CPU Pipelines and Superscalar Organization

Chester Rebeiro
Indian Institute of Technology Madras

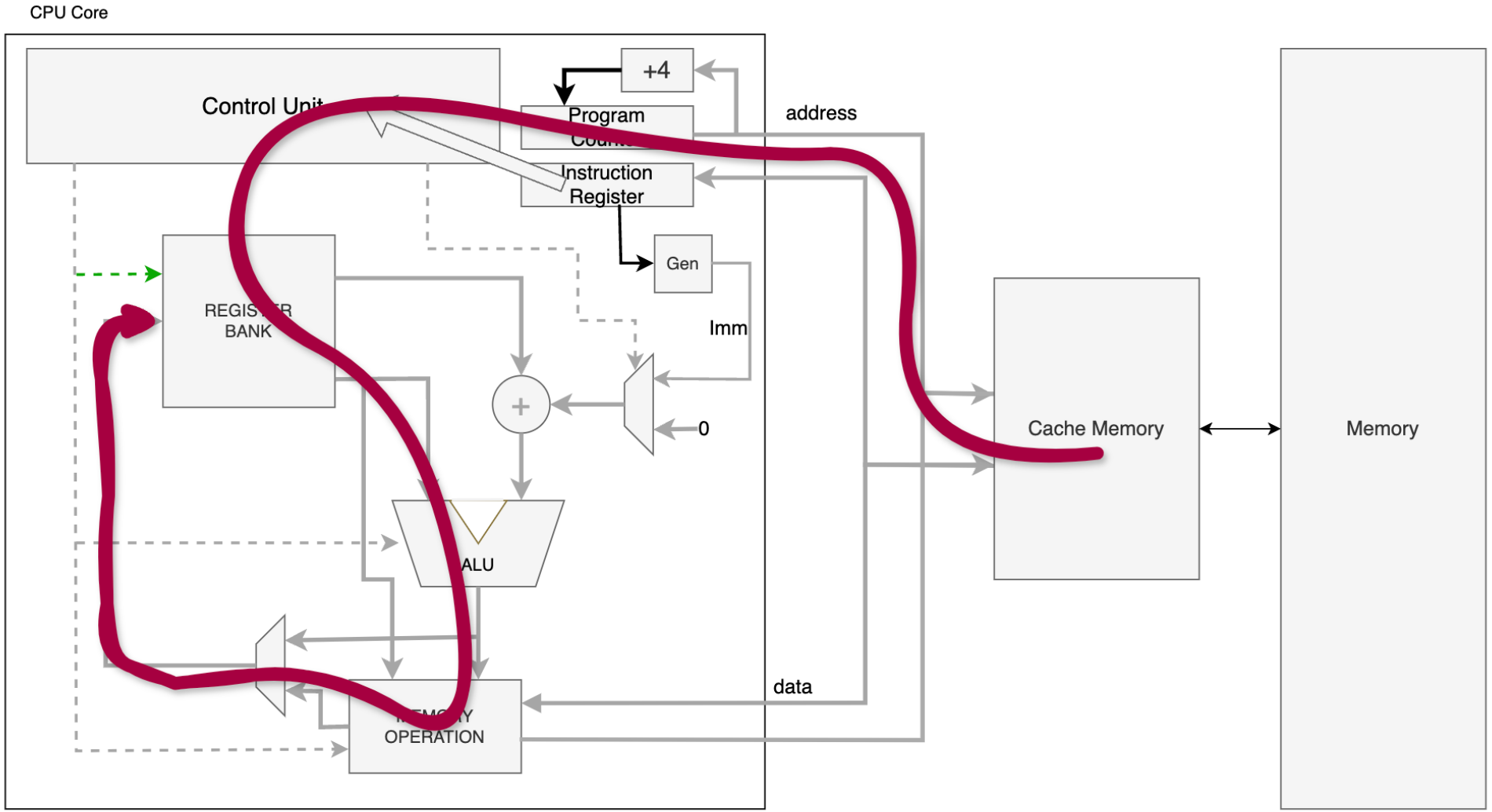
chester@cse.iitm.ac.in

CPU Pipelines

Chester Rebeiro
Indian Institute of Technology Madras

chester@cse.iitm.ac.in

Instruction Execution in a Single Cycle Processor Like a Pipeline

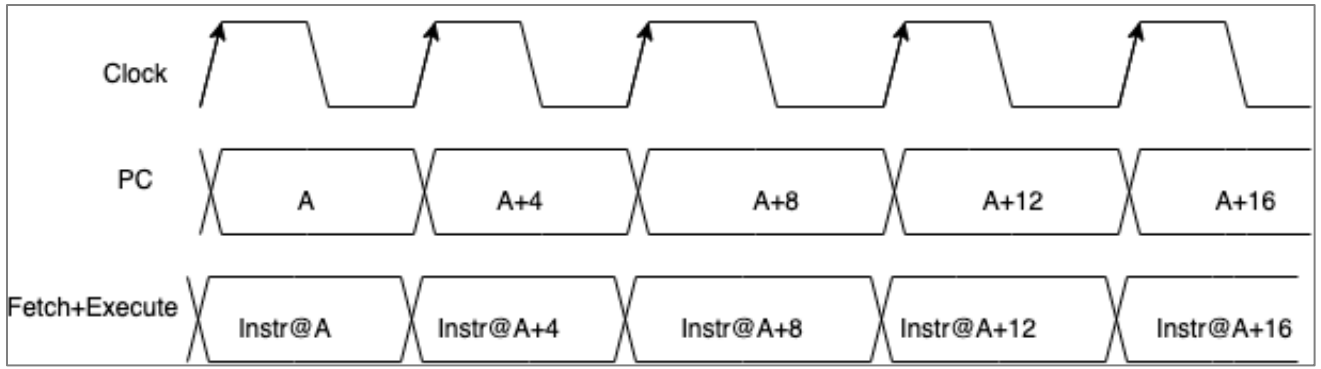


Instruction execution flows through the processor like a pipeline

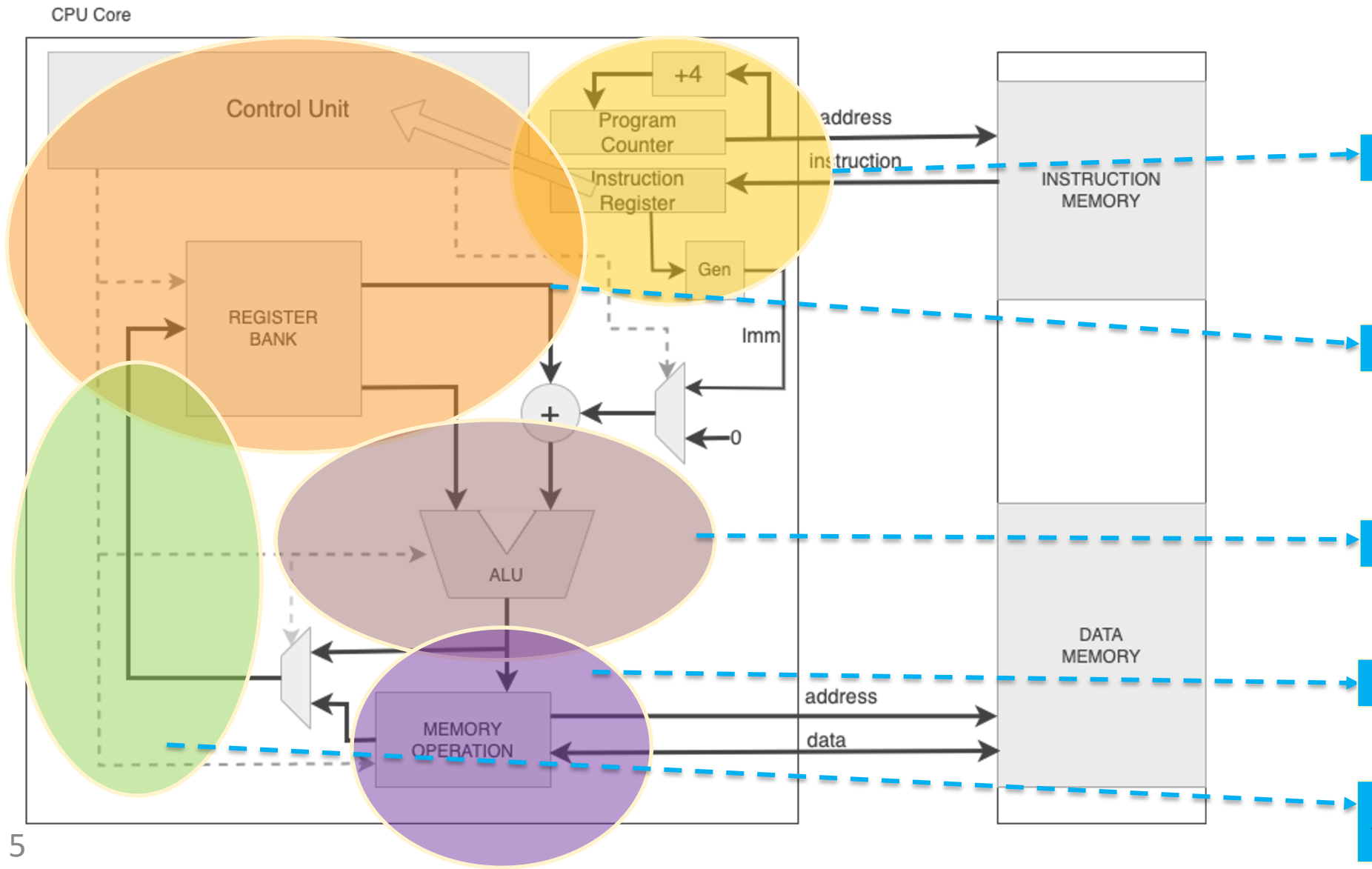
Single Cycle Processor in Practice

Hardly ever used!!!

- Even though instructions execute in 1 clock cycle
- Frequency of clock is very low → frequency set based on the slowest instruction to execute (typically memory operations)

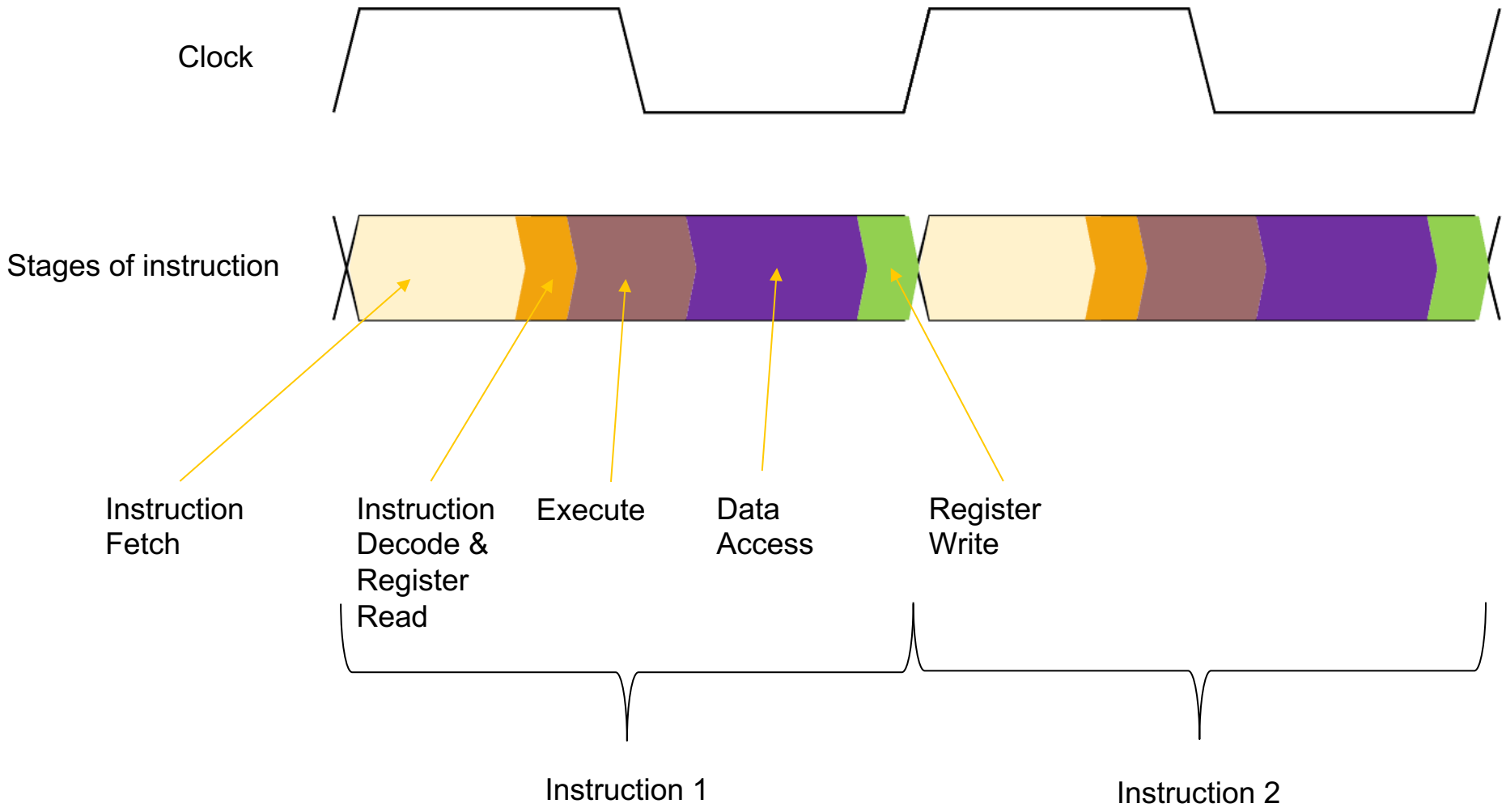


Stages in an instruction's execution: CPU View

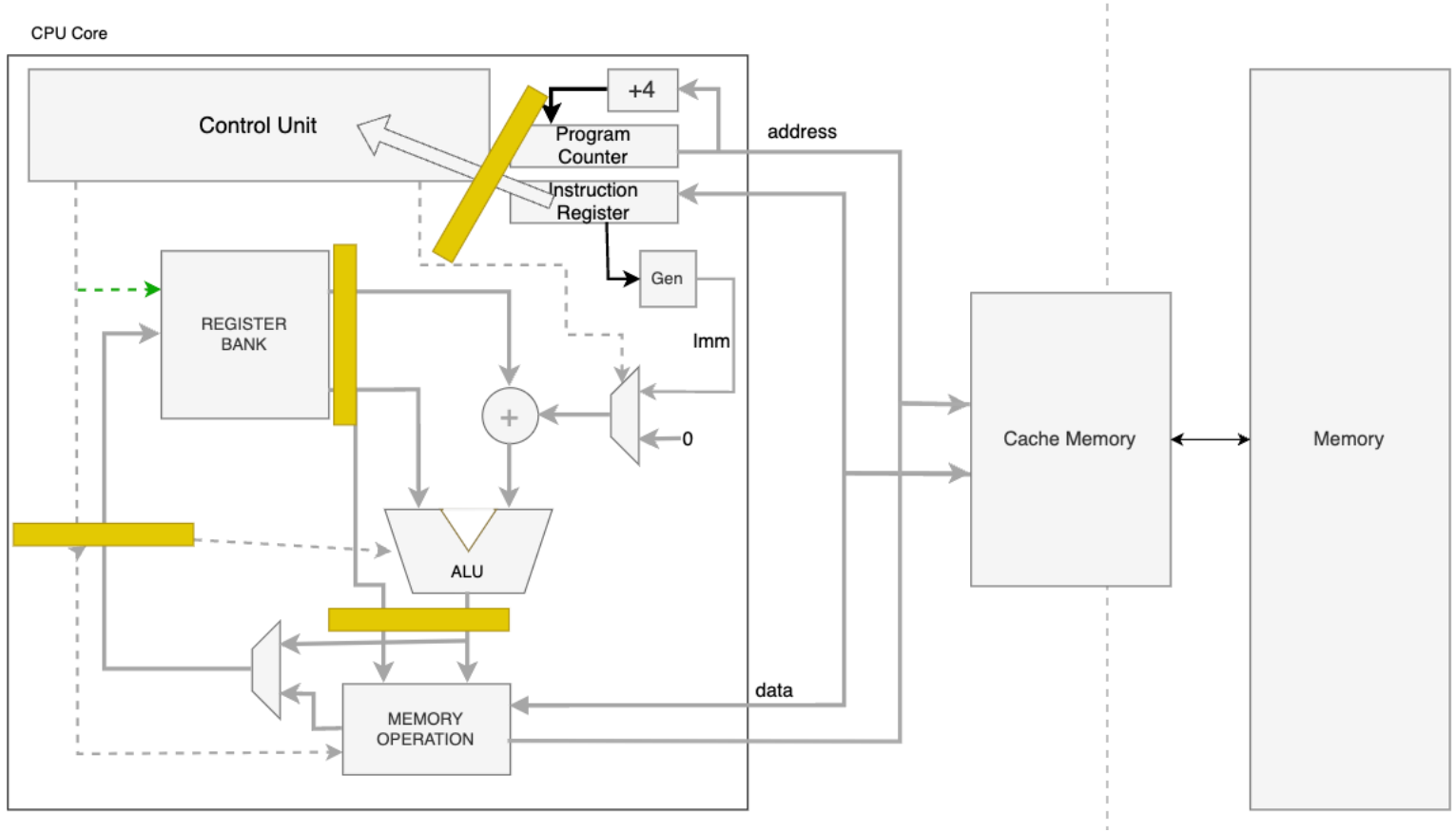


CPU pipeline has multiple stages.

