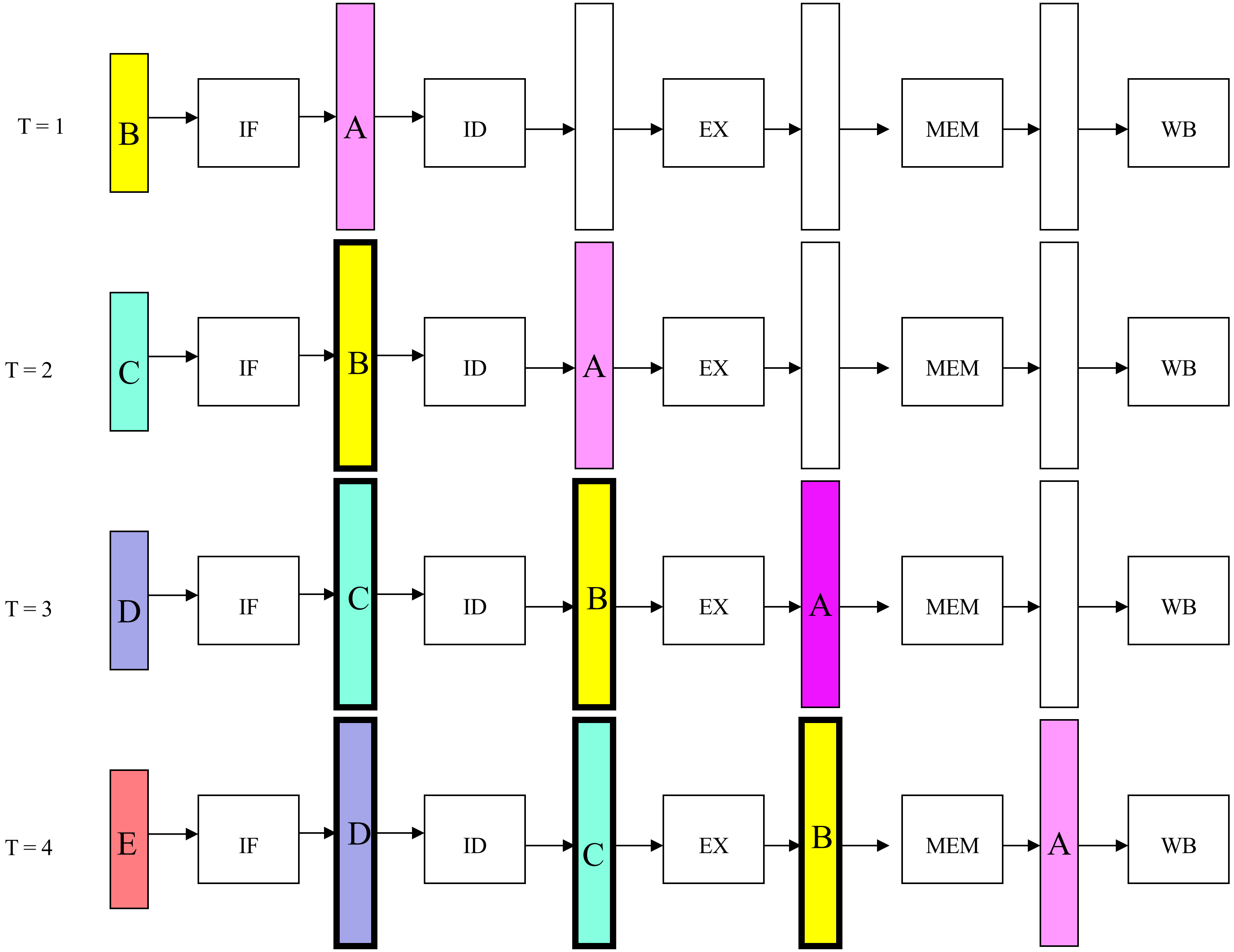
Ideal Pipeline



Ideal Pipeline Summary

Pipeline Depth (Number if stages) :p

MIPS = IPC x Clock Frequency (Hz) x 10^{-6}

CPI = n+ p -1 (cycles) / n (instructions) = 1 + 4/n

Approximately CPI = 1 for large n with no stalls

Clock frequency : Roughly 5 times that of base design

- Limited by the delay through the pipeline stage
  - Faster circuitry
  - Less circuitry with more stages
  - Will be ultimately limited by pipeline register delays
    - In practice useful functions require at least 10 gate delays

- Deep pipelines with very thin stages
  - Clock rate (frequency) increase
  - Instruction throughput (MIPS) increases in ideal pipeline
  - Single instruction latency increase
  - May cause greater number of stall cycles with data and control hazards

Blank Slide

Pipeline Hazards

- What factors force deviations from this ideal pipeline?

- Hazard
  - Condition that disrupts the orderly flow of instructions
  - Requires special attention by hardware and/or software to maintain program correctness and performance

1. Structural Hazards

   Contention for hardware resources

2. Control Hazards

   Disruptions caused by program control flow

3. Data Hazards

   Data dependencies between instructions

Structural Hazards

Contention for hardware resources

- Memory Contention

  Instructions in IF and MEM stages may contend for memory
  We will assume separate instruction and data memories to avoid conflict

- ALU Contention

  Only 1 instruction doing an ALU operation at any time

  No contention in this pipeline

- Register File (RF) Contention
  - Instruction writes to RF while other instruction is reading RF

Register Contention Hazard

Time →	1	2	3	4	5	6	7	8	9	10
ADD R3, R1, R2	IF	ID	EX	MEM	**WB**					
B		IF	ID	EX	MEM	WB				
C			IF	ID	EX	MEM	WB			
SUB R4, R5, R6				IF	**ID**	EX	MEM	WB		
					IF	ID	EX	MEM	WB	
						IF	ID	EX	MEM	WB

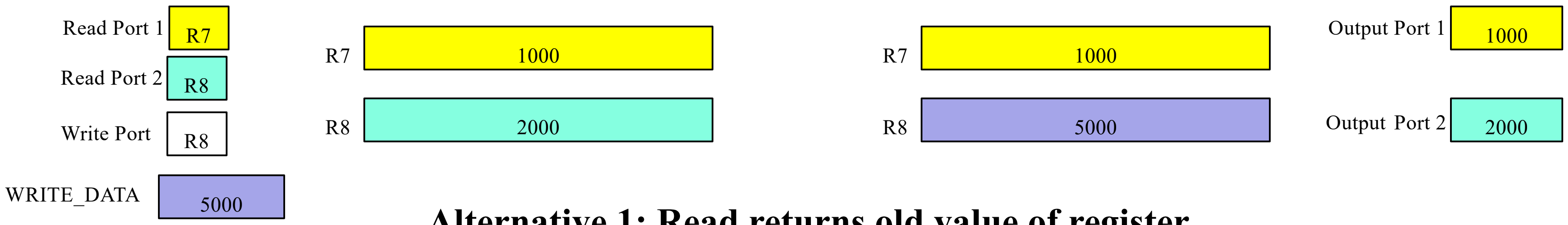
Instructions A (ADD) and D **(SUB)** contend for register file at cycle 5

- ADD is writing to register R3 at cycle 5
- SUB is reading from registers R5 and R6 at cycle 5

- How do we handle the contention for the register file?

Multi-ported Register File Design 1

- 2 Read Ports and 1 Write Port in the Register File
  - At clock edge C1:
    - Read and Write Register numbers and WRITE_DATA are available at register-file input
  - At clock edge C2:
    - Register read values clocked into ID/EX pipeline register. WRITE_DATA clocked into R1
    - What if write register and read register are the same?



