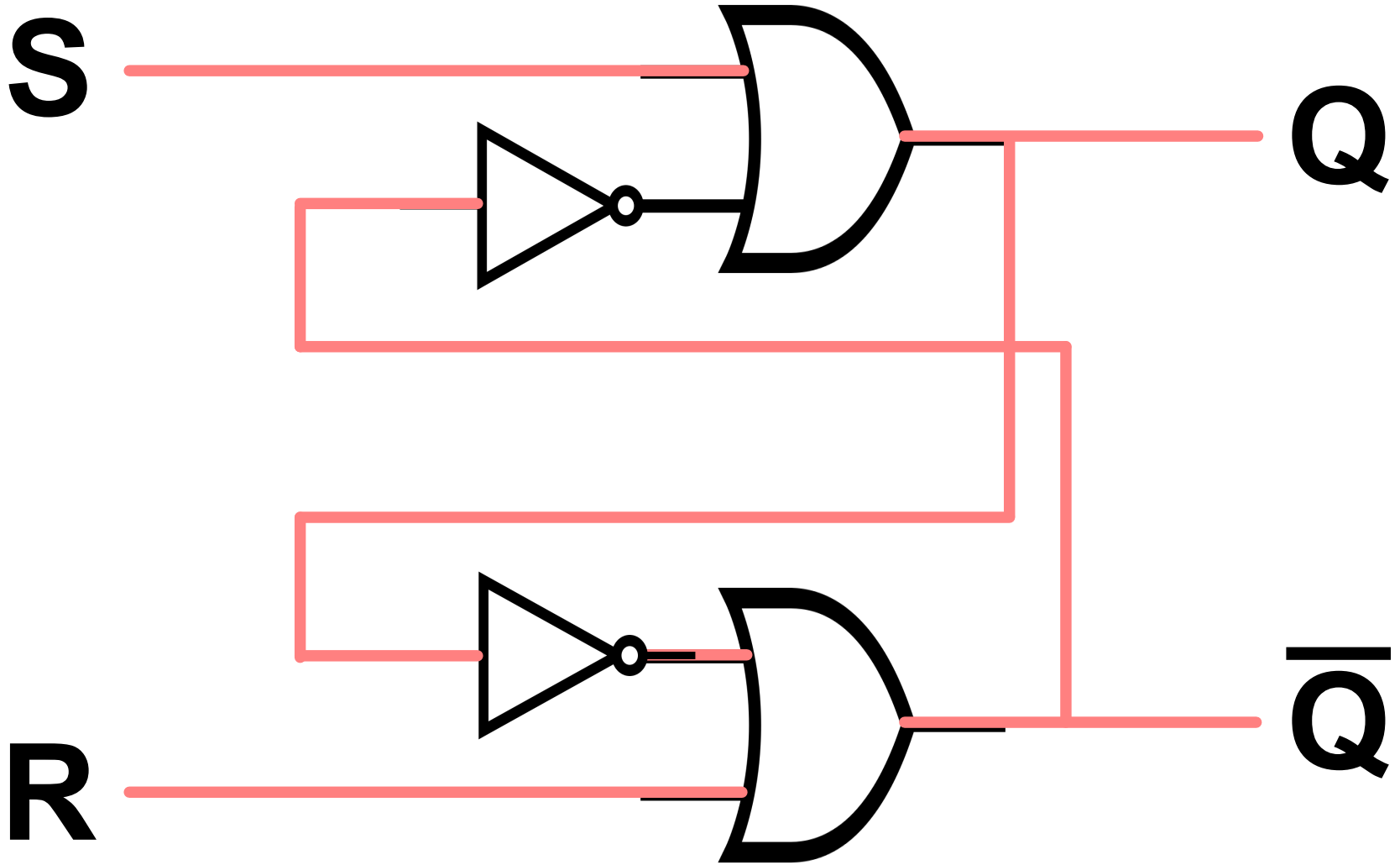
CMP 334 (4/8/19)

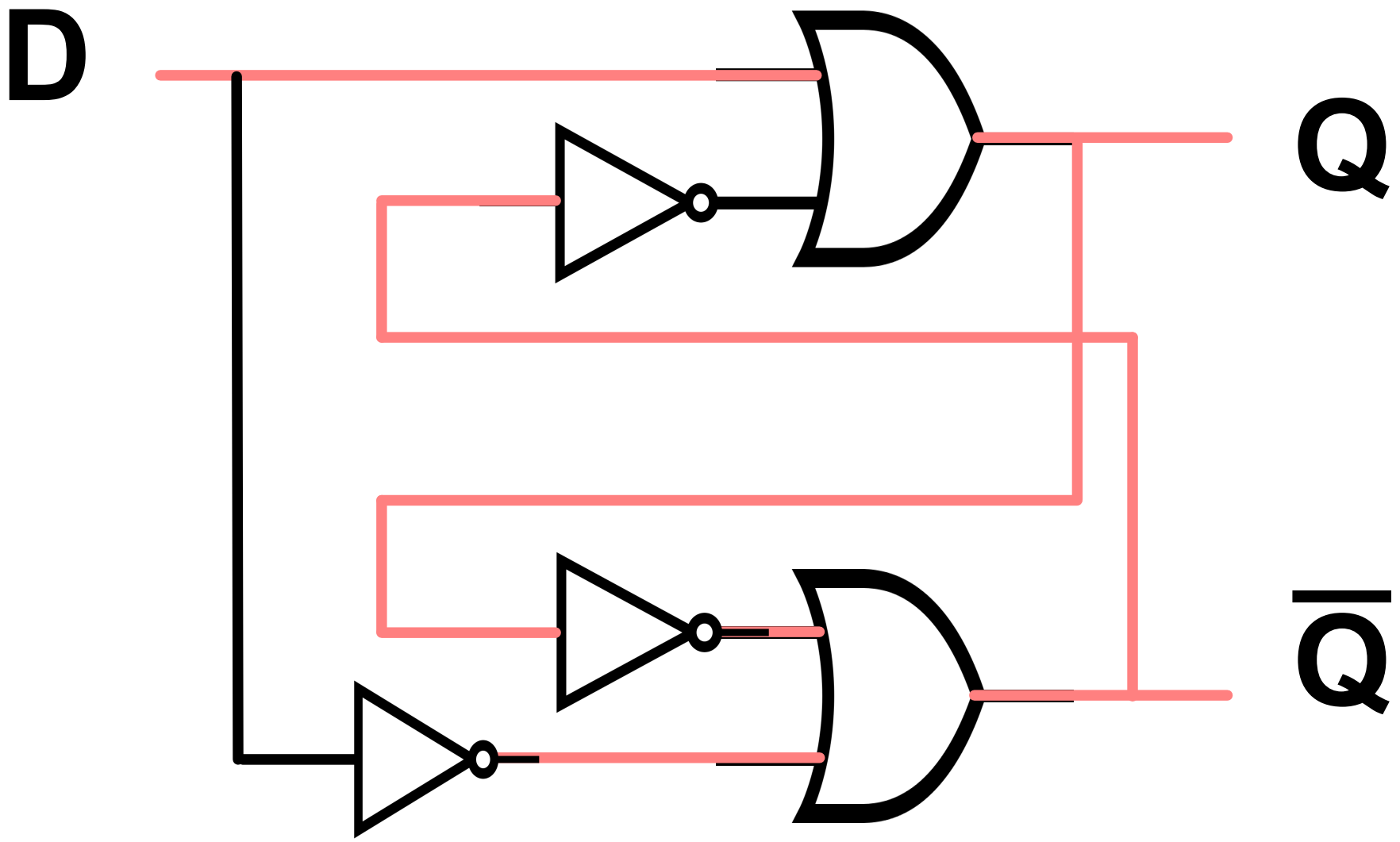
- Synchronous circuits
  - **S-R**, **J-K**, **T** flip flop;
  - Counters; **TOY** instruction cycle counters
- **TOY** AL programming (review)
  - HW 7, HW 11, and HW 12
- **TOY** API "single cycle" implementation
  - Datapath
    - Separate instruction and data memory (caches)
  - Control signals
- **TOY** API pipeline implementations