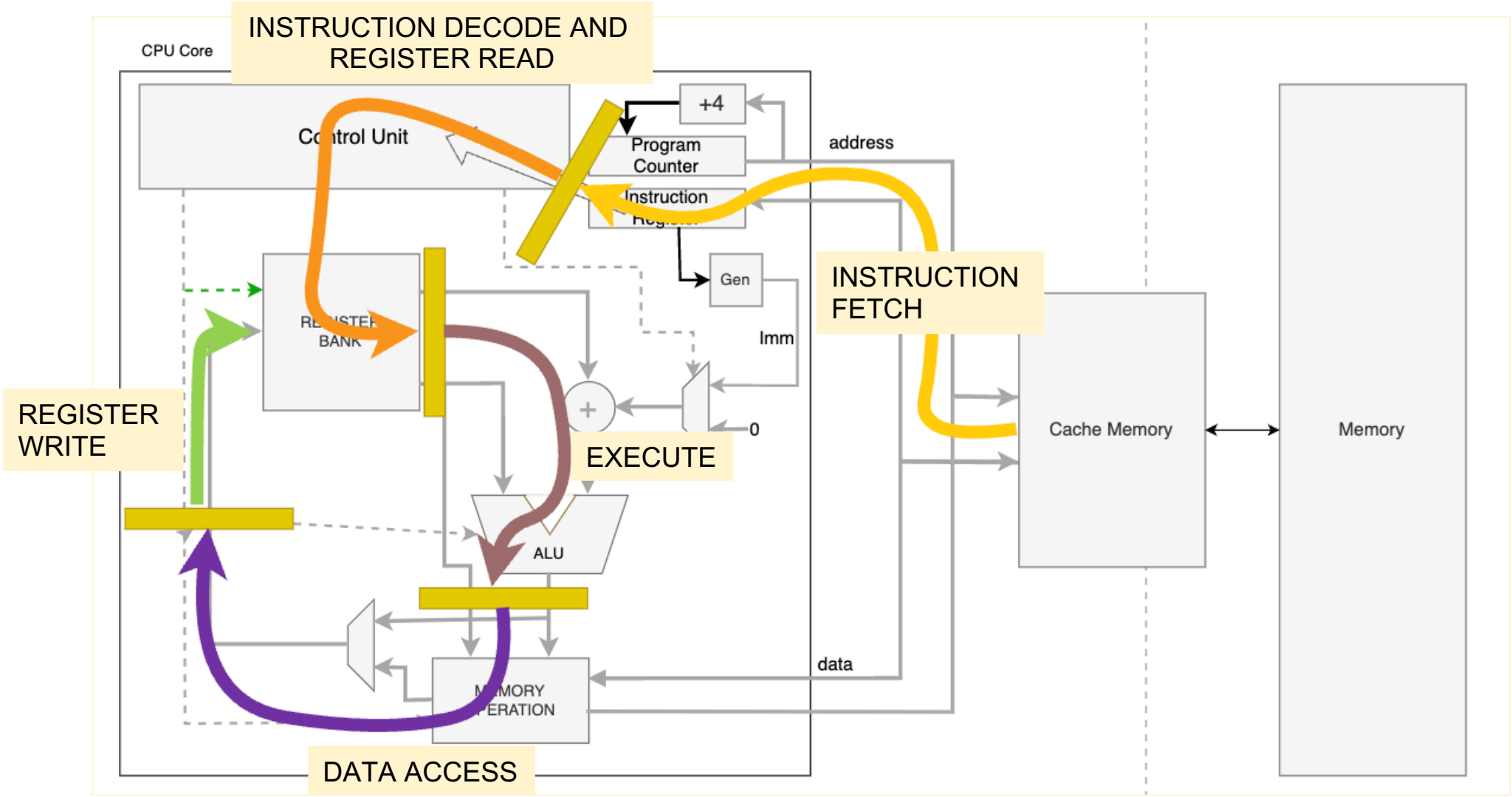
Pipeline Stages in an instruction's execution Timing View



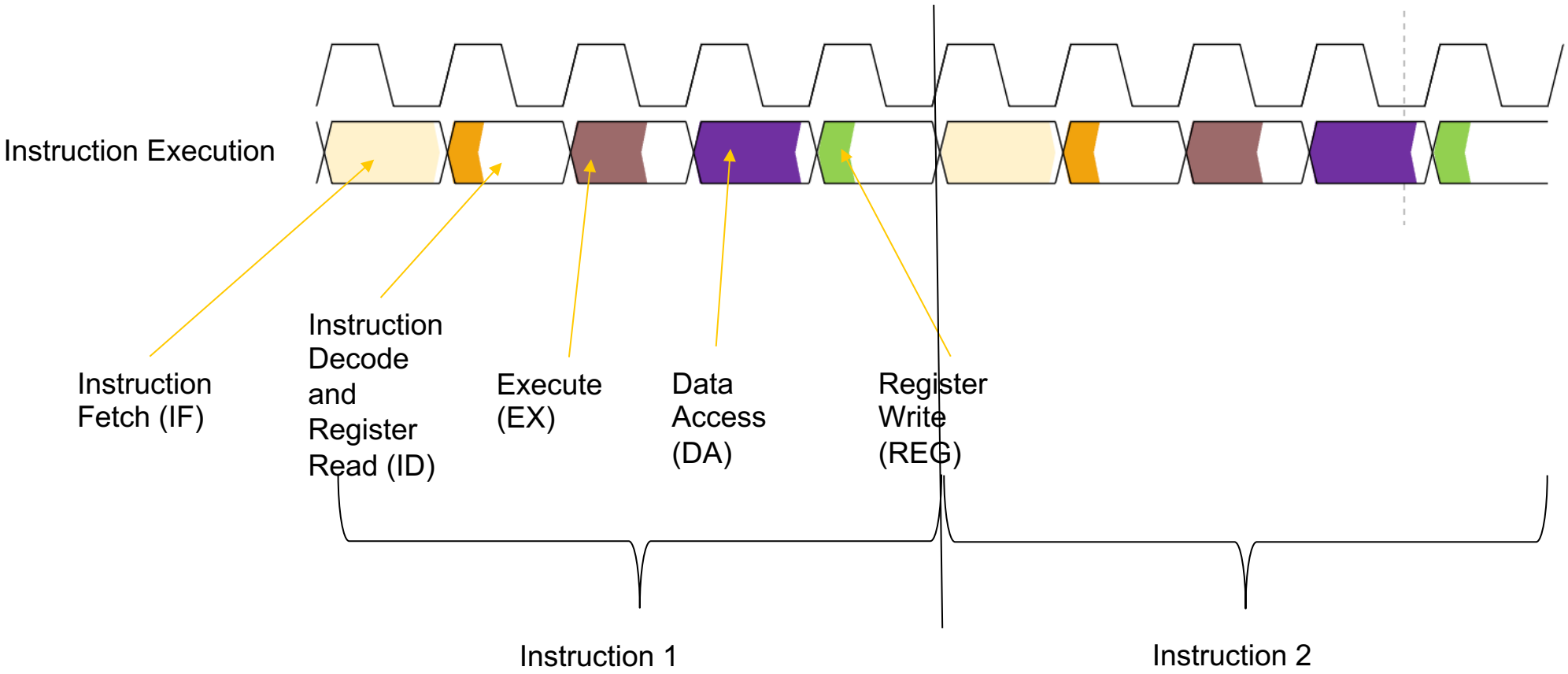
Introducing Pipeline Buffers in the CPU



Introducing Pipeline Buffers in the CPU

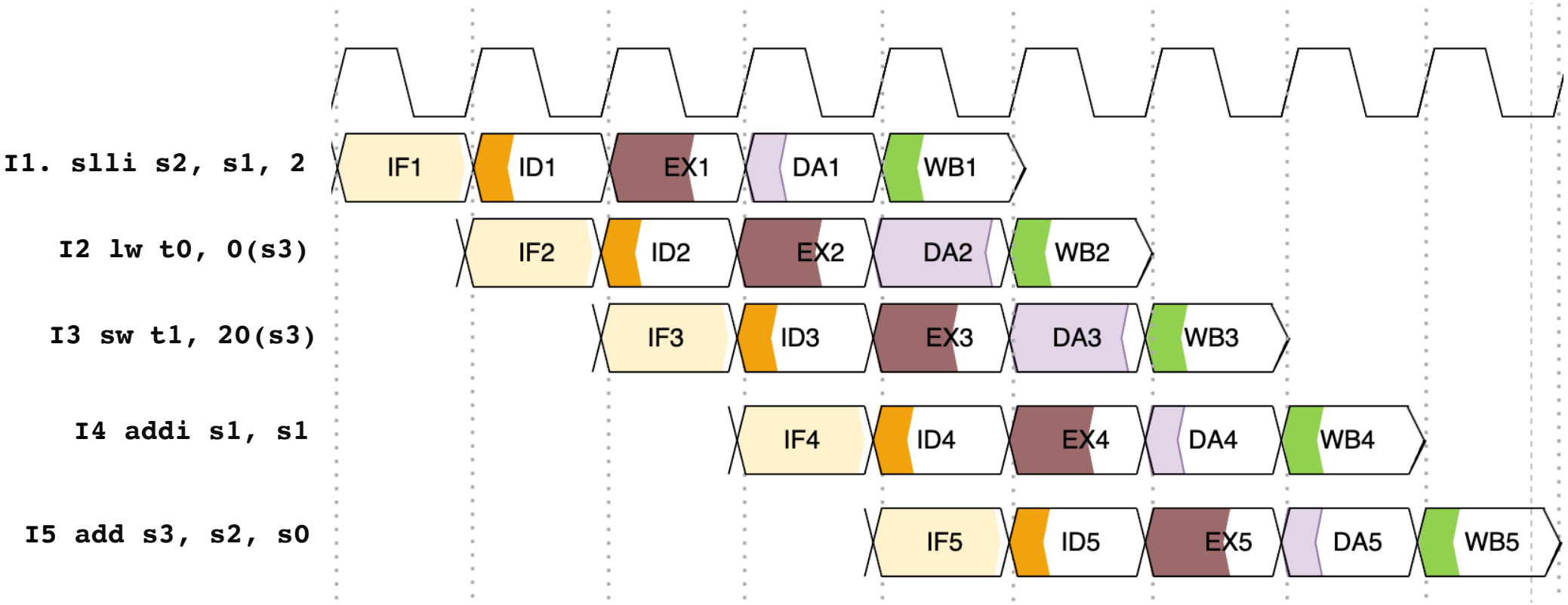


Instruction Execution with Pipeline Buffers



Choose the clock frequency based on the slowest stage. It now takes 5 clock cycles to complete an execution

Multiple Instructions in the Pipeline



Benefits of Pipelining (Ideal case)

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{non pipelined}}}{\text{Number of pipestages}}$$

- Original frequency = F
- Pipelined frequency = 5F
- In ideal state (after the first 5 instructions),
  - 1 instruction completes per clock cycle (1 CPI →Cycle per Instruction)
  - 5x increase in clock frequency
  - 5x increase in throughput (#instructions / time)

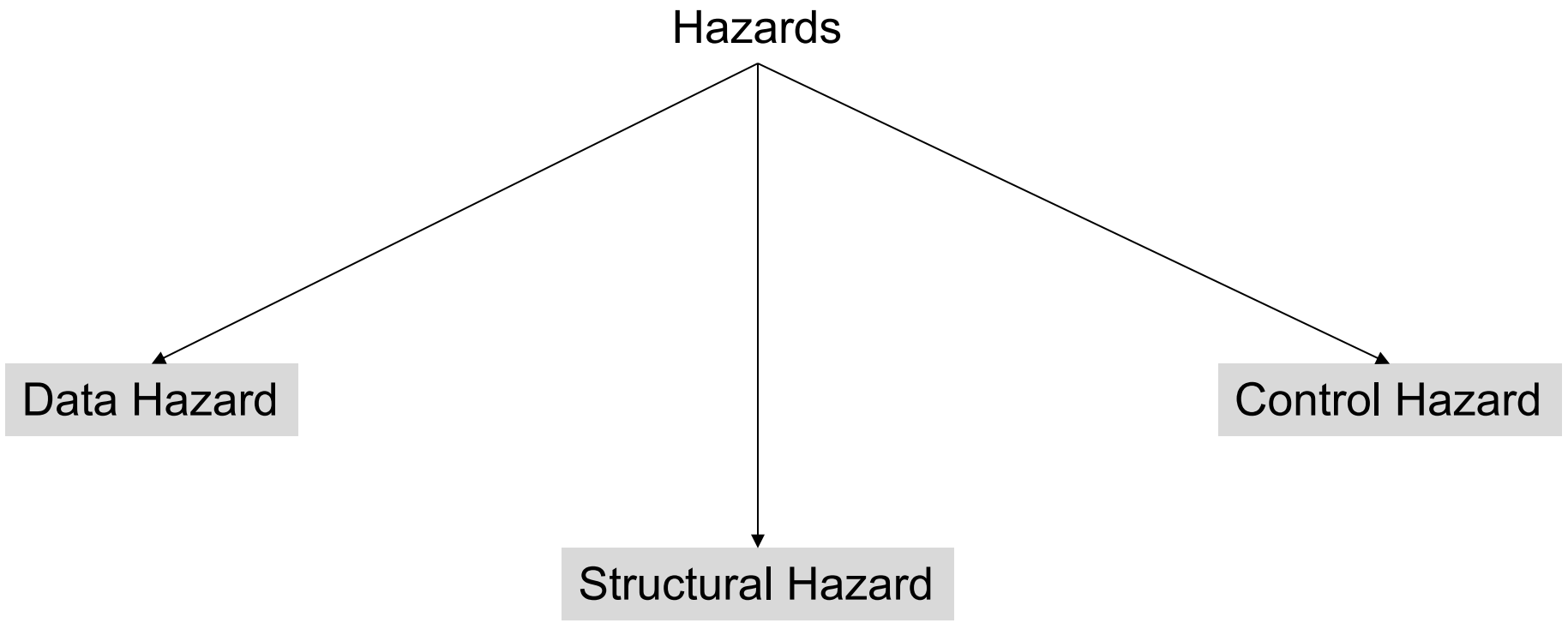
In practice, such speedups are not always possible, because of

(a) Stages are not equally balanced. Thus, the clock frequency is determined by the slowest stage.
(b) hazards

Time between instructions = 1/F

Pipeline Hazards

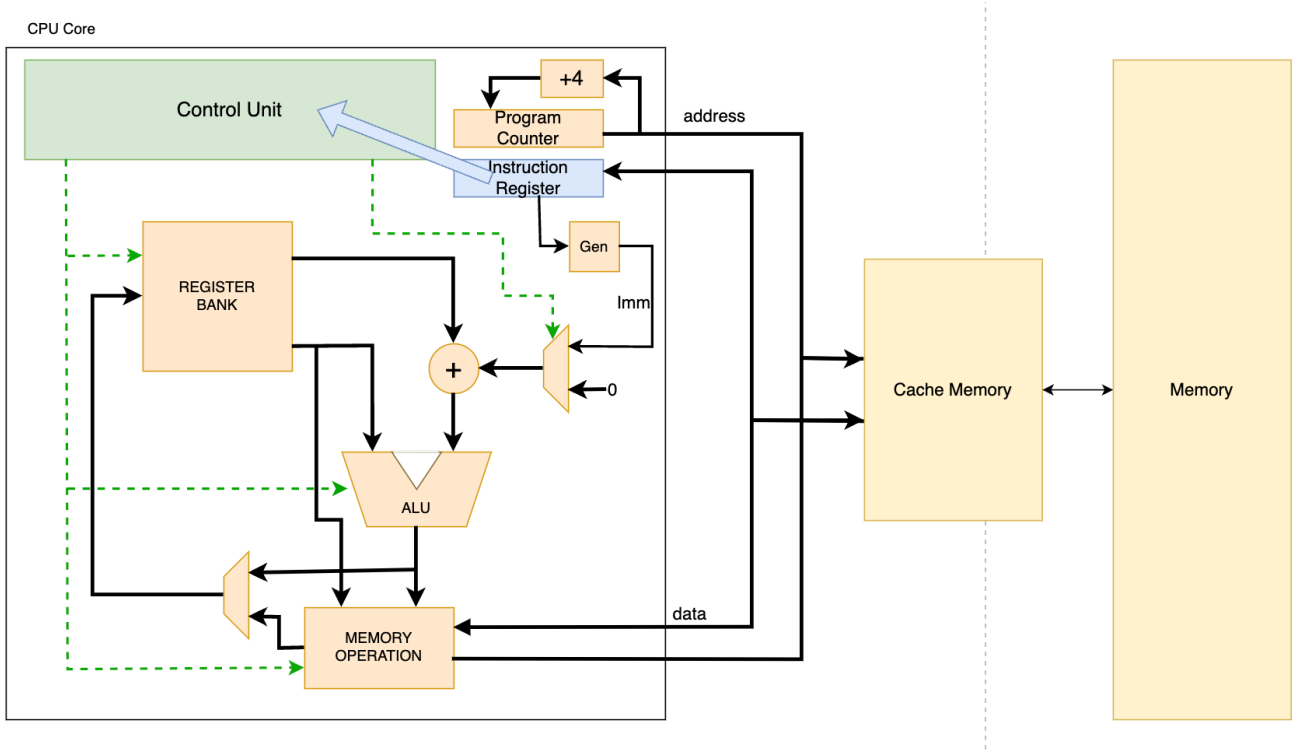
A potential obstruction in the CPUs pipeline that inhibits the next instruction executing in its allocated clock cycle



Structural Hazards

Occurs when the required hardware unit is unavailable for execution at the given clock cycle.

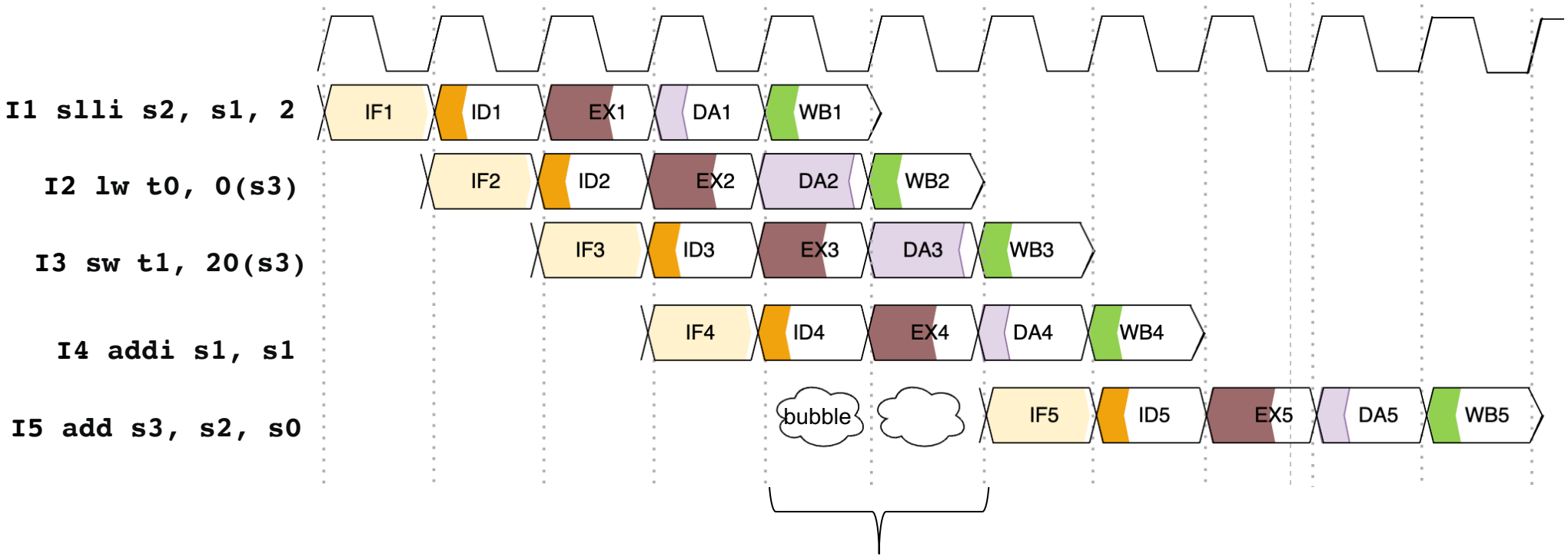
Example. Unified L1 Cache memory



Structural Hazards

Occurs when the required hardware unit is unavailable for execution at the given clock cycle.