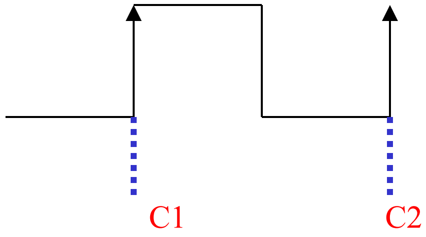
Alternative 1: Read returns old value of register

At C1

At C2

Multi-ported Register File Design 2

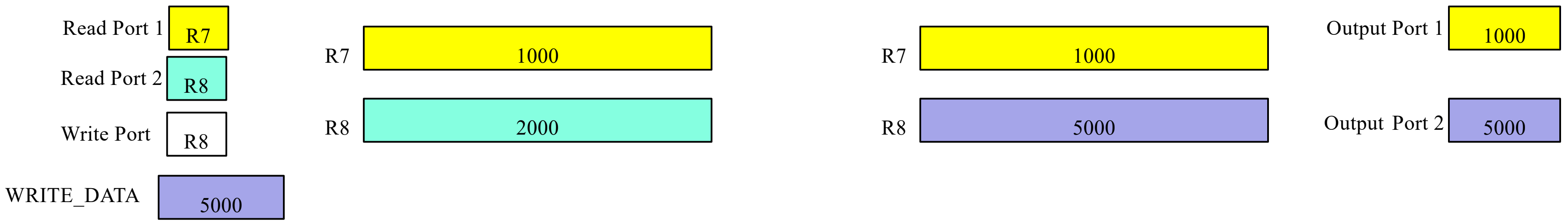
- 2 Read Ports and 1 Write Port in the Register File



READ PORT 1:	R7	
READ PORT 2:	R8	
WRITE PORT:	R8	WRITE_DATA

Write register and Read register is the same.

Alternative 2: Read returns new value of register. (Assume this protocol)



Read Port 1 R7

Read Port 2 R8

Write Port R8

WRITE_DATA 5000

R7 1000

R8 2000

R7 1000

R8 5000

Output Port 1 1000

Output Port 2 5000

At C1

At C2

Blank Slide

Control Hazards

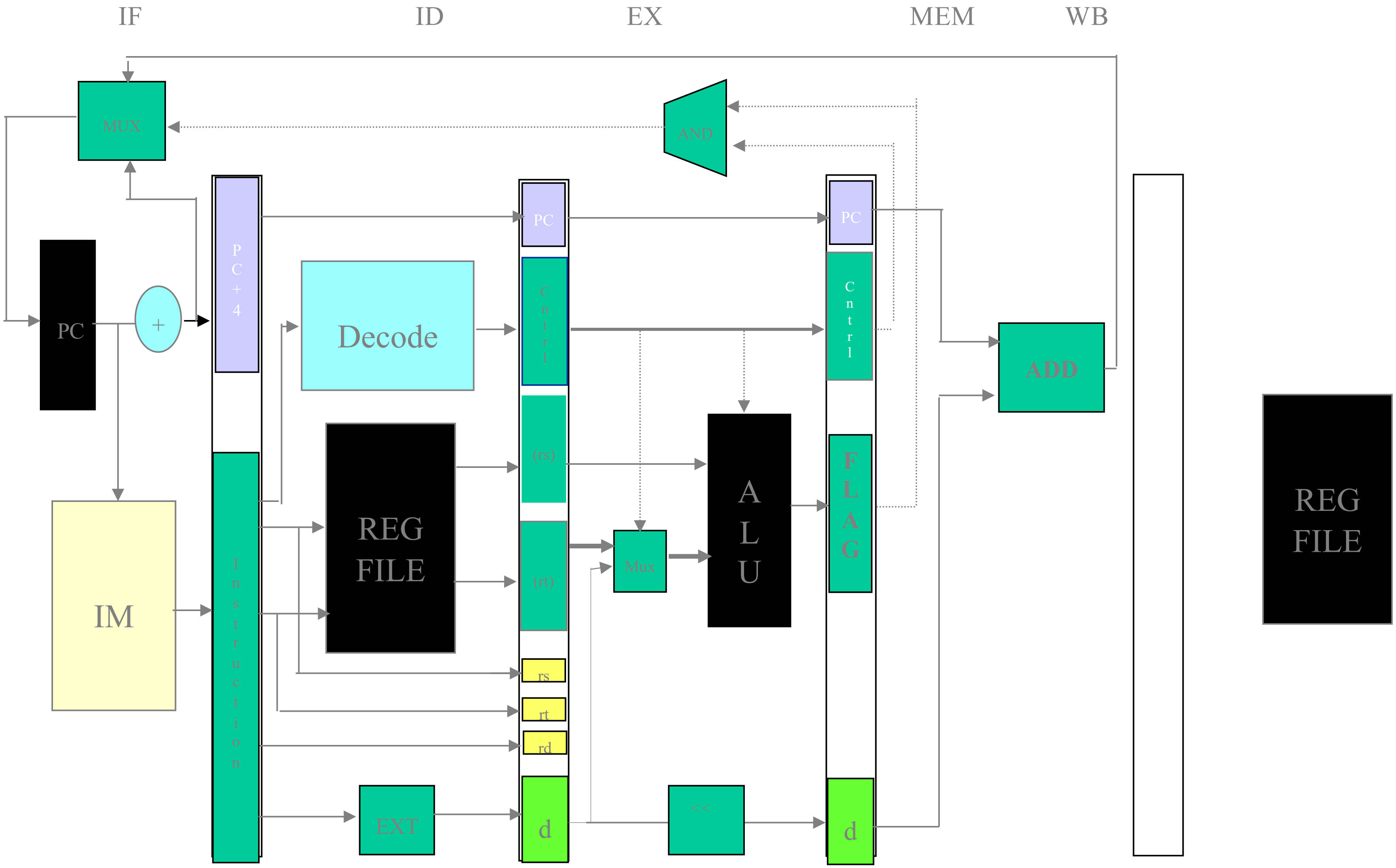
```
A:   beq R2, R3, L1
          B
          C
          D
          E
          ---

L1:       X
```

1. Compute Target Address

2. Determine Branch Outcome – Taken or Not Taken

- In our relaxed design: the branch was resolved only in the MEM stage
- The PC was updated with the target address only at the end of cycle 4
- What happens to the pipeline between fetching BEQ and updating PC with the target address?

Lazy Execution of Branch Equal Instruction: **beq Rt, Rs, d**



Control Hazards

	1	2	3	4	5	6	7	8	9	10	11	12
BEQ	**IF**	**ID**	**EX**	**M**	**WB**							
B		IF	ID	EX	M	WB						
C			IF	ID	EX	M	WB					
D				IF	ID	EX	M	WB				
X					**IF**	**ID**	**EX**	**M**	**WB**			
Y						**IF**	**ID**	**EX**	**M**	**WB**		
Z							**IF**	**ID**	**EX**	**M**	**WB**	

- Instructions B, C, D will enter the pipeline while branch outcome is being determined
- Need to make sure they do not compromise correctness
- Want to reduce the performance degradation

Control Hazards

- Branch Delay Slots
  - Expose the branch delay to the software
  - Compiler puts NOPS in the slots after the branch instruction
  - Compiler puts useful instructions in the slots after branch instruction

- Hardware introduced stall cycles

Branch Delay Slots

beqz R2, R3, L1

NOP
NOP
NOP
B

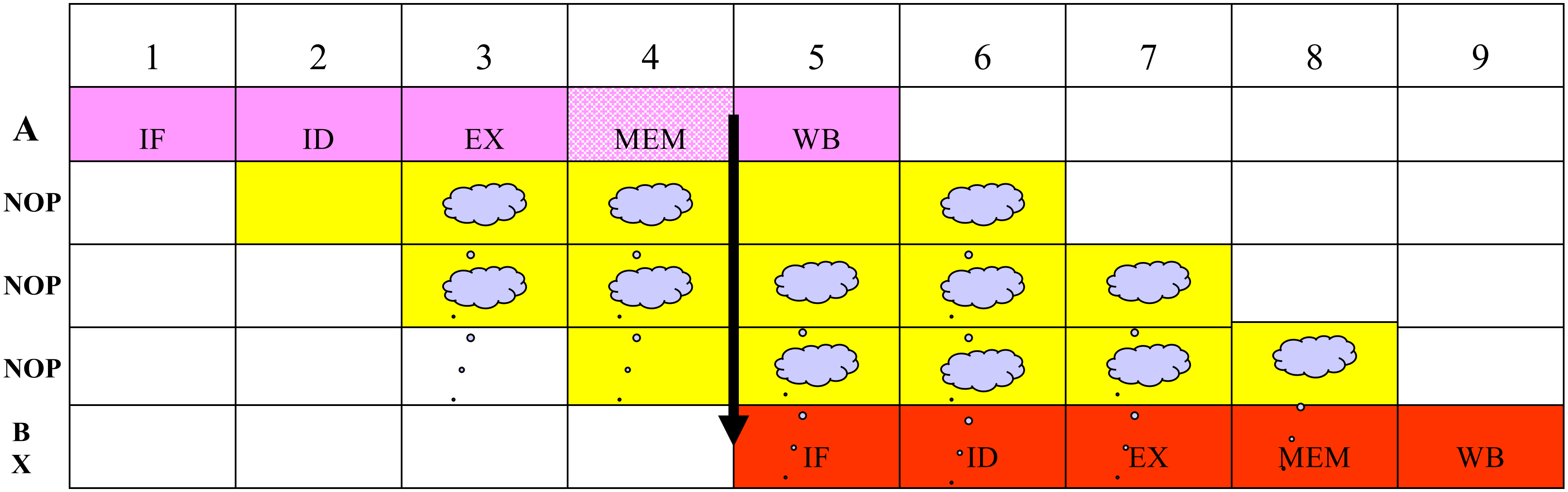
L1: **X**

Possible execution sequences:

Branch Not Taken: **A**, NOP, NOP, NOP, **B**

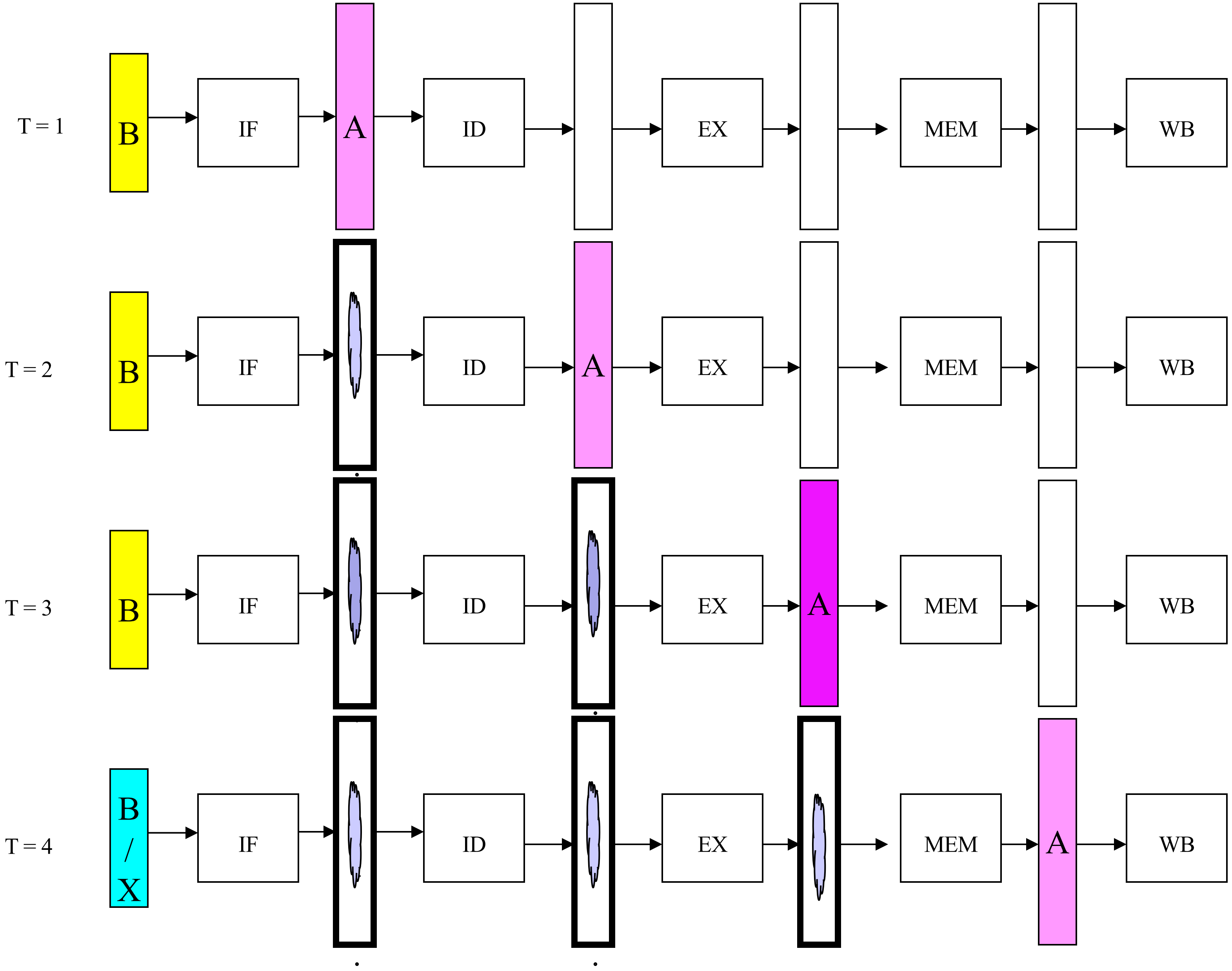
Branch Taken: **A**, NOP, NOP, NOP, **X**

- Adds 3 cycles to execution time for every branch
- Compiler finds useful instructions to put in the Branch Delay Slots



