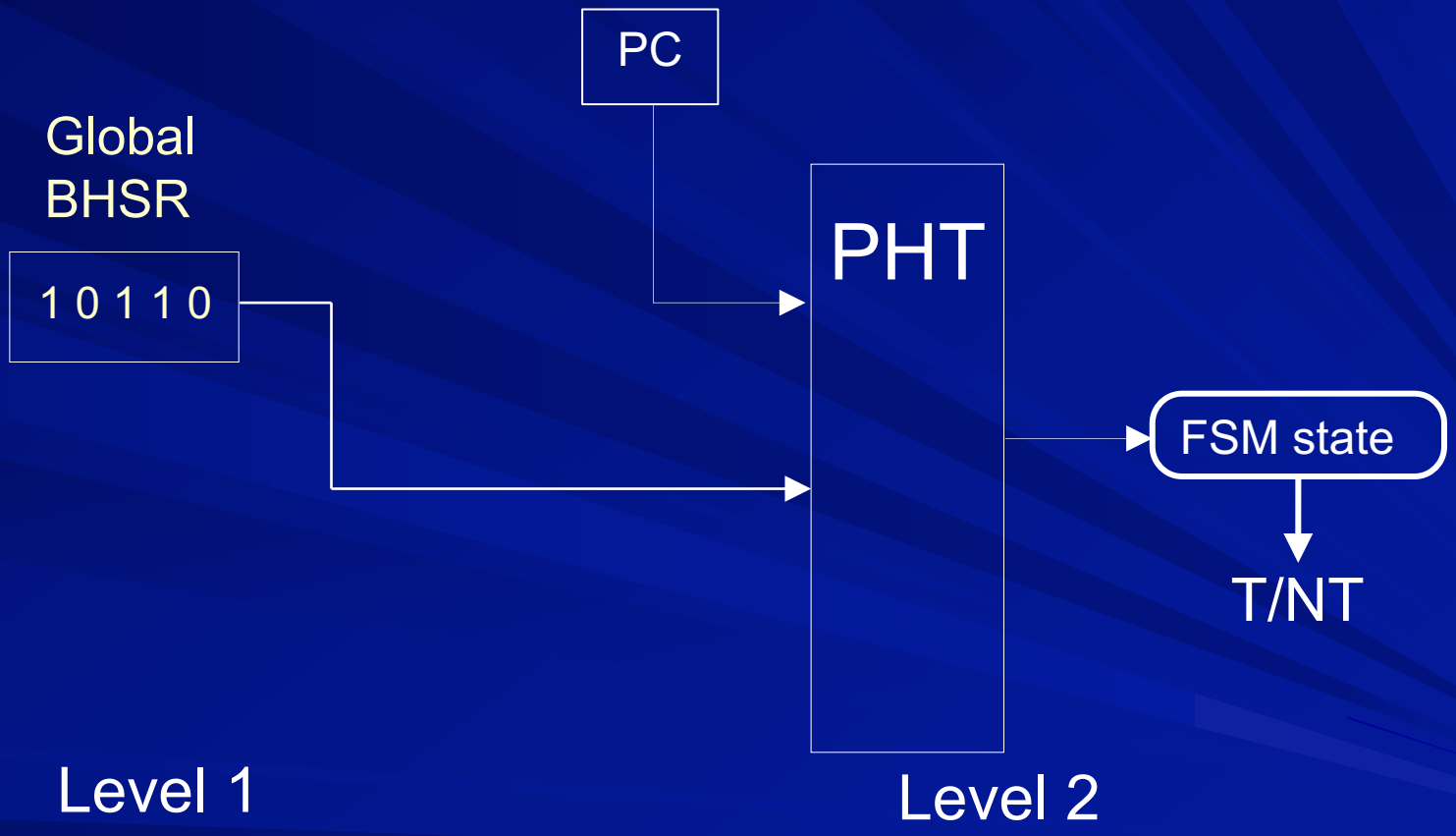
```
B1: if (x)
        ...
B2: if (y)
        ...
    z = x && y
B3: if (z)
        ...
```