Example. Unified L1 Cache memory



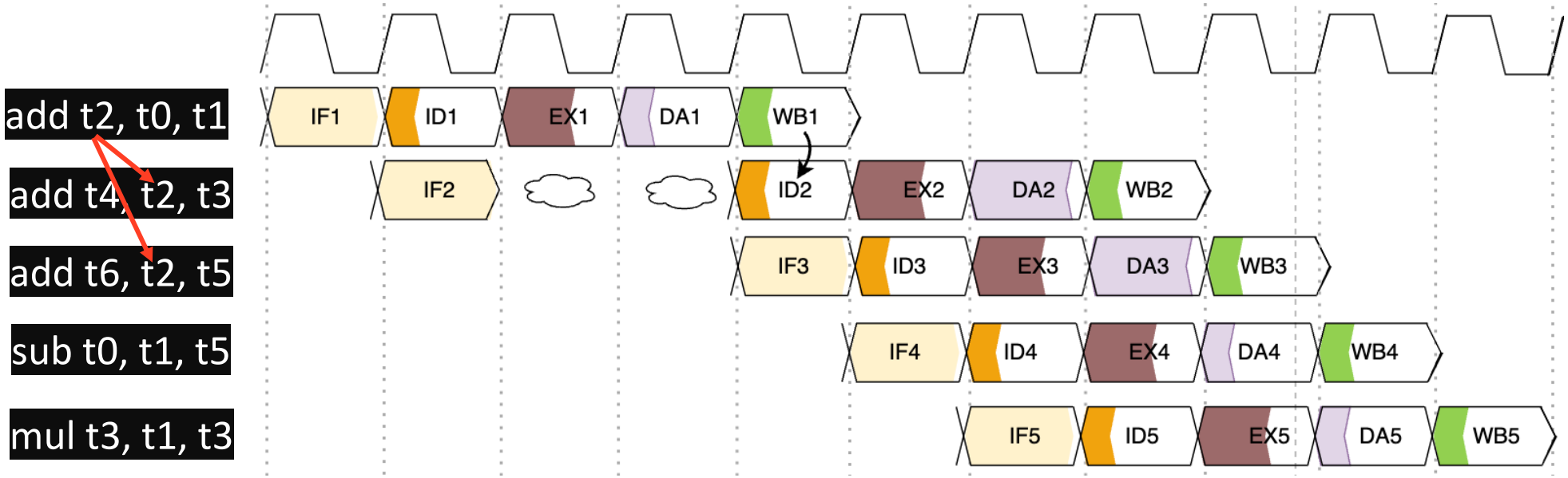
Cache memory used for memory operation request due to the load/store instructions in I2 and I3, therefore I5 is stalled

With a split L1 cache, these stalls don't occur

Data Hazard

Pipeline stalls due to dependency between instructions that are executed in different stages of the pipeline.

Read after Write: Instruction reads a register that is written to by a previous instruction.



Other Data Hazards

Write after Read (WAR): Pipeline writes to a register before a prior instruction reads it.

```
I1    ADD R1, R2, R3

I2    ADD R3, R4, R5
```

I2 updates register R3 before I1 reads it

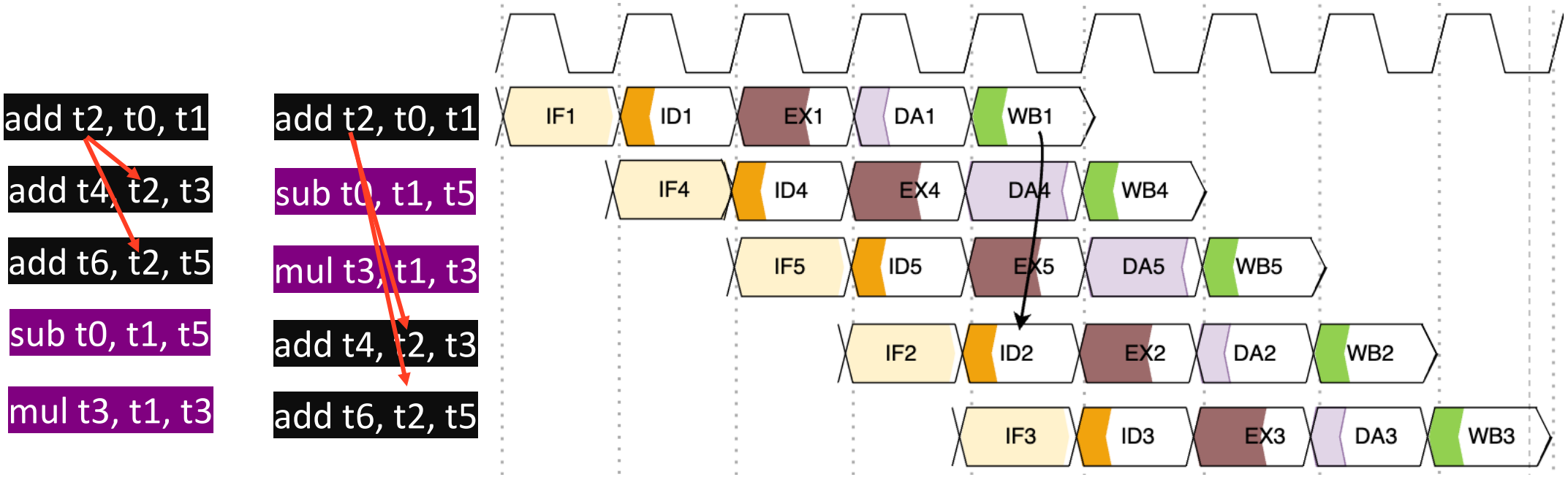
Write after Read (WAW): Pipeline writes to a register before a prior instruction writes to it.

```
I1    ADD R1, R2, R3

I2    ADD R1, R4, R5
```

I2 updates register R1 before I1 updates it

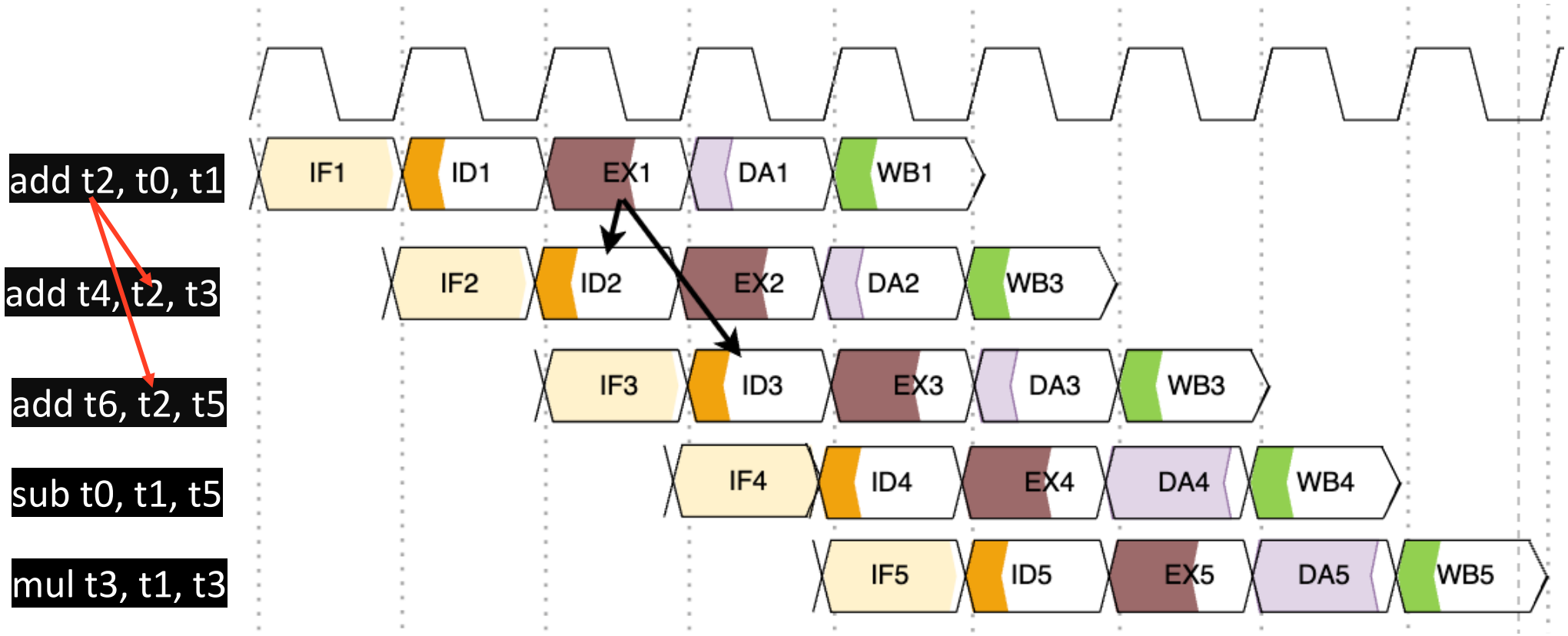
WAR and WAW only possible in superscalar processors. Not in in-order processors

Reducing the impact of Data Hazards
(1) Reordering Instructions

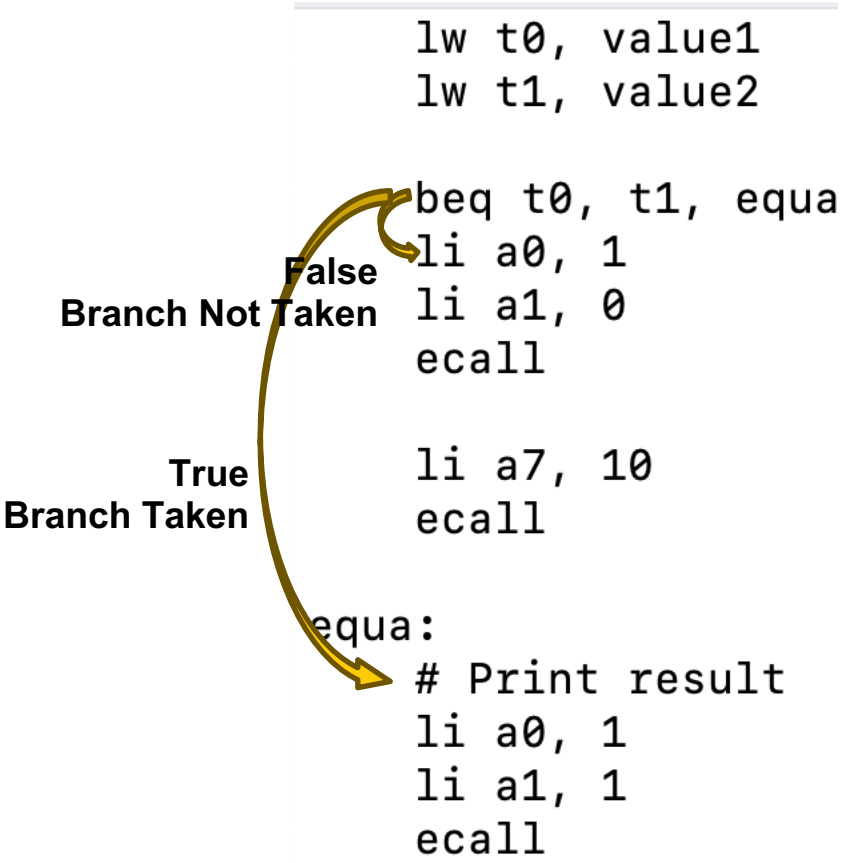


Done by the compiler

Reducing the impact of Data Hazards
(2) Operand Forwarding



Control Hazard



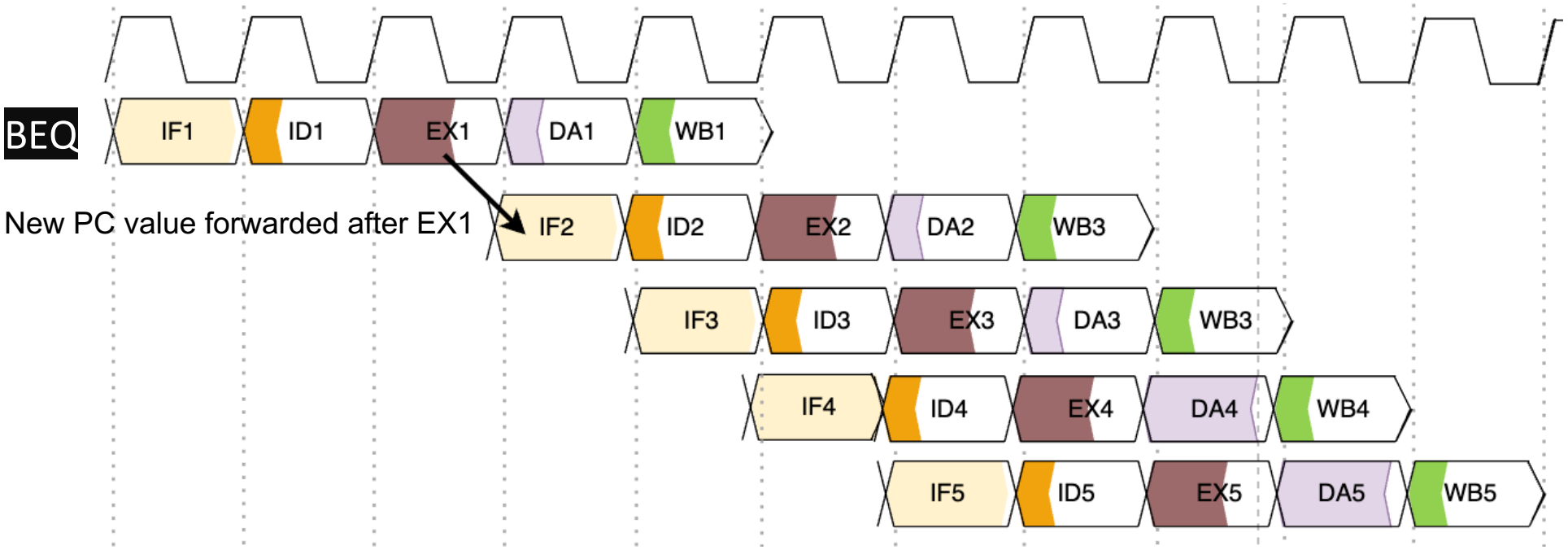
If the instruction is a conditional branch

(eg, BEQ, BNE),

- Fetching the next instruction depends on the branch outcome
- Branch outcomes only known at execution time

Control Hazard: Simple Solutions

Simple solution. Stall until the branch outcome is available



Branches can be very expensive due to the stalls

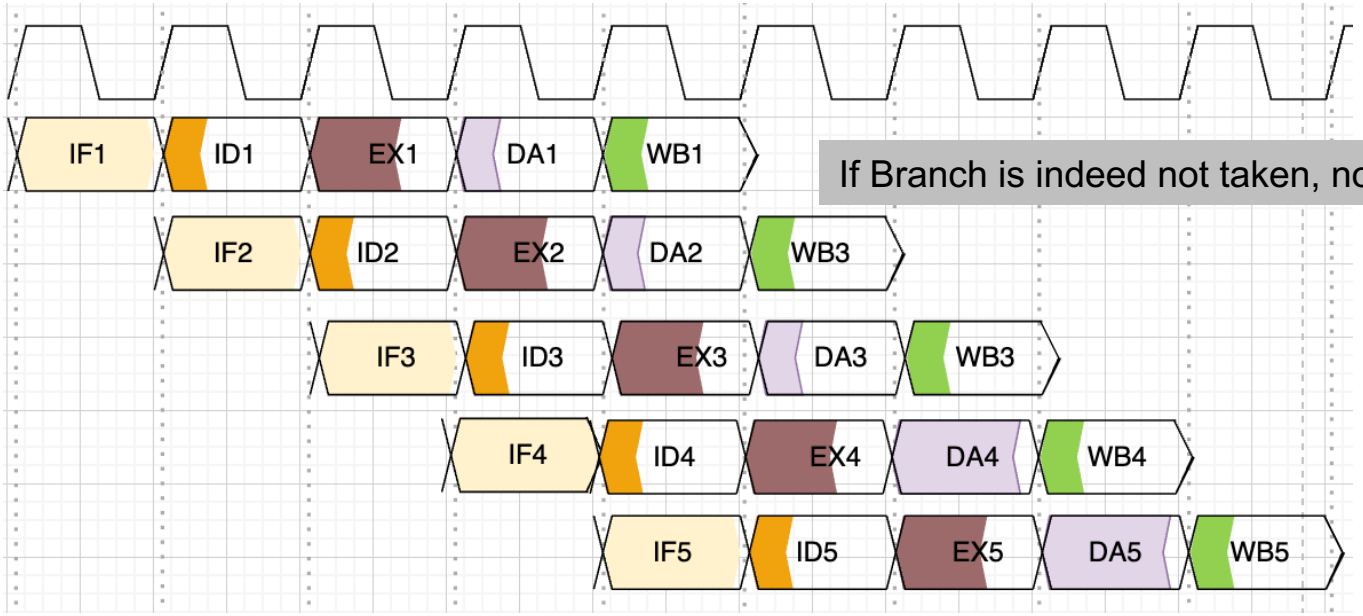
Control Hazard:
Move the Branch condition check earlier

Move the branch condition check to the decode stage, then fewer instructions need to be flushed

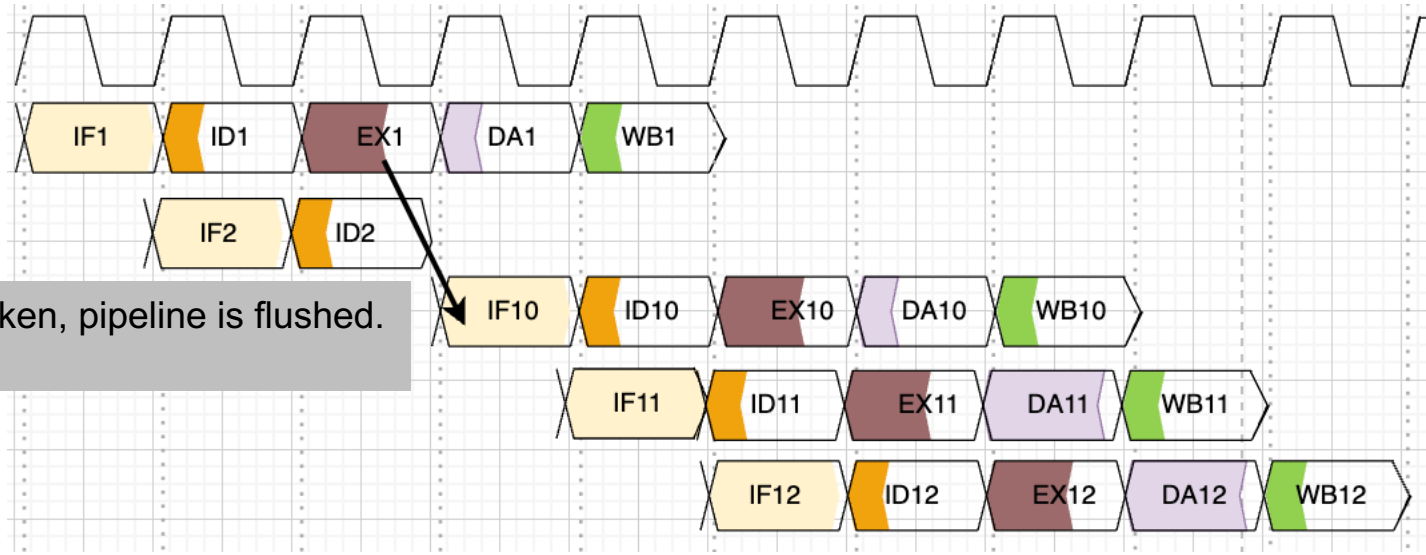
This requires… moving the branch adder to decode stage. Performing the condition check in decode (for example using XOR for equality)

However, this increases chances of hazards. Complicates the hazard detection logic.