	1	2	3	4	5	6	7	8	9
A	IF	ID	EX	MEM	WB				
NOP									
NOP									
NOP									
B **X**					IF	ID	EX	MEM	WB

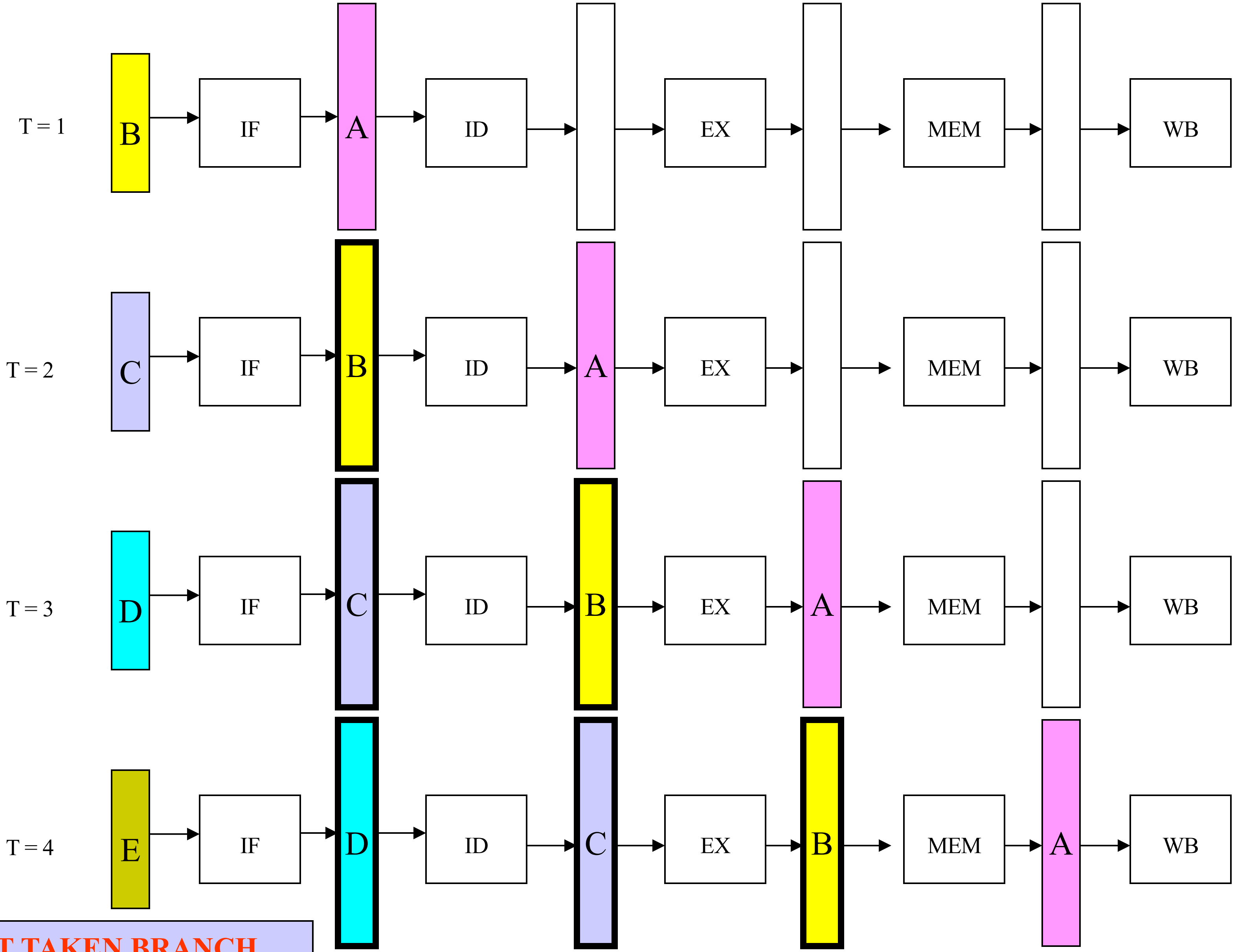
Hardware Induced Stalls



Control Hazards

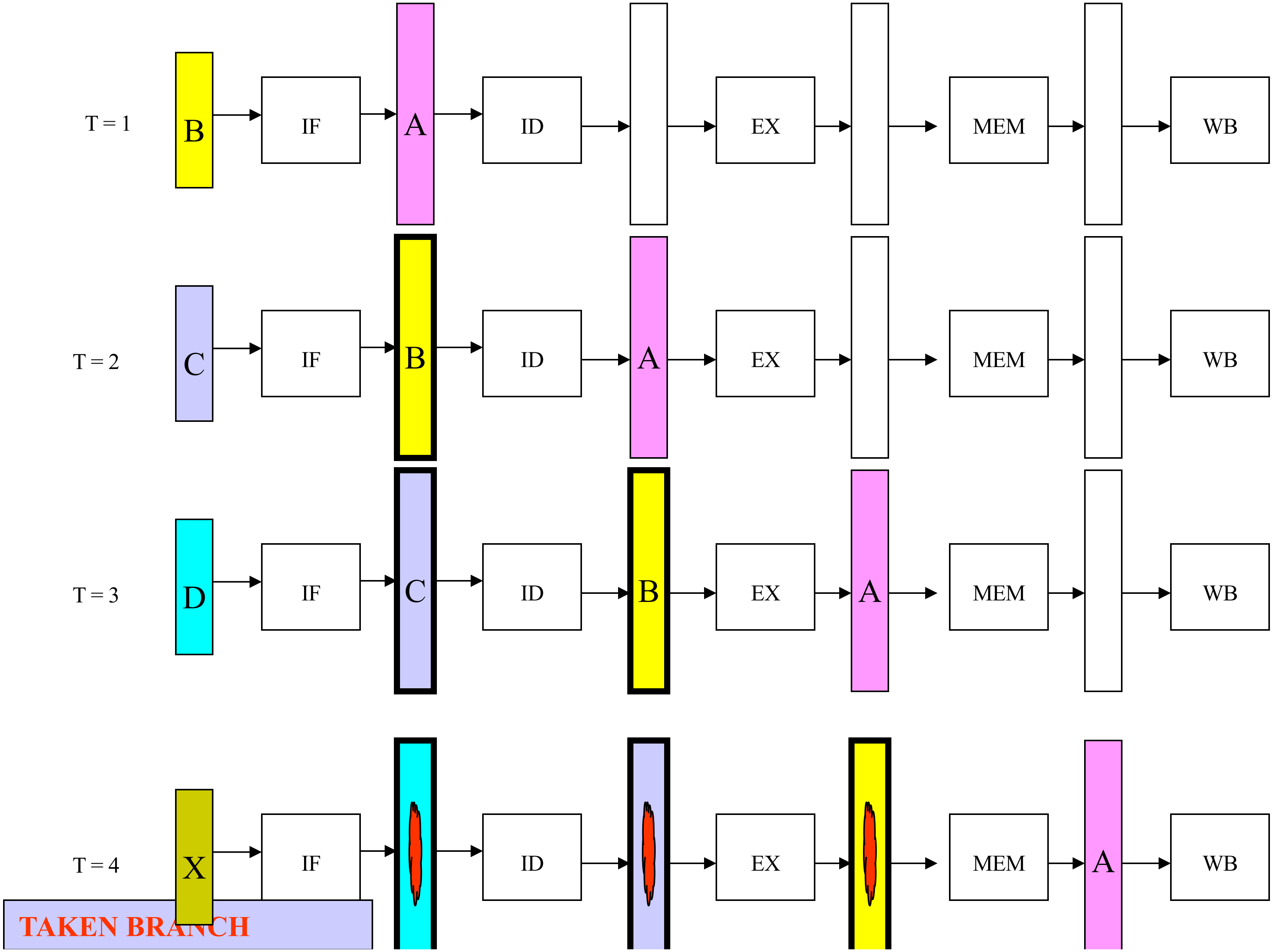
- Speculative Execution
  - Predict whether the branch will be taken or not taken
  - Fetch and execute instructions from predicted address
  - Rollback mechanism to undo wrongly speculated instructions

Predict Branch Not Taken



NOT TAKEN BRANCH

Predict Branch Not Taken



Predict Branch Taken

:

Additional Problem:

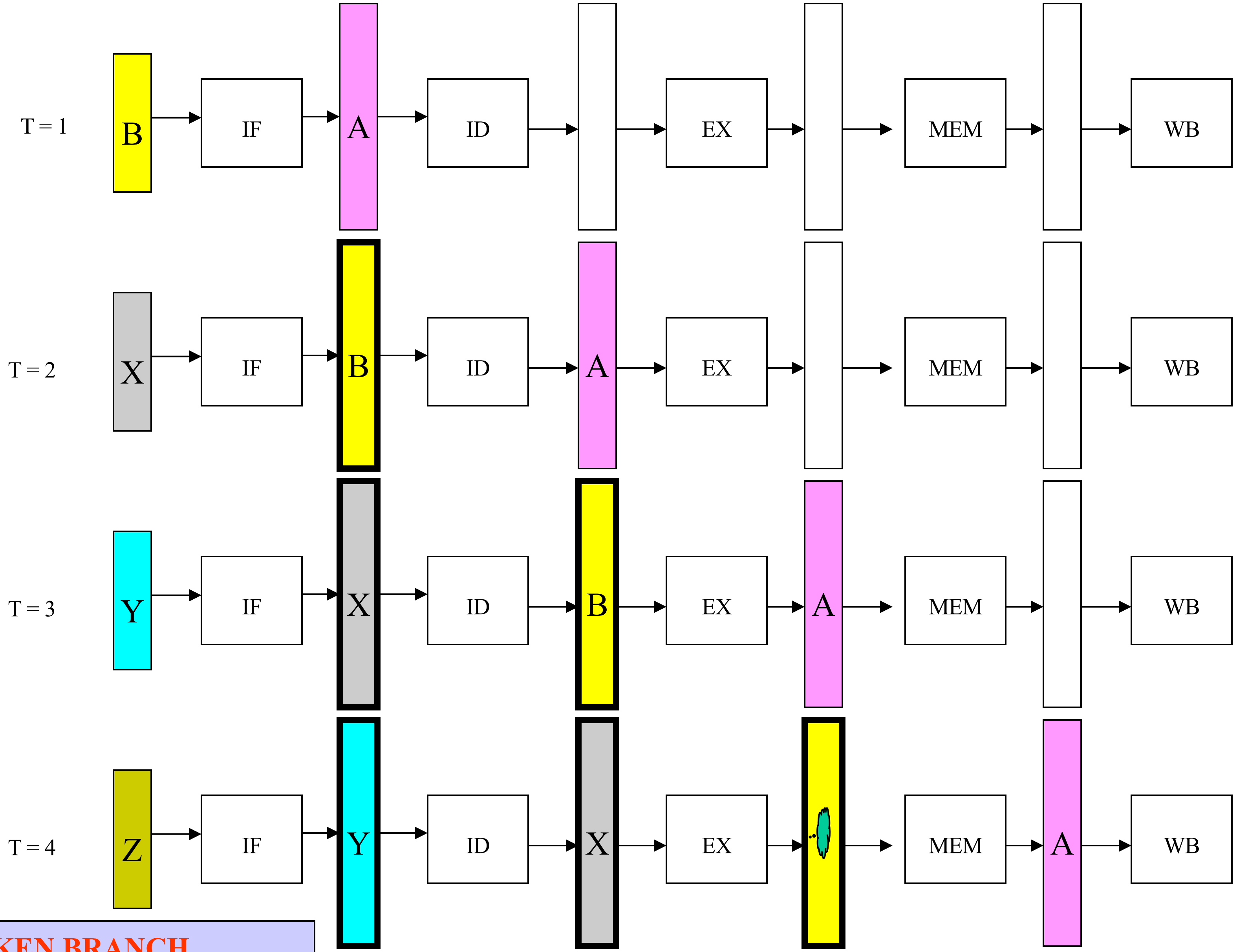
- The target address of the branch is not known (at least) till instruction is decoded
  - What is the address of instruction X?