- 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

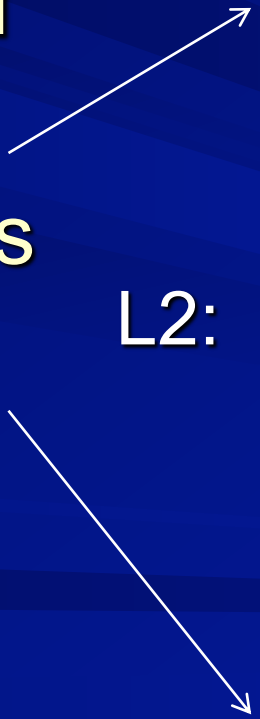
```
        mov   r1, #1          1
        mov   r2, #100        2
L1:     str   r1, [r2, #0]    5       13  .. 77
        add   r1, r1, #1      6
        add   r2, r2, #4      7
        cmp   r1, #11         9
        bne   L1              10      18  .. 82
        sub   r1, r1, #1      85
        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

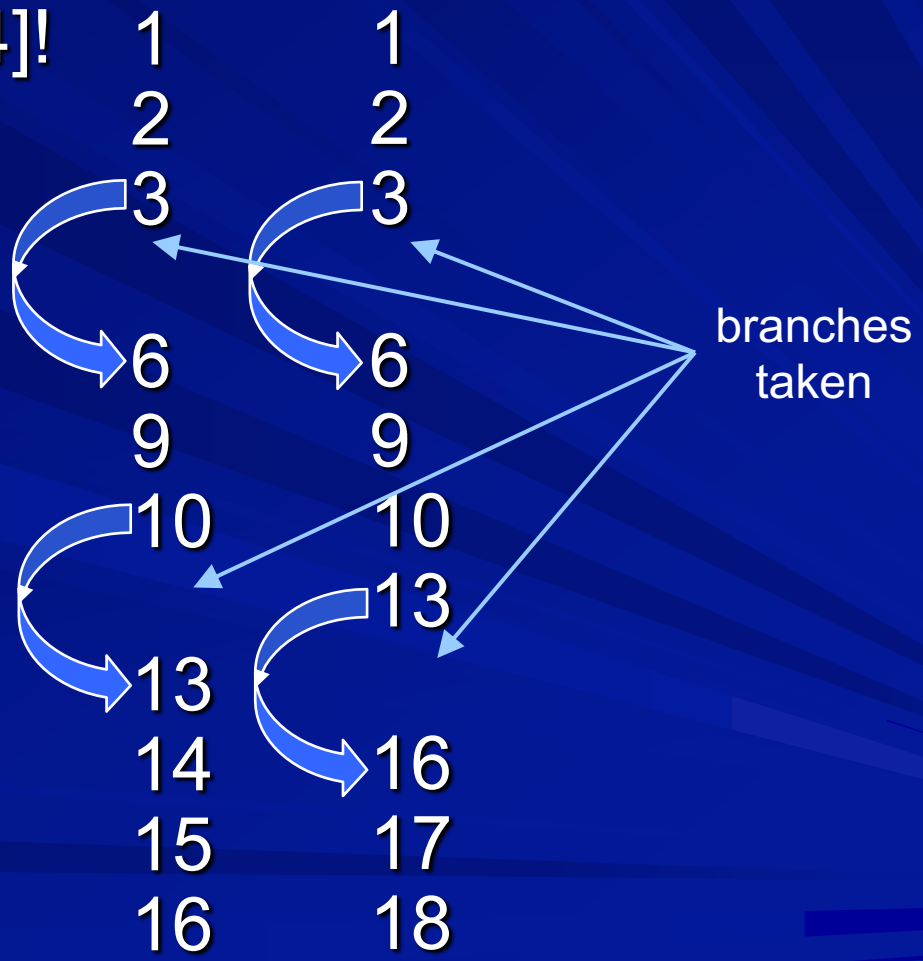
- Original code without stalls 128
- Original code with stalls 254
- Execute inline instr after branch 234
- After first re-ordering 214
- After second reordering 182
- Using delayed branch 162

Branch elimination example

(find max element in array)

```
func:  str   lr, [sp, #-4]!
       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
```

data dependence



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
```

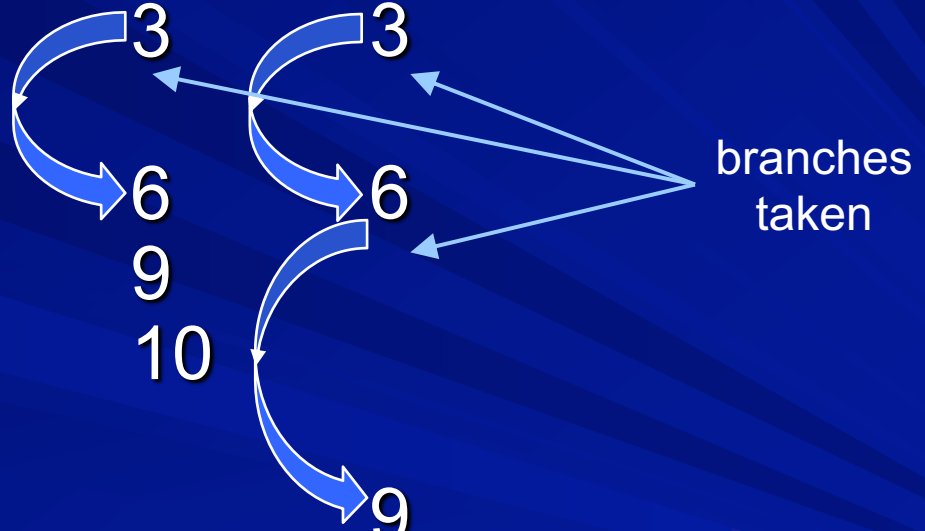
```
L2:   movlo r0, r3
```

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]!   1    1
       cmp r0, r1             2    2
       bne  L1                3    3
       b     Ret
L1:    bhs  L2                6    6
       sub  r1, r1, r0        9
       bl    func             10
       b     Ret
L2:    sub  r0, r0, r1              9
       bl    func                   10
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