Control Hazard: Assume Branch Not Taken



If Branch is indeed not taken, no stalls

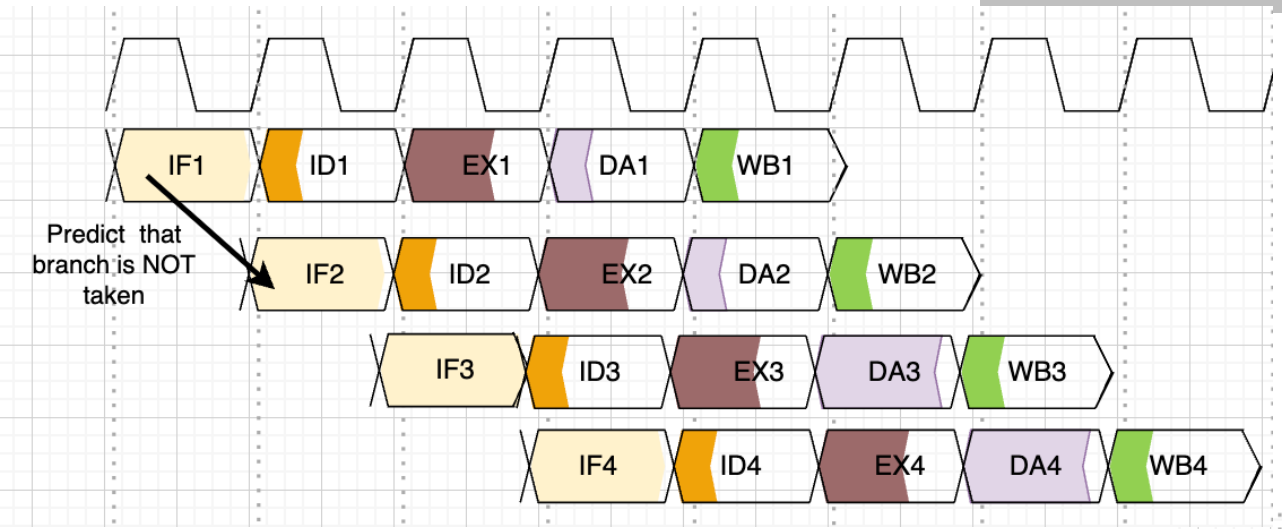


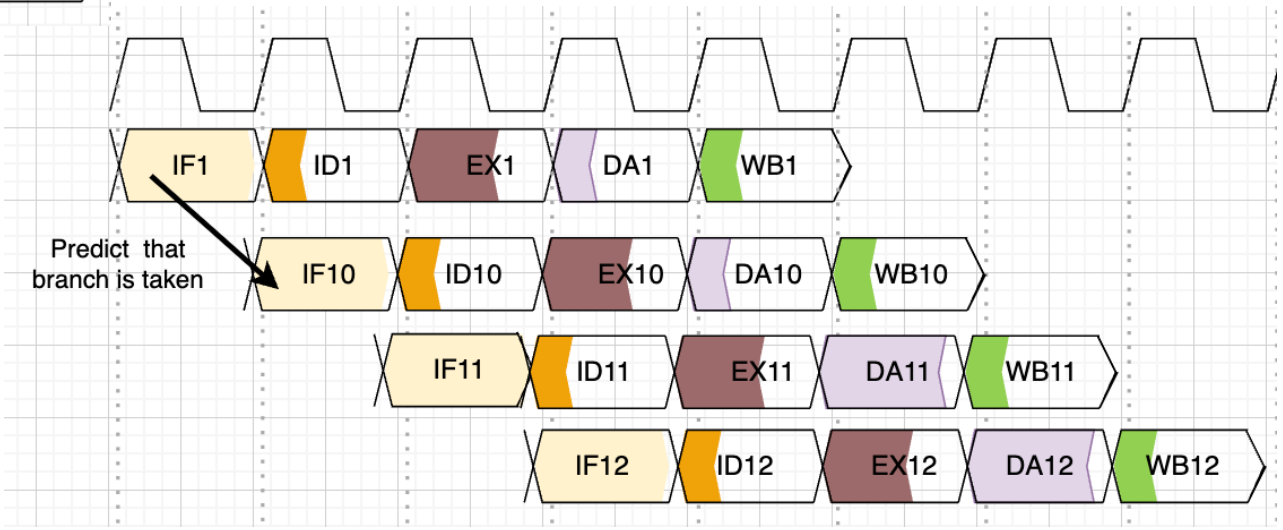
If Branch is taken, pipeline is flushed.
Two stalls

Stalls exist only when branch is taken.
Can we reduce these stalls further?

Dynamic Branch Prediction

Predict if a branch is taken or not!
Ideally, there are no stalls

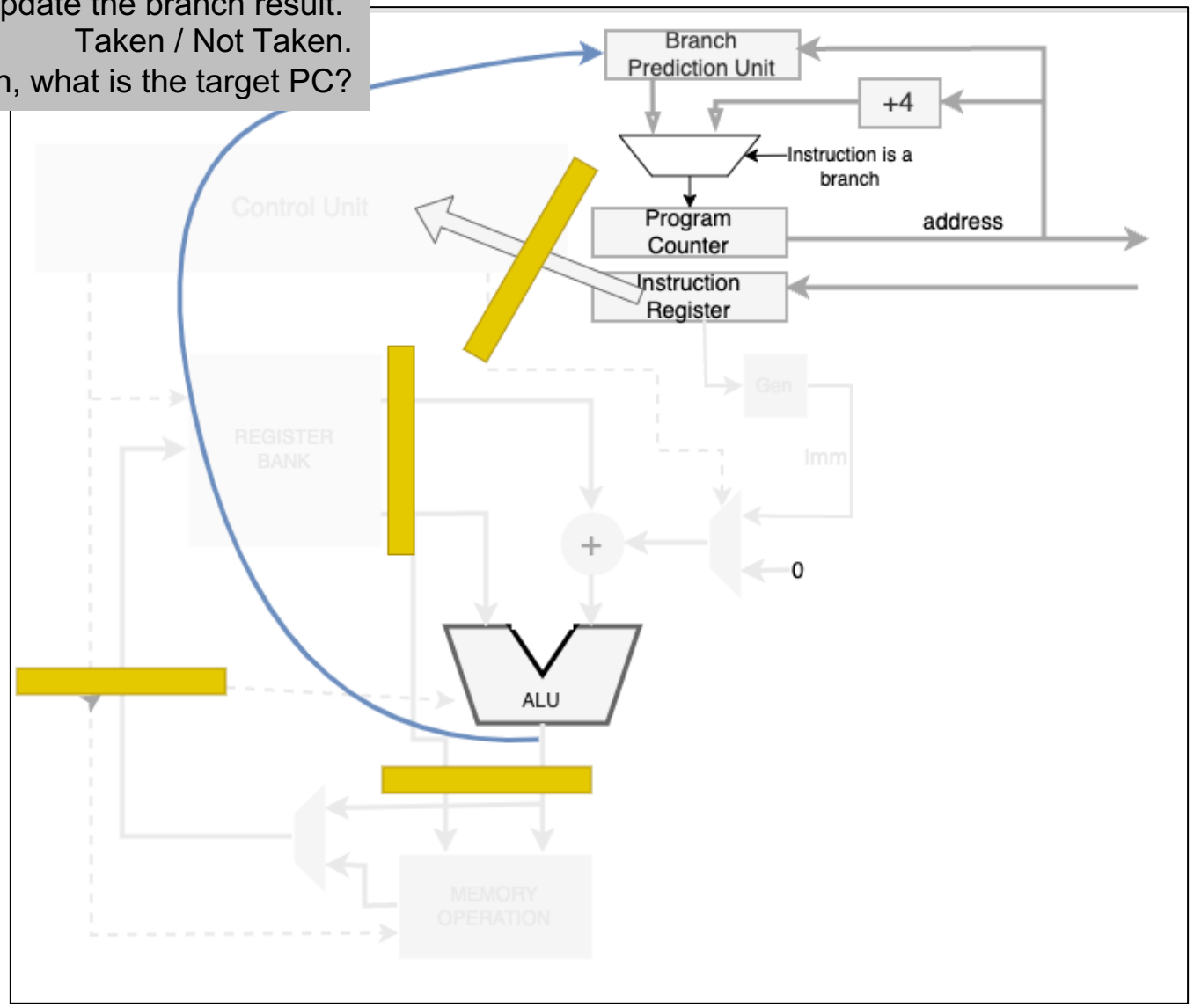




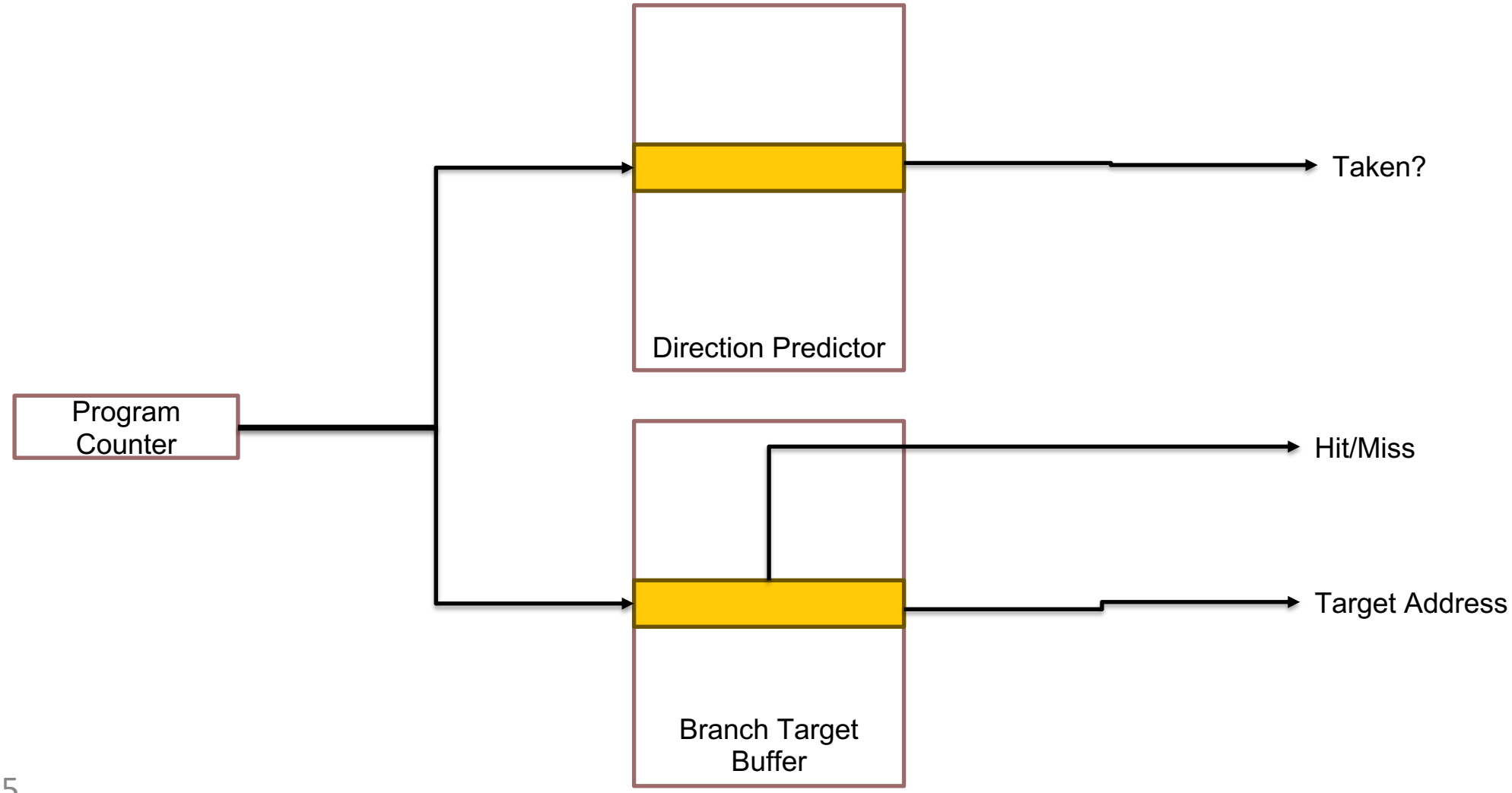
In practice, how do we make this prediction?

The Branch Prediction Unit

Update the branch result.
Taken / Not Taken.
If Taken, what is the target PC?



The Branch Prediction Unit



Direction Predictor

Branch Outcomes tend to repeat!
Learn from history

Branch not taken (BNT) more likely than branch taken (BT)

```
i=10;
while(--i){
   ..
   ..
}
```

Branch taken (BT) more likely than branch not taken (BNT)

```
if(age > 75 || age < 6){
   ..
   ..
}else{
   ..
   ..
}
```

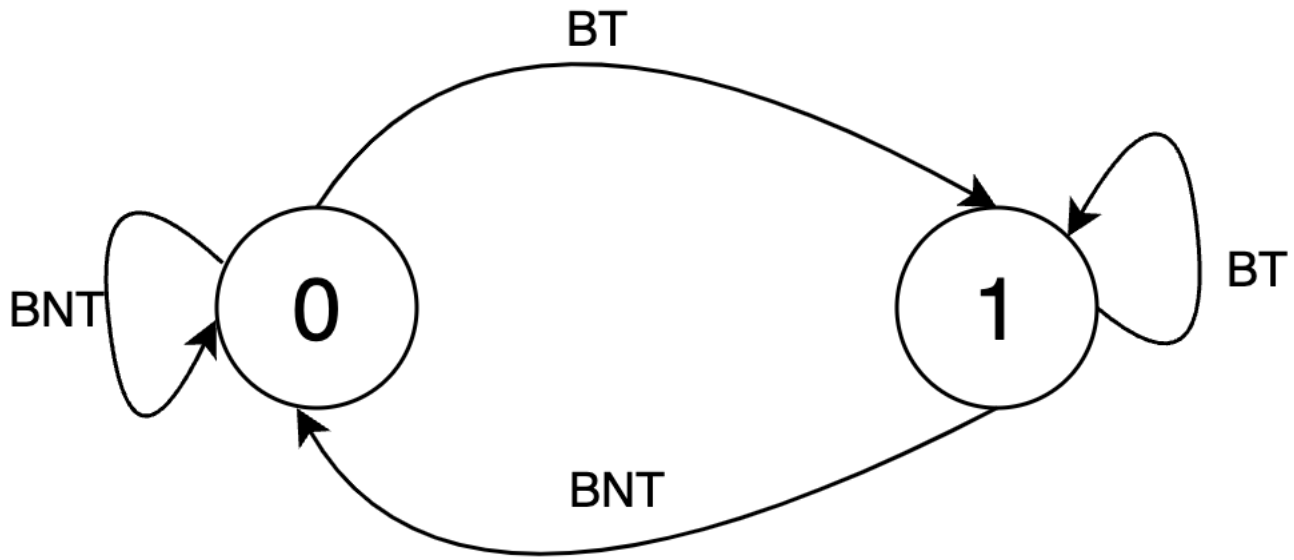
Branch taken (BT) more likely than branch not taken (BNT)

```
i=10;
do{
   ..
   ..
}while(--i);
```

Branch taken (BT) more likely than branch not taken (BNT)

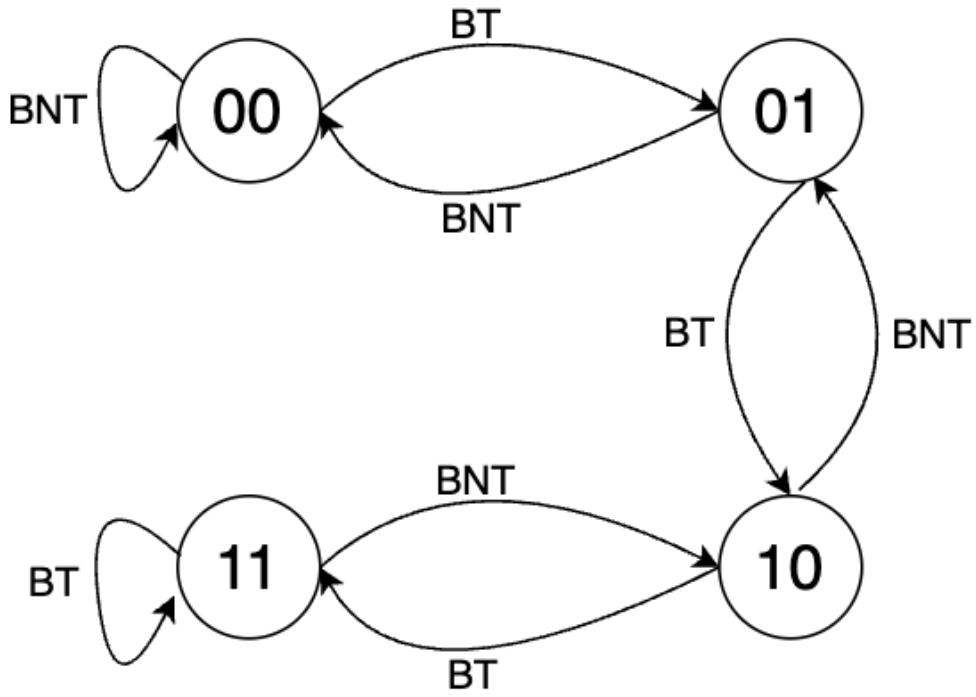
```
if(slno == -1){
   return -ERRORNO;
}
```

Direction Predictor
(1-bit Saturating Counter)



0: Predict Branch Not Taken (BNT)
1: Predict Branch Taken (BT)

Direction Predictor with 2-bit saturating counter



00: Strong Branch Not Taken → Predict (BNT)
01: Weak Branch Not Taken → Predict (BT)
10: Weak Branch Taken → Predict (BT)
11: Strong Branch Taken → Predict (BT)

Branch outcomes may be learnt by observing program execution

Local Prediction

Branch not taken (BNT) more likely than branch taken (BT)

```
i=10;
while(--i){
   ..
   ..
}
```

Branch taken (BT) more likely than branch not taken (BNT)

```
if(age > 75 || age < 6){
   ..
   ..
}else{
   ..
   ..
}
```

Branch taken (BT) more likely than branch not taken (BNT)

```
i=10;
do{
   ..
   ..
}while(--i);
```

Branch taken (BT) more likely than branch not taken (BNT)

```
if(slno == -1){
   return -ERRORNO;
}
```

Global Prediction

Sometimes, branch Outcome is similar to other branches. Here, outcome of if in function2 same as outcome in function1.