HW 13

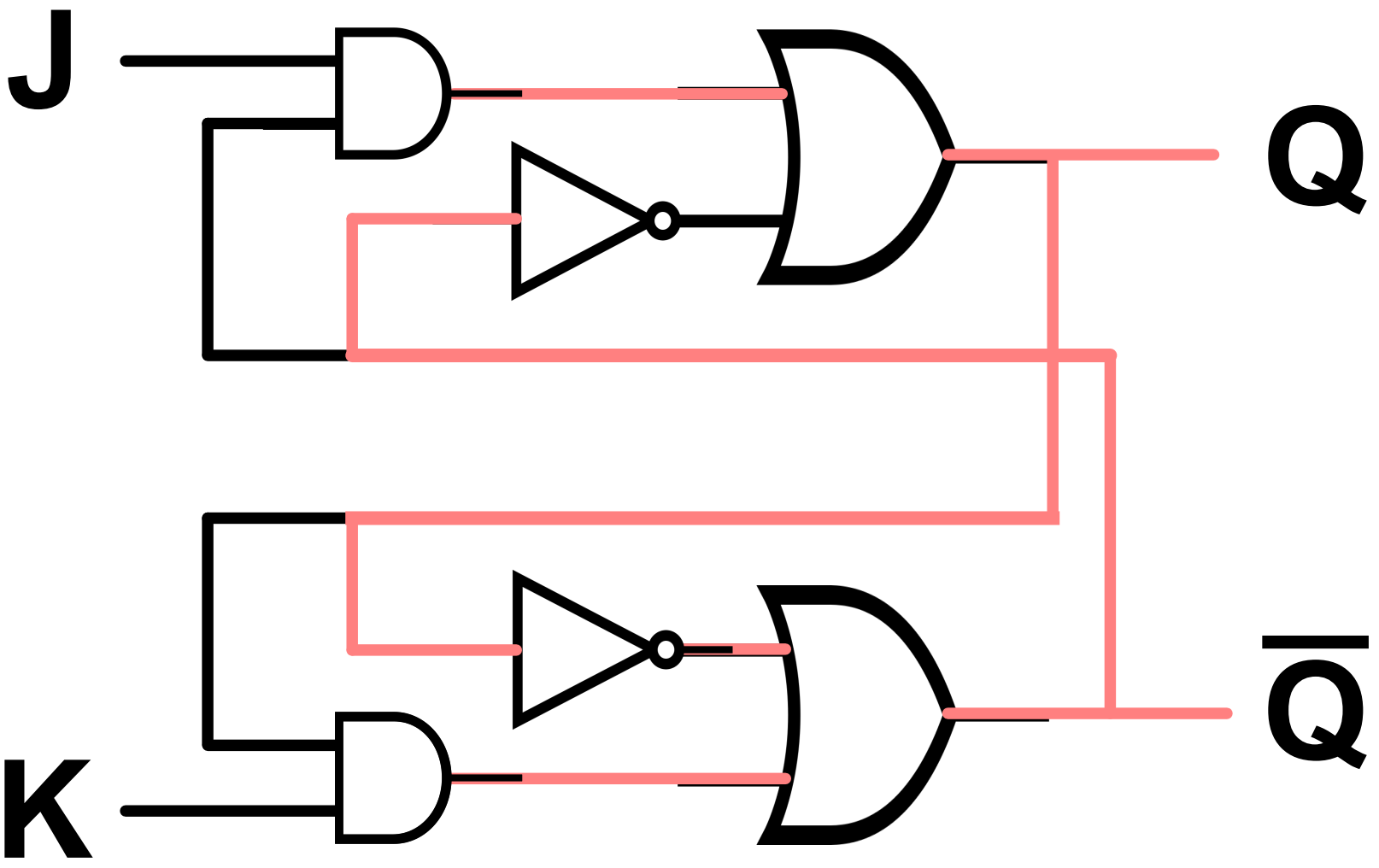
S-R Latch



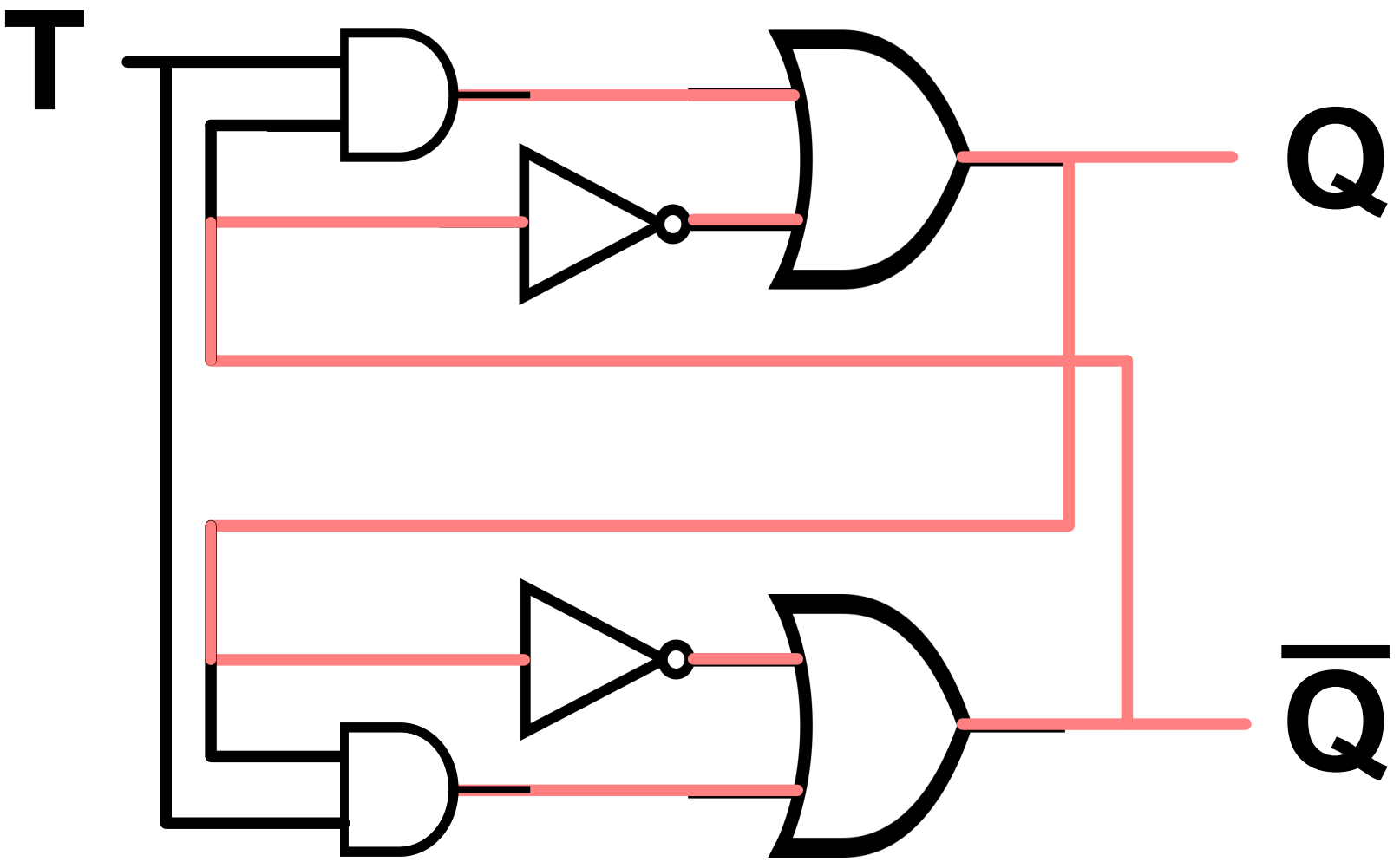
D Latch



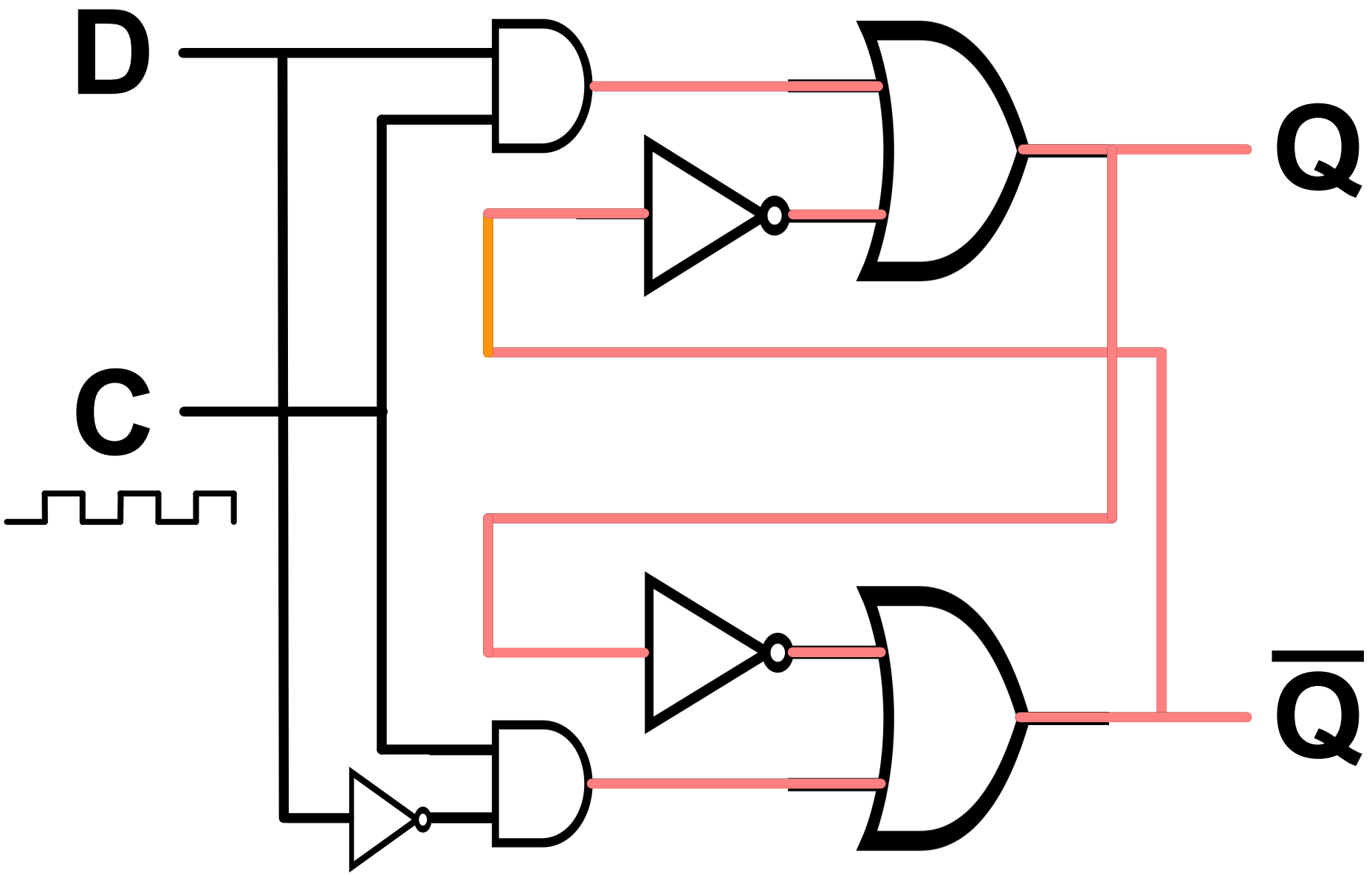
J–K Latch



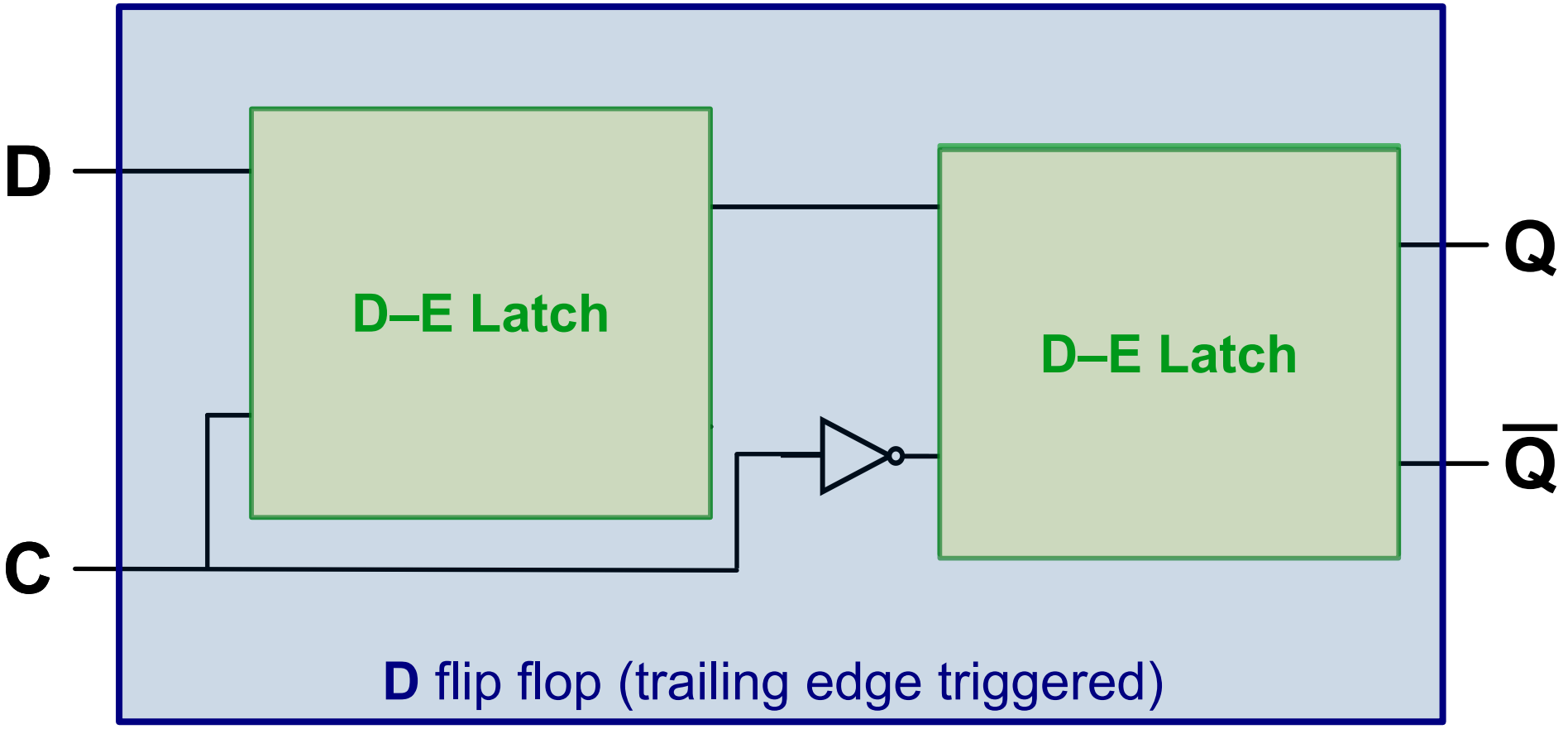
T Latch



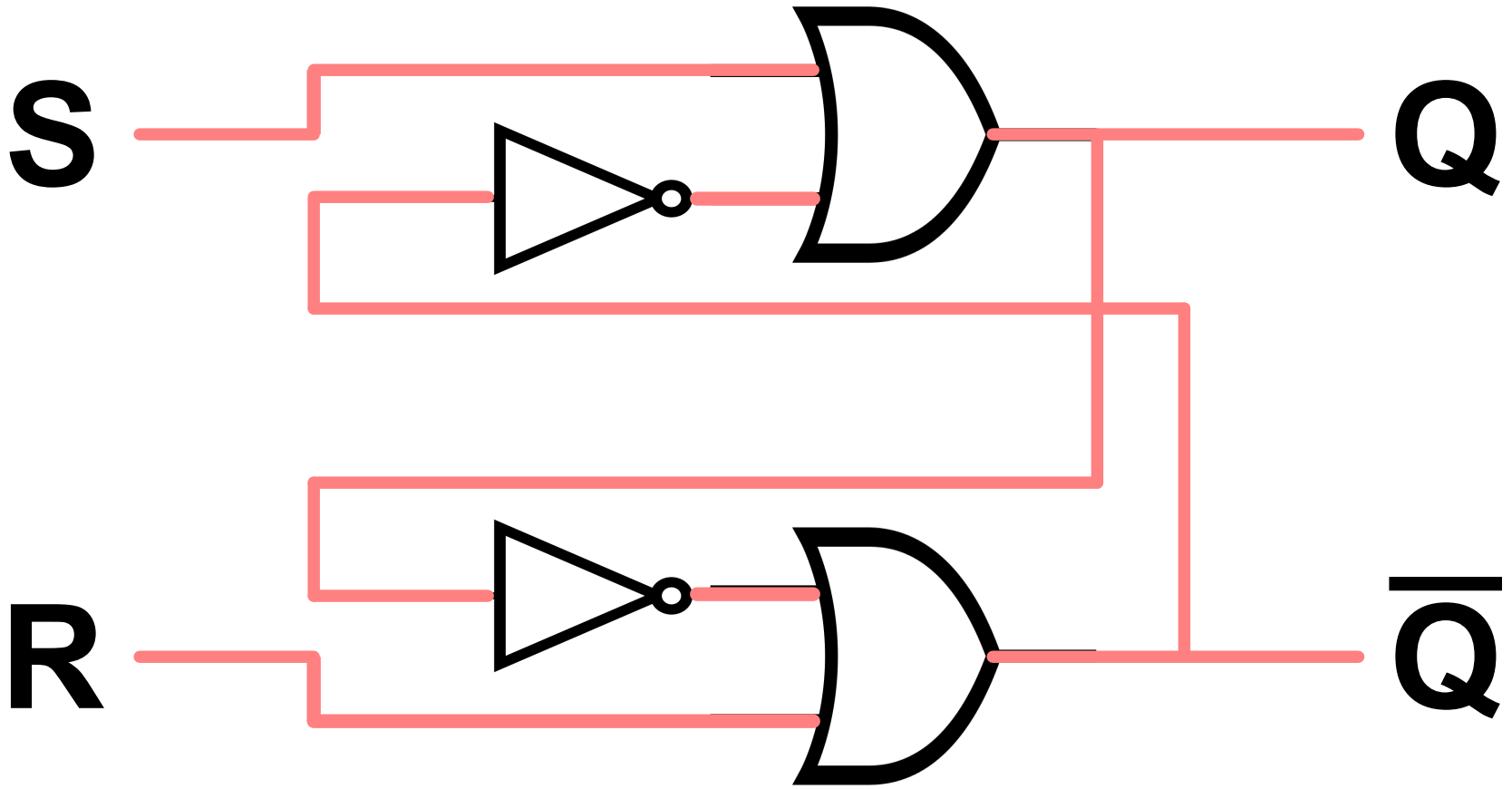
Clocked **D** Latch



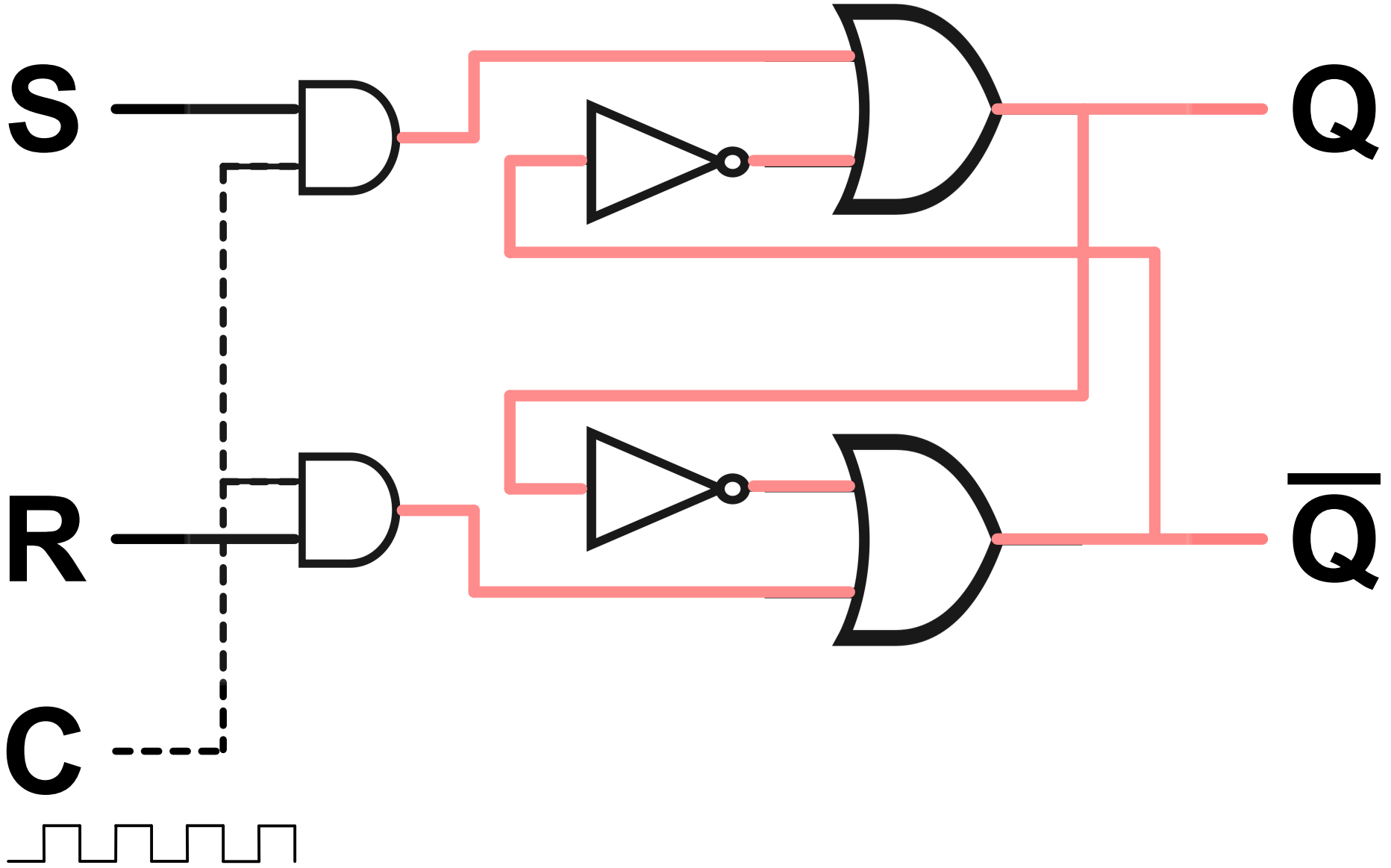
D Flip Flop


D
C
D–E Latch
D–E Latch
Q
$\overline{Q}$
D flip flop (trailing edge triggered)

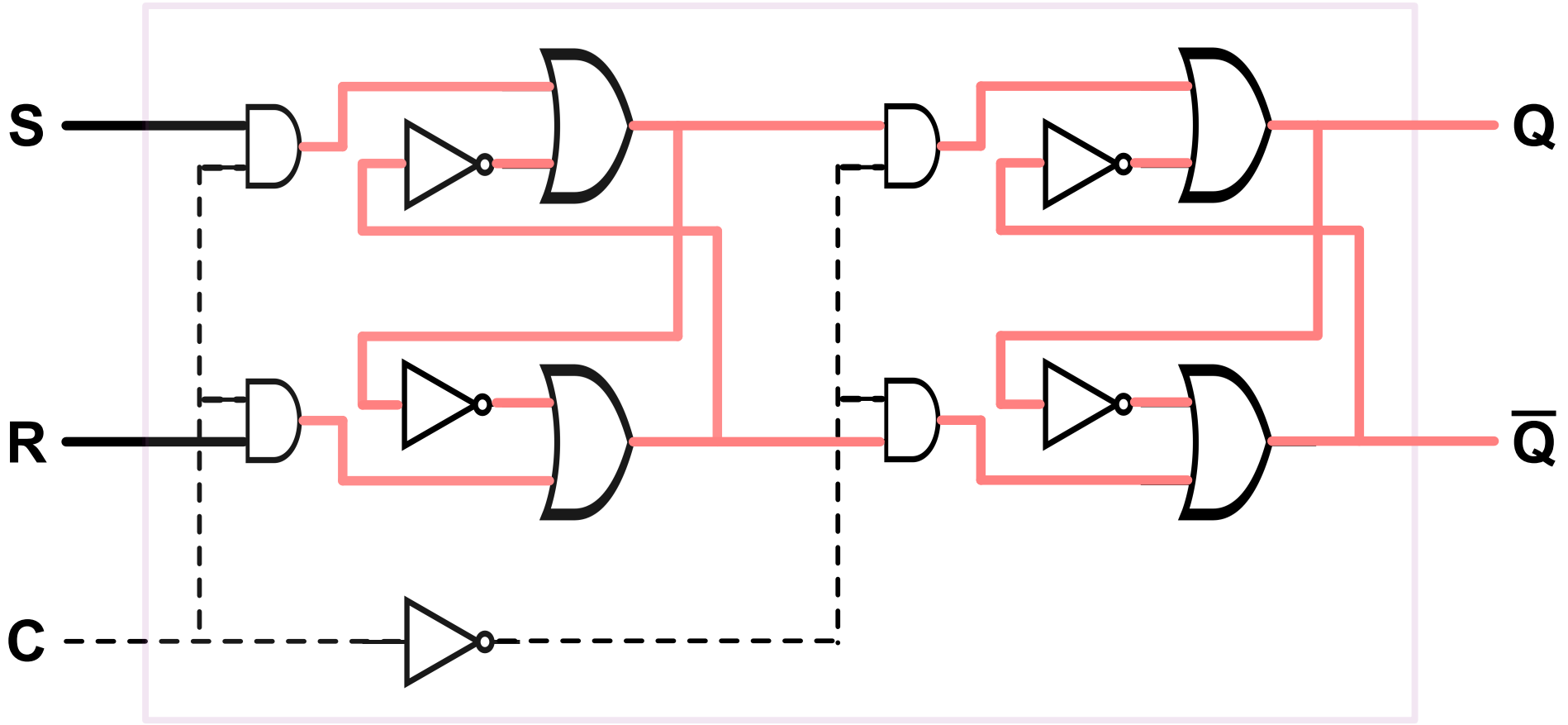
S-R Latch



Clocked **S-R** Latch



S-R Flip Flop

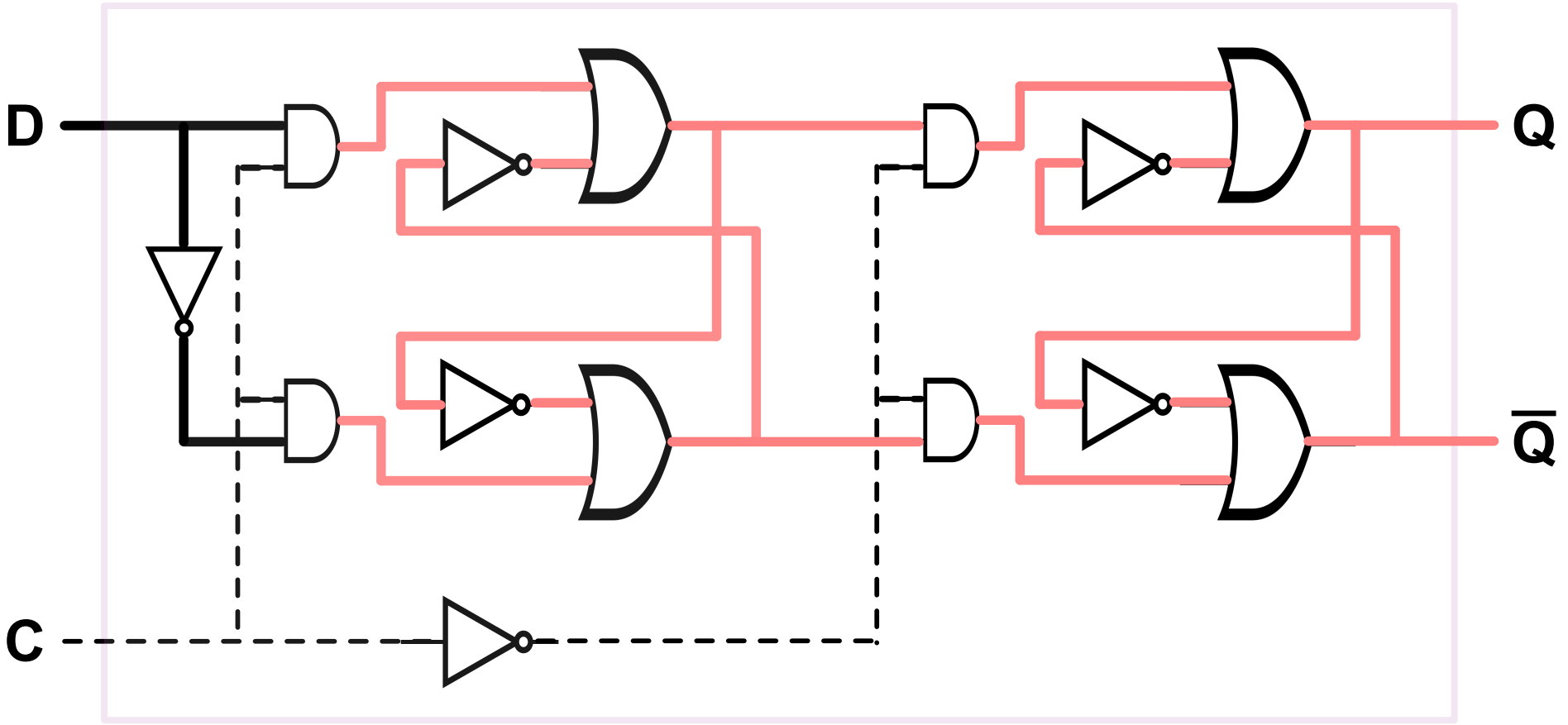


S-R Flip Flop

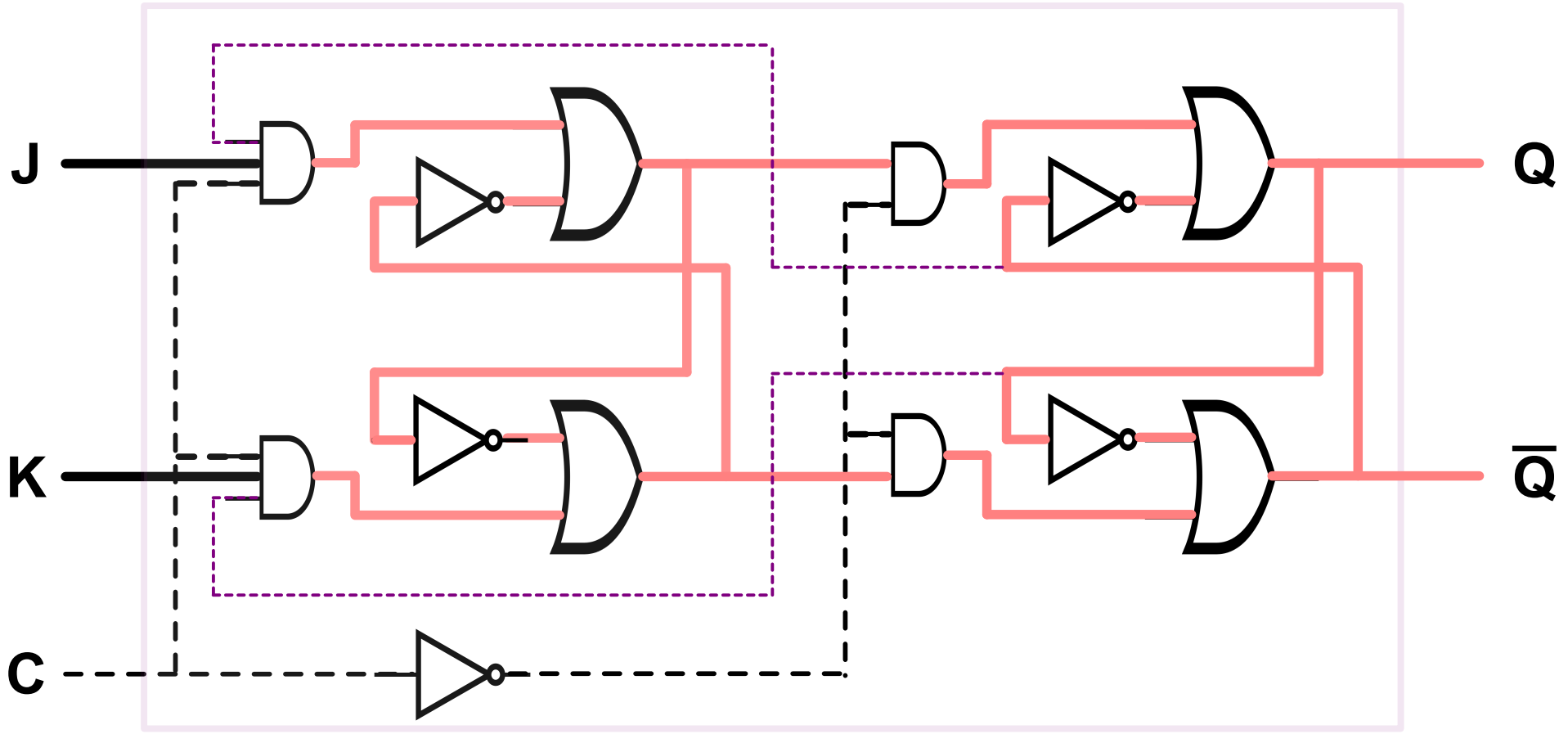
S	R	Q	Q	$\overline{Q}$	
0	0	0	0	1	LATCH
0	0	1	1	0	
0	1	0	0	1	RESET
0	1	1	0	1	
1	0	0	1	0	SET
1	0	1	1	0	

Forbidden

D Flip Flop



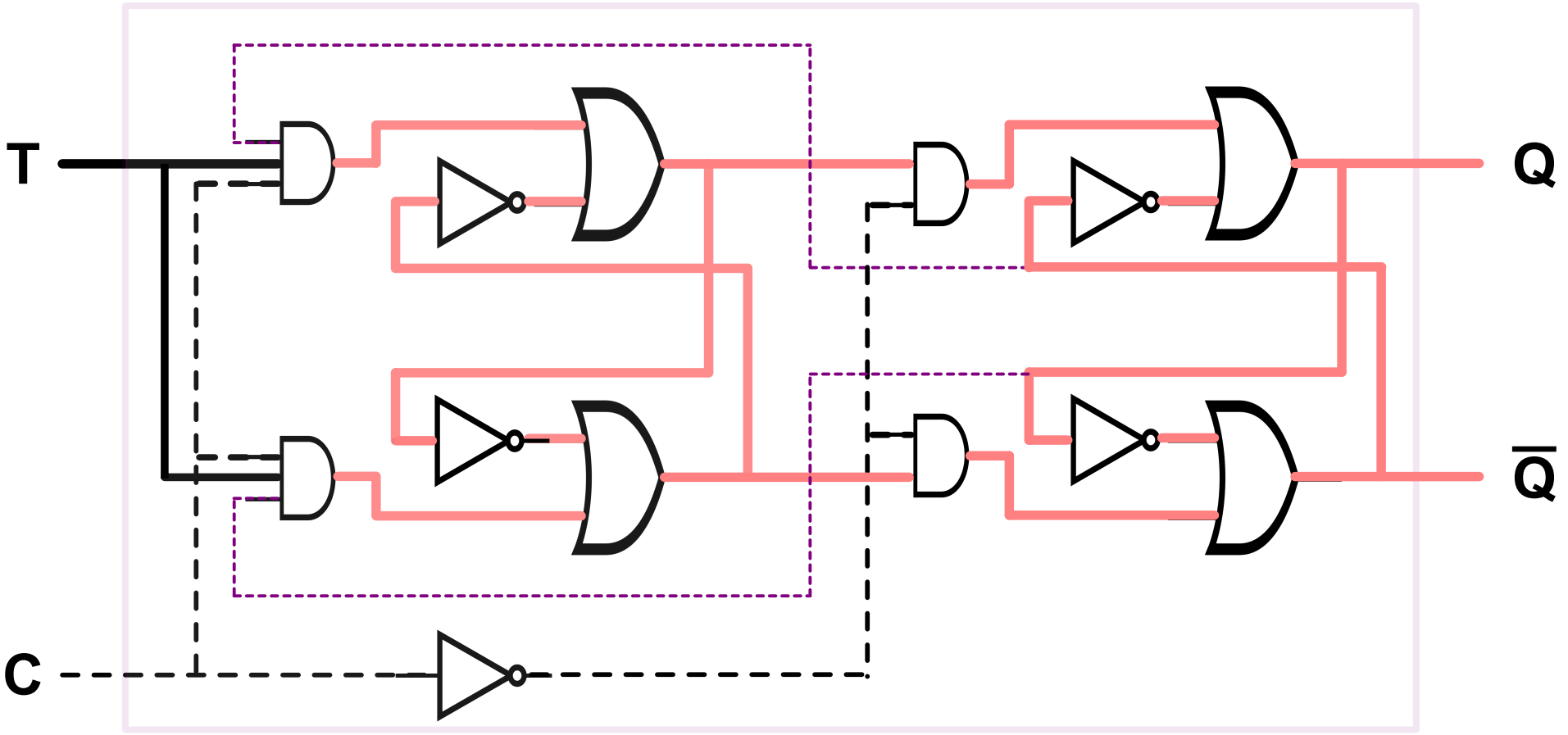
J-K Flip Flop



J-K Flip Flop

J	K	Q	Q	$\overline{Q}$	
0	0	0	0	1	LATCH
0	0	1	1	0	
0	1	0	0	1	RESET
0	1	1	0	1	
1	0	0	1	0	SET
1	0	1	1	0	
1	1	0	1	0	TOGGLE
1	1	1	0	1	

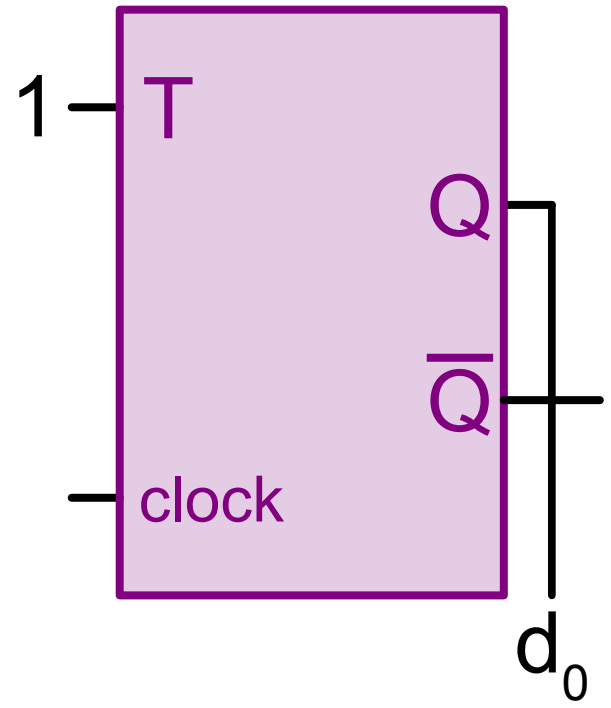
T Flip Flop

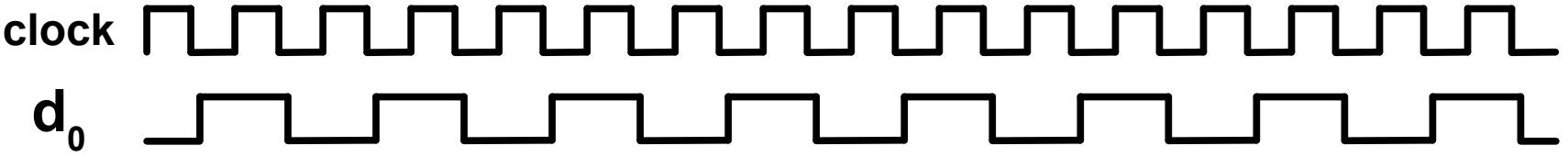


T Flip Flop

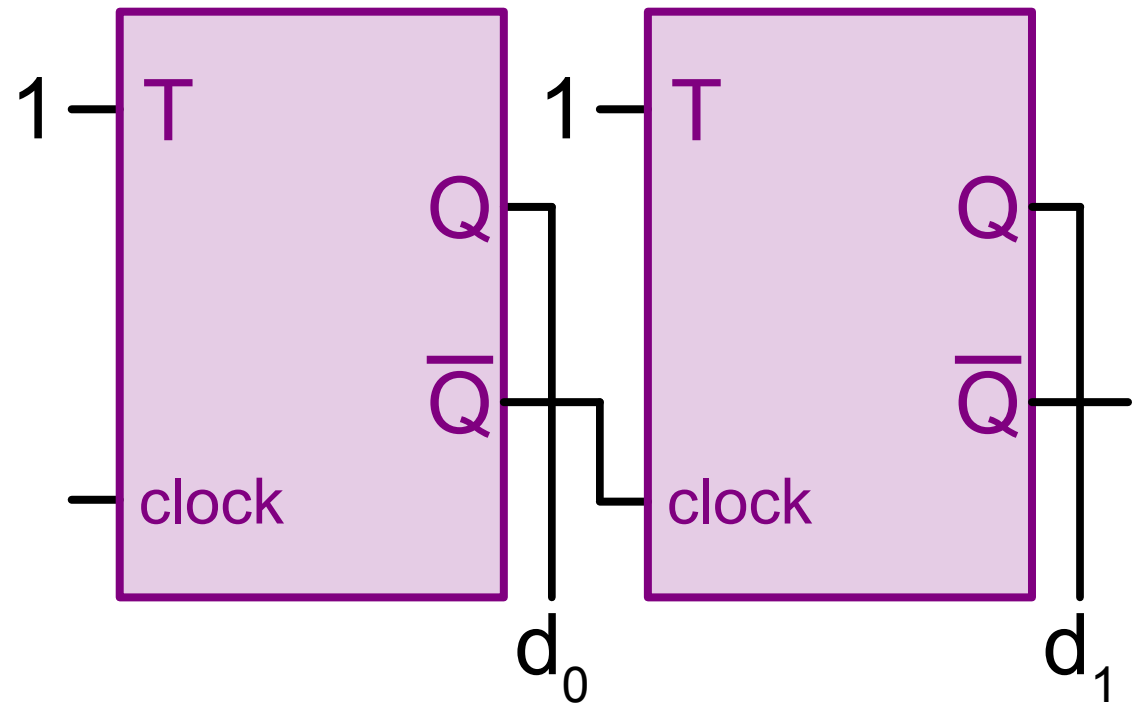
T	T	Q	Q	$\overline{Q}$	
0	0	0	0	1	LATCH
0	0	1	1	0	
Impossible					
Impossible					
1	1	0	1	0	TOGGLE
1	1	1	0	1	

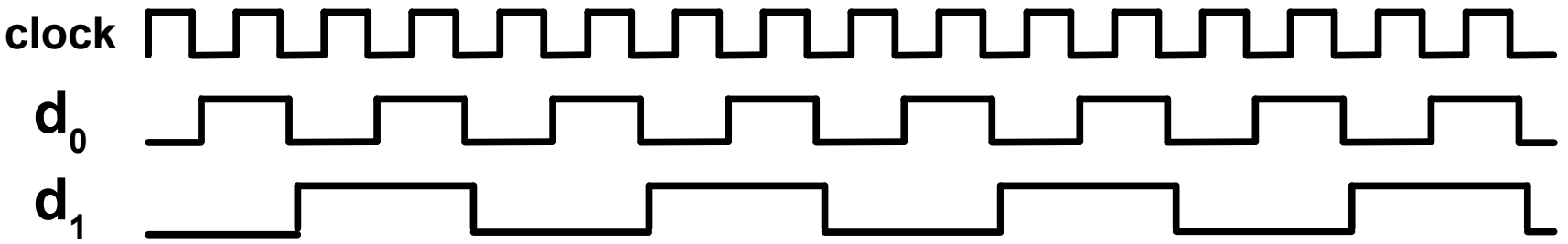
1 Bit Counter (**T** flip flop)