- Compute target address when Branch is in ID stage
  - 1 cycle delay

- In-line instruction after the Branch squashed if prediction is correct

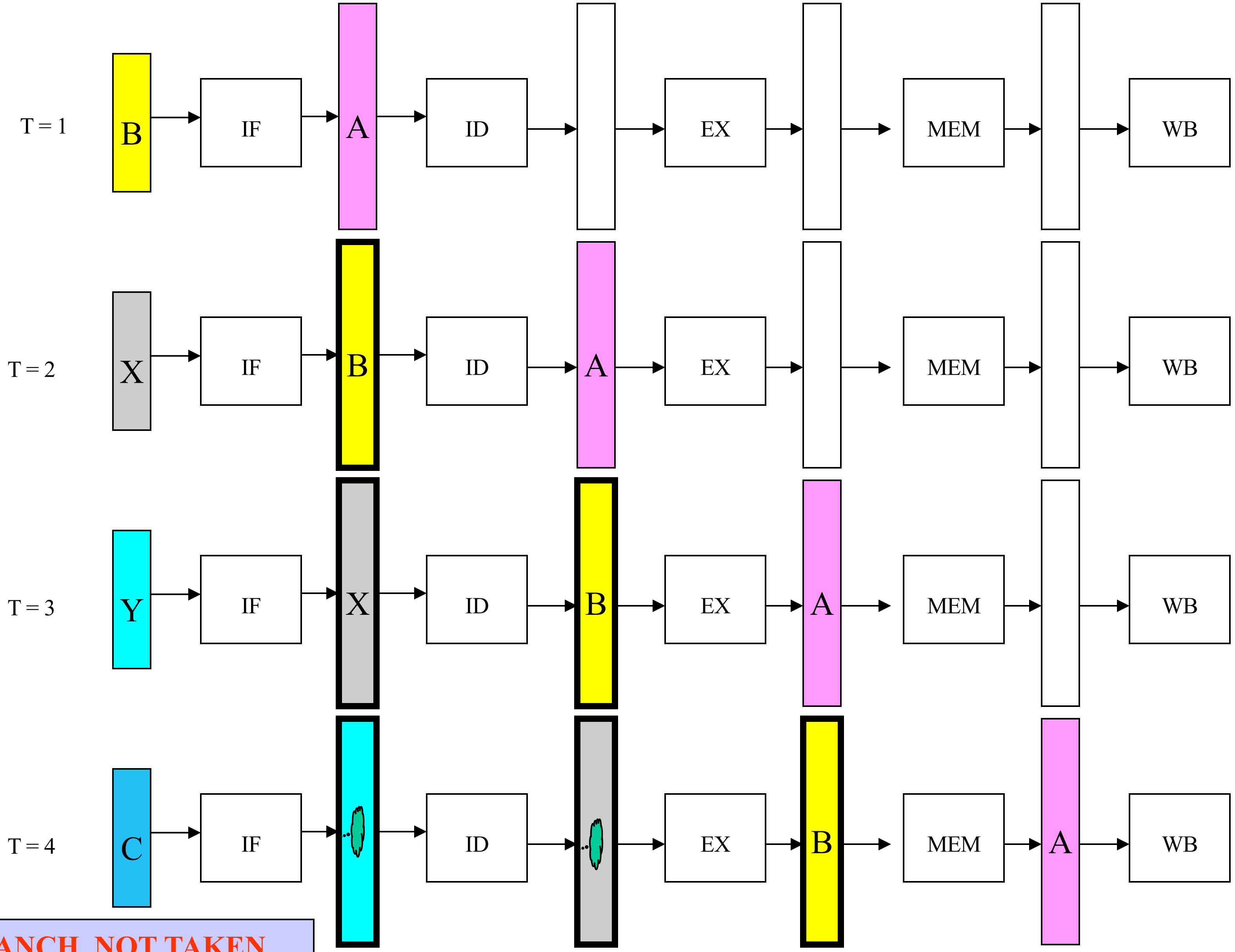
- **Special case**: Unconditional Branch / Jump instruction

Predict Branch Taken



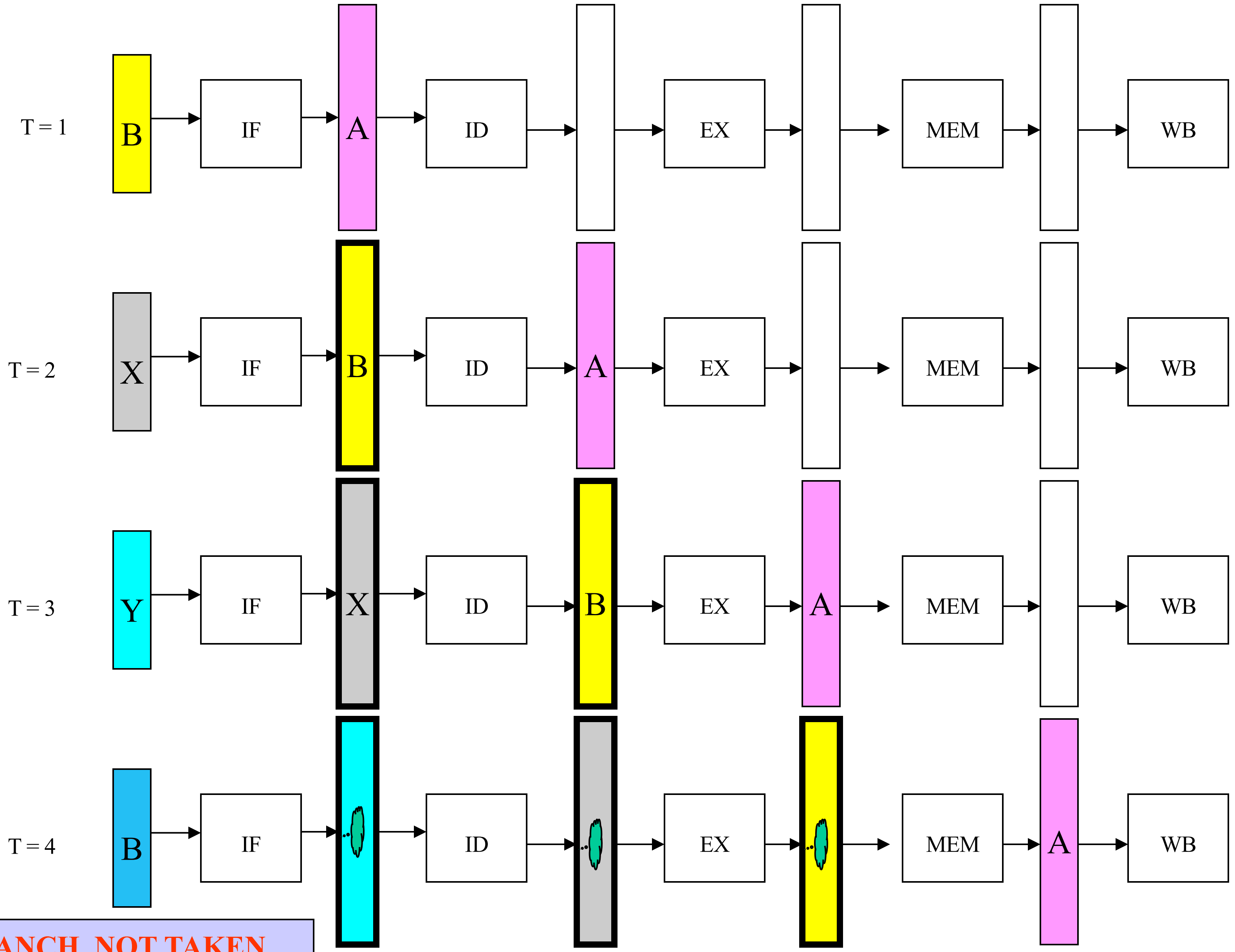
TAKEN BRANCH

Predict Branch Taken



BRANCH NOT TAKEN

Predict Branch Taken



BRANCH NOT TAKEN

Control Hazards

- Speculative Execution
  - Predict whether the branch will be taken or not taken
    - Sophisticated techniques based on past history or hints
      - Hardwired into design
      - Statically fixed at compile Time
      - Run Time
  - Fetch and execute instructions from predicted address
  - Rollback mechanism to undo wrongly speculated instructions
- Hardware Support for Predict Taken
    - Specialized cache to hold target addresses
      - Branch instruction address in PC
        - Look up target address cache
        - If hit: next cycle PC has address of target