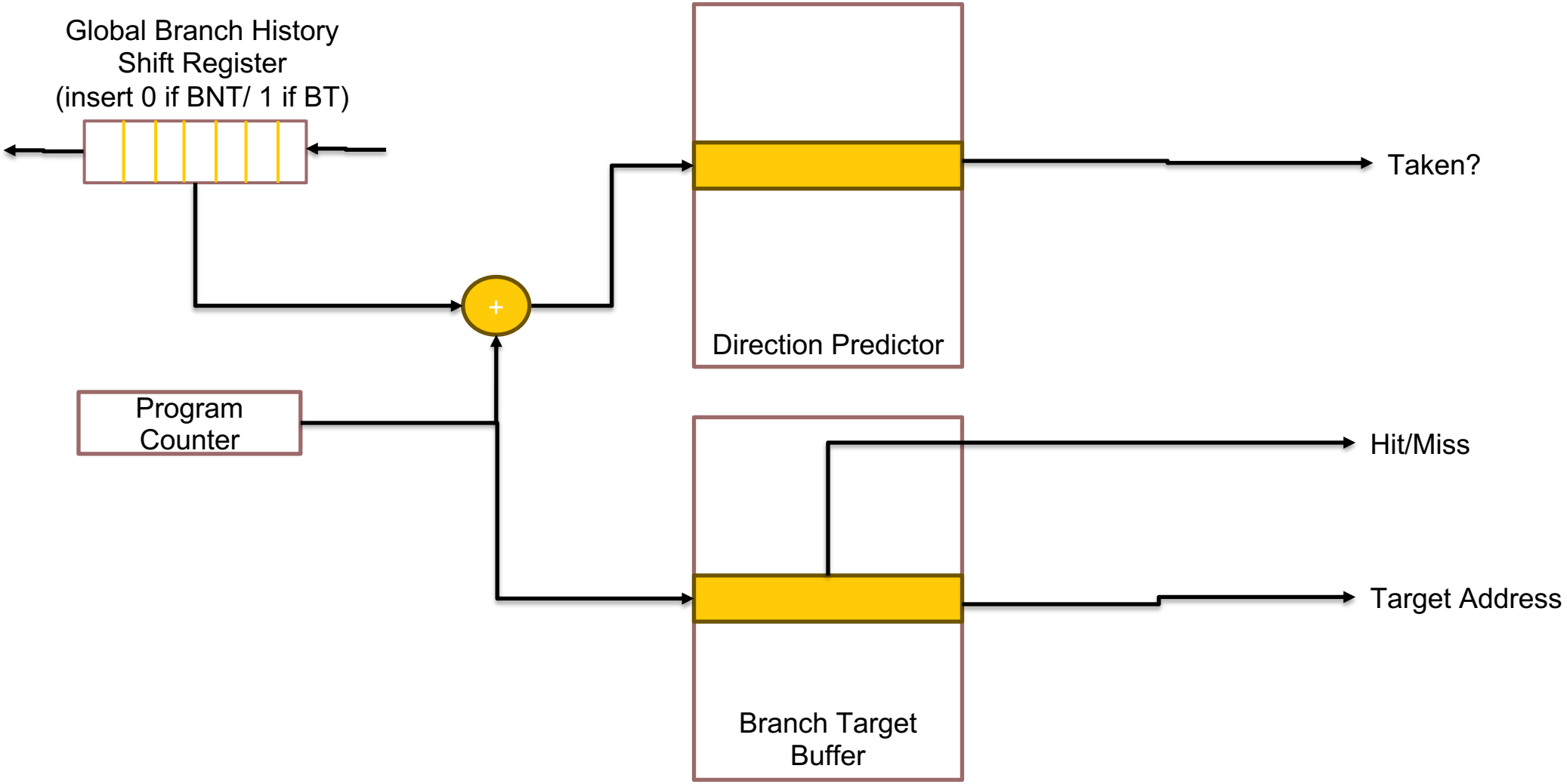
```
int global_flag = 0;

void function1() {
    if (global_flag) {
        ...
    } else {
        ...
    }
}

void function2() {
    if (global_flag) {
        ...
    } else {
        ...
    }
}

int main() {
    global_flag = 1;
    function1();
    function2();
    return 0;
}
```

Path-based Prediction

Sometimes, outcome of a branch is based on the path the program has taken

```
int calculate_sum(int a, int b) {
  if (a > 0) {
    if (b > 0)
      return a + b;
    else
      return a;
  } else {
    if (b > 0)
       return b;
    else
       return 0;
  }
}

int main() {
  int result1 = calculate_sum(5, 10);
  printf("Result 1: %d\n", result1);

  int result2 = calculate_sum(-3, 7);
  printf("Result 2: %d\n", result2);

  int result3 = calculate_sum(-2, -5);
  printf("Result 3: %d\n", result3);

  return 0;
}
```

Adding Context to Branch Prediction: GShare



Performance of Scalar Pipeline:

IPC: Instructions per cycle

$$\text{CPU Performance} = \frac{1}{\text{instruction count}} \times \frac{\text{instructions}}{\text{cycle}} \times \frac{1}{\text{cycle time}} = \frac{\text{IPC} \times \text{frequency}}{\text{instruction count}}$$

Frequency (performance)

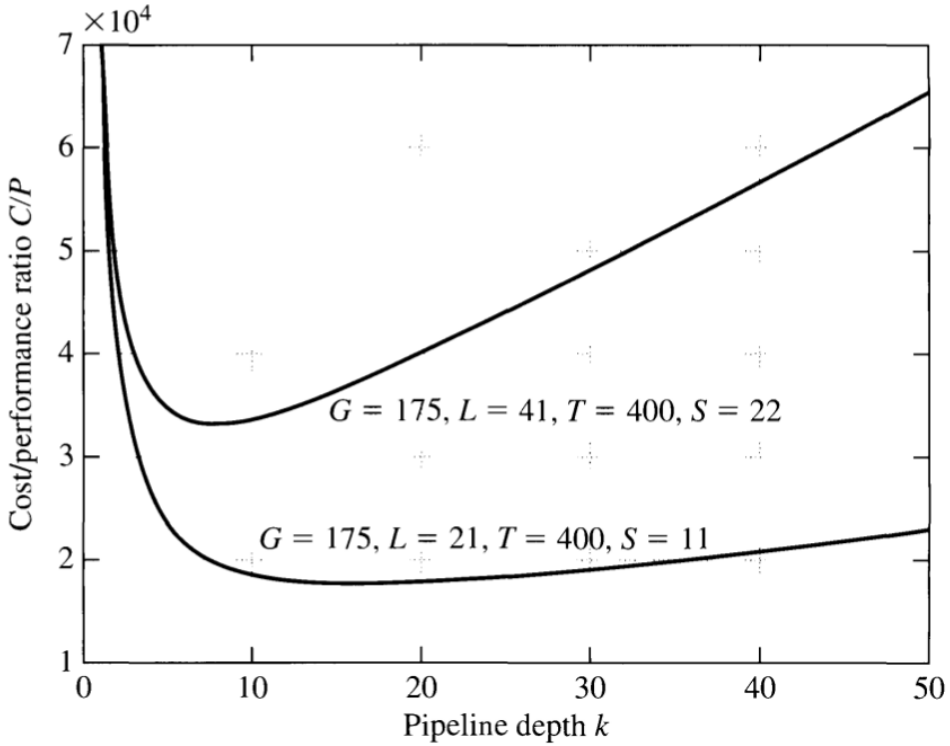
Up to a point, clock frequency can be increased by deeper pipelines

$$C = G + k \times L$$

$$P = \frac{1}{(T/k + S)}$$

$$\frac{C}{P} = \frac{G + k \times L}{\frac{1}{(T/k + S)}}$$

C: Cost;
G: Cost of Non-pipelined design;
k: Number of pipeline stages
S: delay due to pipeline buffer
T: delay of the Non-pipelined design.



Superscalar Organization

Chester Rebeiro
Indian Institute of Technology Madras

chester@cse.iitm.ac.in

Limitations of Scalar Pipeline

$$\text{CPU Performance} = \frac{1}{\text{instruction count}} \times \frac{\text{instructions}}{\text{cycle}} \times \frac{1}{\text{cycle time}} = \frac{\text{IPC} \times \text{frequency}}{\text{instruction count}}$$

For a scalar pipeline, this is bounded by 1

Frequency

(1) Deeper pipelines would mean,

- complexity of forwarding blocks and hazard detection increases.
- Delays due to pipeline buffers start becoming non-significant.
- Stalls due to branch mis-prediction become significant.

(2) Instructions / cycle bounded by 1

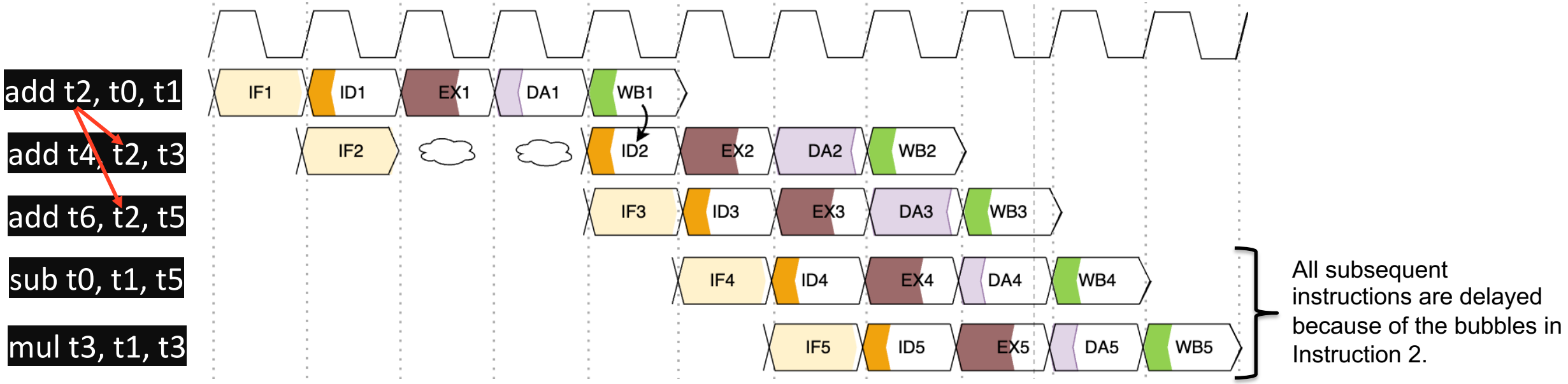
(3) Instructions can be considerably diverse, but the scalar pipeline works best with identical instructions.

- For example, MUL and DIV has considerably longer latencies compared to ADD and SUB. MUL and DIV ideally require multiple clock cycles, while ADD and SUB require a single clock cycle.
- However, in a scalar pipeline, all these instructions should execute identically (same number of delays or same clock cycles)

Limitations of Scalar Pipeline

(4) Performance lost due to rigid pipeline.

- Instructions flow in a lock-step fashion in the scalar pipeline. When an instruction stalls for N clock cycles, all subsequent instructions are delayed by N clock cycles.

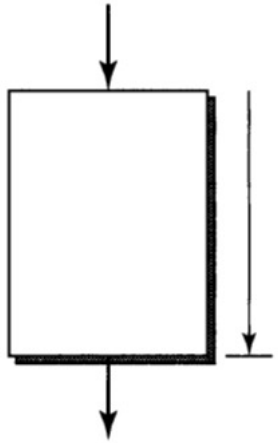


From Scalar to SuperScalar Pipelines

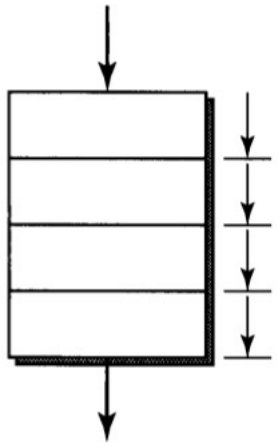
- Overcomes the limitations of Scalar Pipelines
  - (2) No longer bounded by 1 instruction / cycle. Possible to complete >1 instructions/cycle
  - (3) Can efficiently cater to diverse instructions
    - MUL, DIV, ADD, and SUB, do not have to take the same number of clock cycles or have the same delay
  - (4) Rigid pipeline
    - Permits instructions to execute out-of-order, if there are no dependencies. Thus, a stall in one instruction may not delay a subsequent instruction's execution.

Achieved by Diversified Parallel Pipeline

Parallel Pipelines

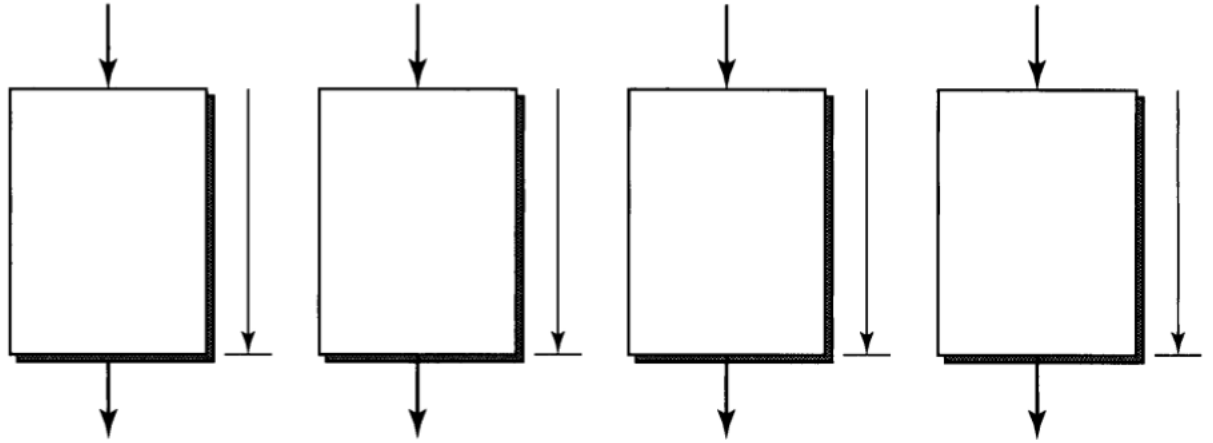


(a) No parallelism

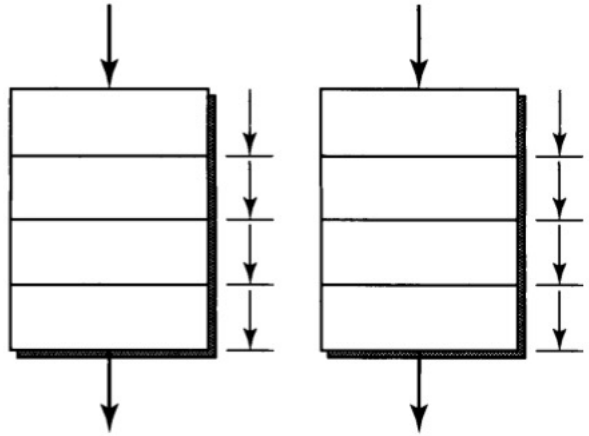


(b) Temporal parallelism

Regular pipeline



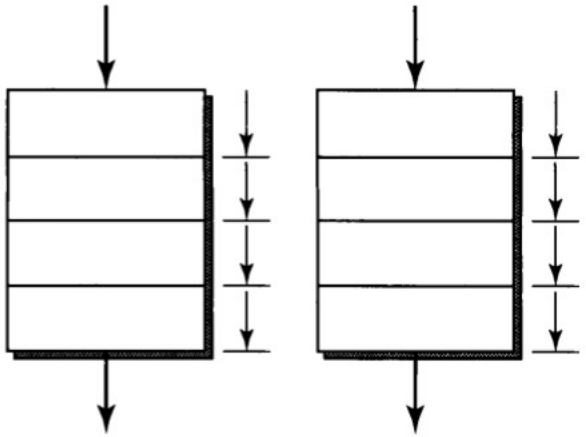
(c) Spatial parallelism



(d) Parallel pipeline

Combines Temporal and Spatial parallelism

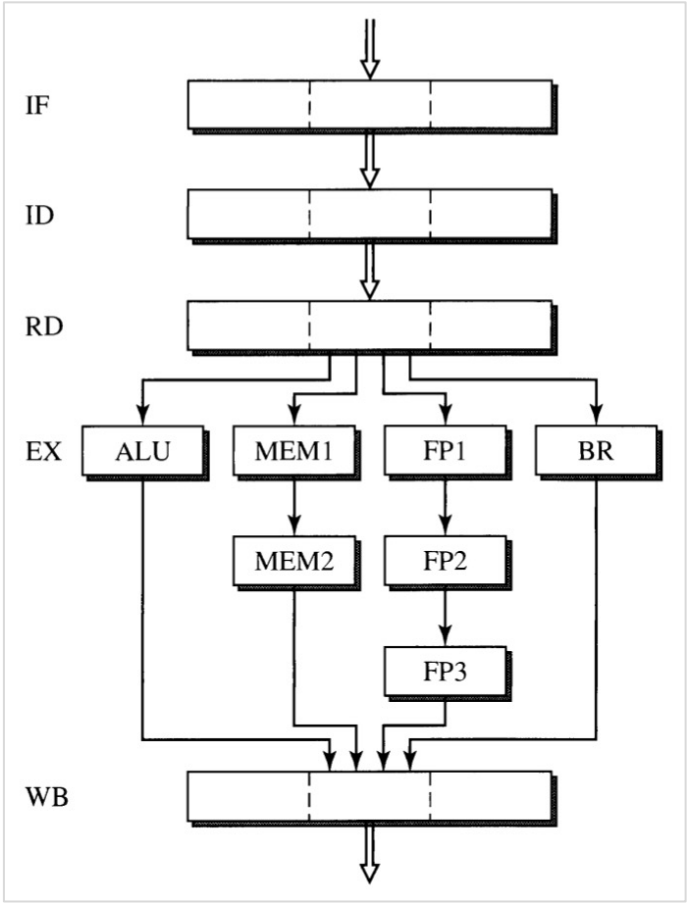
Diversified Pipeline



(d) Parallel pipeline

Parallel Pipelines do not cater to the diversity of instructions – eg. Mul, div, add, and sub have different hardware and latencies

Diversified execution pipes cater to different instructions. Each pipeline has different depth catering to different instructions.

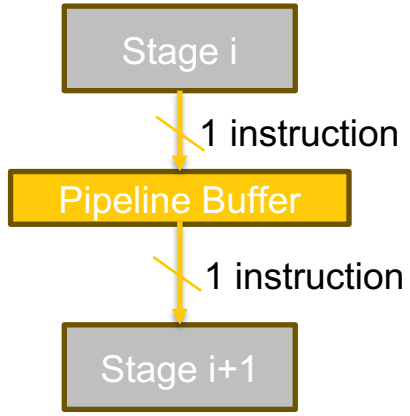


A diversified pipeline with 4 execution pipes

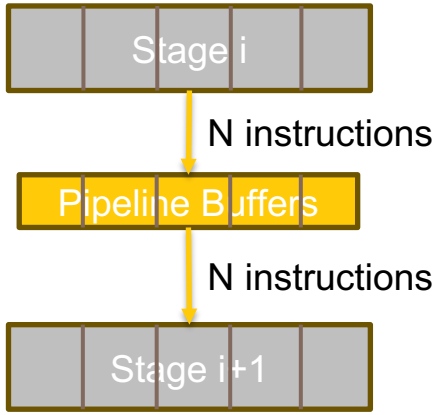
3-issue pipeline: Can issue 3 instructions simultaneously to each stage

Pipeline Buffers in Scalar and Parallel Pipelines

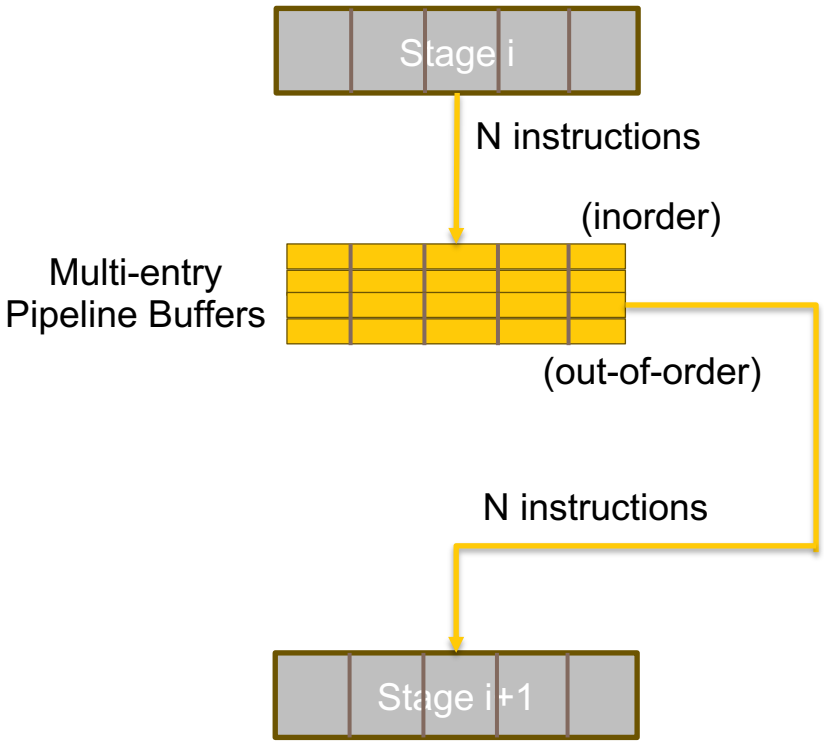
Scalar Pipelines make use of **single pipeline buffers** between stages. These buffers hold single-entry buffers. That latch the result and control signals produced in stage i. These are used in stage i+1.