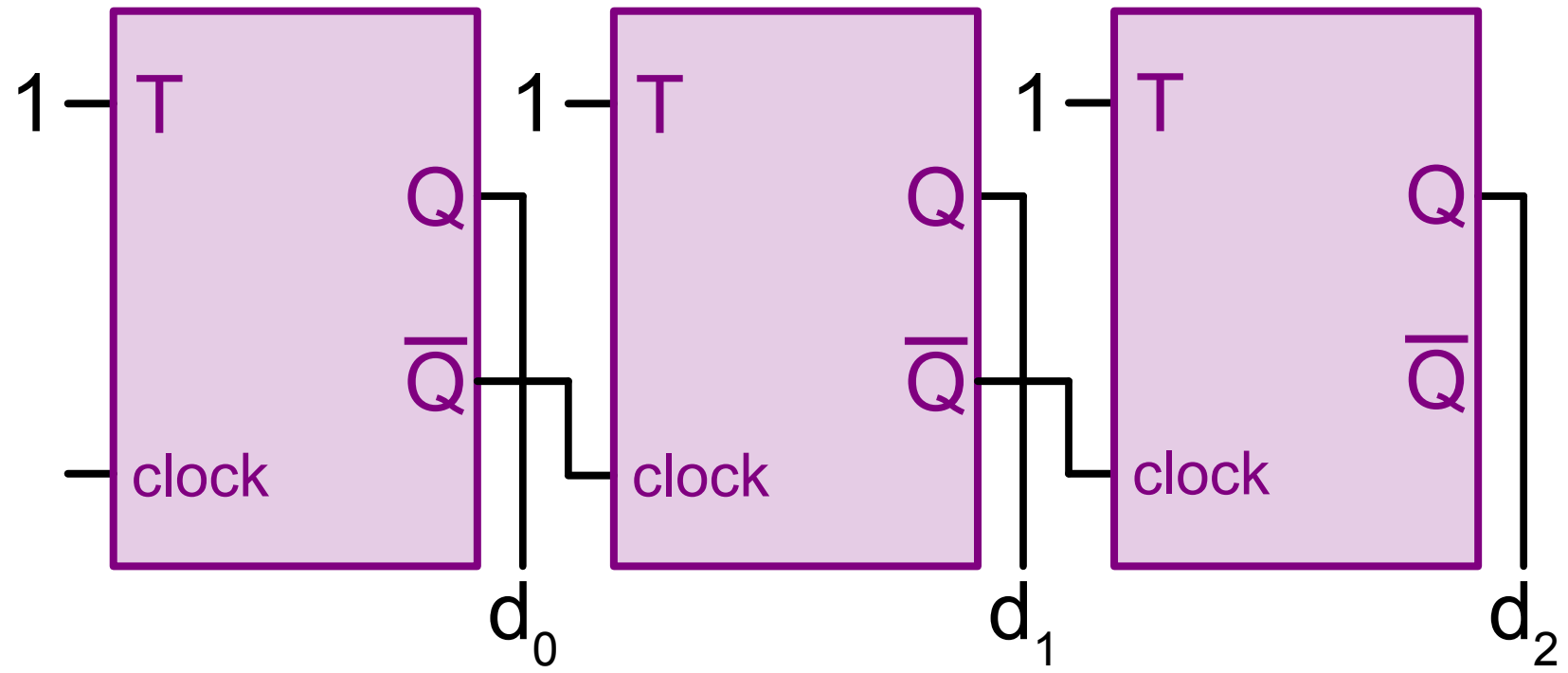


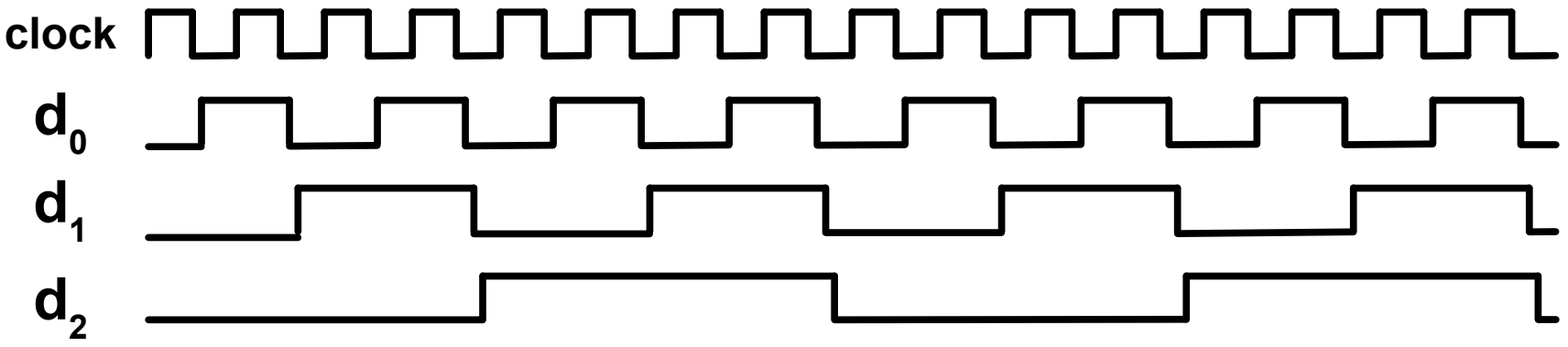
2 Bit Counter



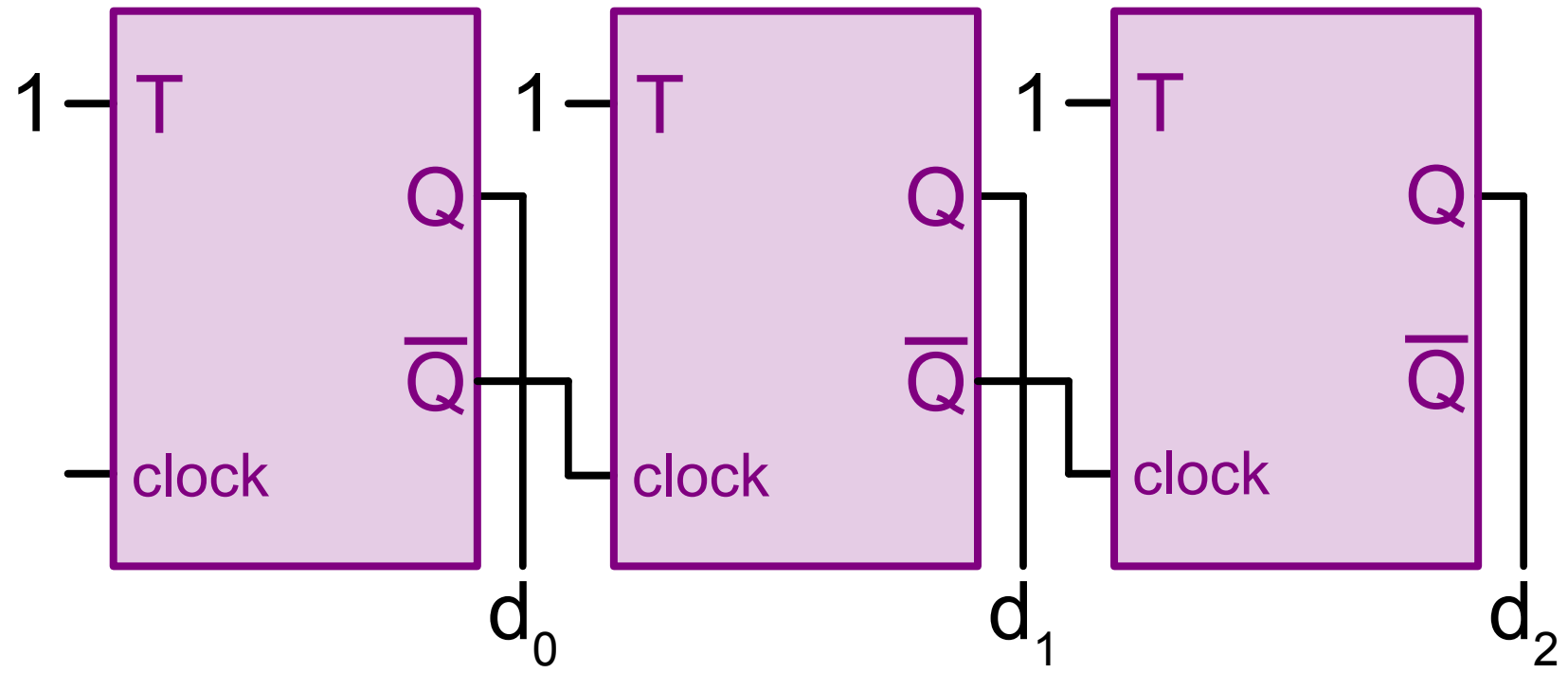


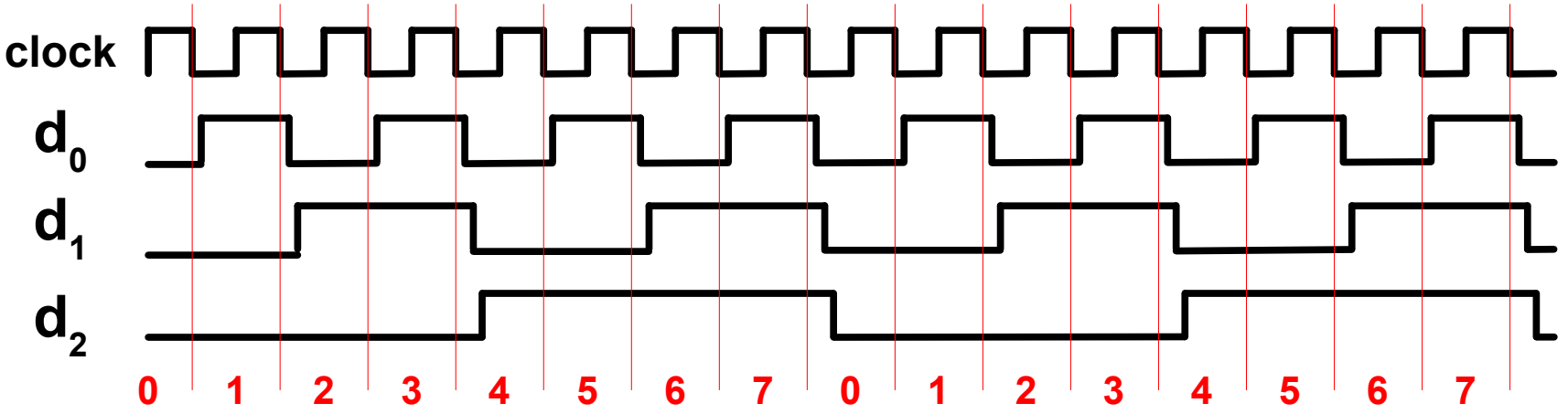
3 Bit Counter



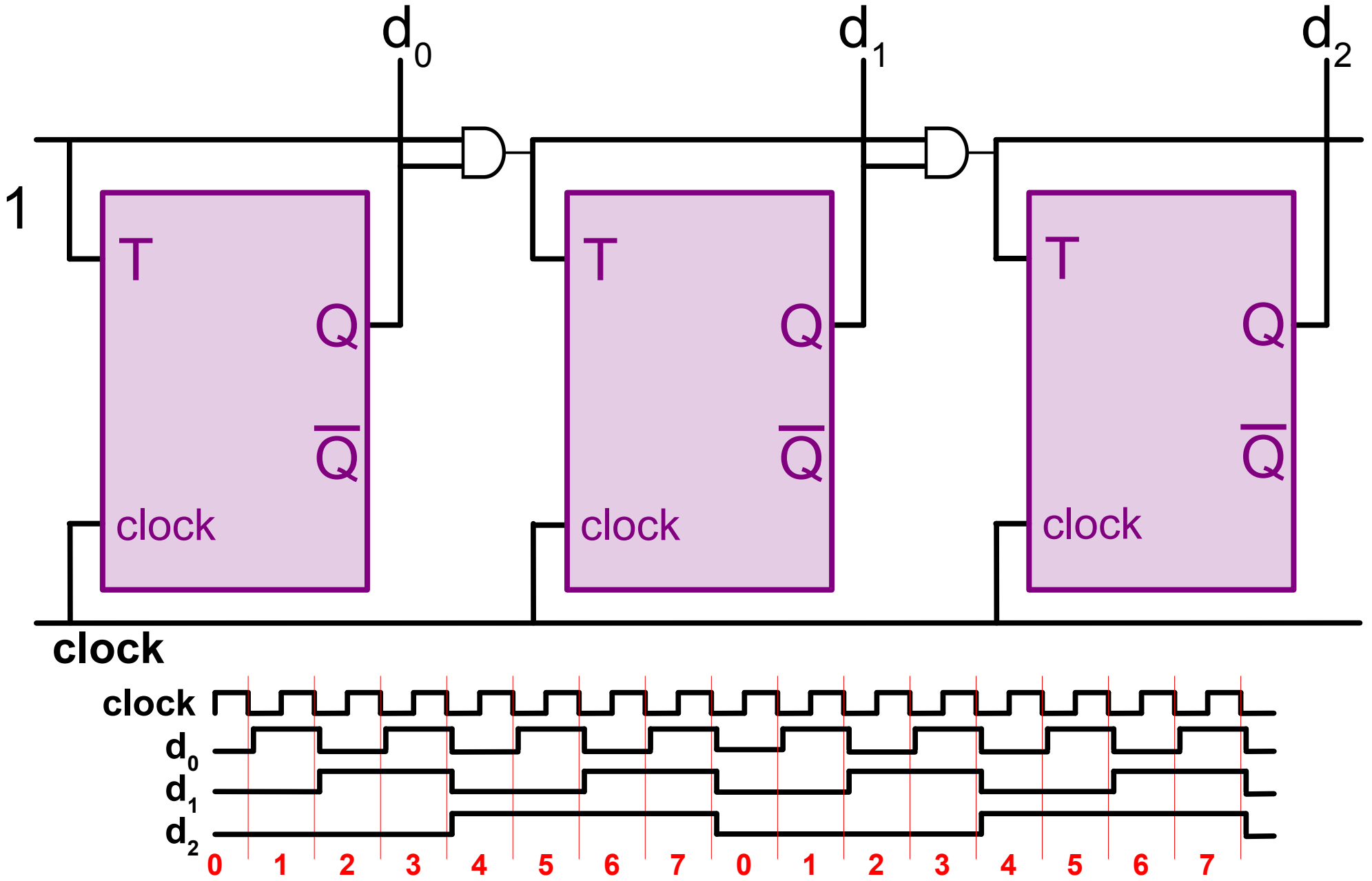


Asynchronous 3-Bit Counter

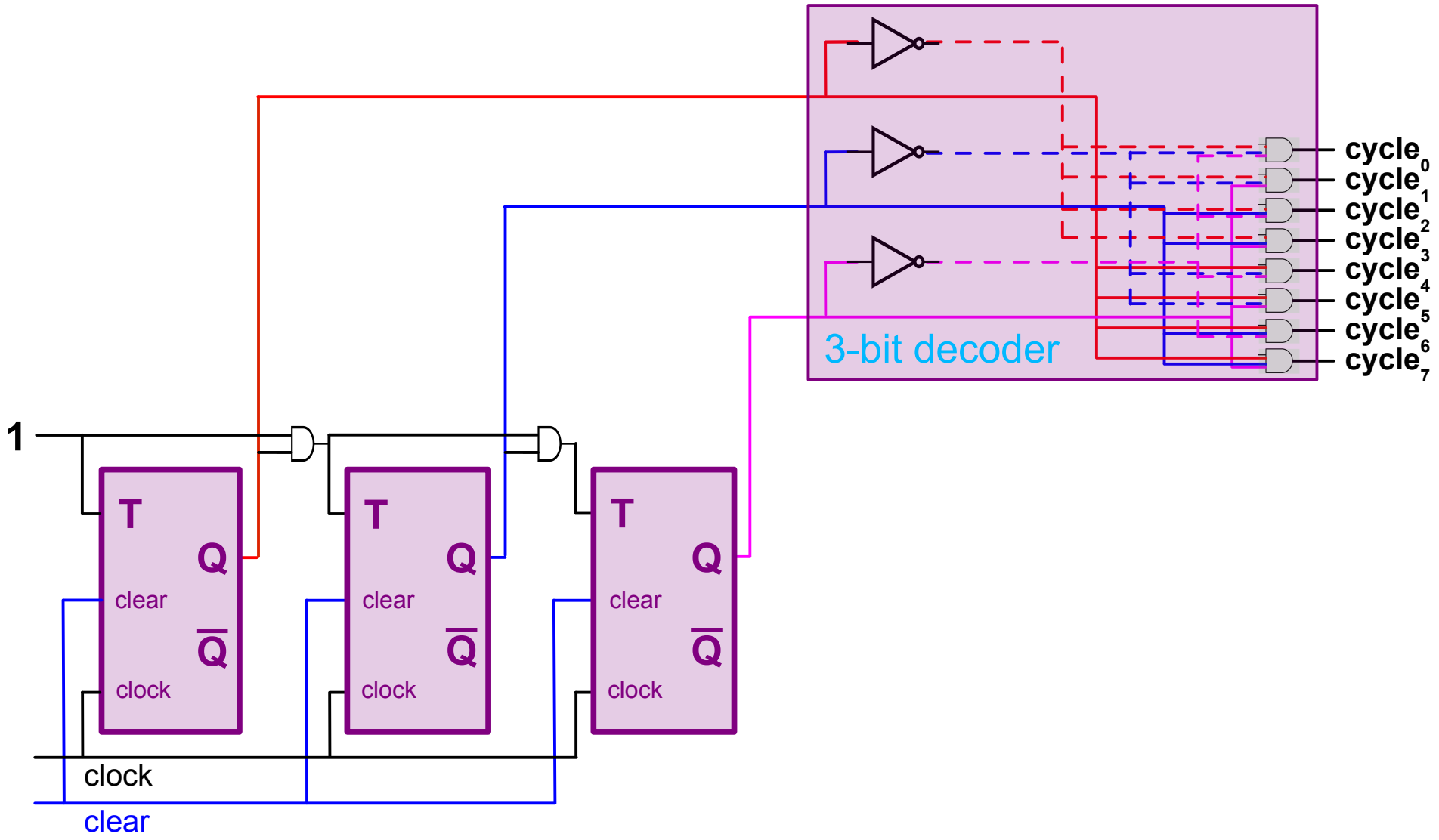




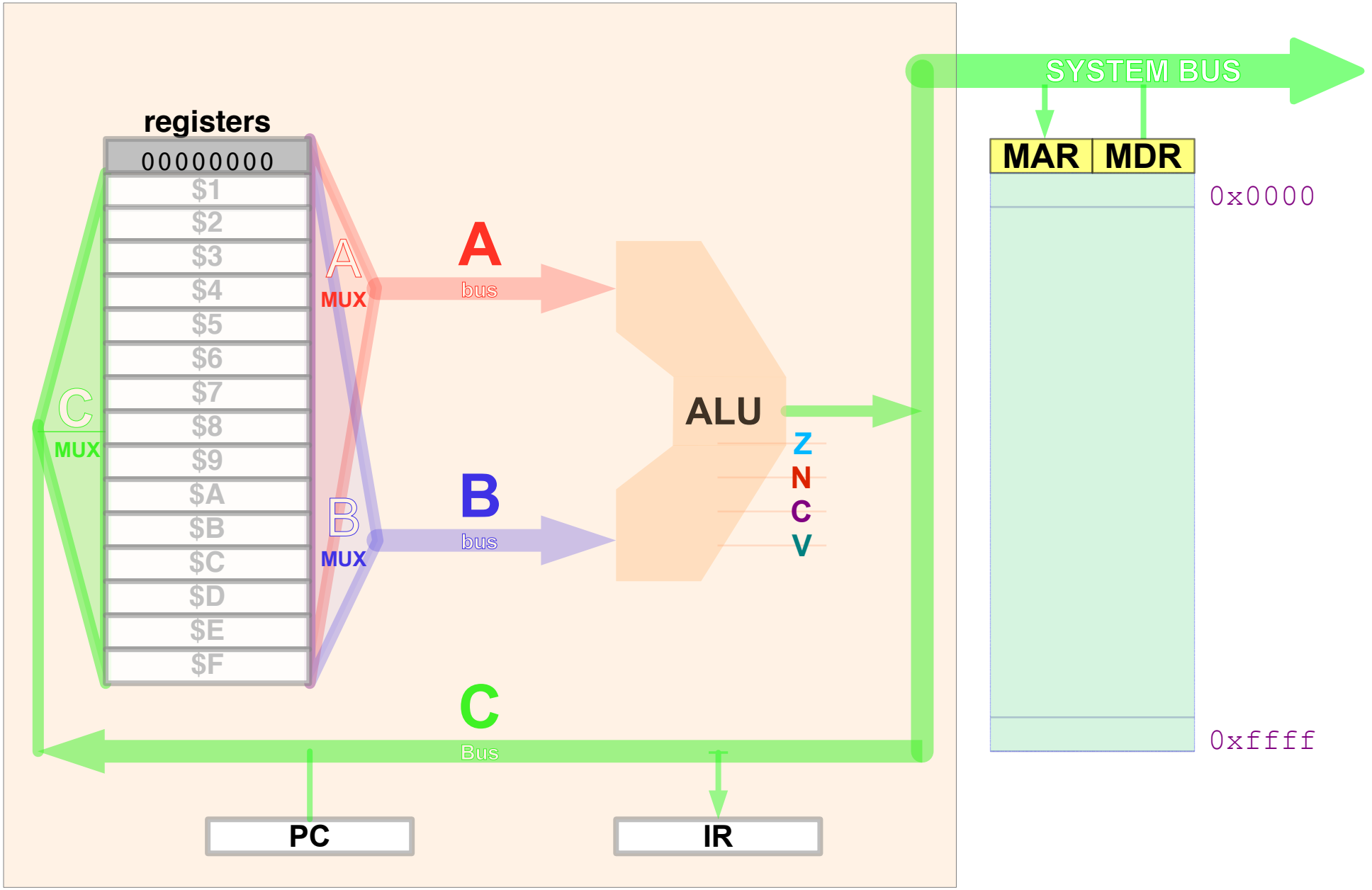
Synchronous 3-Bit Counter



TOY Instruction Cycle Counter



TOY ISA Simple Implementation



TOY Instruction Set Architecture

Reference Card

Immediate values

- s8 — 8-bit signed immediate
- u4 — 4-bit unsigned immediate
- cc — 4-bit condition code

Register File

16 16-bit "registers"

15 real registers: $1 … $F

1 pseudo-register: $0 [$0] = 0

Main Memory

65536 16-bit words

M[n] – n[th] memory address

^M[n] – content of M[n]

Instructions[0]

```
add  $T ← [$A]+[$B]¹
and  $T ← [$A]&[$B]¹
bc   PC ← [PC] + s8                 iff cc
bcl  $L ← [PC], PC ← [$A]¹          iff cc
l    $T ← ^M[[$A]+u4]¹
lwr  $T ← ^M[[$A]+u4]¹              set rsvn
lih  $T15..8 ← s8¹
lis  $T ← s8¹ (sign extended)
nor  $T ← [$A]|[$B]¹
sl   $T ← [$A] << u4¹
srs  $T ← [$A] >> u4¹
sru  $T ← [$A] >>>u4¹
st   M[[$A]+u4] ← [$S]
stc  M[[$A]+u4] ← [$S]⁴             iff rsvn
sub  $T ← [$A]-[$B]¹,²
sys  system call
```

Arithmetic / Logical

1111	**add**	$T	$A	$B
1110	**sub**	$T	$A	$B
1101	**and**	$T	$A	$B
1100	**nor**	$T	$A	$B

Load /Store

1011	**l**	$T	$A	u4
1010	**lwr**	$T	$A	u4
1001	**st**	$S	$A	u4
1000	**stc**	$S	$A	u4

Shift / Branch & Link

0111	**sru**	$T	$A	u4
0110	**srs**	$T	$A	u4
0101	**bcl**	cc	$A	$L
0100	**sl**	$T	$A	u4

Immediate

0011	**lih**	$T	s8
0010	**lis**	$T	s8
0001	**bc**	cc	s8
0000	**sys**	$X	s8

Condition Codes

1111	$\overline{znv} + nv$	SGT
1110	$n\bar{v} + \bar{n}v$	SLT
1101	n	NEG
1100	v	OVF
1011	$\bar{z}c$	UGT
1010	$\bar{c}$	ULT
1001	$\bar{z}$	NE
1000	0	NOP
0111	$z + n\bar{v} + \bar{n}v$	SLE
0110	$nv + \overline{nv}$	SGE
0101	$\bar{n}$	POS
0100	$\bar{v}$	NVF
0011	$z + \bar{c}$	ULE
0010	c	UGE
0001	z	EQ
0000	1	ALL

Notes

[0] PC ← PC+1 ***before*** instruction execution

[1] $0 ***not*** changed ([$0] = 0 always)

[2] Determines flags: z, n, c, v

[4] Determines flag: v

TOY Machine Instructions

ALU instructions

```
add     $, $, $
sub     $, $, $
and     $, $, $
nor     $, $, $
```

Data transfer instructions

```
l       $, $, #
st      $, $, #
```

Load immediate instructions

```
lis     $, #
lih     $, #
```

Conditional branch instructions

```
bc      ?, #
bcl     ?, $, $
```

TOY Machine Instructions

ALU instructions

add	**$**, $, $	:	**$** ← [$] + [$]
sub	**$**, $, $	:	**$** ← [$] − [$]
and	**$**, $, $	:	**$** ← [$] & [$]
nor	**$**, $, $	:	**$** ← ~([$] & [$])

Data transfer instructions

l	**$**, $, #	:	**$** ← MEM[[$] + #]
st	**$**, $, #	:	**$** → MEM[[$] + #]

Load immediate instructions

lis	**$**, #	:	**$** ← # (sign extended)
lih	**$**, #	:	**$** ← # (high bits only)

Conditional branch instructions

bc	**?**, #	:	if **?** **PC** ← **PC** + #
bcl	**?**, $, **$**	:	if **?** **$** ← **PC**, **PC** ← $

HW 7: **W** = **X** + **Y** + **Z**

Assignment: Write a **TOY** assembly language program to add the values of 3 variables, **X**, **Y**, and **Z**, in memory and store the sum in a fourth, **W**.

Details: The variables occupy consecutive words of memory starting with **W**. The address of **W** is in register **$3**. *Do not change* the values in registers **$0** through **$3**. You can use registers **$4** through **$F** as you please.

Your program should consist entirely of addition (**add**), load (**l**), and store (**st**) instructions.

add $7, $5,$6	means	$7 ← [$5] + [$6]
l $4, $C, 8	means	$4 ← [MEM[[$C]+8]]
st $4, $C, 8	means	[$4] → MEM[[$C]+8]

HW 7: **W** = **X** + **Y** + **Z**

Assignment: Write a **TOY** assembly language program to add the values of 3 variables, **X**, **Y**, and **Z**, in memory and store the sum in a fourth, **W**.

Solution:

```
l    $5, $3, 1  : [$5] = [X]
l    $6, $3, 2  : [$6] = [Y]
l    $7, $3, 3  : [$7] = [Z]
add  $8, $5,$6  : [$8] = [X]+[Y]
add  $8, $8,$7  : [$8] = [X]+[Y]+[Z]
st   $8, $3, 0  : [W]  = [X]+[Y]+[Z]
```

HW 7: **W** = **X** + **Y** + **Z**

Assignment: Write a **TOY** assembly language program to add the values of 3 variables, **X**, **Y**, and **Z**, in memory and store the sum in a fourth, **W**.