Control Hazards

A program has 15% conditional branch instructions and 5% jump (unconditional) instructions. 70% of the conditional branches are taken. Target address are computed in the ID stage. What is the CPI assuming (a) predict branch not taken (ii) predict branch taken.

(a) Predict Branch Not Taken

Penalty (per instruction): 15% x [70% x 3 + 30% x 0] + 5% x 1 = 0.315 + 0.05 = 0.365

CPI = 1.0 + 0.365 = 1.365

(b) Predict Branch Taken

Penalty (per instruction): 15% x [70% x 1 + 30% x 3] + 5% x 1 = 0.24 + 0.05 = 0.29

CPI = 1.0 + 0.29 = 1.29

Turn over point?

$3t = t + 3(1-t) = t + 3 - 3t = 3 – 2t$

$t = 3/5$

Blank Slide

Pipeline Hazards

- **Hazard**

  Condition that disrupts the orderly flow of instructions

  Requires special attention by hardware and/or software to maintain program correctness and performance

  - Structural Hazards

    Contention for hardware resources

  - Control Hazards

    Disruptions caused by program control flow

  - **Data Hazards**

    Data dependencies between instructions

Data Dependencies

Dependent instructions: access common storage location

- I_1 and I_2 instructions that access a common storage location
- I_1 occurs before I_2 in the dynamic instruction stream
- No instructions between I_1 and I_2 access the location

- Read-after-Read (**RAR**) dependency
  - I_2 *reads* the location *read* by I_1

    I_1: ADD R1, **R2**, R3

    I_2 : ADD R4, R5, **R2**

  - Both I_1 and I_2 read register R2
  - I_1 and I_2 can execute out-of-order safely

- Read-after-Write (**RAW**) dependency
  - I_2 *reads* the location *written* by I_1
  - No instruction between I_1 and I_2 writes to the same location

    I_1 : ADD **R1**, R2, R3

    I_2 : ADD R4, **R1**, R2

  - I_2 must read the value in register R1 that was written by I_1

Data Dependencies

- Write-after-Read (**WAR**) dependency:
  - I_2 *writes* the location *read* by I_1

    I_1 : ADD R1, **R2**, R3
    I_2 : ADD **R2**, R4, R5

  - I_1 must read the old value of R2 prior to the write by I_2

- Write-after-Write (**WAW**) dependency:

  - I_2 *writes* the location *written* by I_1

    I_1 : ADD **R1**, R2, R3
    I_2 : ADD **R1**, R4, R5

- Write of R1 by I_1 must not occur after the write by I_2
- Final value of R1 due to write by I_2 and not I_1

When is a dependency a hazard?

During execution can there be errors due to concurrent execution of dependent instructions?

Can we have WAW Hazards in the DLX Pipeline?

A : ADD **R1**, R2, R3

B : ADD **R1**, R4, R5

	1	2	3	4	5	6
A	IF	ID	EX	MEM	WB	
B		IF	ID	EX	MEM	WB

Register Writes occur at an only one stage in the pipeline

When is a dependency a hazard?

Can we have WAR Hazards in our DLX pipeline?

WAR Hazard

A : ADD R1, **R2**, R3

B : ADD **R2**, R4, R5

Can the Write to R2 take place before the Read of R2?

	1	2	3	4	5	6
A	IF	**ID**	EX	MEM	WB	
B		IF	ID	EX	MEM	**WB**

Register Reads occur at an earlier stage in the pipeline than Writes

When is a dependency a hazard?

RAW Hazard

A :	ADD	**R1**, R2, R3
B :	ADD	R4, **R1**, R5

- **Hazard possible** since register **reads** occur **earlier** in the pipeline than **writes**
    - **A** writes R1 at cycle 5
    - **B** may read R1 as early as cycle 3

Instruction B reads stale value in R1 before it is updated by A (HAZARD!!)

	1	2	3	4	5	6
A	IF	ID	EX	MEM	**WB**	
B		IF	**ID**	EX	MEM	WB

RAW Hazards

	1	2	3	4	5	6	7
A	IF	ID	EX	MEM	**WB**		
X		IF	ID	EX	MEM	WB	
B			IF	**ID**	EX	MEM	WB

- A and B separated by 1 instruction: Hazard

	1	2	3	4	5	6	7	8
A	IF	ID	EX	MEM	**WB**			
X		IF	ID	EX	MEM	WB		
Y			IF	ID	EX	MEM	WB	
B				IF	**ID**	EX	MEM	WB

- A, B separated by 2 instructions
- Hazard depends on register file design
- No Hazard with split read/write protocol with writes earlier in cycle

Data Hazards in 5-stage In-Order Pipeline

- No WAR or WAW Hazards

- RAW Hazards
  1. Simple solutions to avoid hazard by stalling pipeline
     - Introduce stall cycles (delays) to avoid hazard
       - Delay second instruction till register write is complete

     a) Software introduced stall

     b) Hardware controlled stall

  2. Reducing performance penalty in avoiding RAW hazards

     a) Software

     b) Hardware

Compiler Inserted NOPs

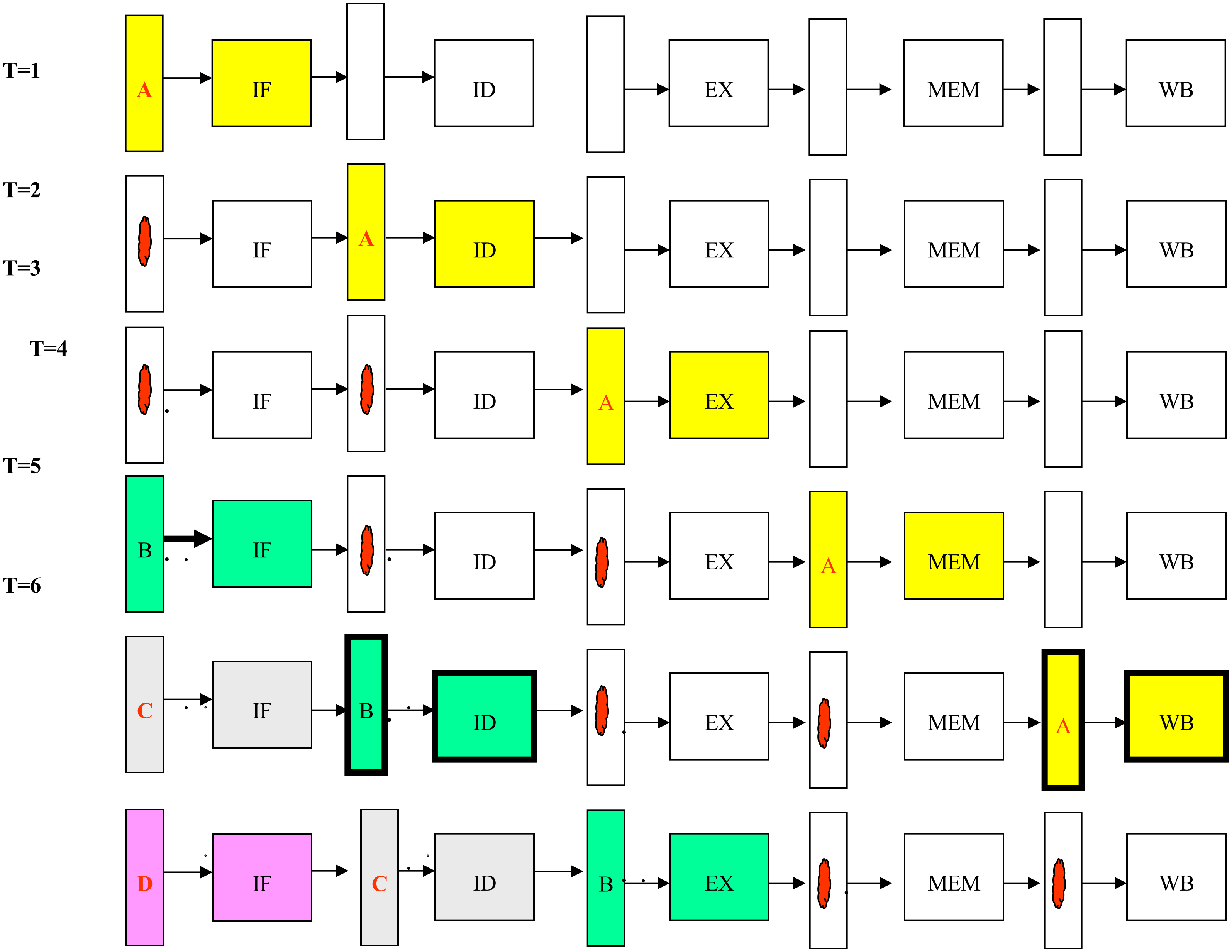
	1	2	3	4	5	6	7	8
A	IF	ID	EX	MEM	**WB**			
NOP		☁	☁	☁	☁	☁		
NOP			☁	☁	☁	☁	☁	
B				IF	**ID**	EX	MEM	WB