Parallel Pipelines make use of **multi-entry pipeline buffers** between stages. Multiple instructions can be latched into buffers. The latch holds result and control signals produced in stage i. These are used in stage i+1.



Multi-entry buffer with reorder

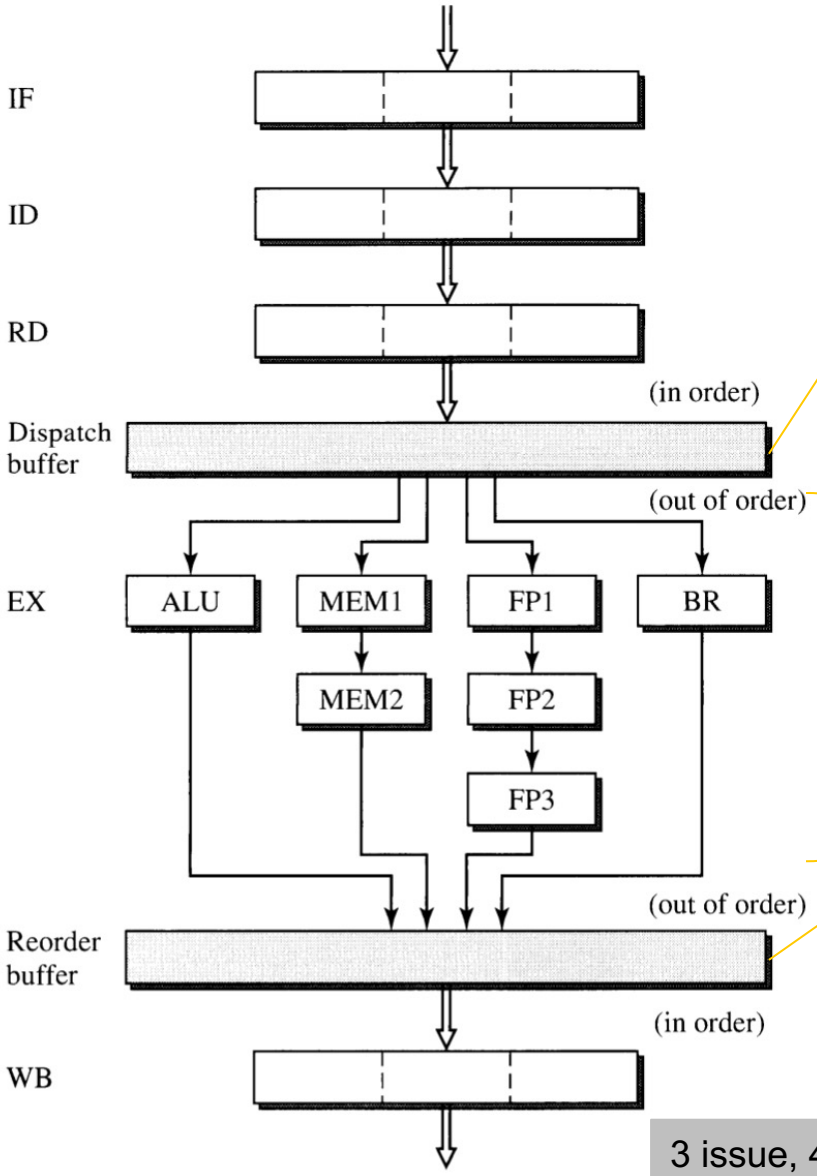


Multiple decoded instructions latched into each multi-entry buffer in every clock cycle by Stage I (in-order)

Multiple instructions read from any entry in the multi-entry buffer in Stage i+1. Not necessarily in the program order.

While in the buffer, instructions can be updated or modified. For example, by forwarding paths.

Multi-entry buffer in a dynamic pipeline



Dispatch Buffer. Loaded with decoded instructions according to program order.

(EX) Execution Stage. 4 pipelines. Each pipeline caters to a class of instructions, has different cycle requirement. Instructions are scheduled out-of-order, based on the availability of a pipeline.

Reorder Buffer. Commits the results in the order of the program to ensure correctness. Instructions which completed early, are buffered until other results are available. Multiple results can be committed in a single clock cycle.

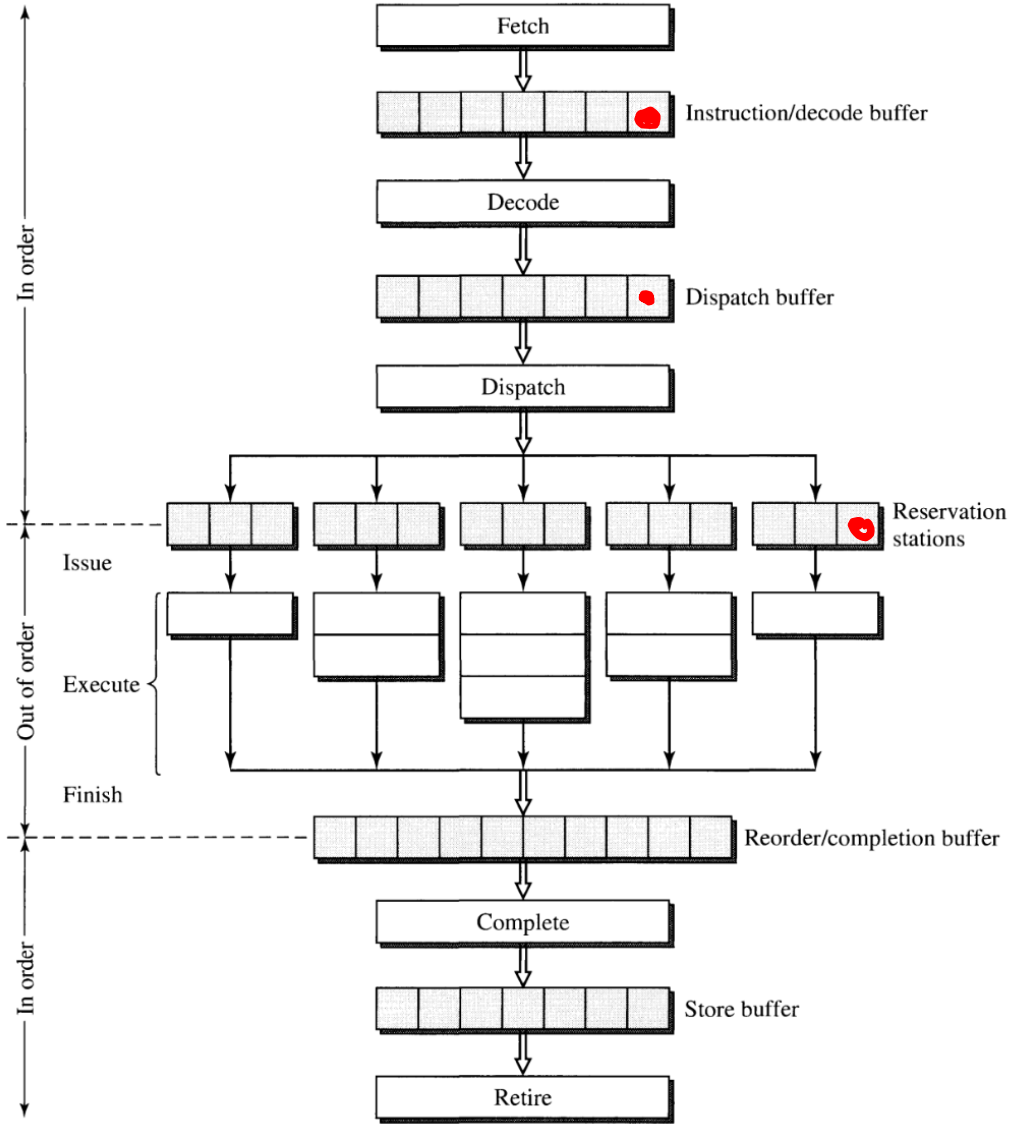
3 issue, 4 execution pipelines

Superscalar Pipeline Overview

TEM: An Example of a Superscalar Pipeline

6 stages: Fetch, Decode, Dispatch, Execute, Complete, Retire

Dispatch stage: Associate instruction with a functional unit (like ALU/FPU etc)
Issuing buffer: used to initiate execution
Execute stage can have multiple pipelines
Complete stage reorders instruction results to ensure the results are in-order

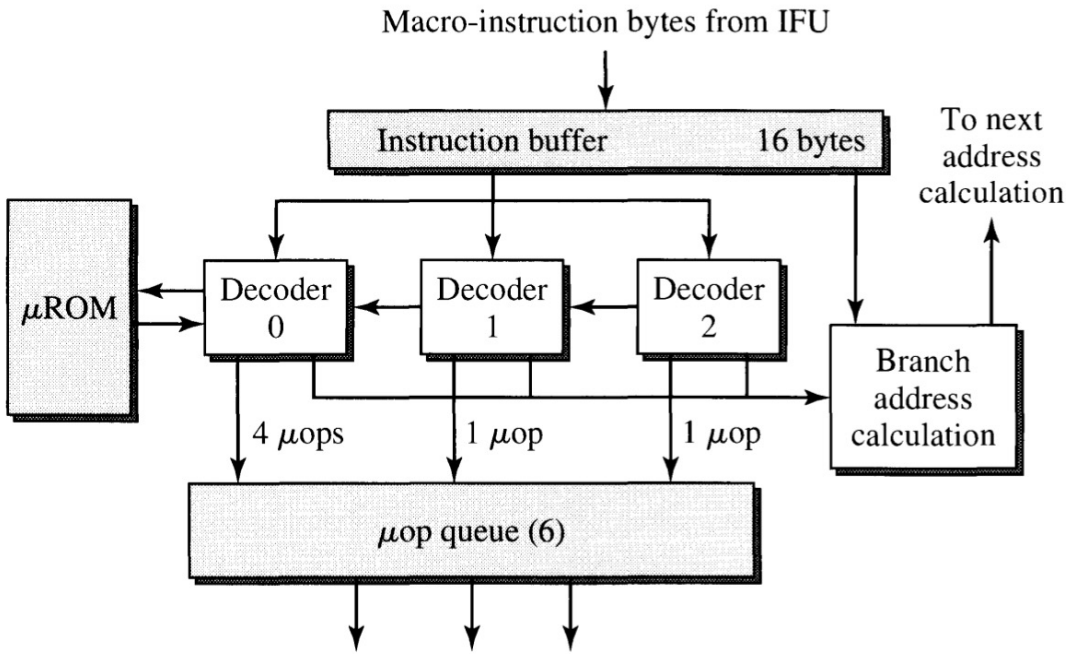


Instruction Fetch Stage

- Unlike scalar pipeline, fetch more than one instruction from Instruction (I)-cache in every clock cycle.
- If s-issue pipeline, IF stage will fetch s-instructions from I-cache
  - If PC points to address A, then the instruction at A and subsequent s-1 instructions are fetched. This is called the **Fetch Group**.
  - ideally multiple instructions from the same cache line is fetched.

Decode Stage

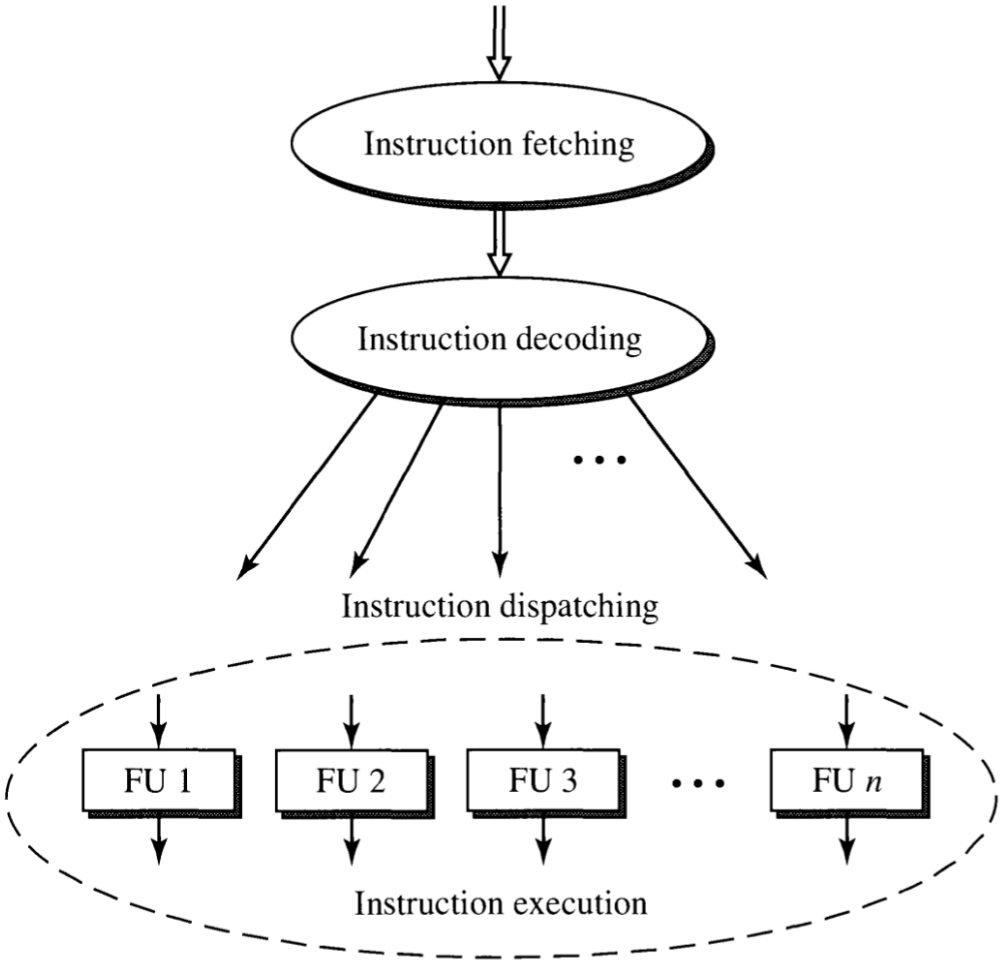
- Unlike, scalar pipeline, superscalar decode stage only decodes the instruction. It does not read the register contents.
- Multiple instructions are simultaneously decoded and dependencies between instructions identified.
  - Early identification of dependencies helps parallelize instructions better
- On RISC machines, the decode stage mainly contains a large number of comparators, multi-ported register files, etc.
- On CISC machines, the variable length instructions make it even more complex. On Intel processors, instructions converted to micro-ops.



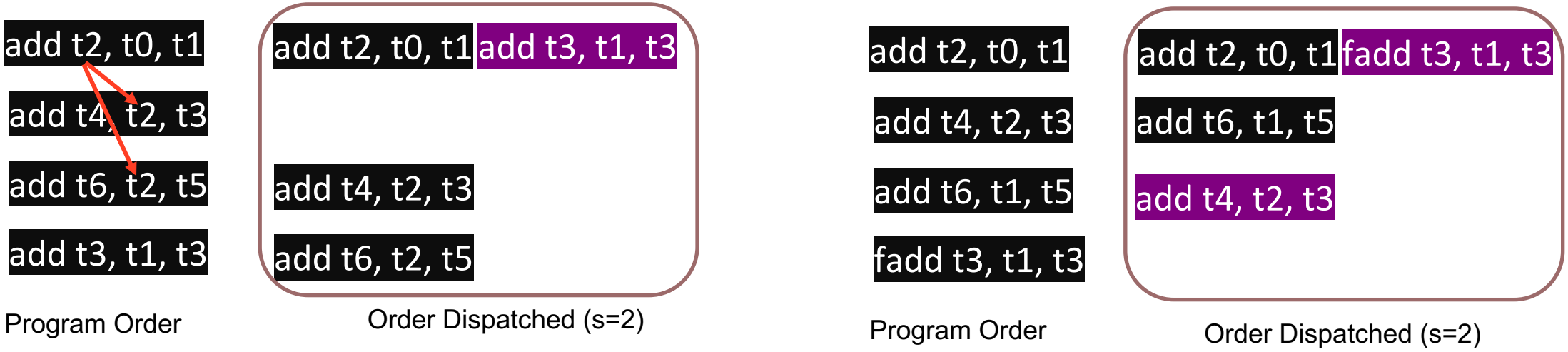
Fetch / Decode unit in Intel P6 processors

Instruction Dispatch

- Between decode and execute. This stage is required because of the diversified pipelines.
- Instructions are dispatched to a pipeline based on,
  - Type of instruction (eg. Load/store dispatched to the memory operations pipeline while add/sub dispatched to the ALU)
  - Availability of operands. All operands needed for an instruction should be available.
  - Availability of a pipeline (eg. some processors have 2 ALUs. Instructions are dispatched to an ALU based on the load on that ALU)