Solution:

```
lis  $3, @W
lih  $3, @W       : [$3] = @W
l    $5, $3, 1    : [$5] = [X]
l    $6, $3, 2    : [$6] = [Y]
l    $7, $3, 3    : [$7] = [Z]
add  $8, $5,$6    : [$8] = [X]+[Y]
add  $8, $8,$7    : [$8] = [X]+[Y]+[Z]
st   $8, $3, 0    : [W]  = [X]+[Y]+[Z]
```

TOY AL Maximum Problem

Problem: Write a **TOY** assembly language program to put the largest *unsigned* value in registers **`$A`**, **`$B`**, and **`$C`** into register **`$F`**.

Comparison, Condition Code, Flags

After **A** – **B** (`sub $0, $x, $y`)

unsigned	**A ≥ B**	UGE	**c**	**A < B**	ULT	$\overline{\mathbf{c}}$
	A ≤ B	ULE	**z**+$\overline{\mathbf{c}}$	**A > B**	UGT	$\overline{\mathbf{z}}$**c**
both	**A = B**	EQ	**z**	**A ≠ B**	NE	$\overline{\mathbf{z}}$
signed	**A ≥ B**	SGE	$\overline{\mathbf{nv}}$+**nv**	**A < B**	SLT	**n**$\overline{\mathbf{v}}$+$\overline{\mathbf{n}}$**v**
	A ≤ B	SLE	**z**+**n**$\overline{\mathbf{v}}$+$\overline{\mathbf{n}}$**v**	**A > B**	SGT	$\overline{\mathbf{znv}}$+**nv**

TOY AL Maximum Problem

Problem: Write a **TOY** assembly language program to put the largest *unsigned* value in registers **$A**, **$B**, and **$C** into register **$F**.

```
add $F, $A, $0    : $F ← $A
                  : $F = max($A)
```

TOY AL Maximum Problem

Problem: Write a **TOY** assembly language program to put the largest *unsigned* value in registers **$A**, **$B**, and **$C** into register **$F**.

```
        add  $F, $A, $0    : $F ← $A
                           : $F = max($A)
        sub  $0, $B, $F    : $B ? max($A)
        bc   ULE, SkipB    : if not $B ≤ max($A)
        add  $F, $B, $0    :    $F ← $B
SkipB                      : $F = max($A, $B)
```

TOY AL Maximum Problem

Problem: Write a **TOY** assembly language program to put the largest *unsigned* value in registers **$A**, **$B**, and **$C** into register **$F**.

```
        add  $F, $A, $0    : $F ← $A
                           : $F = max($A)
        sub  $0, $B, $F    : $B ? max($A)
        bc   ULE, SkipB    : if not $B ≤ max($A)
        add  $F, $B, $0    :    $F ← $B
SkipB                      : $F = max($A, $B)
        sub  $0, $C, $F    : $C ? max($A, $B)
        bc   ULE, SkipC    : if $C > max($A, $B)
        add  $F, $C, $0    :    $F ← $C
SkipC                      : $F = max($A, $B, $C)
```

TOY AL Maximum Fragment

```
        sub  $0, $B, $F    : $B ? max($A)
        bc   ULE, SkipB    : if not $B ≤ max($A)
        add  $F, $B, $0    :     $F ← $B
SkipB
```

HW 11: Count Negative

Assignment: Write a **TOY** AL program fragment that puts the number of (signed) values from registers **`$A`**, **`$B`**, and **`$C`** that are less than **0** into register **`$F`**.

(Note: after execution of this program fragment, **`$F`** will have one of the four values: 0, 1, 2, and 3.)

HW 11: Count Negative

```
/* The corresponding Java method */
int countNegative(int A, int B, int C){
   int F = 0;
   if (A < 0) {
      F = F + 1;
   }
   if (B < 0) {
      F = F + 1;
   }
   if (C < 0) {
      F = F + 1;
   }
   return F;
}
```

HW 11: Count Negative

```
      lis     $9,         1     : $9 = 1
      add     $F, $0, $0        : $F = 0
                                : $F = f()
      sub     $0, $A, $0        :  a ? 0
      bc      SGE, SkipA        :  a < 0
      add     $F, $F, $9        : $F = $F+1
SkipA                           : $F = f(a)
      sub     $0, $B, $0        :  b ? 0
      bc      SGE, SkipB        :  b < 0
      add     $F, $F, $9        : $F = $F+1
SkipB                           : $F = f(a,b)
      sub     $0, $C, $0        :  c ? 0
      bc      SGE, SkipC        :  c < 0
      add     $F, $F, $9        : $F = $F+1
SkipC                           : $F = f(a,b,c)
```

Count Negative Fragment

```
        sub     $0, $B, $0      :  b ? 0
        bc      SGE, SkipB      :  b < 0
        add     $F, $F, $9      : $F = $F+1
SkipB
```

Count Negative Subprogram

Task: Write a subprogram that computes **f**(**a**, **b**, c), the number **f**()'s arguments that are negative. Write a driver that stores into the variable Count the number of negative values in the variables X, Y, and Z.

Subprogram interface:

Label: `CountNeg`

On entry:

Register **`$1`** is the return address of the caller.

Registers **`$A`**, **`$B`**, and **`$C`** contain values **a**, **b**, and **c**.

On exit:

Register **`$F`** contains **f**(**a**, **b**, **c**) .

Registers **`$4`** to **`$E`** may have changed.

Registers **`$0`** to **`$3`** and main memory are unchanged.

Count Negative Subprogram

```
CountNeg
      lis     $9,       1       : $9 = 1
      add     $F, $0, $0        : $F = 0
                                : $F = f()
      sub     $0, $A, $0        :  a ? 0
      bc      SGE, SkipA        :  a < 0
      add     $F, $F, $9        : $F = $F+1
SkipA                           : $F = f(a)
      sub     $0, $B, $0        :  b ? 0
      bc      SGE, SkipB        :  b < 0
      add     $F, $F, $9        : $F = $F+1
SkipB                           : $F = f(a,b)
      sub     $0, $C, $0        :  c ? 0
      bc      SGE, SkipC        :  c < 0
      add     $F, $F, $9        : $F = $F+1
SkipC                           : $F = f(a,b,c)
      bcl     ALL,$1, $0        :  return
```

Count Negative Driver

```
lis     $A,       @X       :
lih     $A,       @X       : $A = @x
1       $A, $A,    0       : $A =  x

lis     $B,       @Y       :
lih     $B,       @Y       : $B = @y
1       $B, $B,    0       : $B =  y

lis     $C,       @Z       :
lih     $C,       @Z       : $C = @z
1       $C, $C,    0       : $C =  z

lis     $F, @CountNeg      :
lih     $F, @CountNeg      : $F = @countNeg
bcl     ALL,$F, $1         : $F = f(x,y,z)

lis     $D, @Count         :
lih     $D, @Count         : $D = @Count
st      $F, $D,    0       : Count = f(x,y,z)
```

HW 12: Array Count Subprogram

Assignment: Write a **TOY** AL subprogram that returns in register **`$F`** the number of *unsigned* values in an array that are greater than some specified cut-off value.

Subprogram interface:

Label: `ArrayCount`

On entry:

Register **`$1`** is the return address of the caller.
Register **`$A`** is **`@A`** (memory address of the array **`A`**).
Register **`$B`** is **`n`** (# of elements in the array **`A`**).
Register **`$C`** is **`k`** (cut-off value).

On exit:

Register **`$F`** is # of elements of **`A`** greater than **`k`**.
Registers **`$4`** to **`$E`** may have changed.
Registers **`$0`** to **`$3`** and main memory are not changed.

Array Processing Pattern

```
/* The analogous Java method */
int arraySomething (int[] A, ...){

   < your initialization >
   for (int j=0; j < A.length; j=j+1) {
      int t = A[j];

      < your code goes here >
      < do something to array element t >

      A[j] = t;          // only if t ≠ A[j]
   }
   return result;
}
```

```
ArrayIncrement
      lis   $9,        1       : $9 = 1 = c
      add   $F, $0, $0         : $F = 0 = f
      add   $8, $0, $0         : $8 = 0 = j
Loop
      sub   $0, $8, $B         :  j ? n
      bc    UGE, Done
      add   $7, $A, $8         : $7 = @A[j]
      l     $6, $7,  0         : $6 =  A[j] = t
```

Do something with t ($6)

```
      st    $6, $7,  0         :  A[j] ← A[j] + k
      add   $8, $8, $9         :  j = j+1
      bc    ALL, Loop
Done
      bcl   ALL,$1, $0         :  return to caller
```

```
ArrayIncrement
      lis   $9,        1       : $9 = 1 = c
      add   $F, $0, $0         : $F = 0 = f
      add   $8, $0, $0         : $8 = 0 = j
Loop
      sub   $0, $8, $B         :  j ? n
      bc    UGE, Done
      add   $7, $A, $8         : $7 = @A[j]
      add   $8, $8, $9         :  j = j+1
      l     $6, $7,  0         : $6 =  A[j-1] = t
```

Do something with t ($6)

```
      st    $6, $7,  0         :  A[j-1] ← A[j-1] + k
      bc    ALL, Loop
Done
      bcl   ALL,$1, $0         :  return to caller
```

Array Maximum Subprogram

Task: Write a **TOY** AL subprogram that returns in register **`$F`** the largest *signed* values in an array.

Subprogram interface:

Label: `ArrayMax`

On entry:

Register **`$1`** is the return address of the caller.

Register **`$A`** is **`@A`** (memory address of the array **`A`**).

Register **`$B`** is **`n`** (# of elements in the array **`A`**).

On exit:

Register **`$F`** is the value of the maximal element of **`A`**.

Registers **`$4`** to **`$E`** may have changed.

Registers **`$0`** to **`$3`** and main memory are not changed.

Array Maximum in Java

```
/* The corresponding Java method */
int arrayMaximum (int[] A){
   int F = Integer.MIN_VALUE;
   for (j=0; j < A.length; j=j+1) {
      if (A[j] > F) {
         F = A[j];
      }
   }
   return F;
}
```

Array Maximum Subprogram

```
ArrayMax
        lis   $9,        1    : $9 = 1
        add   $8, $0, $0      : $8 = 0 = j
        lis   $F, 0x8000      : $F = 0
        lih   $F, 0x8000      : $F = max A[0..0)
Loop                          : $F = max A[0..j)
        sub   $0, $8, $B      :  j ? n
        bc    UGE, Done
        add   $7, $A, $8      : $7 = @A[j]
        l     $6, $7,  0      : $6 =  A[j] = t
        sub   $0, $6, $F      : A[j] ? max A[0..j)
        bc    SLE, Skip       : A[j] > max A[0..j)
        add   $F, $6, $0      : $F = A[j]
Skip                          : $F = max A[0..j+1)
        add   $8, $8, $9      :  j = j+1
        bc    ALL, Loop       : $F = max A[0..j)
Done
        bcl   ALL,$1, $0      :  return to caller
```

Array Maximum Fragment

```
        sub   $0, $6, $F    : A[j] ? max A[0..j)
        bc    SLE, Skip     : A[j] > max A[0..j)
        add   $F, $6, $0    : $F = A[j]
Skip
```

Array Sum Subprogram

Task: Write a **TOY** AL subprogram that returns the sum of the values in an array **A** in register **$F**.

Subprogram interface:

Label: `ArraySum`

On entry:

Register **$1** is the return address of the caller.
Register **$A** is **@A** (memory address of the array **A**).
Register **$B** is **n** (# of elements in the array **A**).

On exit:

Register **$F** is the sum of the values in **A** .
Registers **$4** to **$E** may have changed.
Registers **$0** to **$3** are unchanged.
Main memory is unchanged.

Array Sum Java Method

```
/* The analogous Java method */
int arraySum(int[] A, int C){
   int f = 0;
   for (int j=0; j<A.length; j=j+1) {
      f = f + A[j];
   }
   return f;
}
```

Array Sum Subprogram

```
ArraySum
   lis  $9,  1          : $9 = 1
   add  $F, $0, $0      : $F = 0 = f
   add  $8, $0, $0      : $8 = 0 = j
Loop                    : $F = sum A[0..j)
   sub  $0, $8, $B      :  j ? n
   bc   UGE, Done
   add  $7, $A, $8      : $7 = @A[j]
   l    $6, $7,  0      : $6 =  A[j] = t
   add  $F, $F, $6      : $F = sum A[0..j+1)
   add  $8, $8, $9      :  j = j+1
   bc   ALL, Loop       : $F = sum A[0..j)
Done
   bcl  ALL,$1, $0      :  return to caller
```

Array Sum Fragment

```
add $F, $F, $6    : $F = sum A[0..j+1)
```

Array Increment Subprogram

Task: Write a **TOY** AL subprogram that adds a specified value to every element in an array.

Subprogram interface:

Label: `ArrayIncrement`

On entry:

Register **`$1`** is the return address of the caller.
Register **`$A`** is **`@A`** (memory address of the array **`A`**).
Register **`$B`** is **`n`** (# of elements in the array **`A`**).
Register **`$C`** is **`k`** (the specified value).

On exit:

Register **`$F`** is the # elements of **`A`** that equal to **`k`**.
Registers **`$4`** to **`$E`** may have changed.
Registers **`$0`** to **`$3`** are not changed.
Main memory is unchanged, *except that*
Each element of **`A`** has been increased by **`k`**.

Array Increment in Java

```
/* The analogous Java method */
int arrayIncrement (int[] A, int k){
   int c = 1;
   int f = 0;
   for (int j=0; j < A.length; j=j+c) {
      int t = A[j];
      if (t == k) {
         f = f + c;
      }
      t = t + k;
      A[j] = t;
   }
   return f;
}
```

```
ArrayIncrement
      lis   $9,        1       : $9 = 1 = c
      add   $F, $0, $0         : $F = 0 = f
      add   $8, $0, $0         : $8 = 0 = j
Loop
      sub   $0, $8, $B         :  j ? n
      bc    UGE, Done
      add   $7, $A, $8         : $7 = @A[j]
      l     $6, $7,  0         : $6 =  A[j] = t
      sub   $0, $6, $C         :  A[j] ? k
      bc    NE, Skip           :  A[j] == k
      add   $F, $F, $9         : $F = $F + 1
Skip                           :
      add   $6, $6, $C         :  t = A[j] + k
      st    $6, $7,  0         :  A[j] ← A[j] + k
      add   $8, $8, $9         :  j = j+1
      bc    ALL, Loop
Done
      bcl   ALL,$1, $0         :  return to caller
```

Array Increment Fragment

```
      sub   $0, $6, $C      :  A[j] ? k
      bc    NE, Skip        :  A[j] == k
      add   $F, $F, $9      : $F = $F + 1
Skip                        :
      add   $6, $6, $C      :  t = A[j] + k
```

HW 12: Array Count Subprogram

Assignment: Write a **TOY** AL subprogram that returns in register **`$F`** the number of *unsigned* values in an array that are greater than some specified cut-off value.

Subprogram interface:

Label: `ArrayCount`

On entry:

Register **`$1`** is the return address of the caller.
Register **`$A`** is **`@A`** (memory address of the array **`A`**).
Register **`$B`** is **`n`** (# of elements in the array **`A`**).
Register **`$C`** is **`k`** (cut-off value).

On exit:

Register **`$F`** is # of elements of **`A`** greater than **`k`**.
Registers **`$4`** to **`$E`** may have changed.
Registers **`$0`** to **`$3`** and main memory are not changed.

HW 12: Array Count

```
/* count the elements of an array A
that are greater than k */
int arrayCount(int[] A, int k) {
   int c = 1;
   int f = 0;
   for (int j = 0; j<A.length; j=j+1) {
      int t = A[j];
      if (t > k) {
         f = f+1;
      }
   }
   return f;
}
```

HW 12: Array Count

```
ArrayCount
      lis  $9,  1          : $9 is 1 = c
      add  $8, $0, $0      : $8 is j = 0
      add  $F, $0, $0      : $F: count[0..j)
Loop                       : $F: count[0..j)
      sub  $0, $8, $B      :  j ? n
      bc   UGE, Done       :  j < n
      add  $7, $8, $A      : $7 = @A[j]
      l    $6, $7,  0      : $6 = A[j]




      add  $8, $8, $9      :  j = j+1
      bc   ALL, Loop       : $F: count[0..j)
Done
      bcl  ALL, $1, $0     : return to caller
```

HW 11: Count Negative

```
      lis     $9,         1   : $9 = 1
      add     $F, $0, $0      : $F = 0
                              : $F = f()
      sub     $0, $A, $0      :  a ? 0
      bc      SGE, SkipA      :  a < 0
      add     $F, $F, $9      : $F = $F+1
SkipA                         : $F = f(a)
      sub     $0, $B, $0      :  b ? 0
      bc      SGE, SkipB      :  b < 0
      add     $F, $F, $9      : $F = $F+1
SkipB                         : $F = f(a,b)
      sub     $0, $C, $0      :  c ? 0
      bc      SGE, SkipC      :  c < 0
      add     $F, $F, $9      : $F = $F+1
SkipC                         : $F = f(a,b,c)
```

Count Negative Fragment

```
      sub     $0, $A, $0   :  a ? 0
      bc      SGE, SkipA   :  a < 0
      add     $F, $F, $9   : $F = $F+1
SkipA
```

Count Negative Fragment

```
      sub     $0, $A, $0   :  a ? 0
      bc      SGE, SkipA   :  a < 0
      add     $F, $F, $9   : $F = $F+1
SkipA

      sub     $0, $6, $C   :  A[j] ? k
```

Count Negative Fragment

```
      sub     $0, $A, $0   :  a ? 0
      bc      SGE, SkipA   :  a < 0
      add     $F, $F, $9   : $F = $F+1
SkipA

      sub     $0, $6, $C   :  A[j] ? k
      bc      ULE, Skip    :  A[j] > k

Skip
```

Count Negative Fragment

```
      sub     $0, $A, $0   :  a ? 0
      bc      SGE, SkipA   :  a < 0
      add     $F, $F, $9   : $F = $F+1
SkipA

      sub     $0, $6, $C   :  A[j] ? k
      bc      ULE, Skip    :  A[j] > k
      add     $F, $F, $9   : $F = $F+1
Skip
```

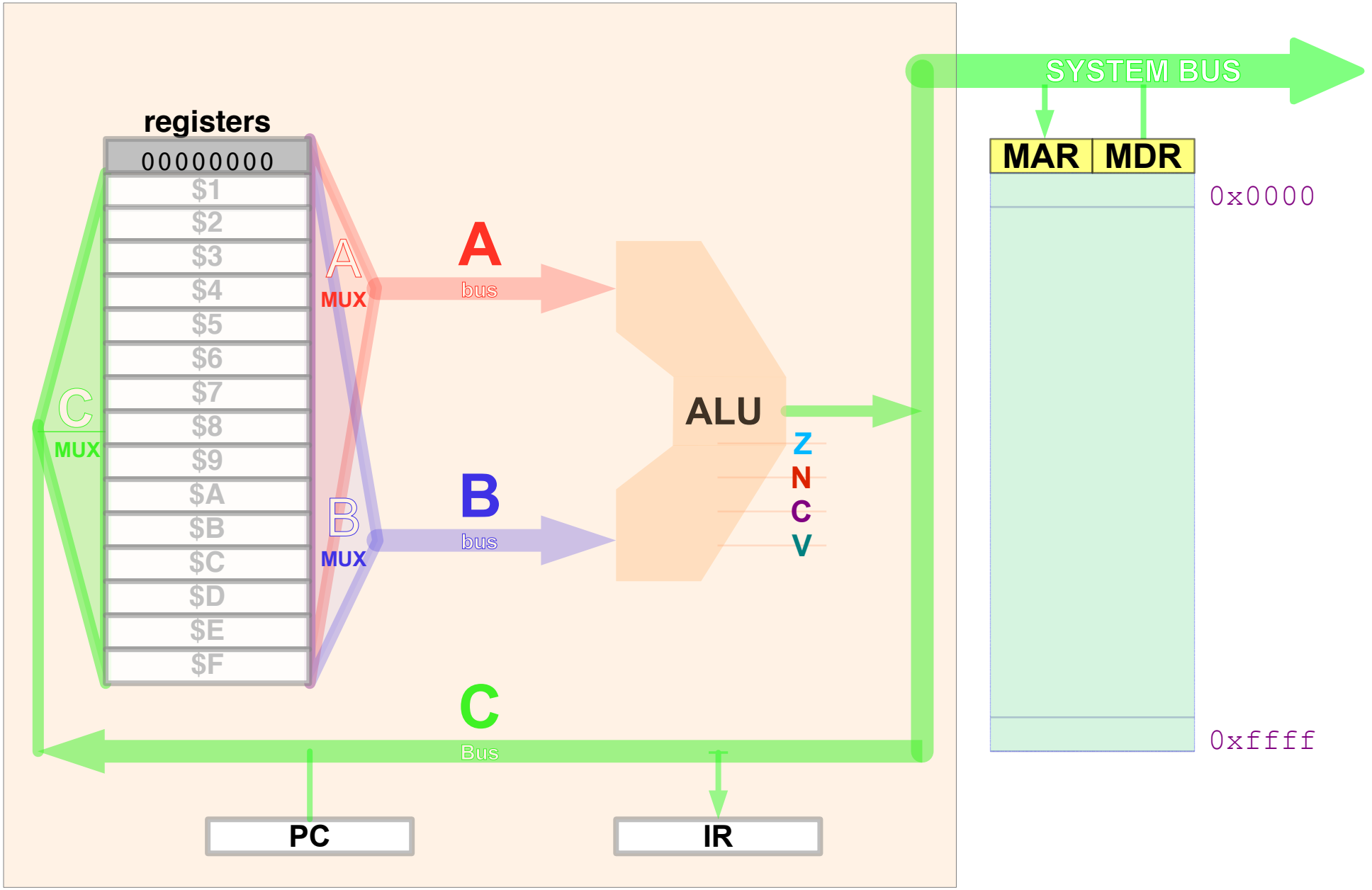
Array Count Fragment

```
      sub     $0, $6, $C   :  A[j] ? k
      bc      ULE, Skip    :  A[j] > k
      add     $F, $F, $9   : $F = $F+1
Skip
```

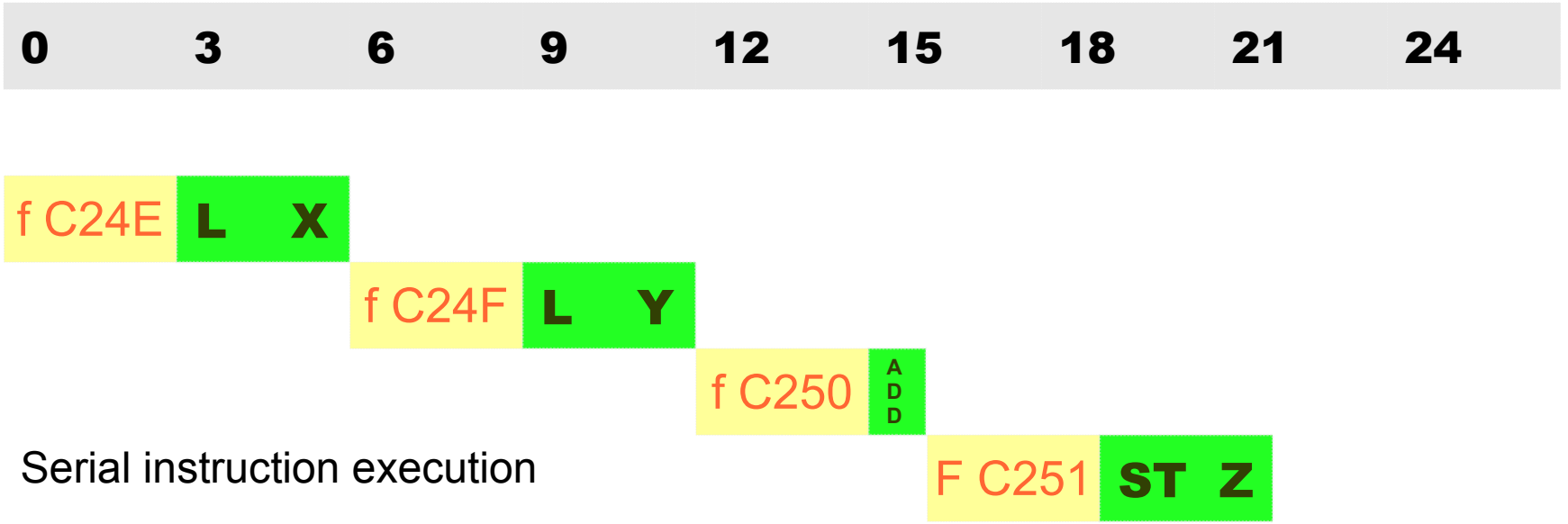
HW 12: Array Count

```
ArrayCount
      lis  $9,  1          : $9 is 1 = c
      add  $8, $0, $0      : $8 is j = 0
      add  $F, $0, $0      : $F: count[0..j)
Loop                       : $F: count[0..j)
      sub  $0, $8, $B      :  j ? n
      bc   UGE, Done       :  j < n
      add  $7, $8, $A      : $7 = @A[j]
      l    $6, $7,  0      : $6 = A[j]
      sub  $0, $6, $C      :  A[j] ? k
      bc   ULE, Skip       :  A[j] > k
      add  $F, $F, $9      : $F = $F+1
Skip                       : $F: count[0..j+1)
      add  $8, $8, $9      :  j = j+1
      bc   ALL, Loop       : $F: count[0..j)
Done
      bcl  ALL, $1, $0     : return to caller
```

TOY ISA Simple Implementation



Pipeline Speedup ≈ 1.47



TOY Pipeline Structural Hazard

load **cycle 3**:

```
1111 → ALU controller;        000 → ALU
1010 → A MUX,          [$A] → A bus → ALU
0000000000000011             → B bus → ALU
 ALU          → C bus → system bus → MAR
```