Consecutive instructions A, B forced apart by 2 NOPs to avoid RAW hazard

A : ADD **R1**, R2, R3
NOP
NOP
B : ADD R4, **R1**, R5.

- NOP add 1 cycle to the execution time

Compiler Inserted NOPS



Compiler Inserted NOPs

Example

Suppose 30% of ALU instructions are followed by a dependent ALU instruction, and 10% are separated from a dependent instruction by one independent instruction. Assume 90% of instructions are ALU instructions

NOPS add 1 cycle to the execution time

Worst-case CPI = 1.0 + 90% x [30% x 2 + 10% x 1] = 1.63

Hardware-Controlled Pipeline Stall

A : ADD **R1**, R2, R3

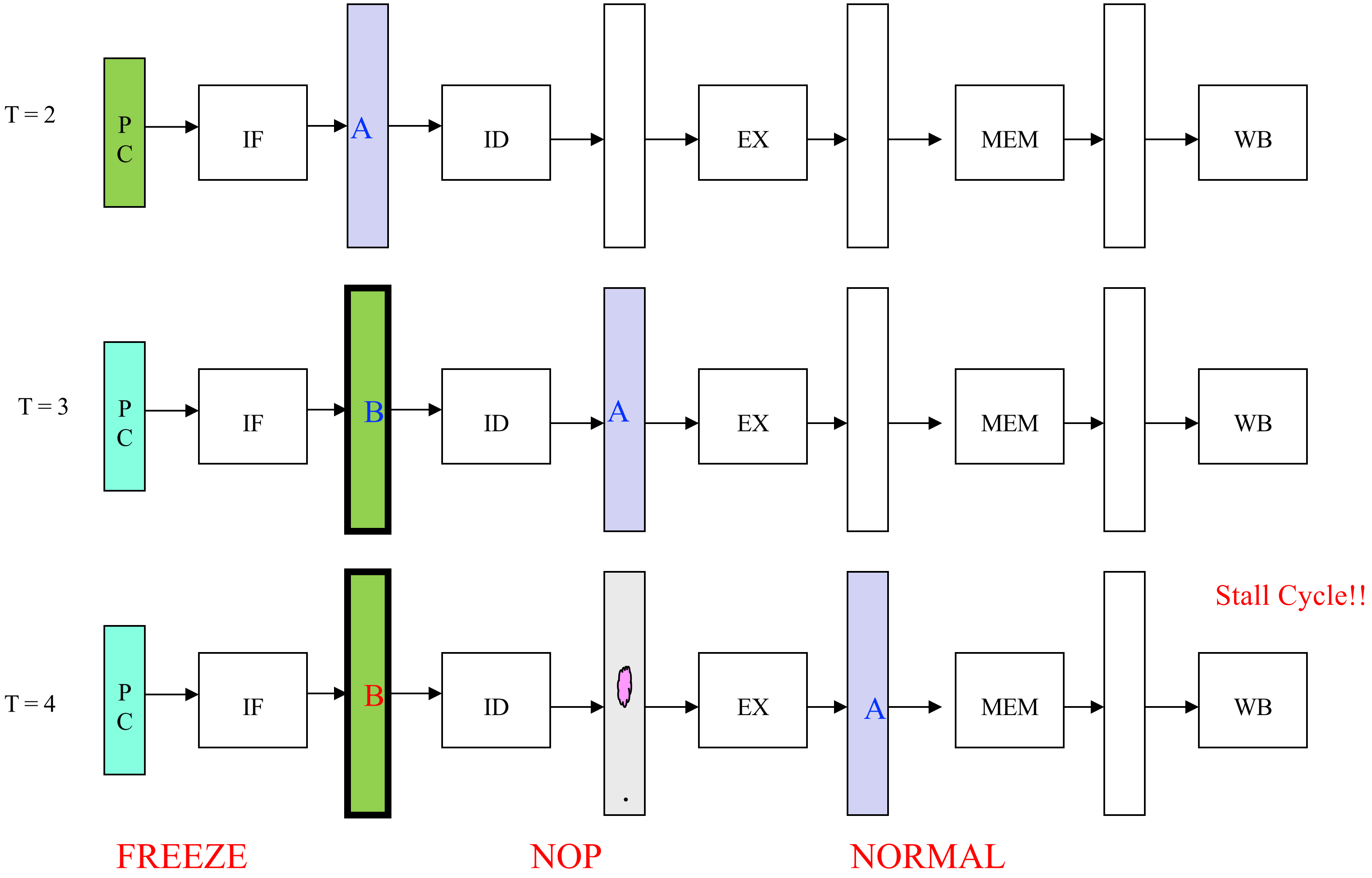
B : ADD R4, **R1**, R2

- Hazard Detection unit detects hazardous RAW dependency
- Stalls the necessary stages to avoid the hazard
- Delays B by 2 cycles