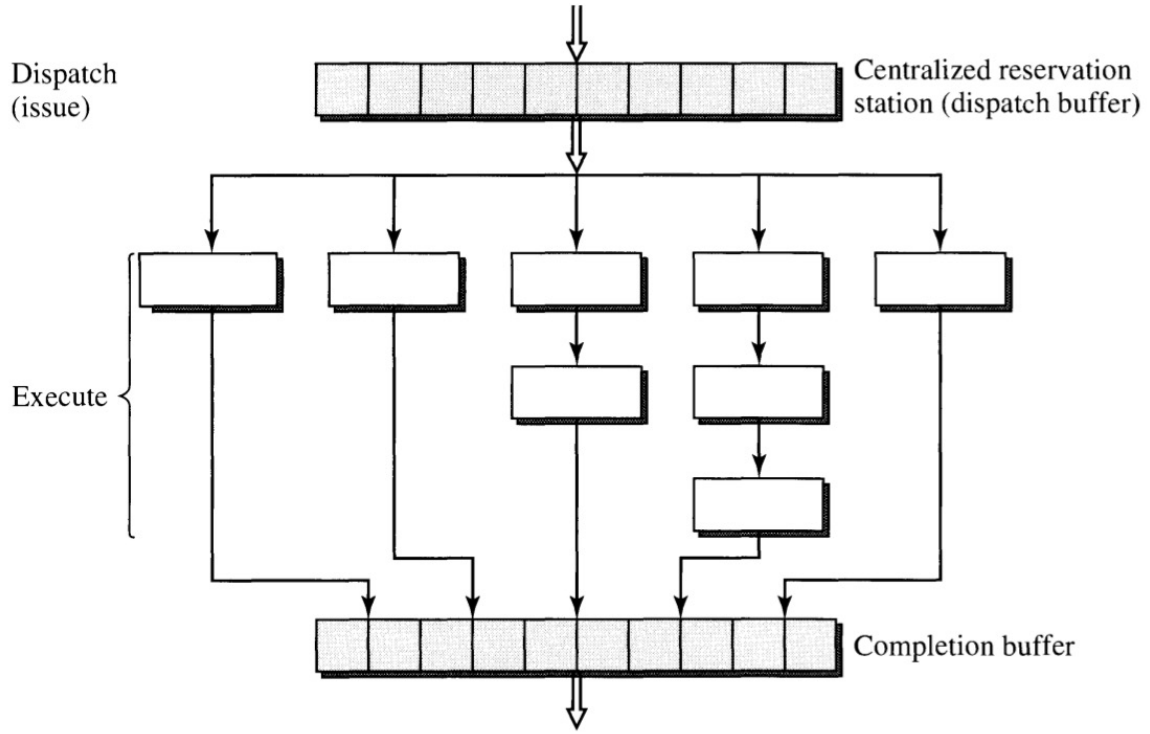


Instruction Dispatching

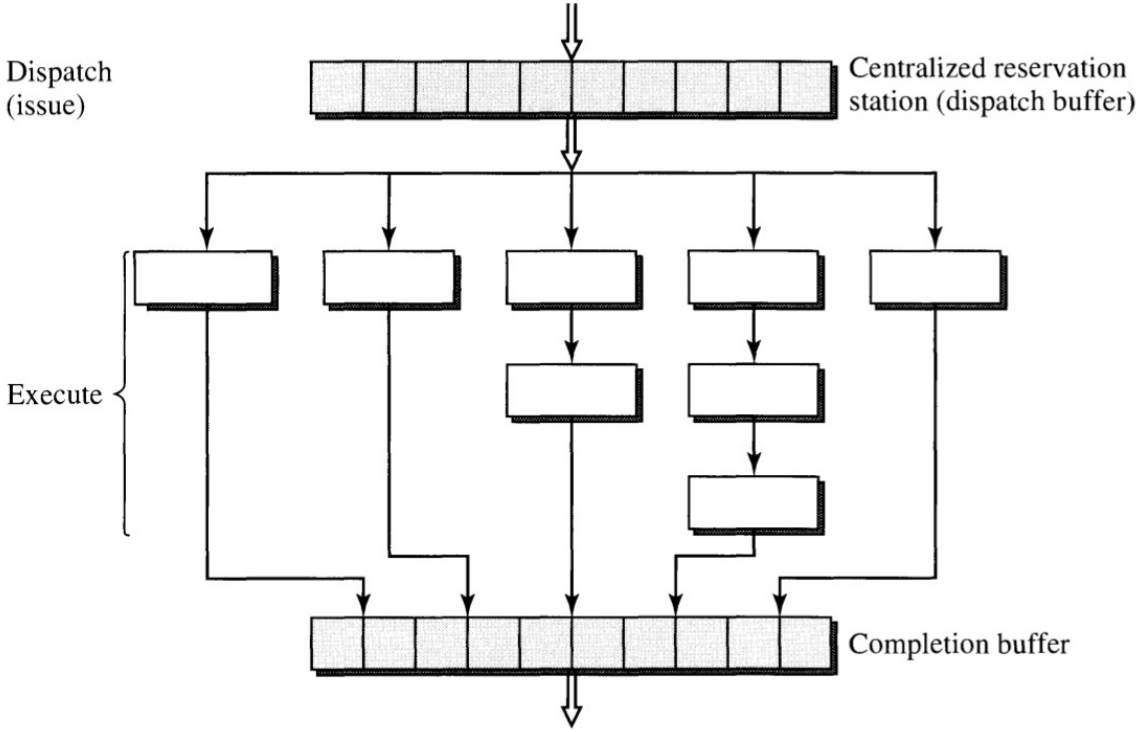


Dispatch Buffer

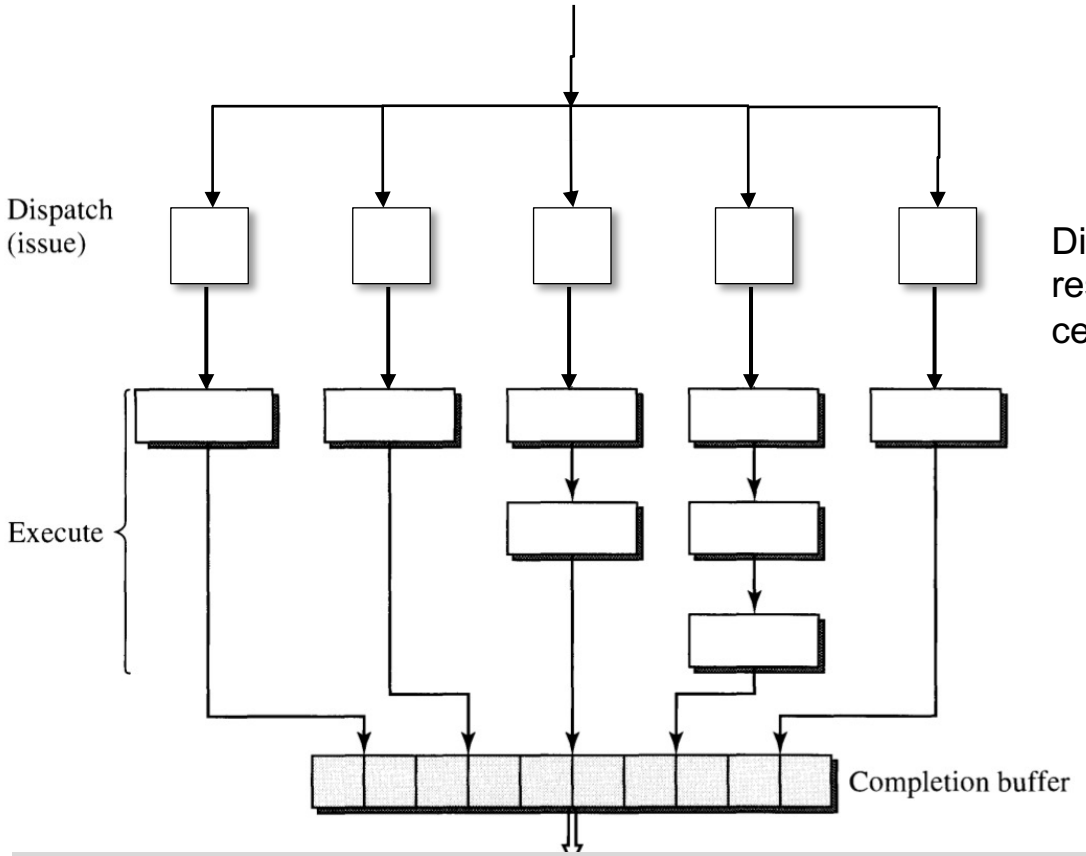
- All operands (from registers) needed for an instruction should be available for an instruction to be dispatched and the pipeline must be available.
- Until then, instruction is stored in the dispatc buffer, other instructions which are independent and whose operands are ready can be dispatched.
- Thus, instructions may be dispatched out-of-order, as soon as its operands are available.
- s instructions can be dispatched simultaneou (s is the issue)
- Dispatch Buffer also called Reservation Statio..



Dispatch Buffer Centralized or Distributed



Centralized Reservation Station:
Hold all instructions regardless of type
More complexity (multi-ported buffers)
better utilization or resources



Distributed
reservation
centers

Distributed Reservation Station:
Simple single ported multi-entry buffers that store instructions corresponding to a specific functional unit
Utilization of buffers may not be uniform, causing stalls

Execution Stage

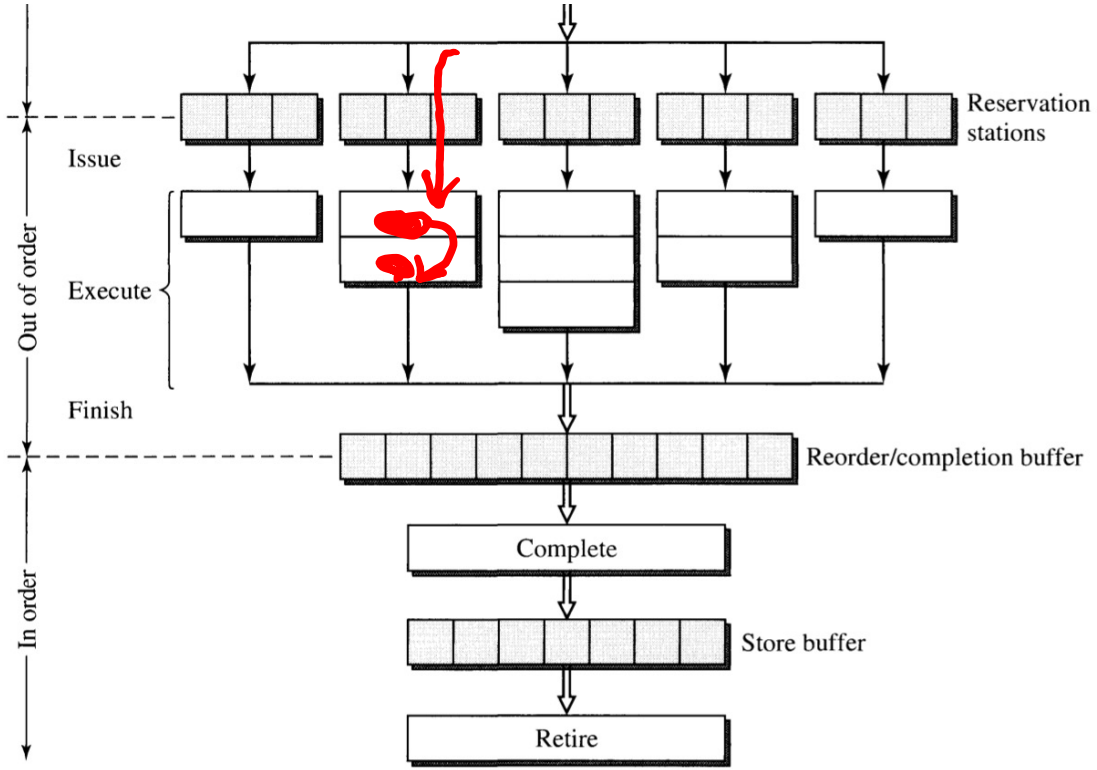
- Can have multiple similar functional units, for example two independent pipes for ALUs
- Can perform compound arithmetic operations, like (AxB) + C
- Support for Single Instruction Multiple Data types

What is the right composition of functional units?
4-2-4 rule is one example: For every 4 ALU units, have 2 Branch units, 4 memory units (in practice, have fewer memory units due to memory bottlenecks)

How much of parallelism vs pipelining is appropriate?

Instruction Completion and Retiring

- Instruction is ‘completed’ when it finishes execution
  - Such instructions are stored in a completion buffer
  - Results stored in temporary registers
- Store Instruction are ‘retired’ when the D-cache is updated.



An example of Superscalar instruction flow

- Consider a 3-issue superscalar processor with stages: Instruction Fetch, Decode, Dispatch, Execute, and Writeback
- There is one 2-stage load/store unit, 1-pipeline stage ALU, and 1-pipeline stage mul/div unit
- Consider the following instructions.
- Outline the flow of instructions through this CPU.

```
lw a0, 0(x2)
lw a1, 4(x2)
add a2, a0, a1
sub a3, a1, a0
lw a0, 8(x2)
div a5, a3, a2
```

Instruction flow in the CPU

lw a0, 0(x2)	IF1	ID1	DIS1	EX1	EX1	WB1			
lw a1, 4(x2)	IF2	ID2	DIS2		EX2	EX2	WB2		
add a2, a0, a1	IF3	ID3	DIS3				EX3	WB3	
sub a3, a1, a0		IF4	ID4	DIS4			EX4	WB4	
lw a0, 8(x2)		IF5	ID5	DIS5		EX5	EX6	WB5	
div a5, a3, a2		IF6	ID6	DIS6				EX6	WB6

3 issue, therefore at-most 3 instructions move from one stage to the next in each clock cycle

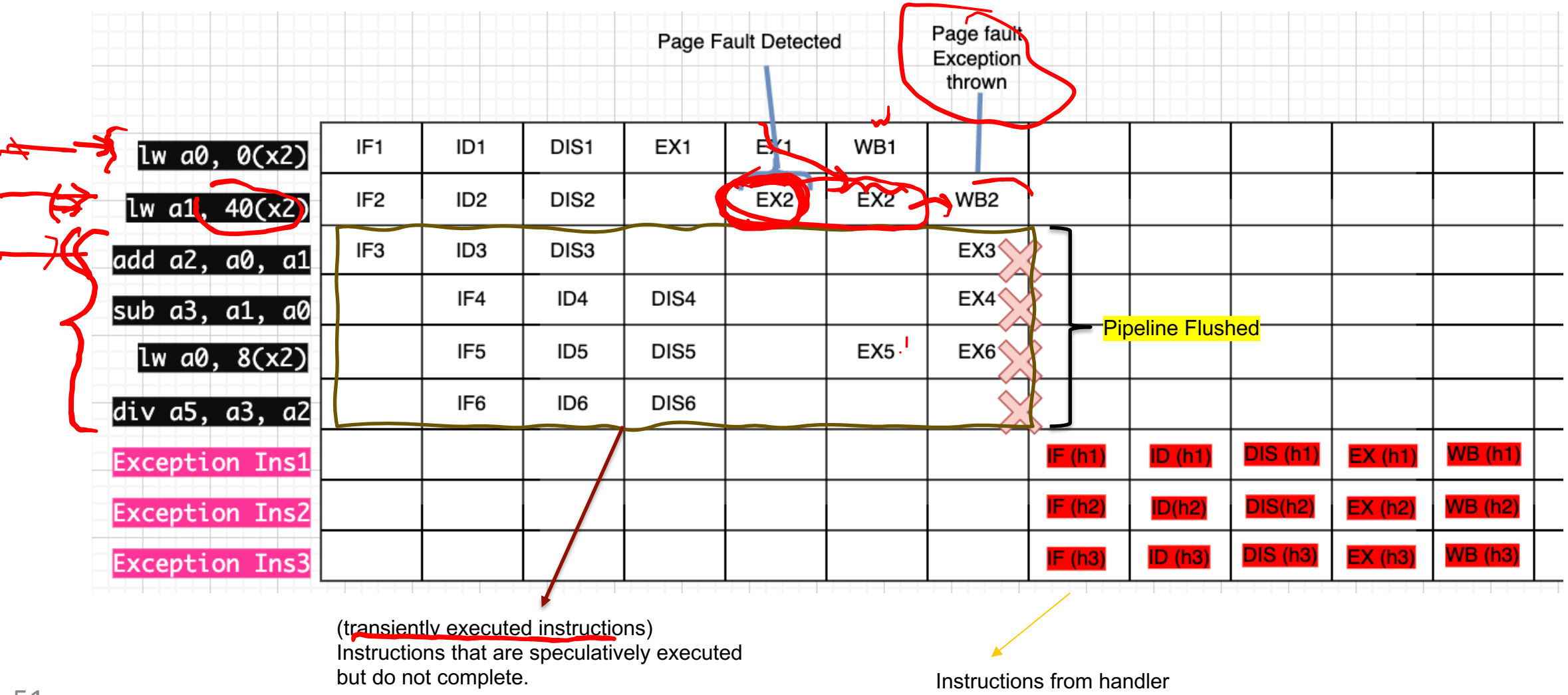
Can't move Instr 2 to Execute in this cycle, because there is only one load/store pipeline

Can't move Instr 3 to Execute in this cycle, because of dependencies with Instr 1 and Instr 2.

Executing Instr 5 out-of-order because it does not have any dependencies. We can't execute it in previous clock cycle, because of EX2 occupies the load/store pipeline

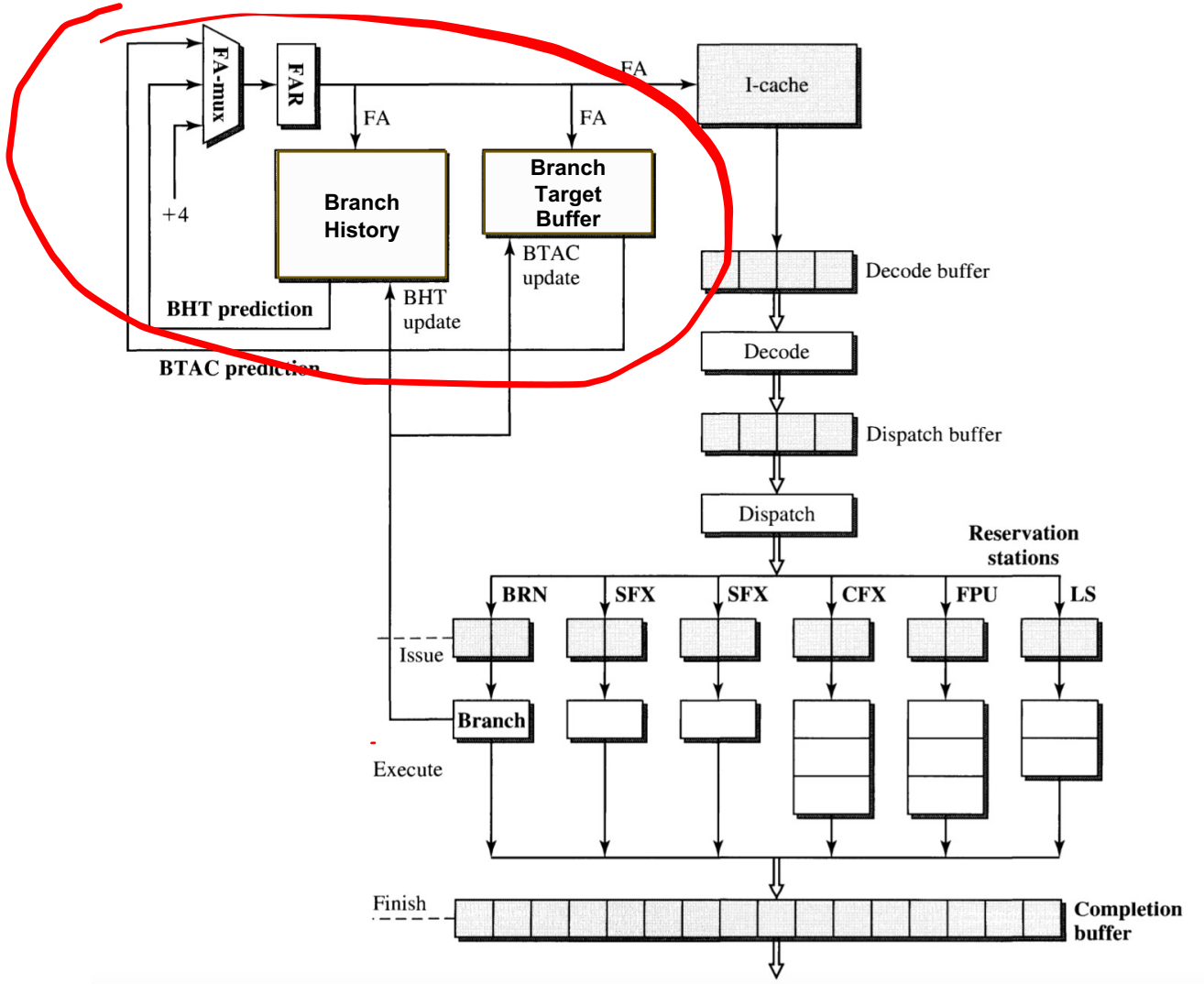
Writebacks always happen inorder

Instruction flow in the CPU (with an exception)