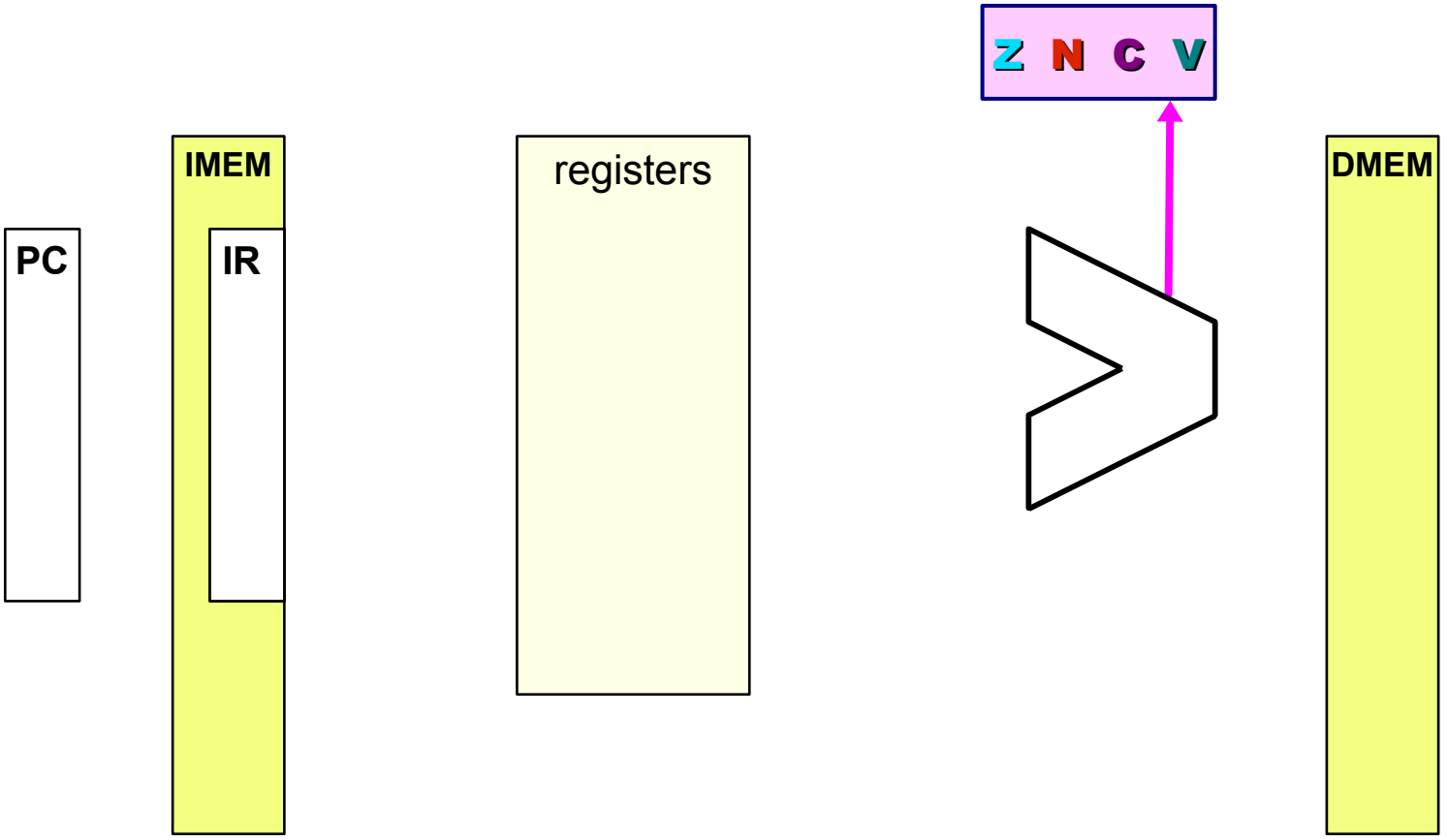
fetch **cycle** 0:

```
[PC]          → C bus → system bus → MAR
```

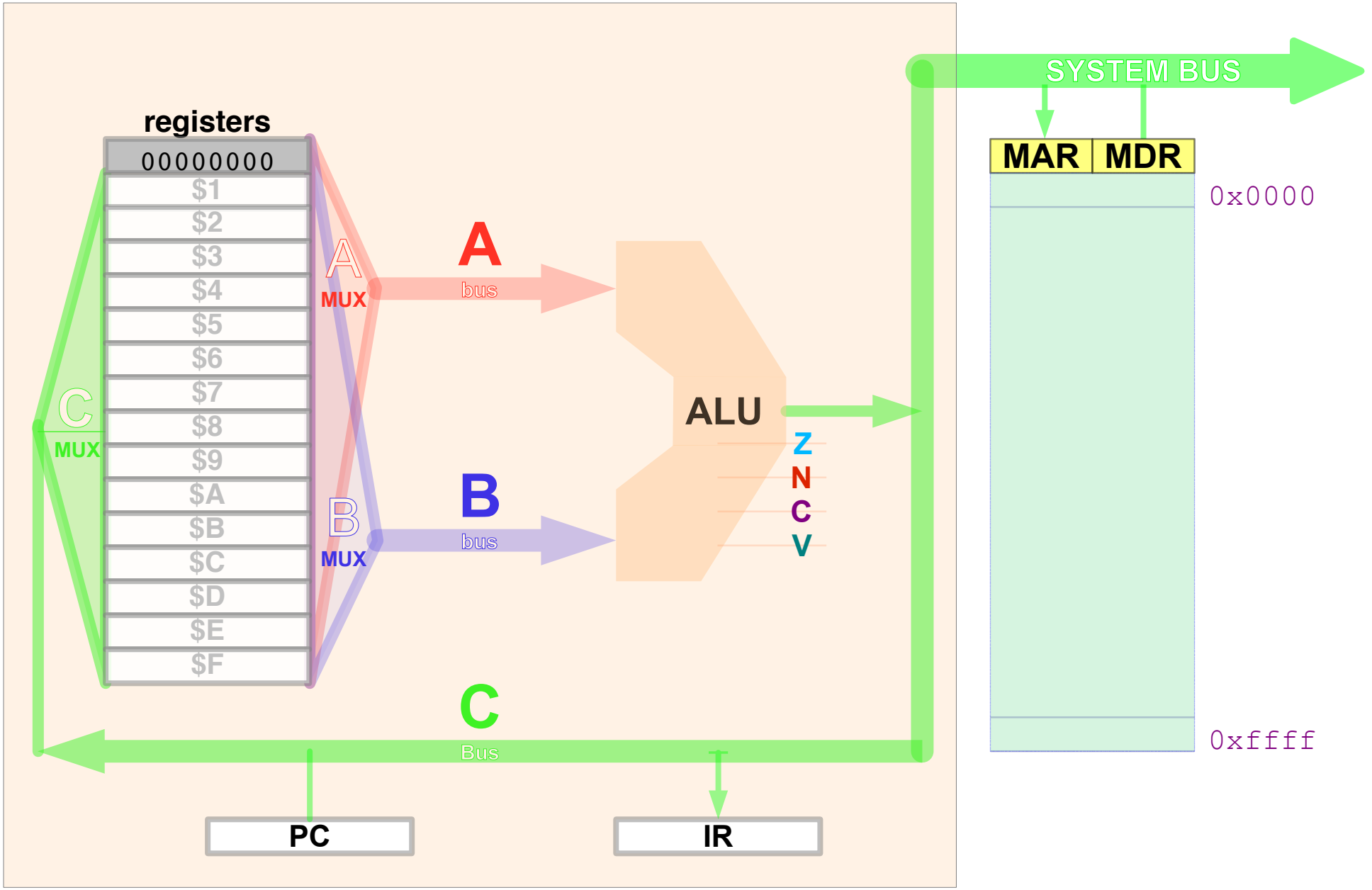
Pipeline performance improvements require **processor redesign** (microarchitecture)

- Step 1: single cycle instruction execution
- Step 2: pipelined instruction execution
  - Overlap execution of successive instructions

TOY Processor Redesign



TOY ISA Simple Implementation



TOY Instruction Fetch

cycle 0:

[**PC**] → C bus → system bus → **MAR**

cycle 1:

memRead

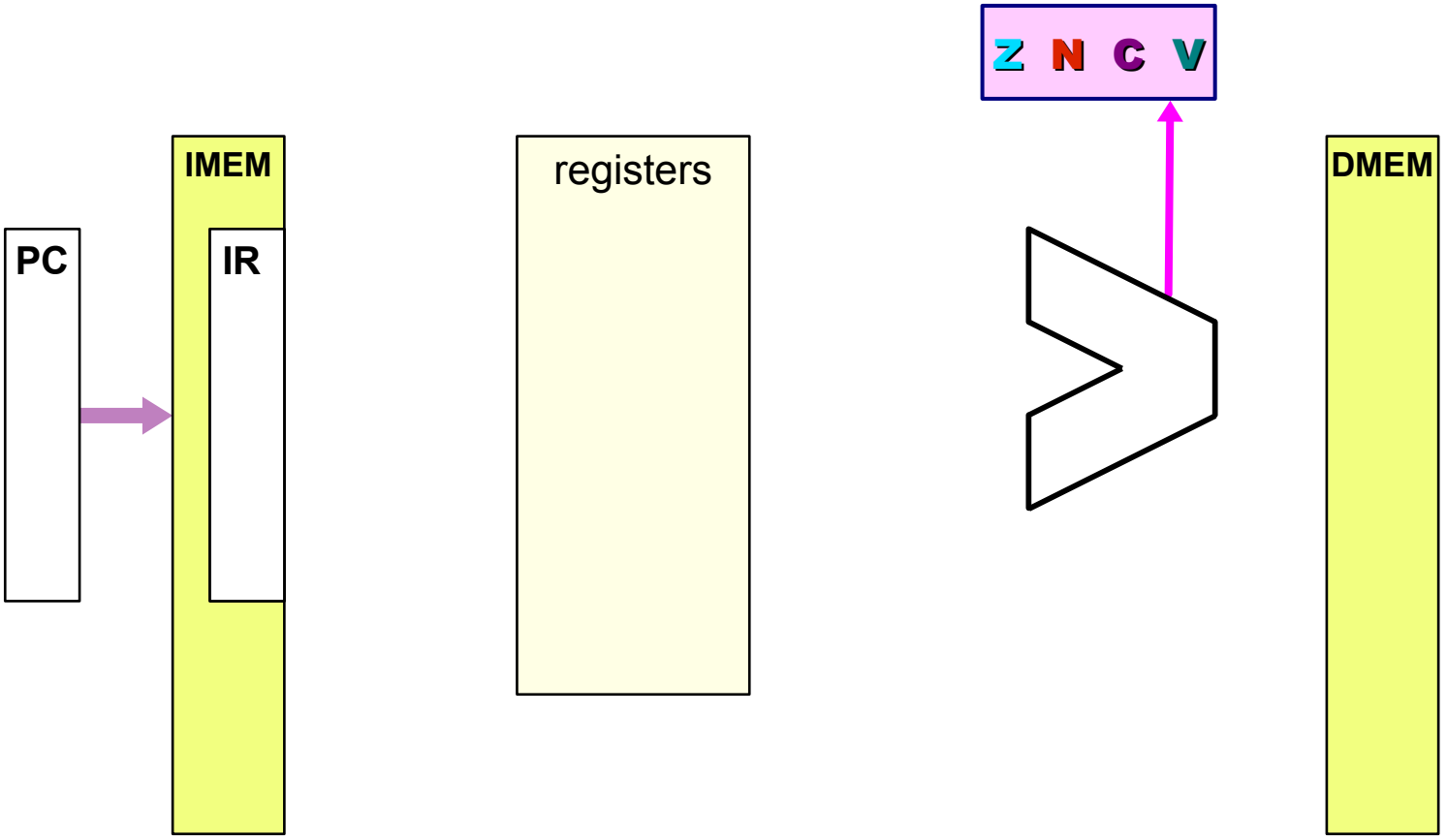
Memory[**MAR**] → **MDR**

[**PC**] + 1 → **PC**

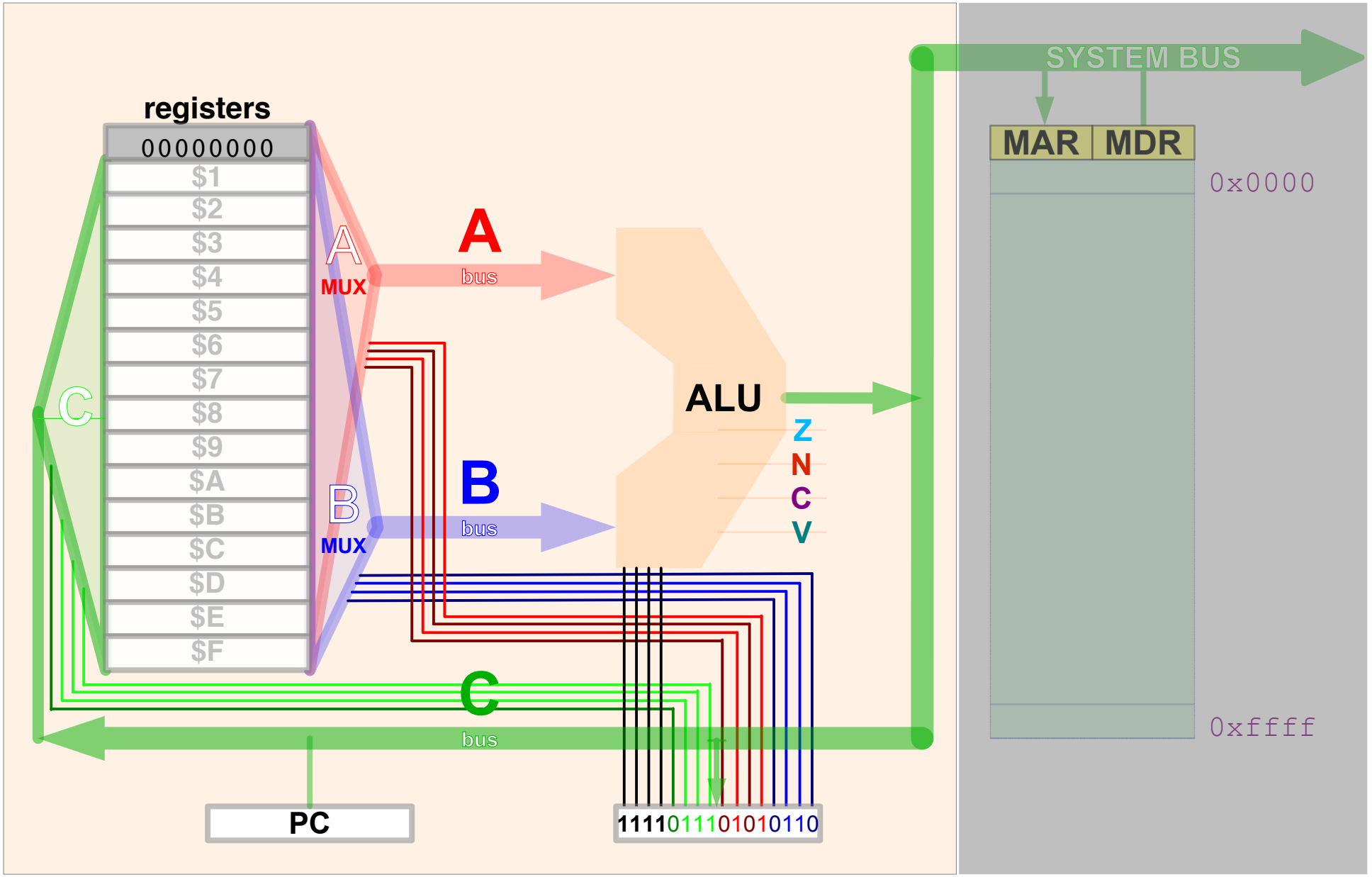
cycle 2:

[**MDR**] → system bus → C bus → **IR**

TOY Redesign w. `fetch`



add $7, $5, $6


registers
00000000
$1
$2
$3
$4
$5
$6
$7
$8
$9
$A
$B
$C
$D
$E
$F
C
A
MUX
B
MUX
A
bus
B
bus
C
bus
ALU
Z
N
C
V
PC
1111011101010110
SYSTEM BUS
MAR
MDR
0x0000
0xffff

add Instruction Execution

add $7, $5, $6 **1111** 0111 0101 0110

cycle 3:

1111 → **ALU** controller; 000 → **ALU**

0101 → A MUX; [$5] → A bus → **ALU**

0110 → B MUX; [$6] → B bus → **ALU**

regWrite

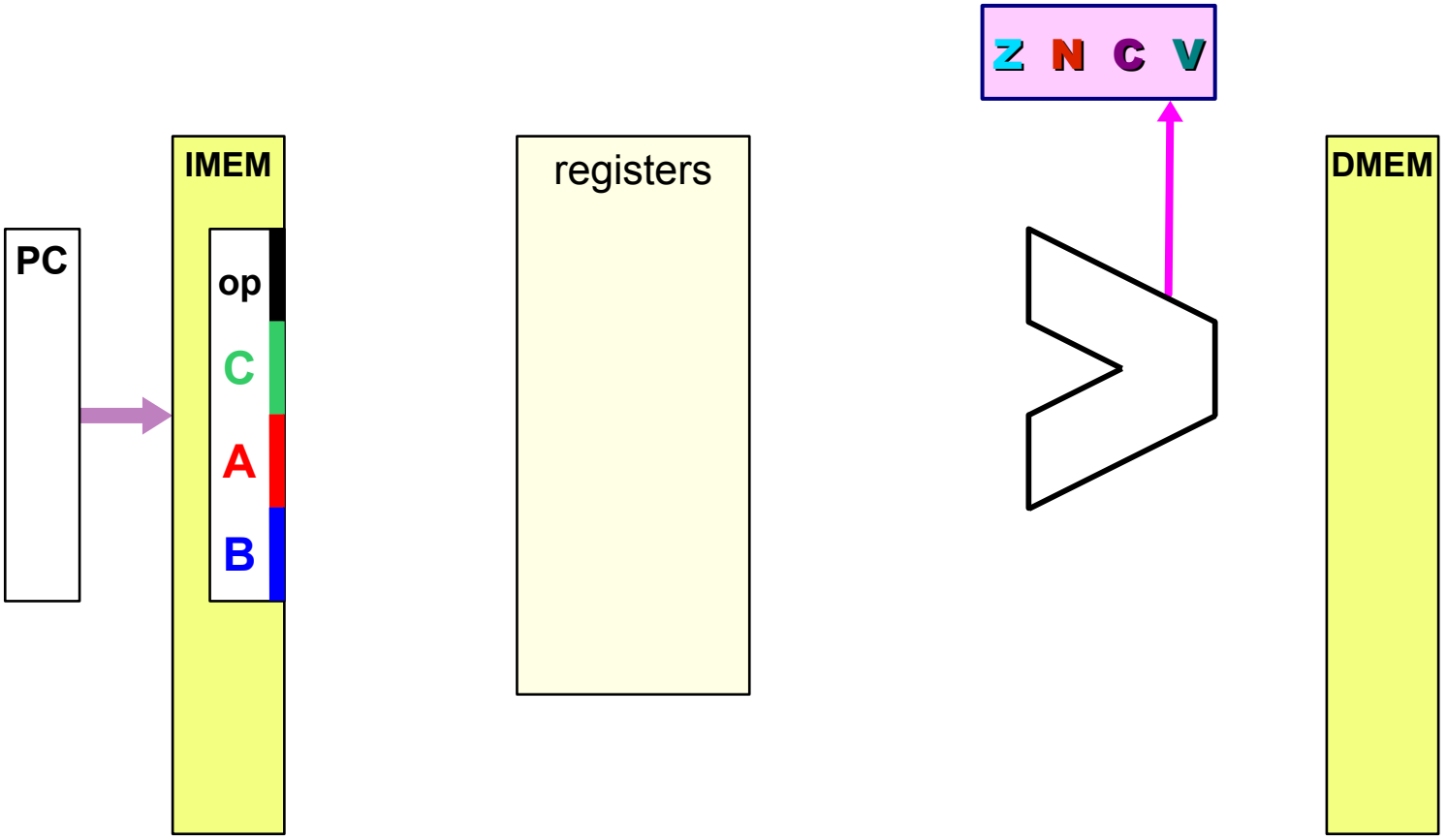
0111 → C decode **ALU** → C bus → $7

zncvWrite

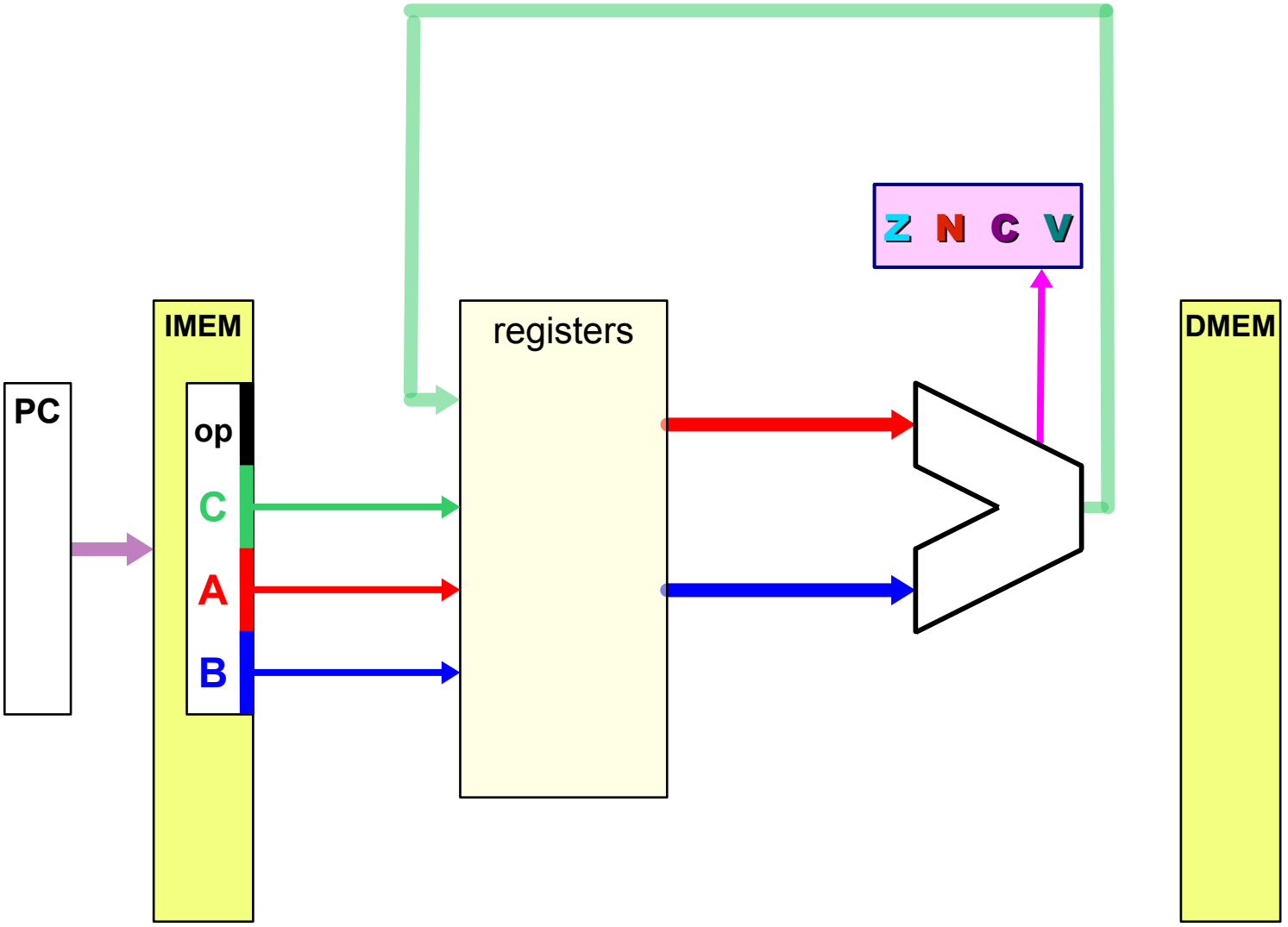
ALU flags → **Z**, **N**, **C**, **V** registers

nextInstruction (reset the cycle counter)

TOY Redesign **IR**



TOY Redesign w. ALU Instructions



load Instruction Execute

l $1, $A, 3 **1011** 0001 1010 0011

cycle 3:

1111 → **ALU** controller; 000 → **ALU**

1010 → A MUX, [$A] → A bus → **ALU**

0000000000000011 → B bus → **ALU**

ALU → C bus → system bus → **MAR**

cycle 4: **MemRead**

Memory[**MAR**] → **MDR**

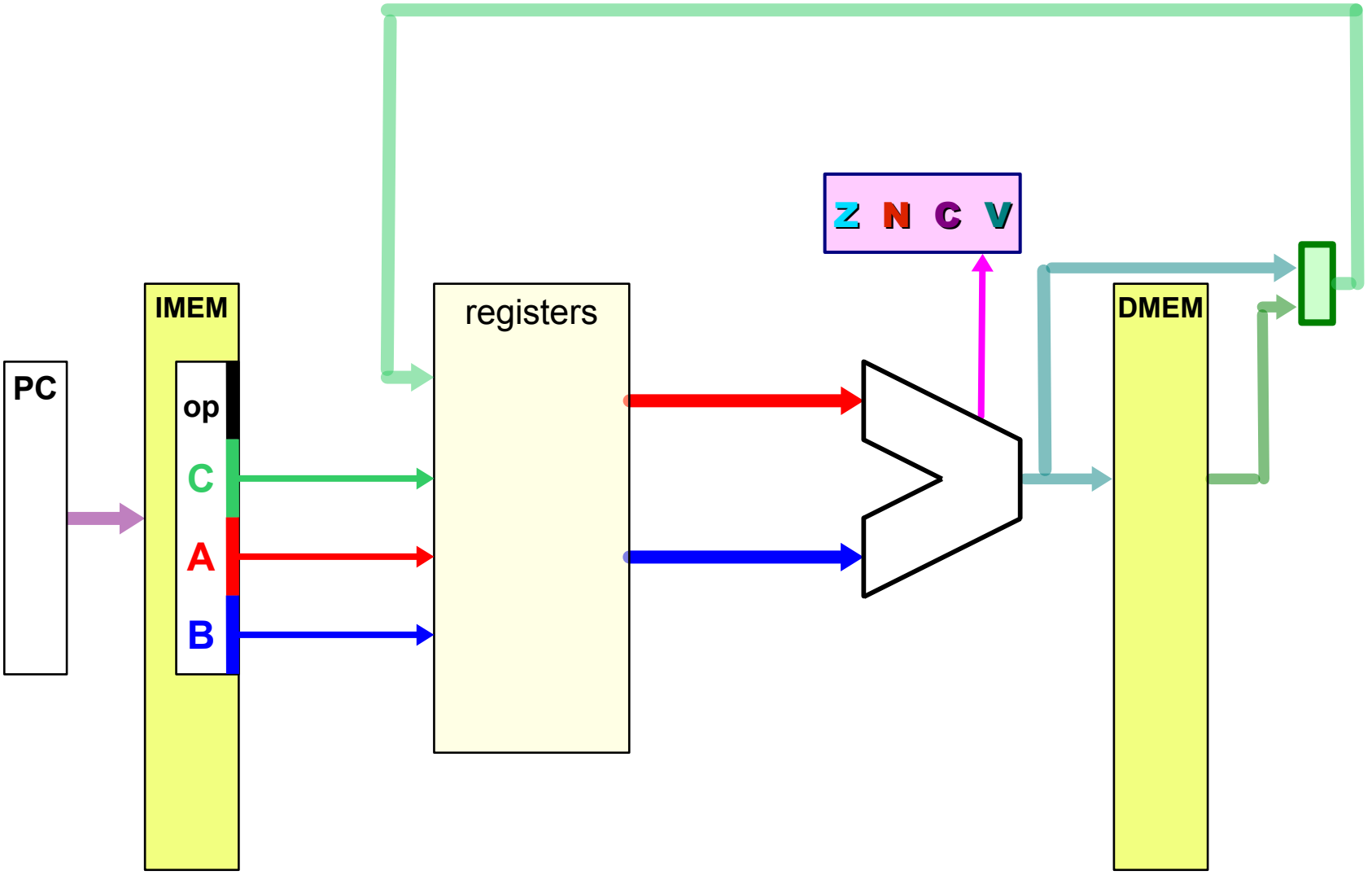
cycle 5: **RegWrite**

0001 → C decode

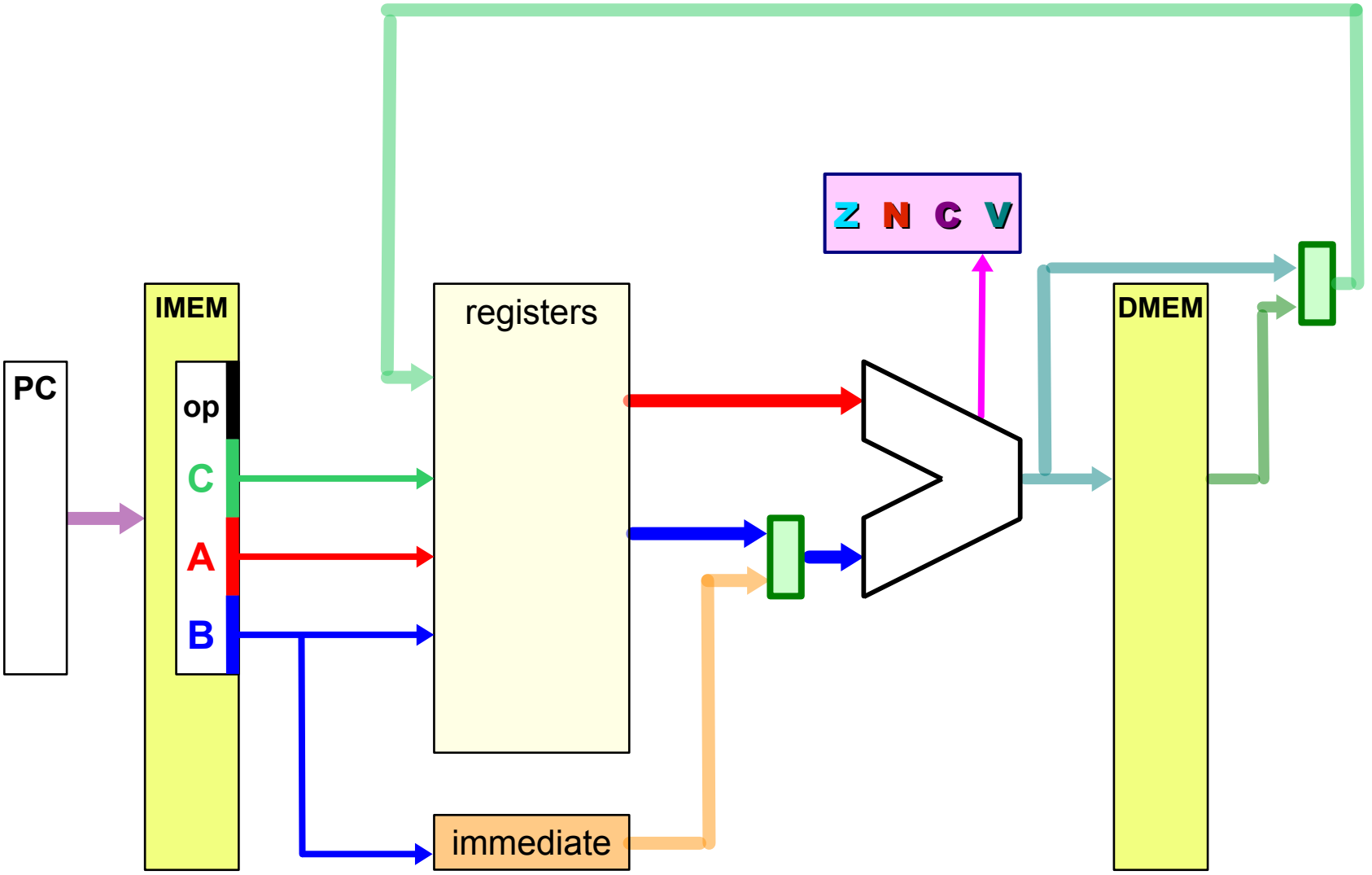
[**MDR**] → system bus → C bus → $1

nextInstruction

TOY Redesign w. `load` ?



TOY Redesign w. `load`



store Instruction Execute

st $1, $A, 9 **0011** 0001 1010 1001

cycle 3:

1111 → **ALU** controller; 000 → **ALU**

1010 → A MUX, [$A] → A bus → **ALU**

0000000000001001 → B bus → **ALU**

ALU → C bus → system bus → **MAR**

cycle 4:

1111 → **ALU** controller; 000 → **ALU**

0001 → A MUX, [$1] → A bus → **ALU**

0000 → B MUX, 0000 → B bus → **ALU**

ALU → C bus → system bus → **MDR**

cycle 5: **memWrite**

[**MDR**] → Memory[**MAR**]

nextInstuction

store Instruction Execute

st $1, $A, 9 **0011** 0001 1010 1001

cycle 3:

1111 → **ALU** controller; 000 → **ALU**

1010 → A MUX, [$A] → A bus → **ALU**

0000000000001001 → B bus → **ALU**

ALU → C bus → system bus → **MAR**

cycle 4:

1111 → **ALU** controller; 000 → **ALU**

0001 → A MUX, [$1] → A bus → **ALU**

0000 → B MUX, 0000 → B bus → **ALU**

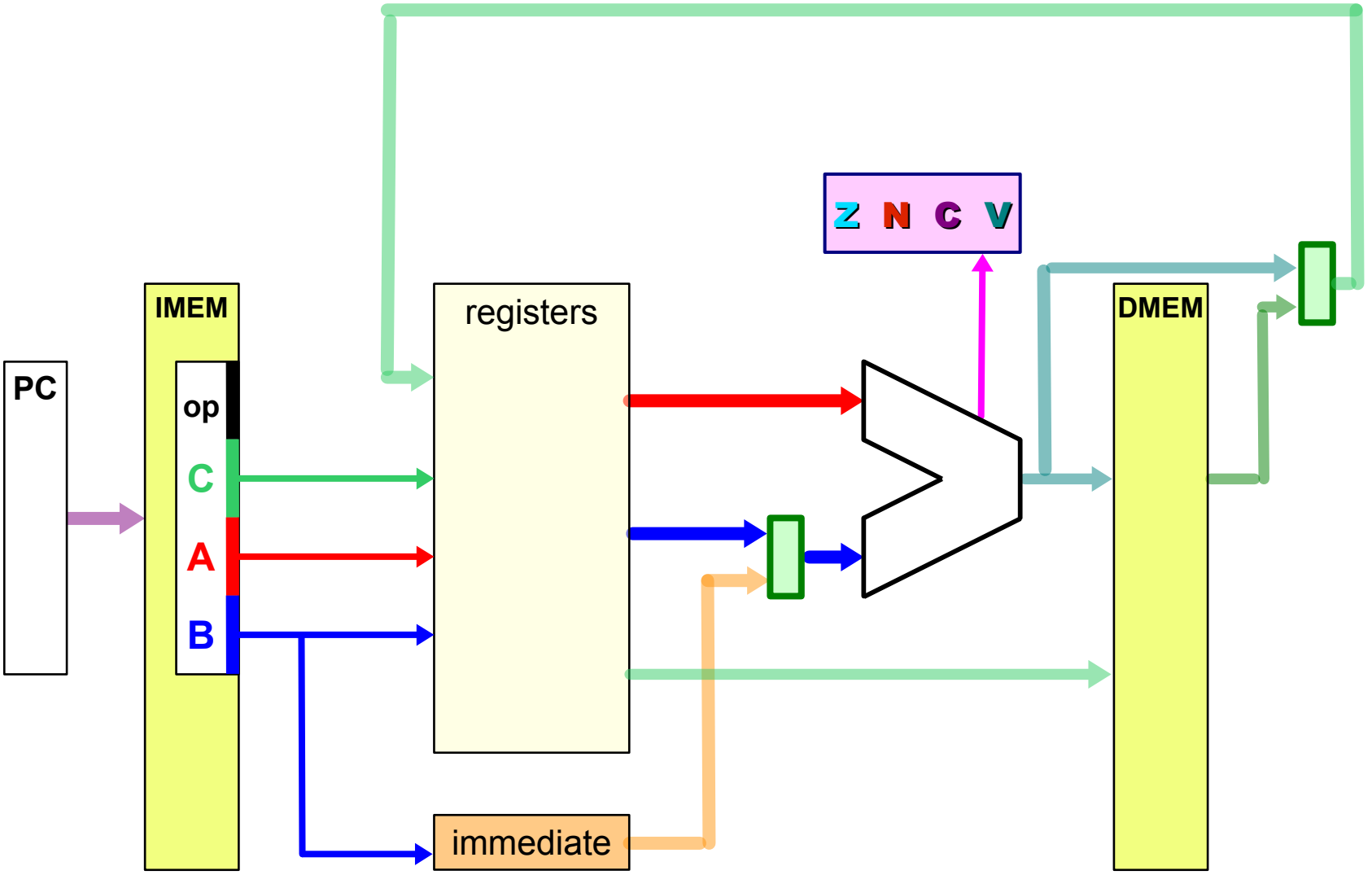
ALU → C bus → system bus → **MDR**

cycle 5: **memWrite**

[**MDR**] → Memory[**MAR**]

nextInstuction

TOY Single Cycle w. **st**ore



load immediate signed

lis $8, 0x9C **1011** 1000 1001 1100

cycle 3:

1111 → **ALU** controller; 000 → **ALU**

0000 → A MUX, 0000 → A bus → **ALU**

1001 1100 → B bus[7..0] → **ALU**

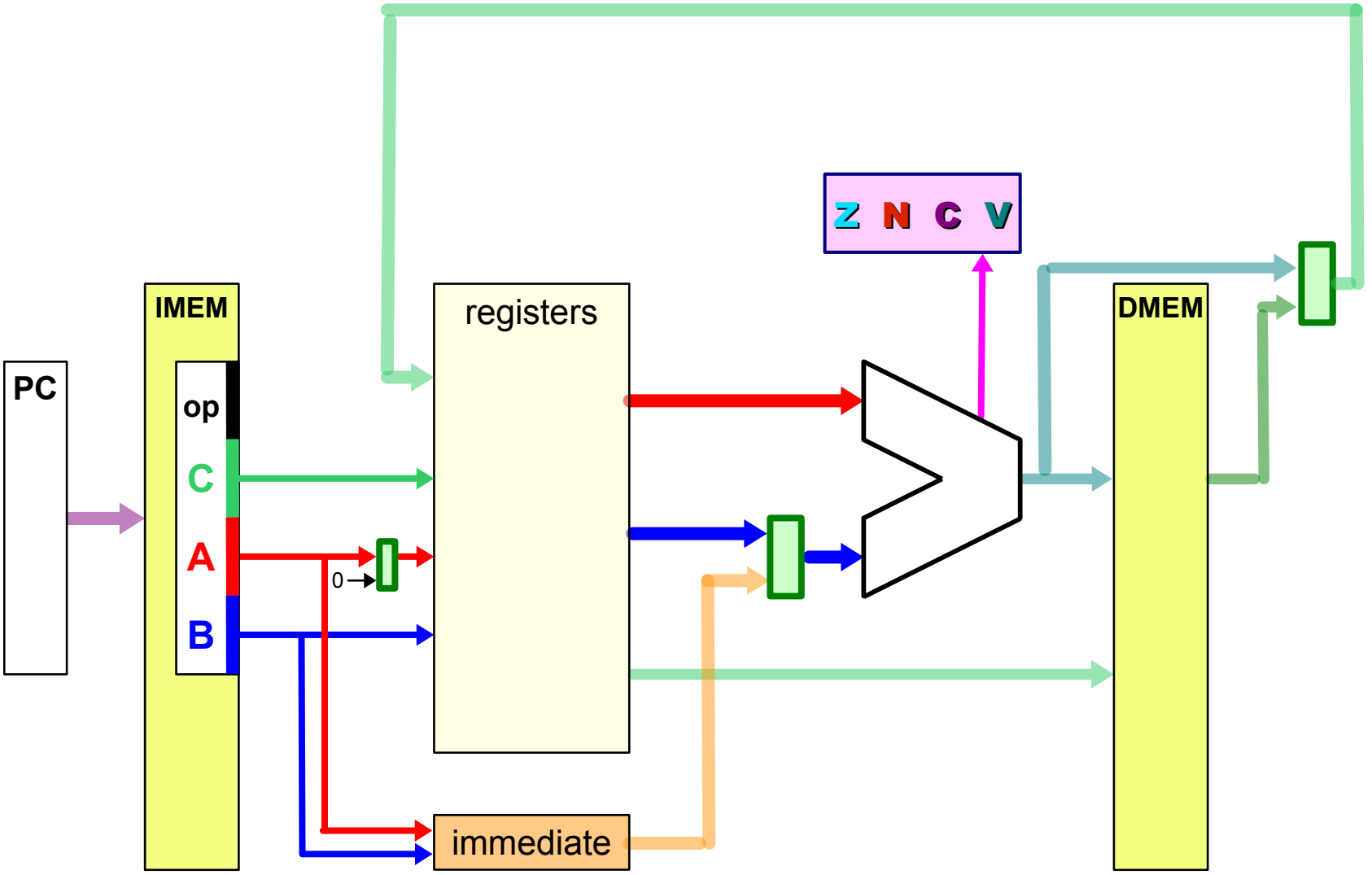
1111 1111 → B bus[F..8] → **ALU**

regWrite

1000 → C decode, **ALU** → C bus → $8

nextInstruction (reset the cycle counter)

TOY Redesign w. `lis`



load **i**mmediate **h**igh

lih $8, 0x37 **1011** 1000 0011 0111

cycle 3:

1101 → **ALU** controller; 001 → **ALU**

1000 → A MUX

1111 1111 → A bus[F..8] → **ALU**

[$8] → A bus[7..0] → **ALU**

0011 0111 → B bus[F..8] → **ALU**

1111 1111 → B bus[7..0] → **ALU**

regWrite

1000 → C decode, **ALU** → C bus → $8

nextInstruction (reset the cycle counter)

load immediate high

lih $8, 0x37 **1011** 1000 0011 0111

cycle 3:

1101 → **ALU** controller; 001 → **ALU**

1000 → A MUX

1111 1111 → A bus[F..8] → **ALU**

[$8] → A bus[7..0] → **ALU**

0011 0111 → B bus[F..8] → **ALU**

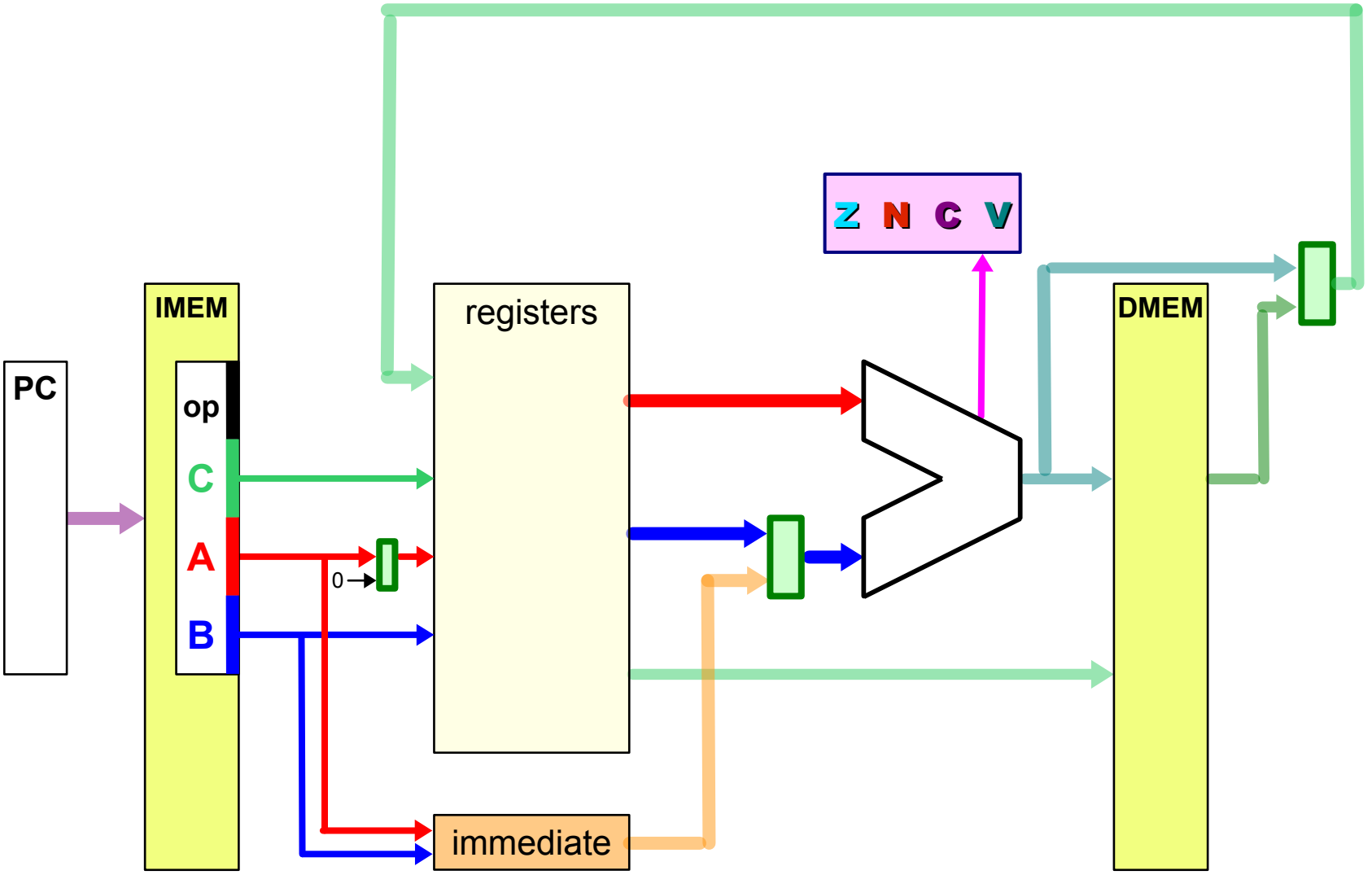
1111 1111 → B bus[7..0] → **ALU**

regWrite

1000 → C decode, **ALU** → C bus → $8

nextInstruction (reset the cycle counter)

TOY Redesign w. `lih` (?)



Branch Conditional

BC ULT, 0xA7 **0001** 1010 **1**010 0111

cycle 3:

1111 → **ALU** controller; 000 → **ALU**

[**PC**] → A bus → **ALU**

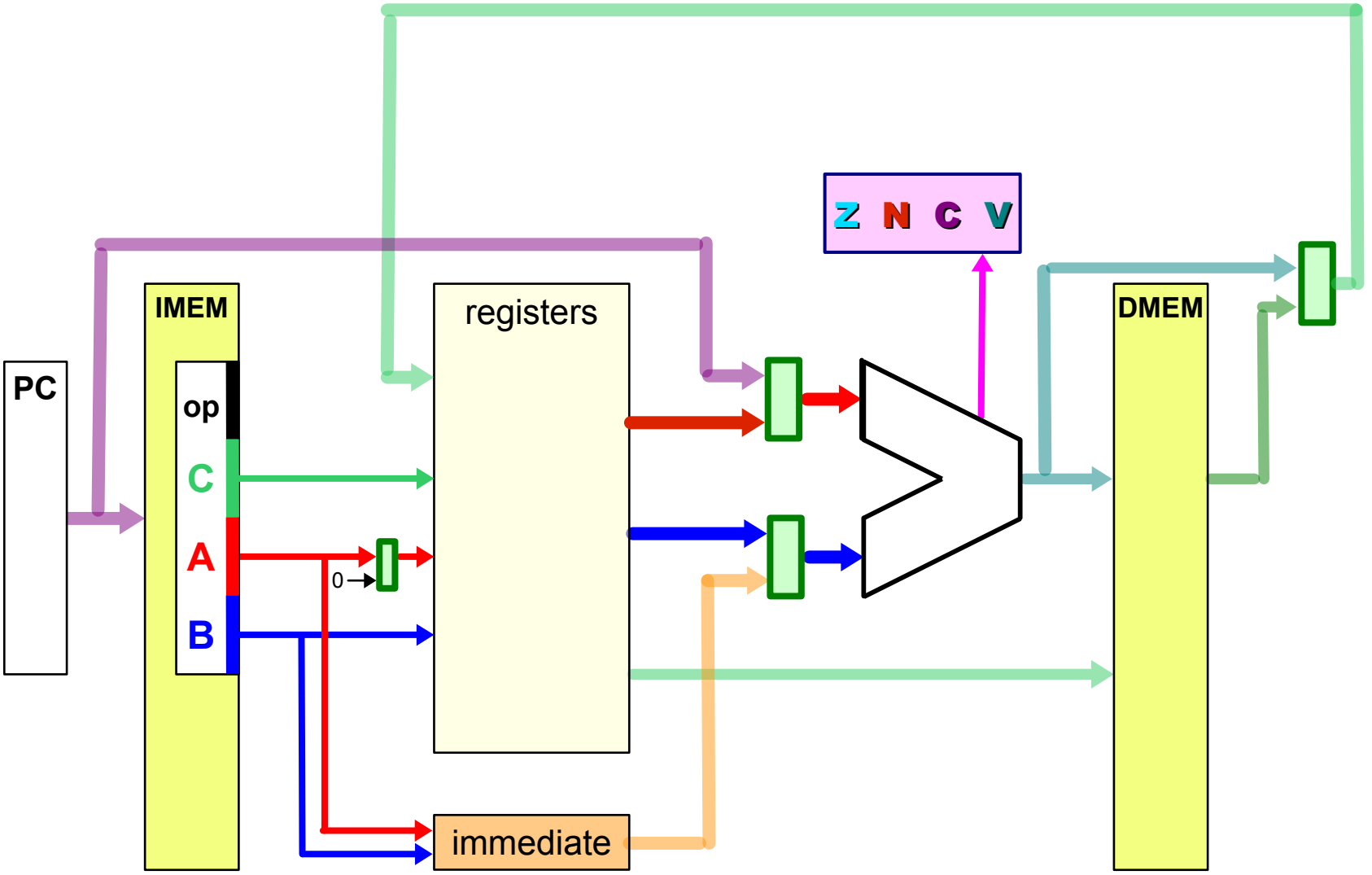
1111 1111 1010 0111 → B bus → **ALU**

1010 → CONDITION unit → branch?