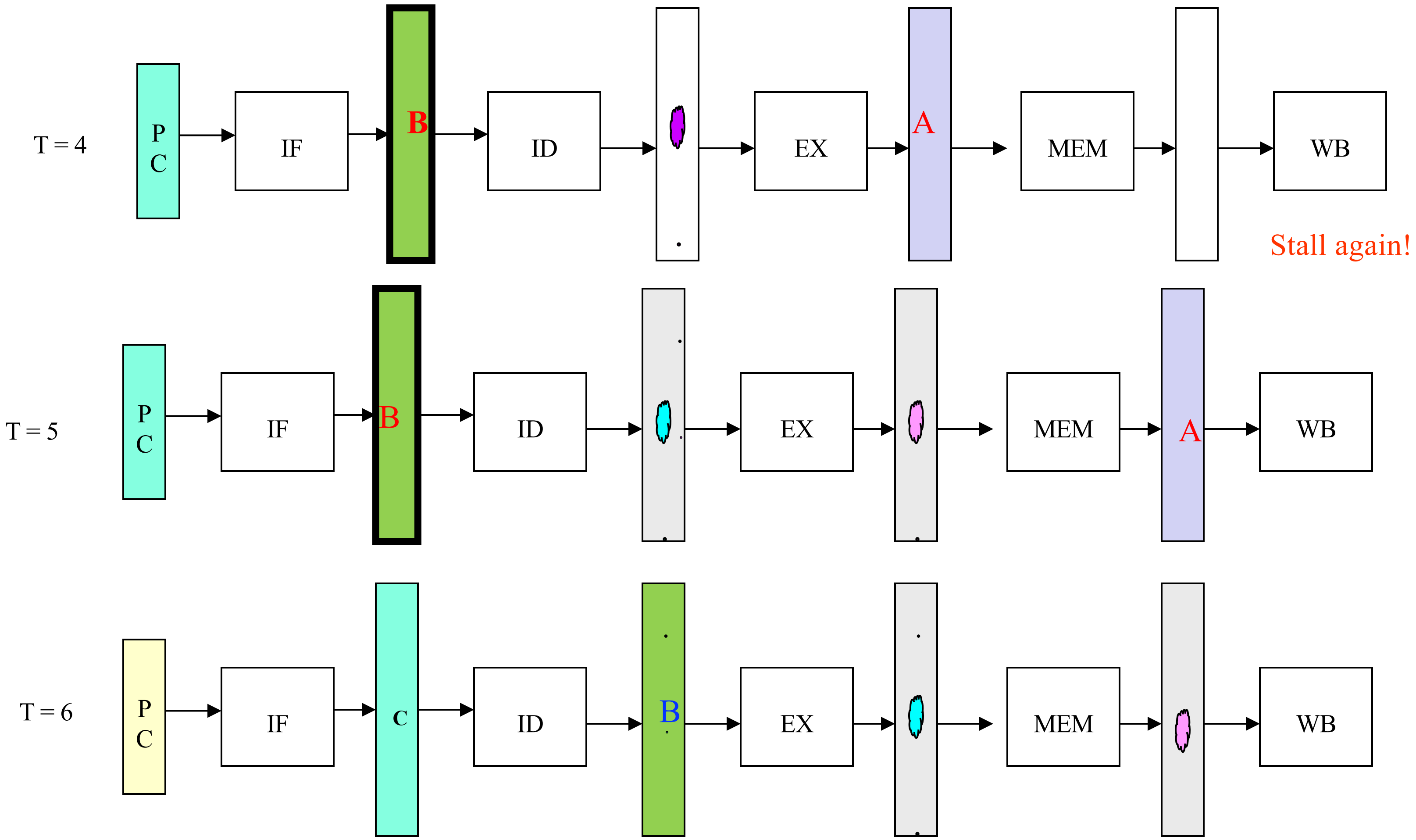
	1	2	3	4	5	6	7	8	9
A	IF	ID	EX	MEM	**WB**				
B		IF	ID	ID	**ID**	EX	MEM	WB	
C			IF	IF	IF	ID	EX	MEM	WB
D						IF	ID	EX	MEM

Hardware Controlled Pipeline Stall

🟪	**A** :	ADD	**R1**, R2, R3
🟩	**B** :	ADD	R4, **R1**, R2
🟦	C:		

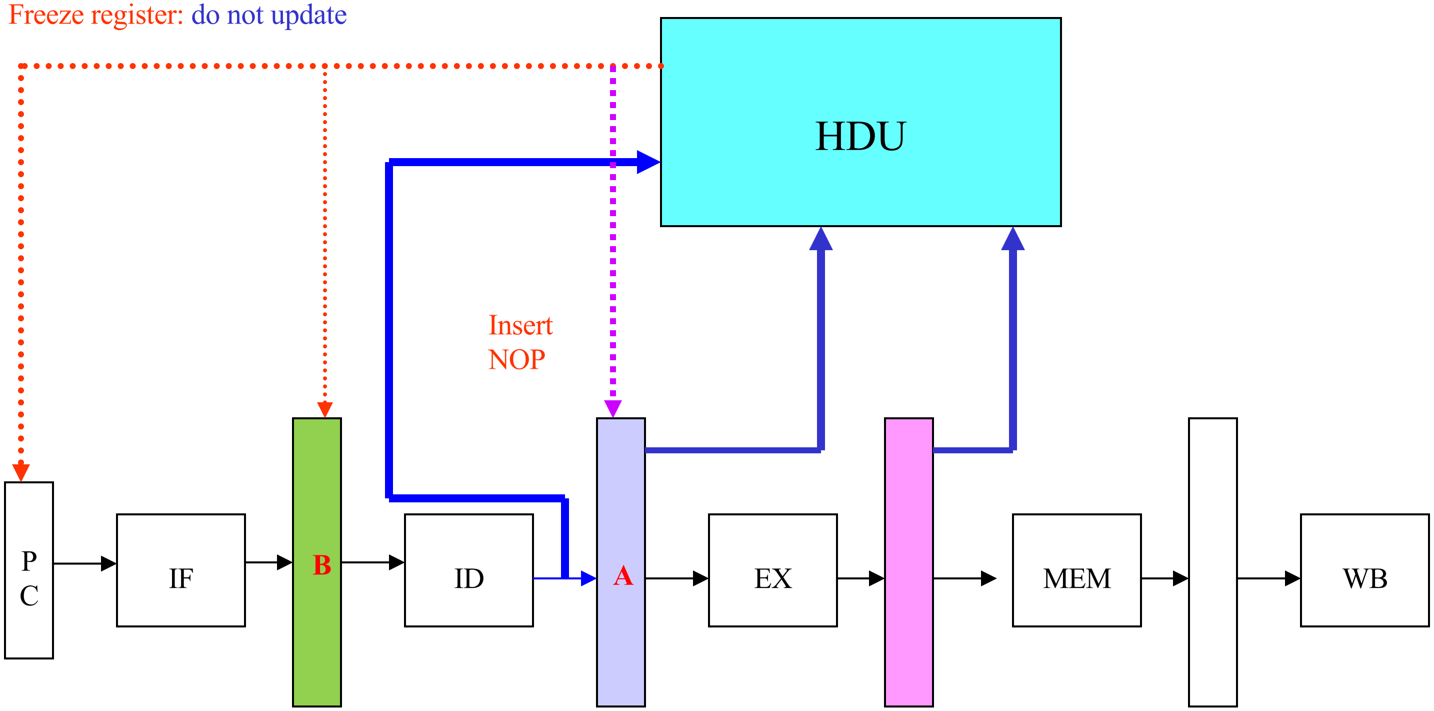


Hardware Controlled Pipeline Stall



- Instruction B held in IF/ID register until A reaches WB stage
- Internally generated NOPs propagated forward while B is stalled

Hazard Detection Unit



Stall Pipeline if instruction in **IF/ID register** reads register W and the instruction in either the **ID/EX register** or the **EX/MEM register** will write register W

W is in either the rt (RI) or rd (RR) field of the writing instruction and the rs or rt field of the reading instruction

Operation of Hazard Detection Unit

Compare Register numbers of the

READ REGISTER of instruction in IF/ID Pipeline Register

with the

WRITE REGISTER of the instruction in the ID/EX Pipeline Register and

WRITE REGISTER of the the instruction in the EX/MEM Pipeline Register

If any of the comparisons succeed: Insert Stall Cycle

- FREEZE PC and IF/ID Pipeline Register
- Insert NOP into ID/EX Pipeline Register

	Write Register	Read Register
R-R	rd	rs, rt
R-I	rt	rs
LD	rt	rs
SD	--	rs, rt
Bcc	--	rs
Bcc	--	rs, rt

RAW Hazards: Correctness + Performance

1. Reduce or eliminate stall cycles

    - Software
        - Compiler Optimizations
            - Restructure code to fill delay slots with independent instructions

    - Hardware
        - Forwarding (Register bypass)
            - Provide alternate data paths within the pipeline to communicate values
            - Instruction gets value directly from source instruction bypassing the register

    - Combination
        - Load Delay Slot

2. Overlap stall cycles with other useful operations

Performance Issues

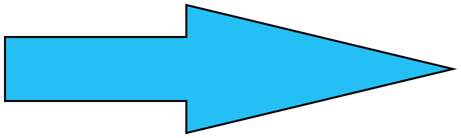
NOPS and stalls uselessly consume cycles (and energy) and reduce throughput

- **Software**
  - Reorganize assembly code
  - Move an independent instruction in the delay slot (where the NOP was inserted)

```
ADD    R1, R2, R3
SUB    R4, R1, R5
XOR    R3, R2, R7
AND    R8, R7, R7
```

Original Code

Compiler Optimization



```
ADD       R1, R2, R3
XOR       R3, R2, R7
AND       R8, R7, R7
SUB       R4, R1, R5
```

Optimized Code

- Challenge: Find enough independent instructions within a basic block