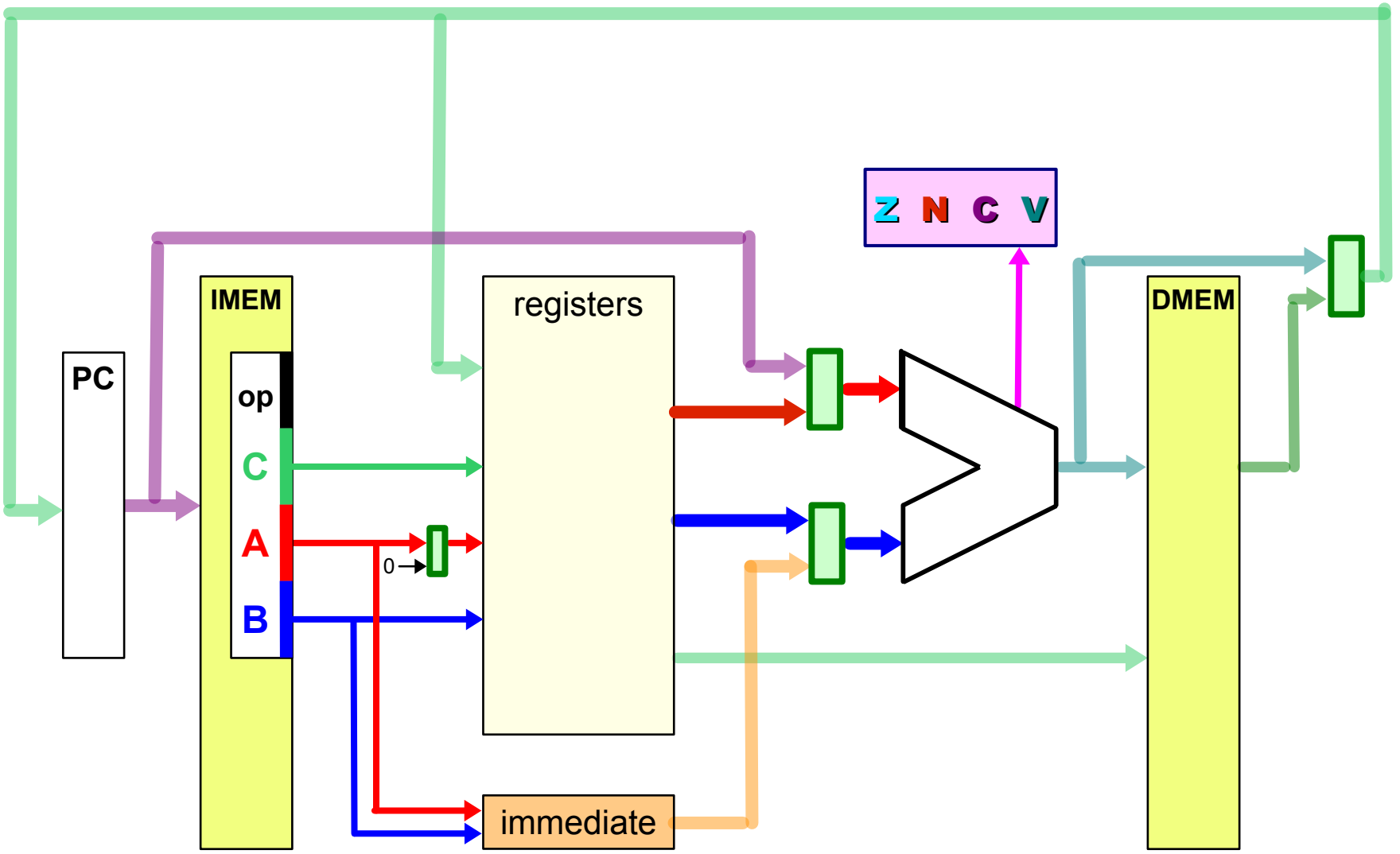
if branch? **ALU** → C bus → **PC**

nextInstruction (reset the cycle counter)

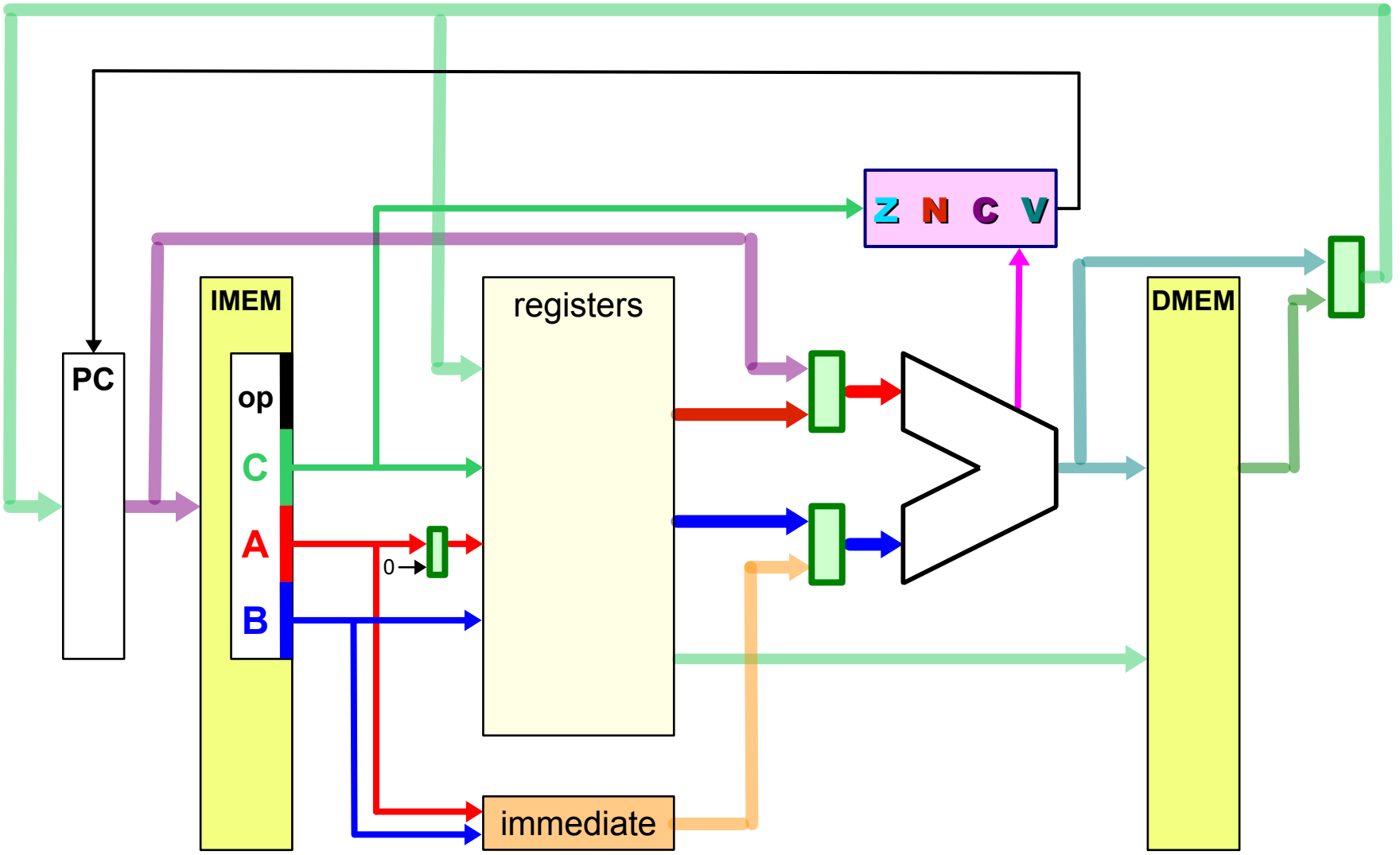
TOY Redesign w. **bc** (1)



TOY Redesign w. `bc` (2)



TOY Redesign w. `bc`



branch **c**onditional & **l**ink

bcl SLE, $F, $1 **0001** 0110 1111 0001

cycle 3: **regWrite**

1111 → **ALU** controller, 000 → **ALU**

[**PC**] → A bus → **ALU**

0000 → B MUX, 0000 → B bus → **ALU**

0001 → C decode, **ALU** → C bus → $1

cycle 4:

1111 → **ALU** controller; 000 → **ALU**

1111 → A MUX, [$F] → A bus → **ALU**

0000 → B MUX, 0000 → B bus → **ALU**

0110 → CONDITION unit → branch?

if branch? **ALU** → C bus → **PC**

nextInstruction (reset the cycle counter)

branch conditional & link

bcl SLE, $F, $1 **0001** 0110 1111 0001

cycle 3: **regWrite**

1111 → **ALU** controller, 000 → **ALU**

[**PC**] → A bus → **ALU**

0000 → B MUX, 0000 → B bus → **ALU**

0001 → C decode, **ALU** → C bus → $1

cycle 4:

1111 → **ALU** controller; 000 → **ALU**

1111 → A MUX, [$F] → A bus → **ALU**

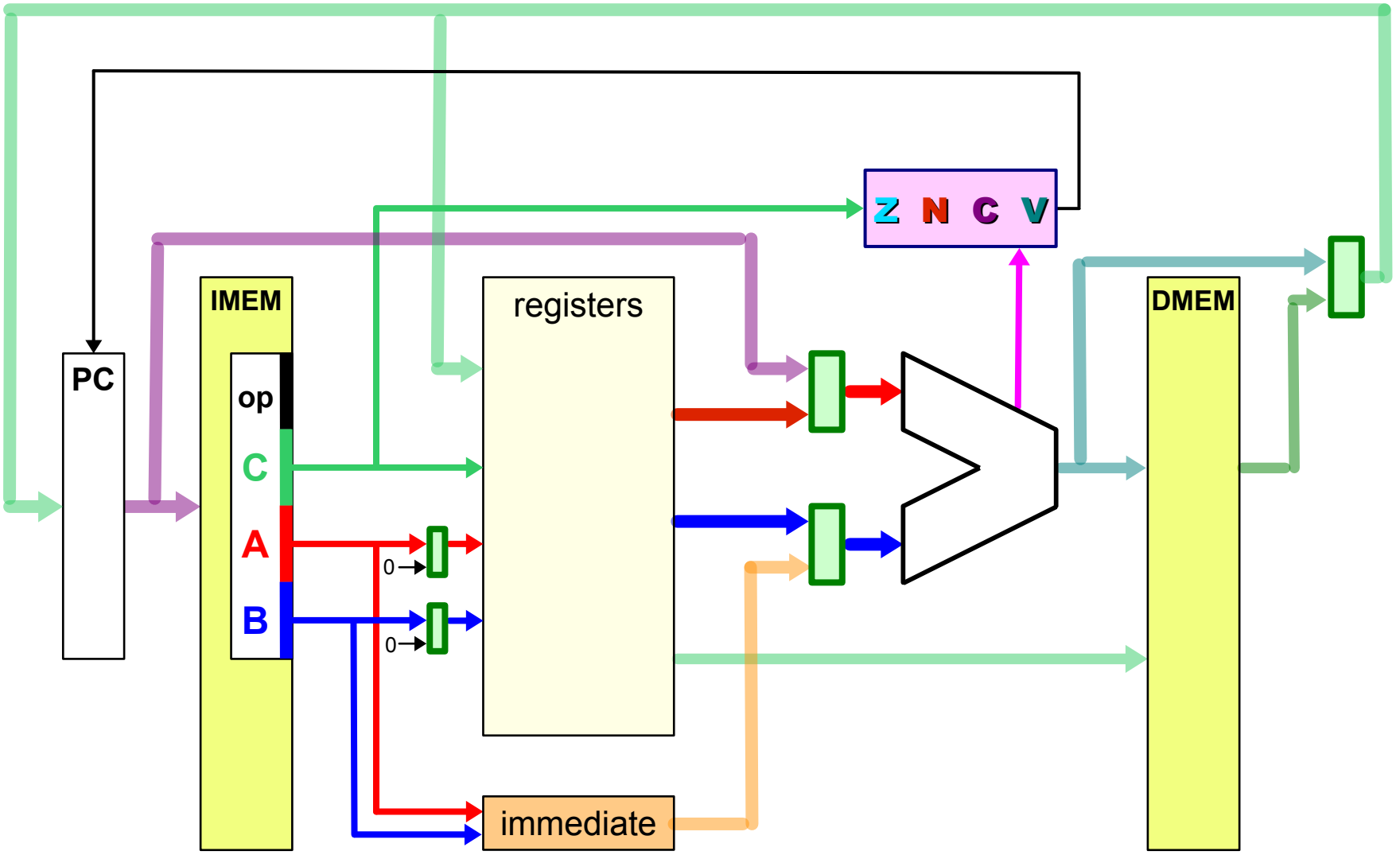
0000 → B MUX, 0000 → B bus → **ALU**

0110 → CONDITION unit → branch?

if branch? **ALU** → C bus → **PC**

nextInstruction (reset the cycle counter)

TOY Redesign w. `blc` (?)



branch **c**onditional & **l**ink

bcl ALL, $1, $0 **0001** 0110 0001 0000

cycle 3:

1111 → **ALU** controller, 000 → **ALU**

0001 → A MUX, [$1] → A bus → **ALU**

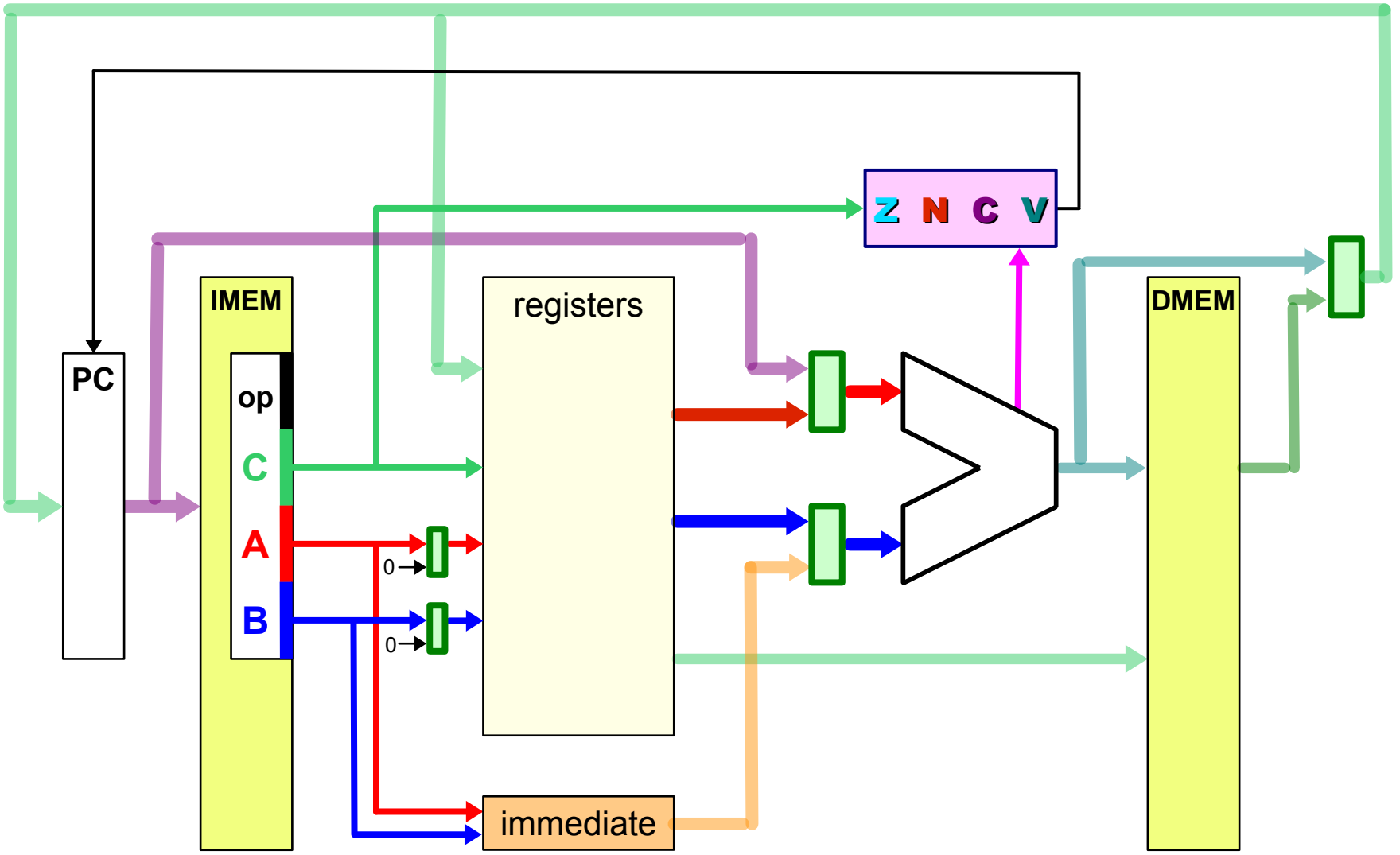
0000 → B MUX, 0000 → B bus → **ALU**

0000 → CONDITION unit → branch?

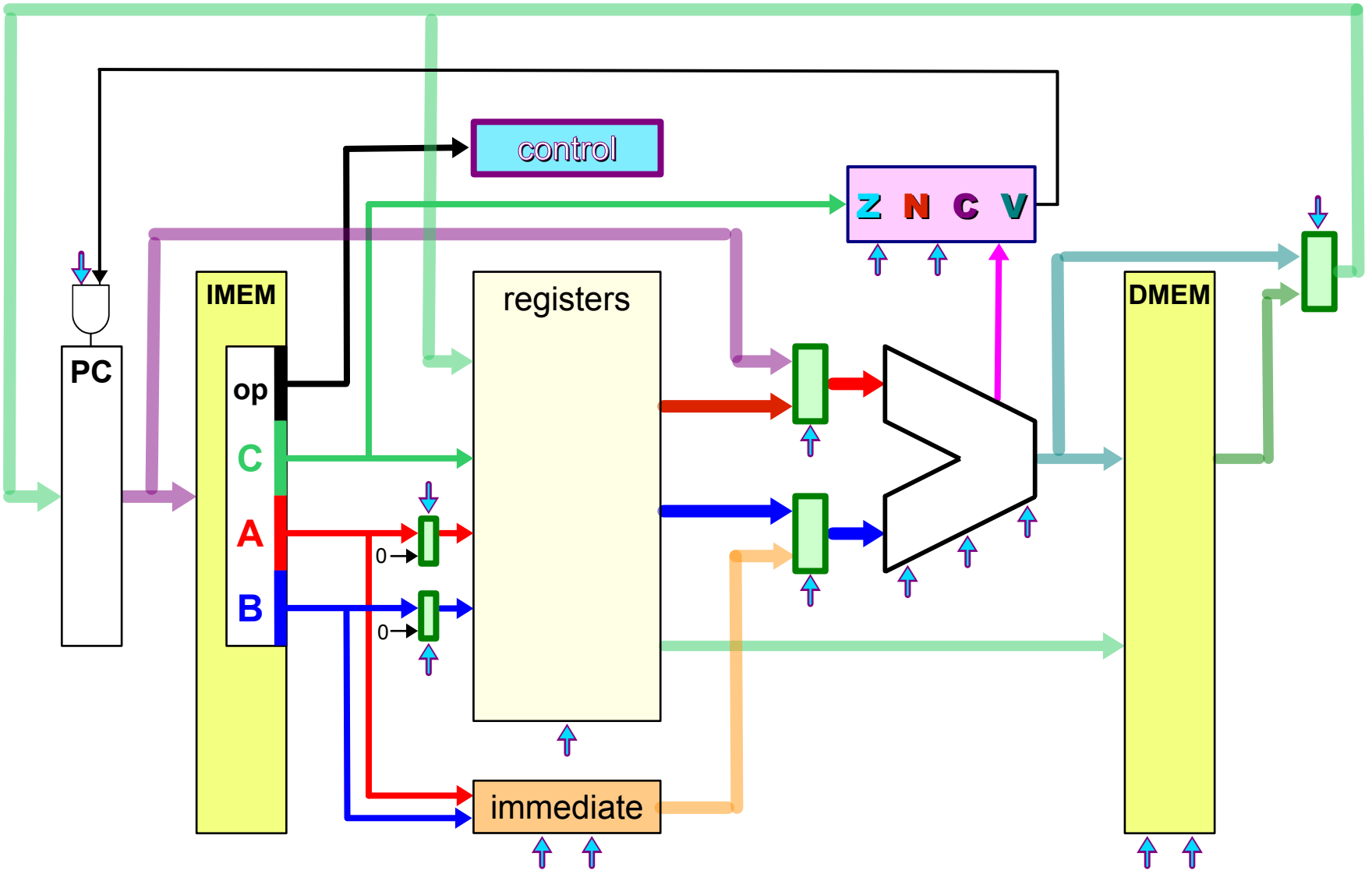
if branch? **ALU** → C bus → **PC**

nextInstruction (reset the cycle counter)

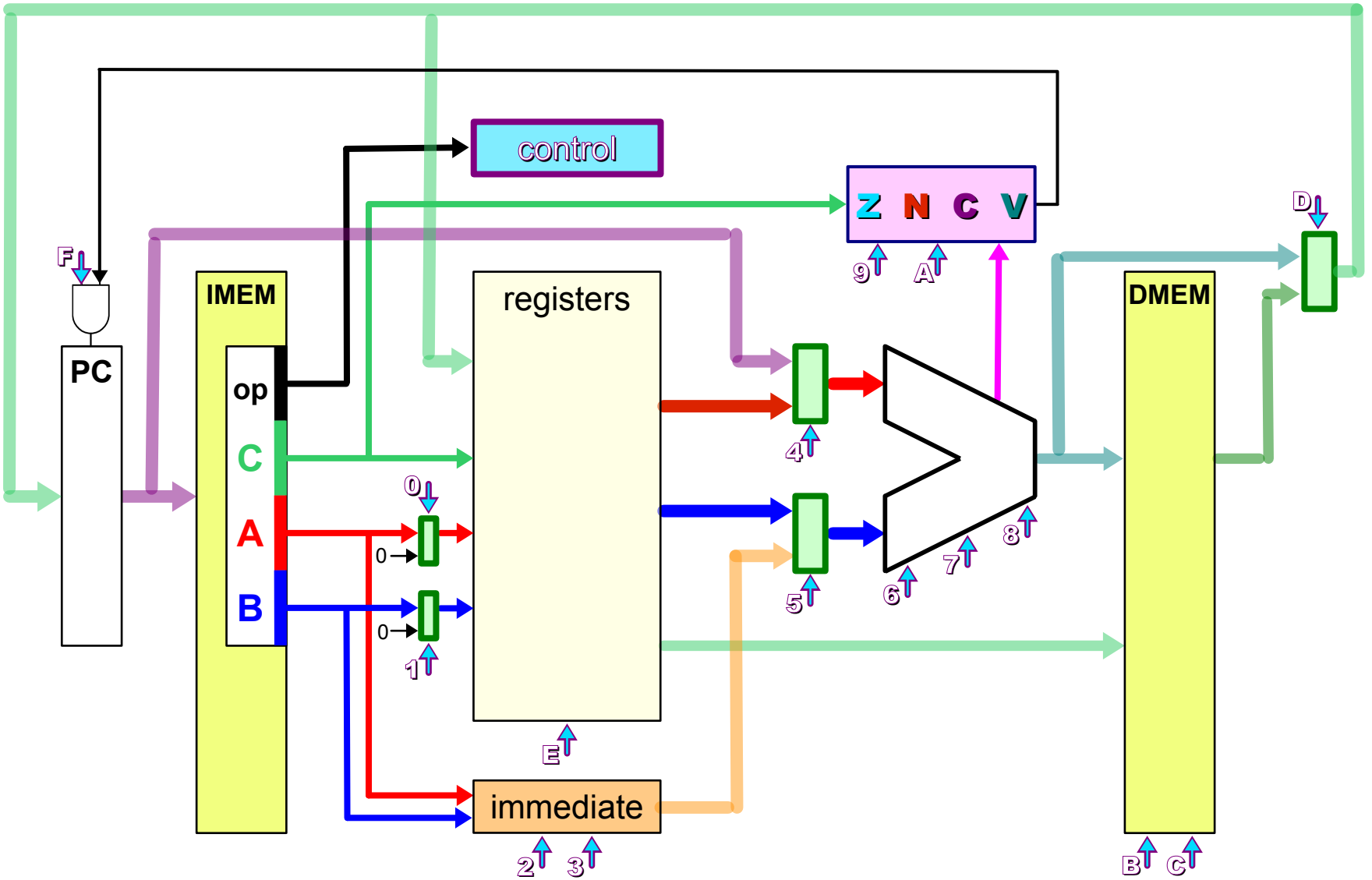
TOY Redesign *Datapath*



TOY Redesign w. **Control**



TOY Redesign w. **Control**



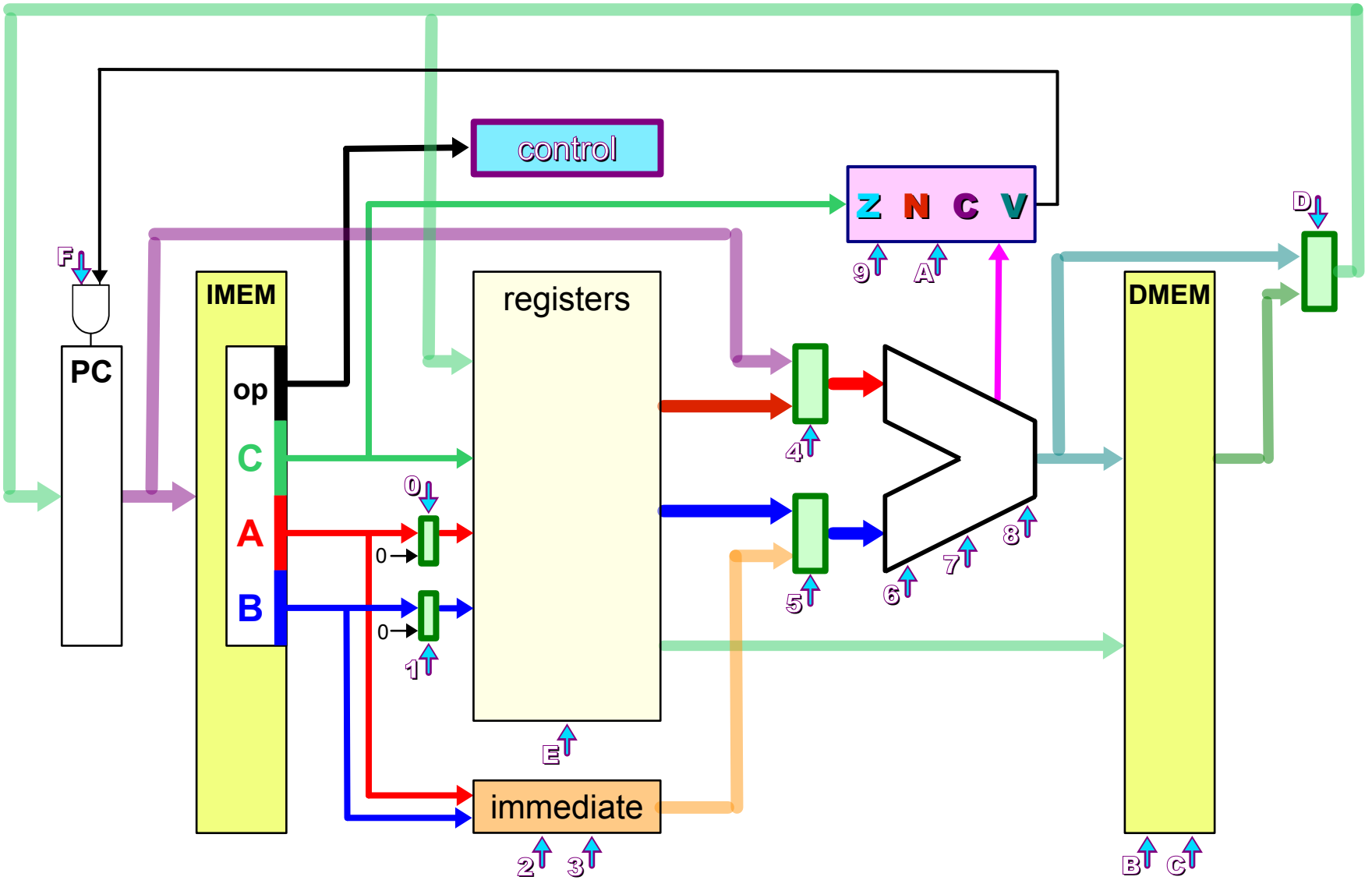
TOY Redesign Control Signals

	?	0	1
0	**A** register address	**IR**[7..4]	0
1	**B** register address	**IR**[3..0]	0
2	Immediate input	**IR**[7..0]	0000 **IR**[3..0]
3	Immediate output	(sign extended) [7..0]	[F..8]
4	**B** bus source	**B** register	Immediate value
5	**A** bus source	**A** register	**PC**
6	Invert **A** bus	no	yes
7	Invert **B** bus	no	yes
8	ALU output	Arithmetic	Logical
9	Condition reg write	no	**Z N**
A	Condition reg write	no	**C V**
B	Read data memory	no	yes
C	Write data memory	no	yes
D	**C** bus source	**ALU**	Data memory
E	Write data memory	no	yes
F	Branch instruction	no	yes

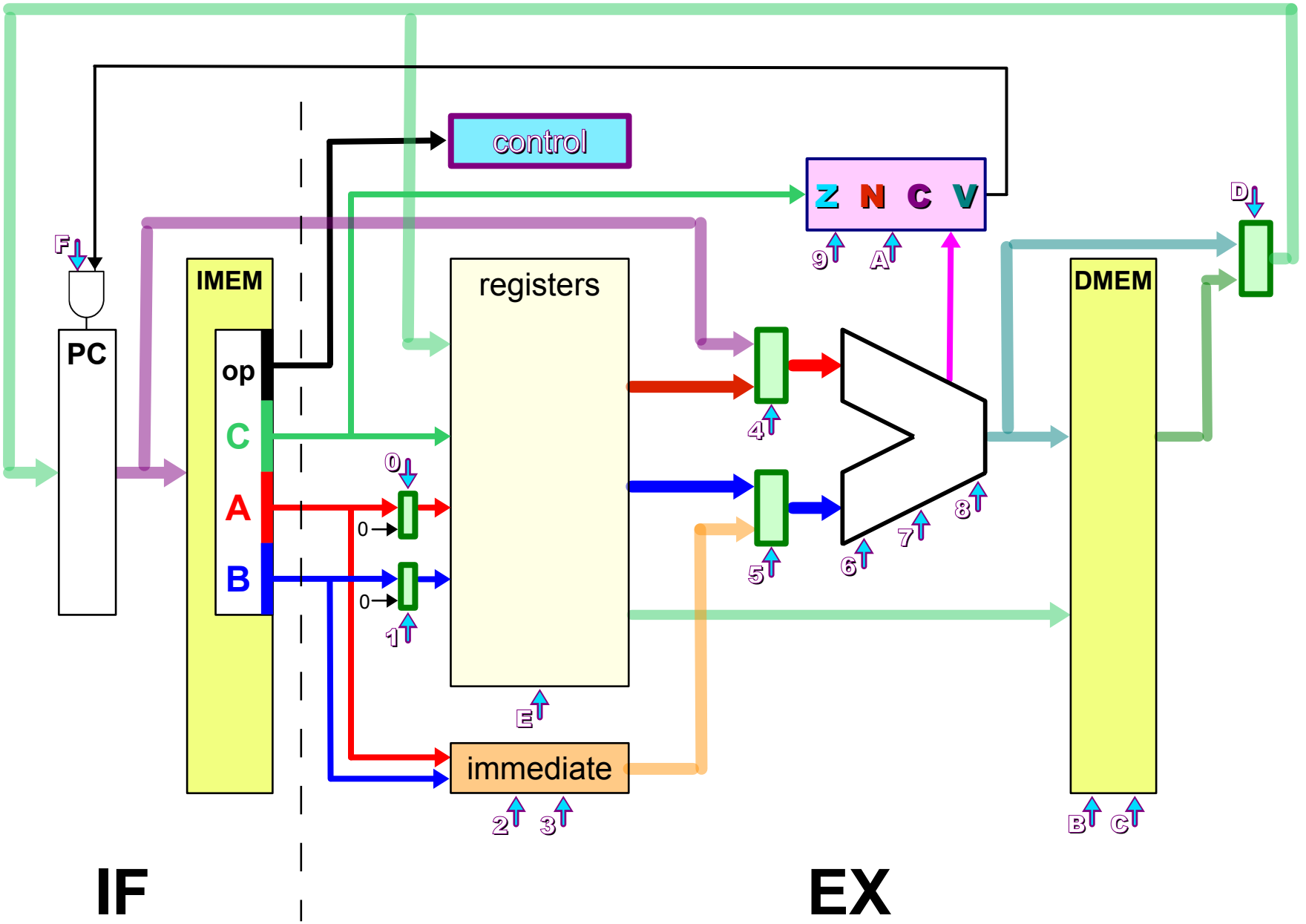
TOY Redesign Control Signals

	?	0	1
0	**A** register address	**IR**[7..4]	0
1	**B** register address	**IR**[3..0]	0
2	Immediate input	**IR**[7..0]	0000 **IR**[3..0]
3	Immediate output	(sign extended) [7..0]	[F..8]
4	**B** bus source	**B** register	Immediate value
5	**A** bus source	**A** register	**PC**
6	Invert **A** bus	no	yes
7	Invert **B** bus	no	yes
8	ALU output	Arithmetic	Logical
9	Condition reg write	no	**Z N**
A	Condition reg write	no	**C V**
B	Read data memory	no	yes
C	Write data memory	no	yes
D	**C** bus source	**ALU**	Data memory
E	Write data memory	no	yes
F	Branch instruction	no	yes

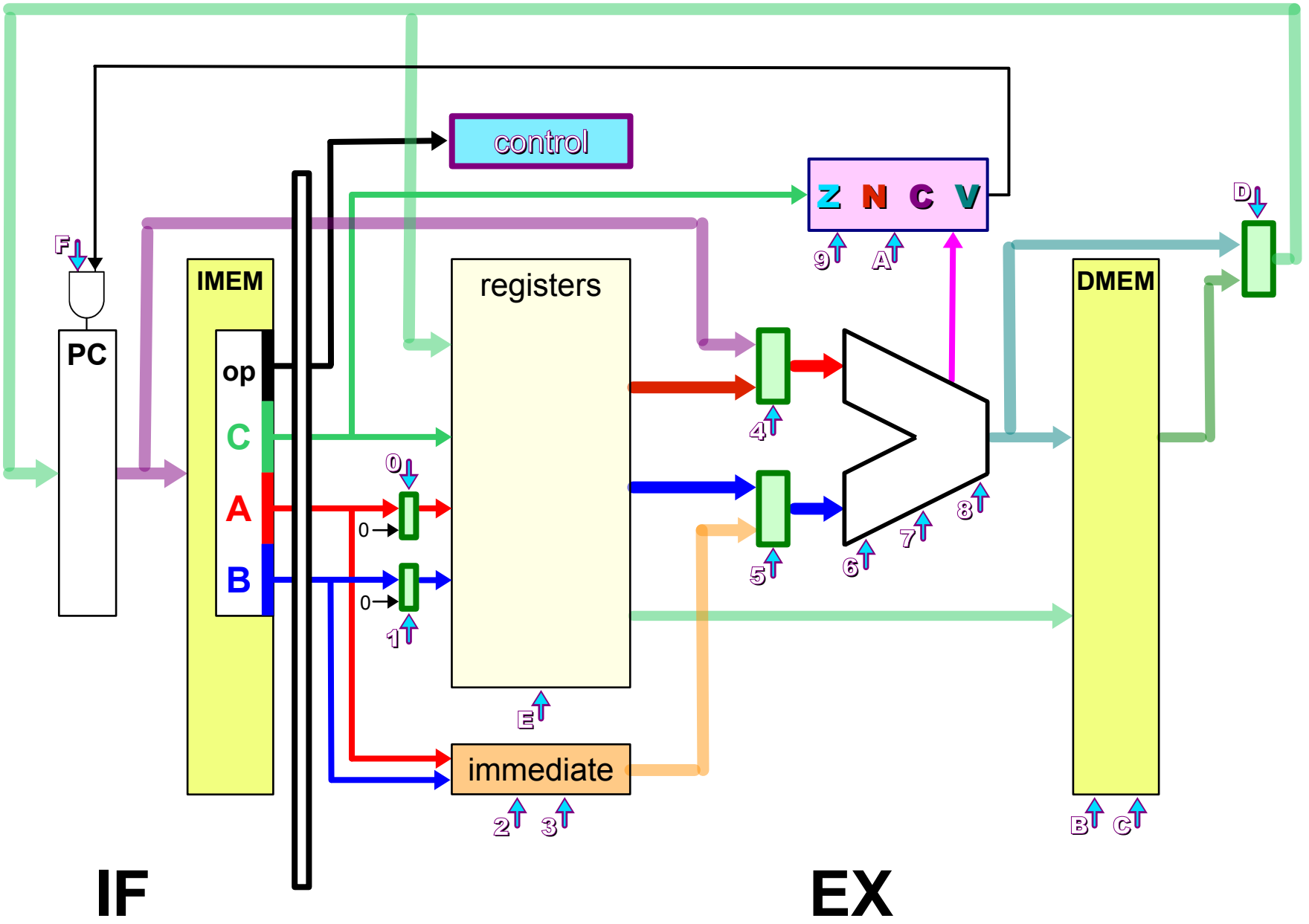
TOY Single Cycle Implementation



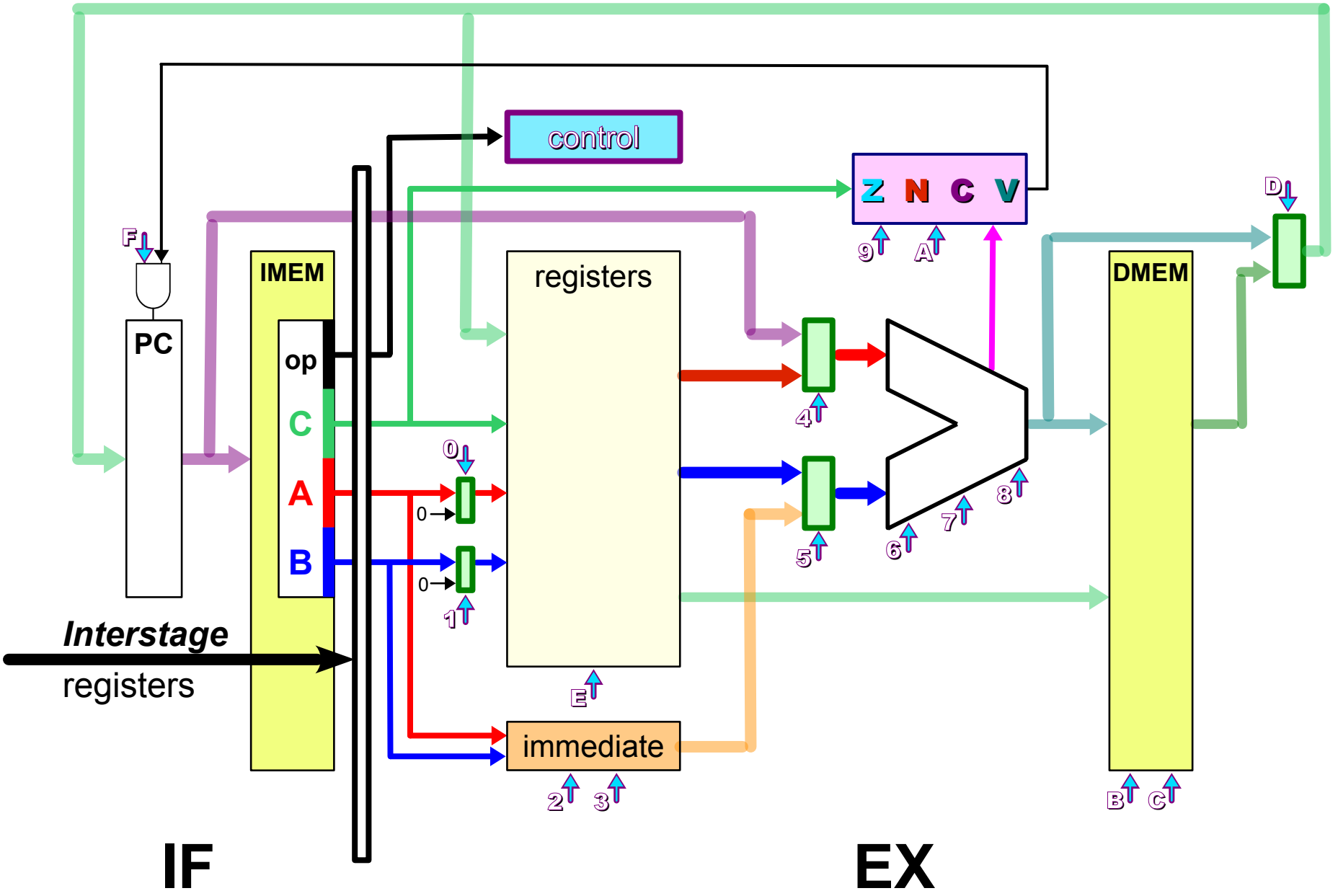
TOY Pipeline Implementation?



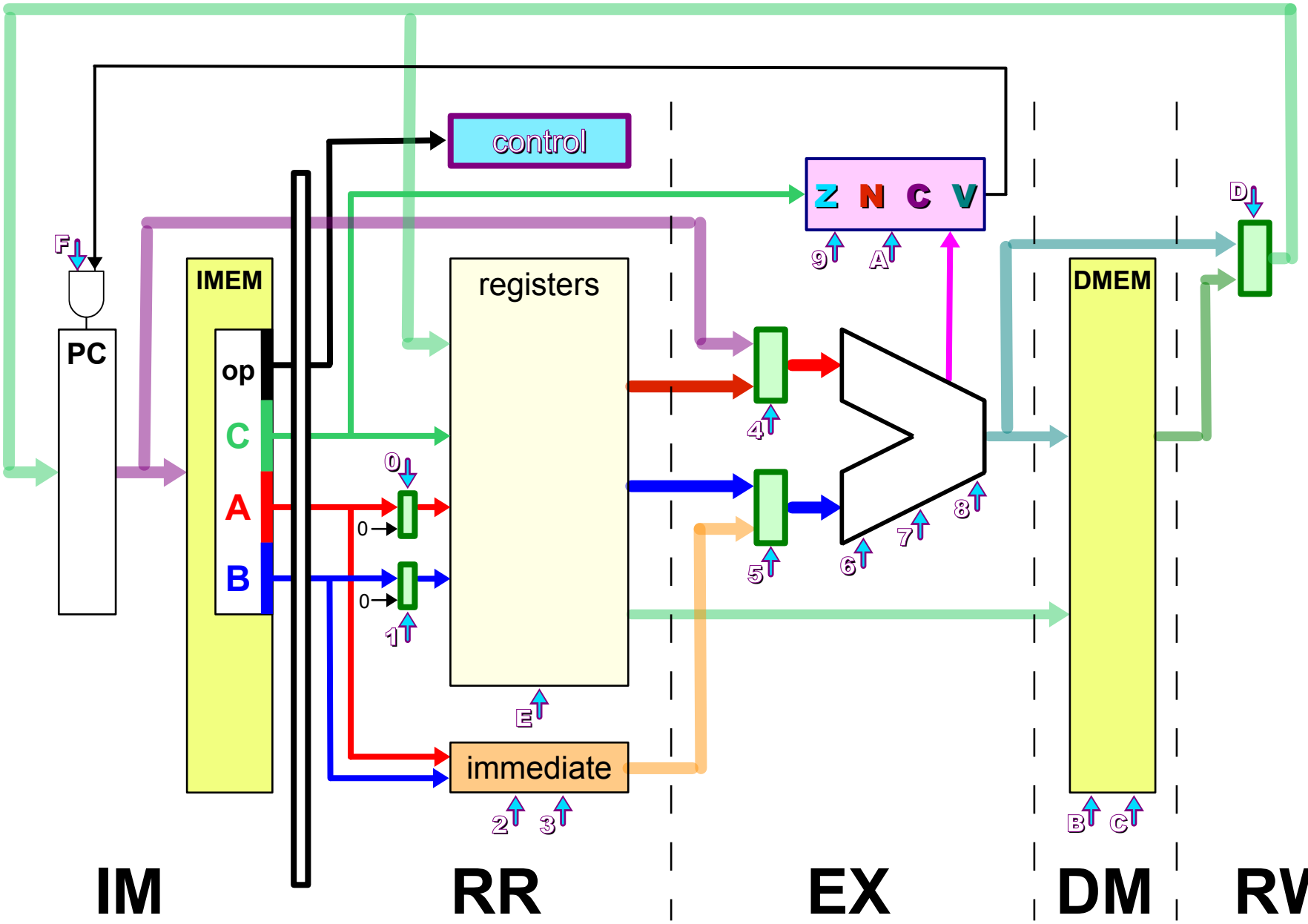
TOY 2 Stage Pipeline



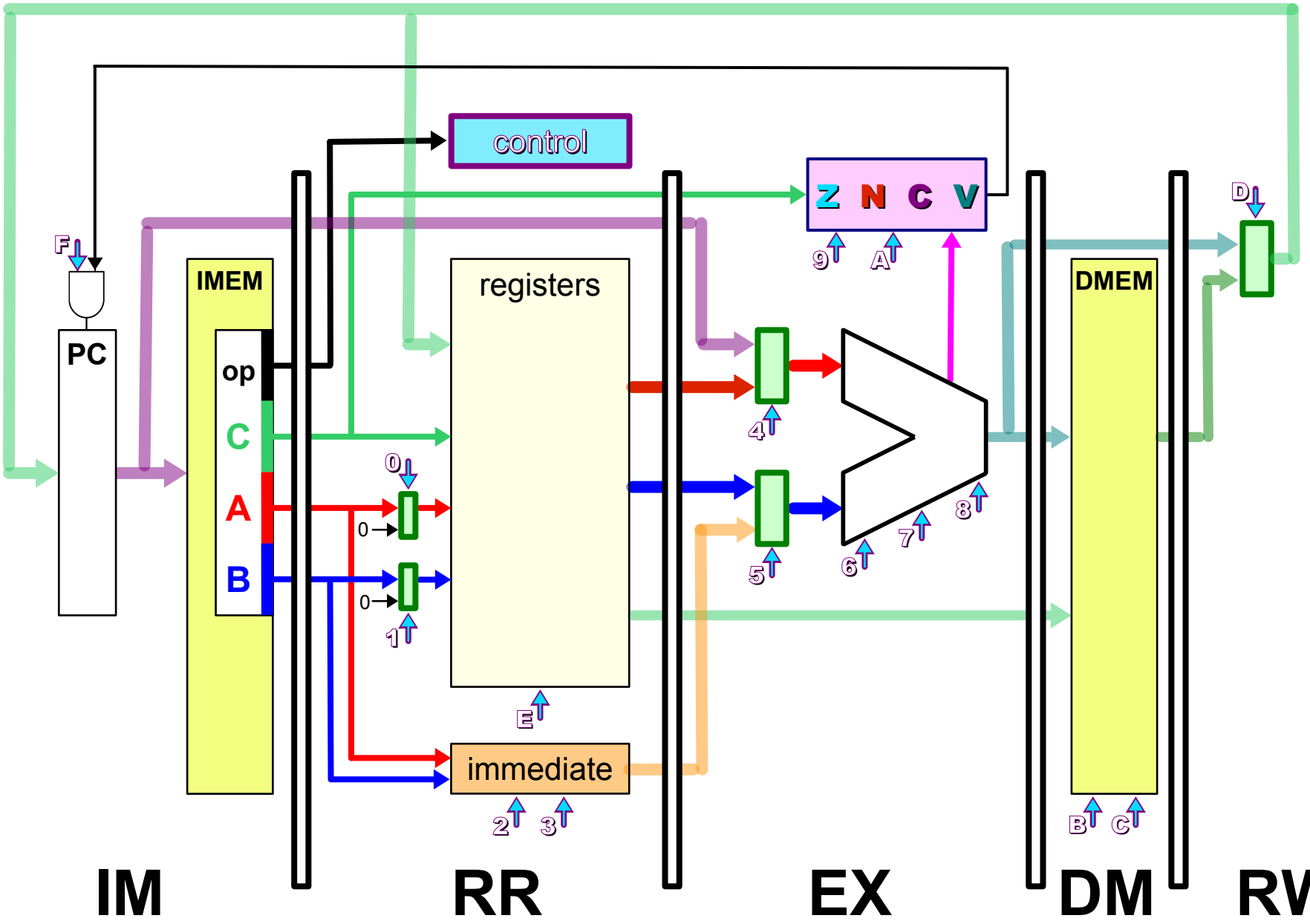
TOY 2 Stage Pipeline



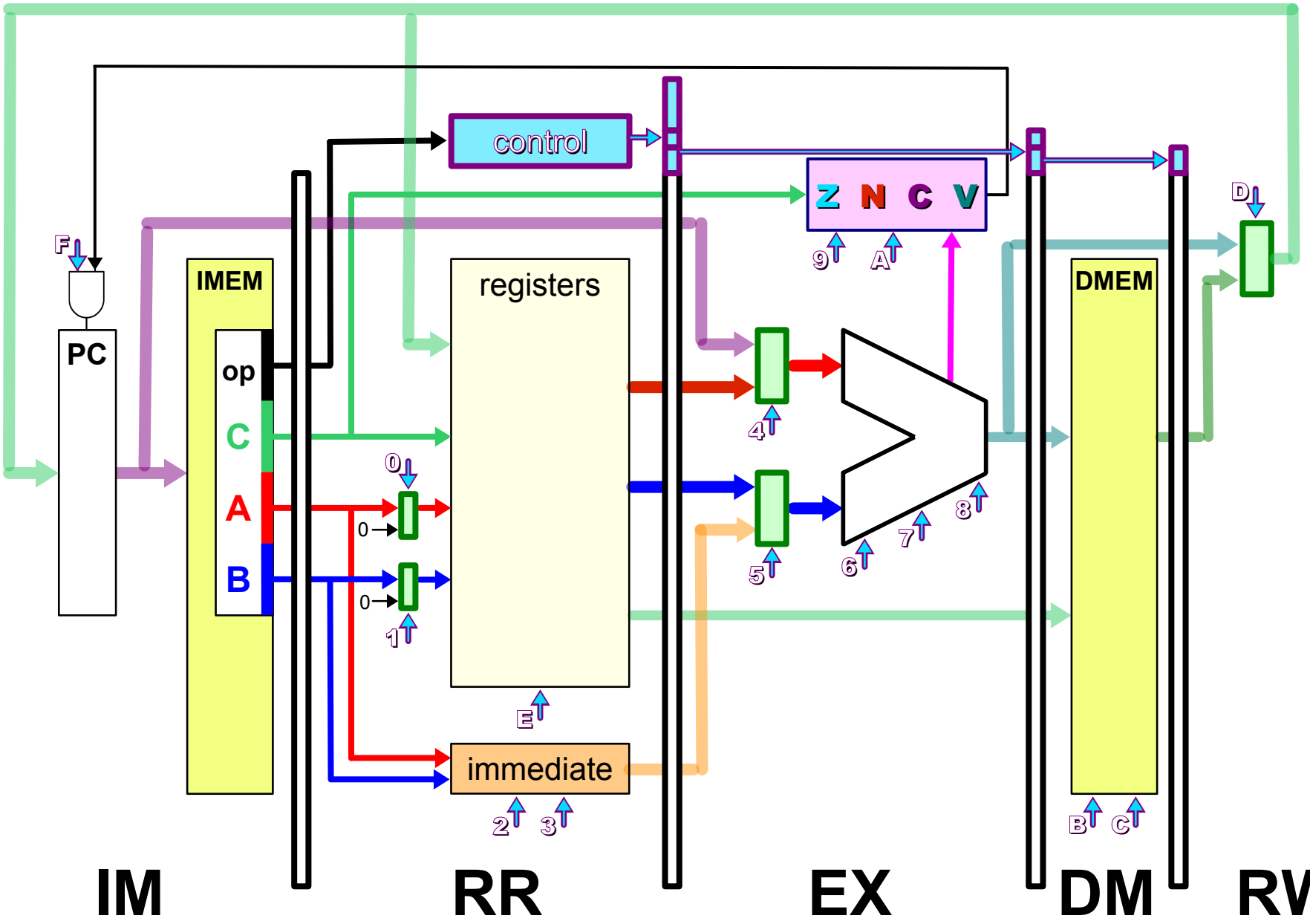
TOY 5 Stage Pipeline ??



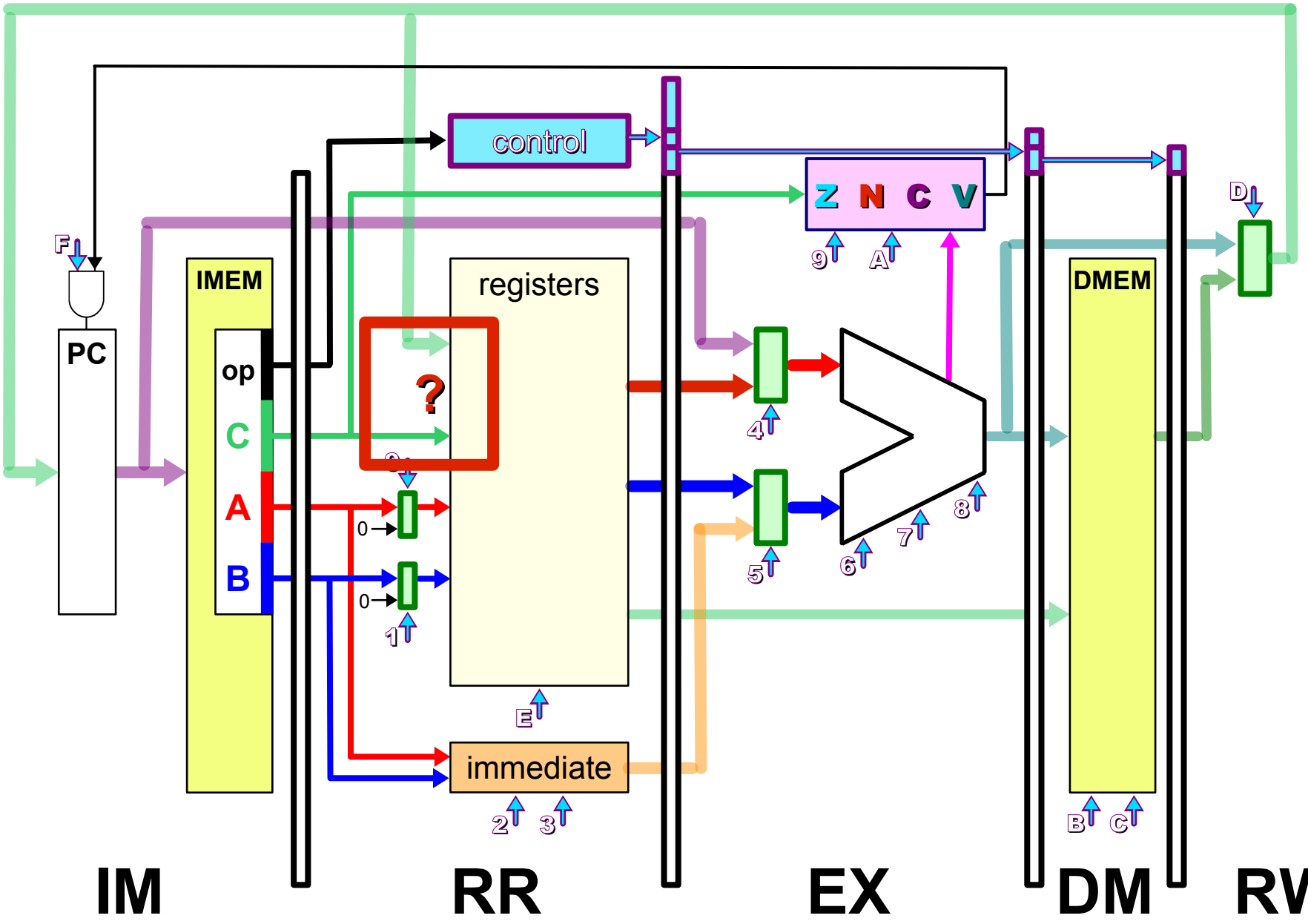
TOY 5 Stage Pipeline ?



TOY 5 Stage Pipeline ?



TOY 5 Stage Pipeline ?



TOY 5 Stage Pipeline !