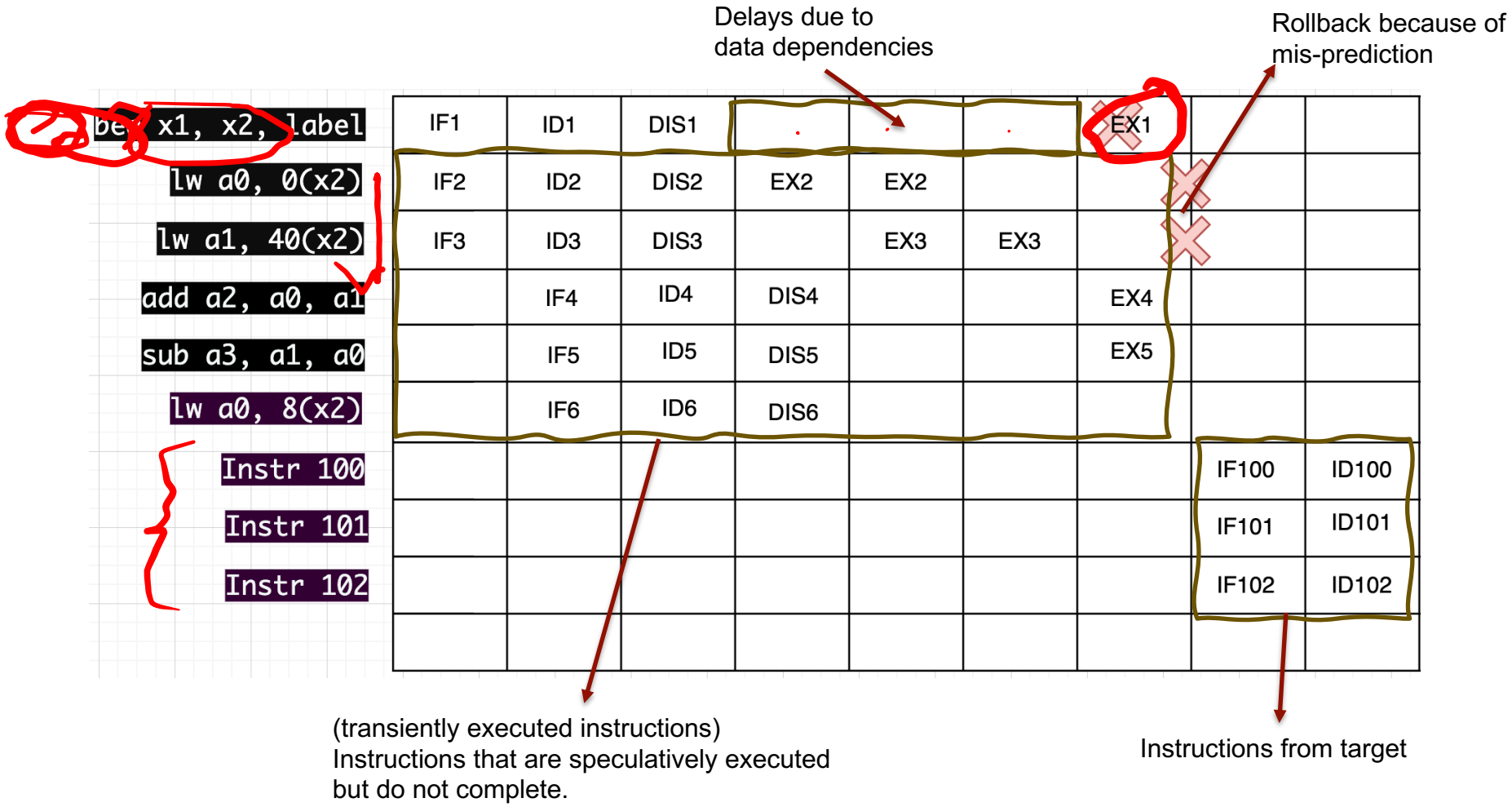


Branches are Expensive
Speculative Execution

Extensive use of the branch prediction

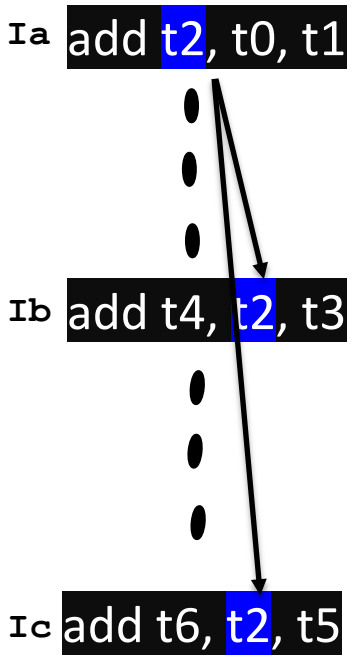


Speculatively Execution Example



Revisiting RAW Data Hazards

Read after Write
(True Dependencies)



Register t2 read after it is written.

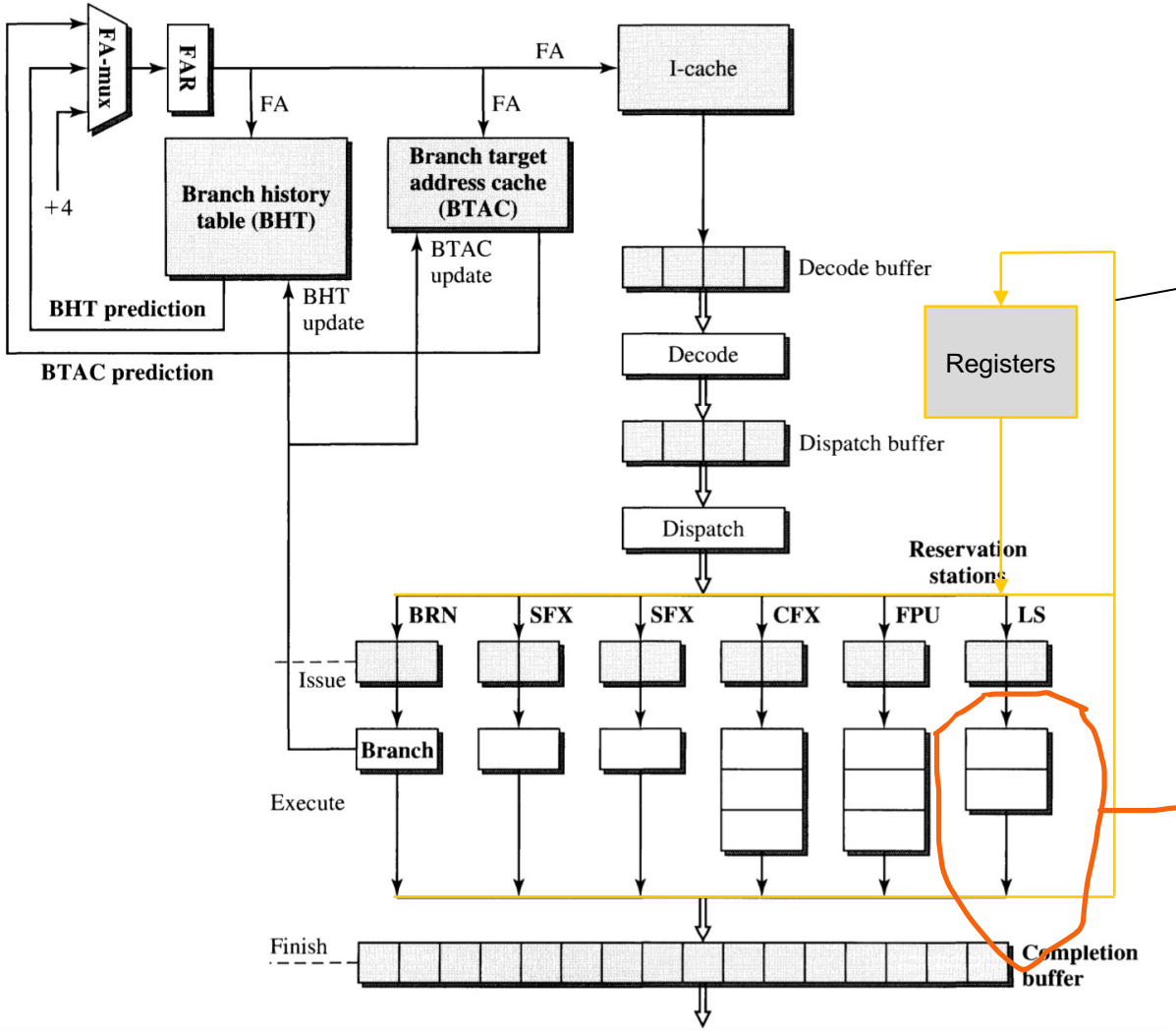
In scalar pipeline with N stages,
`Ia, Ib` and `Ic` should be within a window of N instructions in order to have stall in the pipeline.

In super-scalar pipelines stalls can occur based on the number of **in-flight** instructions.
If there can be a maximum of M in-flight instruction, Ia, Ib and Ic should be within a window of M instructions to have stall in the pipeline.

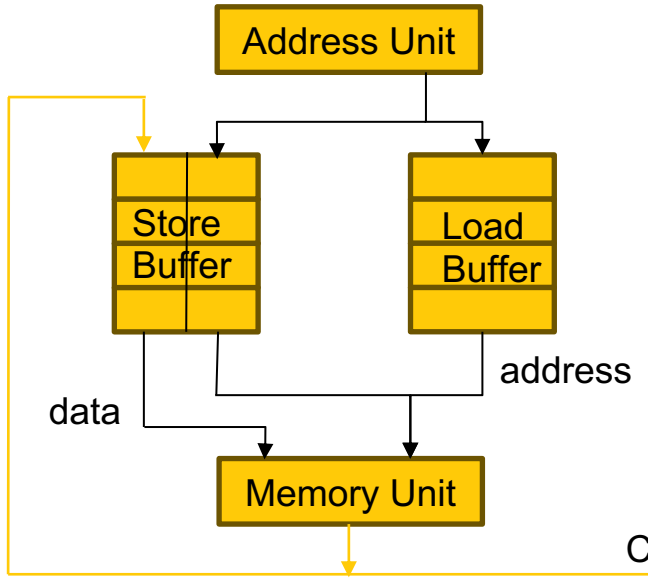
RAW addressed by
(a) forwarding
(b) out-of-order execution
(c) Speculation (mostly in memory operations)

In-flight instructions: are instruction which are decoded but not retired. They are partially processed, but the results are not present as yet.

Common Data Bus to enable forwarding



Common Data Bus: Results stored back in the reservation station soon as they are ready



Revisiting WAR Data Hazards

Write after Read (WAR): Pipeline writes to a register before a prior instruction reads it. (False dependency)

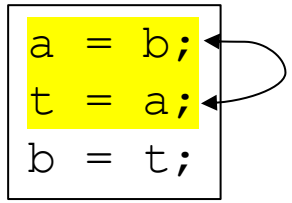
```
I1    ADD R1, R2, R3
          .
          .
          .
I2    ADD R3, R4, R5
```

Example

```
swap(a, b){
    t = a;
    a = b;
    b = t;
}
```

Register operations

```
ADD X1, X2, X0
ADD X2, X3, X0
ADD X3, X1, X0
```



Order in which instructions are dispatched for execution (viz-a-viz out-of-order)

Revisiting WAW Data Hazard

Write after Read (WAW): Pipeline writes to a register before a prior instruction writes to it. ((False dependency)

```
I1    ADD R1, R2, R3

I2    ADD R1, R4, R5
```

I2 updates register R1 before I1 updates it

Example

```
u = w * y;
t = a * b;
    o
    o
t = c + d;
    o
    o
```

```
u = w * y;
t = c + d;
    o
    o
    o
t = a * b;
    o
```

```
MUL X1, X2, X3;
ADD X4, X5, X6
    o
    o
    o
ADD X4, X7, X8
    o
```

Order in which instructions are dispatched for execution (viz-a-viz out-of-order)

Register Renaming for WAR and WAW Hazards

Main cause of WAR and WAW hazards is insufficient registers in CPU, due to which registers are reused.

We can use temporary registers to solve the problem. During Writeback, we writeback from the temporary results

```
ADD X1, X2, X0
ADD T1, X3, X0
ADD T2, X1, X0
```

```
ADD T1, X5, X6

ADD T2, X7, X8
```

Handling False Dependencies with Register Renaming

False dependencies occur due to insufficient registers

Fixed by have a large number of ‘physical registers’ and
a renaming table that maps ‘architectural registers’ to ‘physical registers’

Renaming Table
(RT)

R1	P1
R2	P2
R3	P3
R4	P4
R5	P5
R6	P6
R7	P7
R8	P8

Architectural registers
(visible to the programmer)

Physical Registers

Free / Inuse

P1		F
P2		
P3		
P4		
P5		
P6		
P7		
P8		
P9		
P10		
P11		
:		
:		
:		
:		
P128		

False Dependencies Example

Free / Inuse

```
ADD R2, R2, R2
SUB R1, R1, R2
ADD R2, R4, R4
MUL R3, R3, R2
MUL R2, R6, R6
ADD R5, R5, R2
```

WAW Dependencies

```
ADD R2, R2, R2
SUB R1, R1, R2
ADD R2, R4, R4
MUL R3, R3, R2
MUL R2, R6, R6
ADD R5, R5, R2
```

WAR Dependencies

Handling False Dependencies with Register Renaming

```
ADD R2, R2, R2      ADD
SUB R1, R1, R2      SUB
ADD R2, R4, R4      ADD
MUL R3, R3, R2      MUL
MUL R2, R6, R6      MUL
ADD R5, R5, R2      ADD
```

Renaming Table (RT)

R1	
R2	
R3	
R4	
R5	
R6	
R7	
R8	

Physical Registers

Free / Inuse

P1		F
P2		
P3		
P4		
P5		
P6		
P7		
P8		
P9		
P10		
P11		
:		
:		
:		
:		
P128		

Handling False Dependencies with Register Renaming

Fetched

```
ADD R2, R2, R2
SUB R1, R1, R2
ADD R2, R4, R4
MUL R3, R3, R2
MUL R2, R6, R6
ADD R5, R5, R2
```

Renamed

```
ADD R2, R2, R2
SUB R1, R1, R2
ADD R2, R4, R4
MUL R3, R3, R2
MUL R2, R6, R6
ADD R5, R5, R2
```

Renaming Table (RT)

R1	P1
R2	P2
R3	P3
R4	P4
R5	P5
R6	P6
R7	P7
R8	P8

Physical Registers

P1		F
P2		
P3		
P4		
P5		
P6		
P7		
P8		
P9		
P10		
P11		
:		
:		
:		
:		
P128		

1

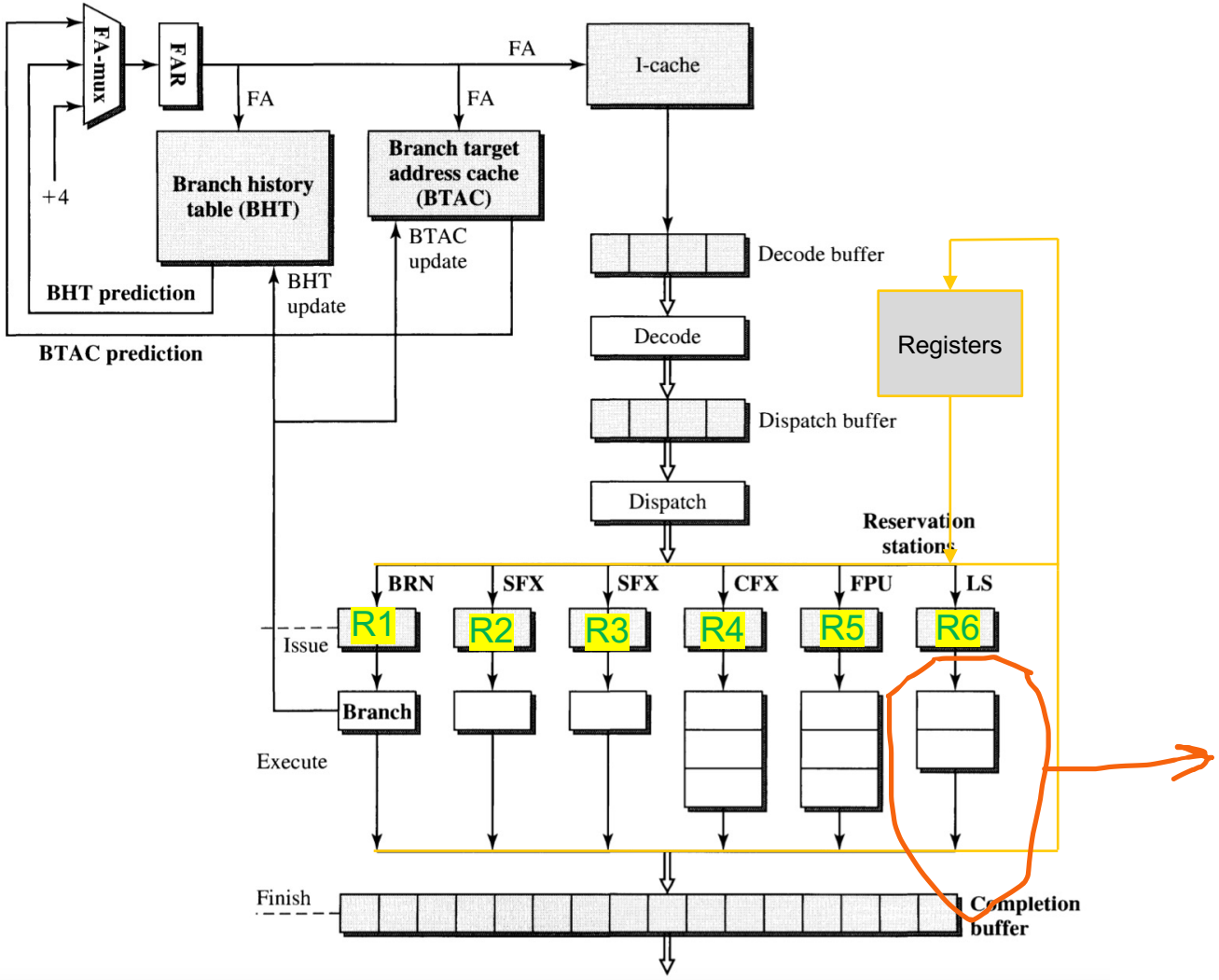
download animation here.

Realizing Register Renaming

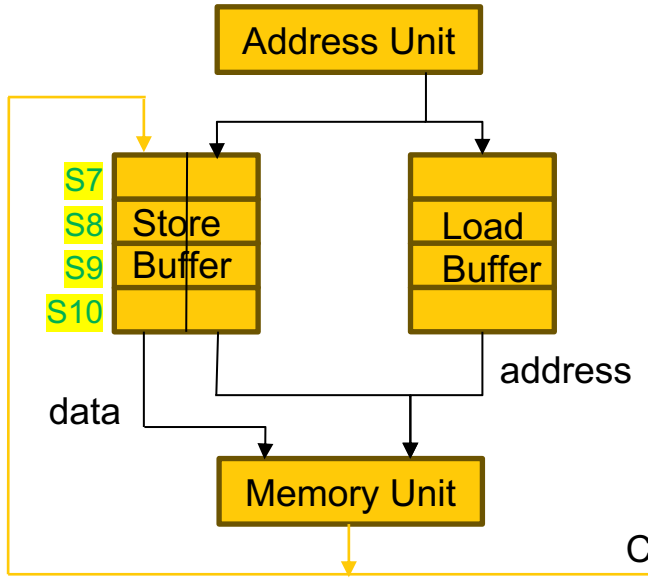
Extending architectural registers (Increase the size of the register bank)

Use virtual registers

Virtual Registers



The reservation station number, store buffer, form the virtual set of registers



Reservation Station Contents

Op	The operation to perform on source operands S1 and S2
Qj and Qk	The reservation stations that will produce the corresponding source operand; A value of 0 indicates that the source operand is already available in Vj or Vk, or is unnecessary
Vj and Vk	The values of the source operand.
A	Information about memory address calculation for a load or store. Initially the immediate field of the instruction is stored here. After the address calculator, the effective address is stored here
Busy	Reservation station and its accompanying functional unit are occupied.

Example of Reservation Station Utilization

LW x6, 32(x9)	IF1	D1	DIS1	EX1	EX2	WB1				
LW x2, 44(x10)	IF2	D2	DIS2			EX1	EX2	WB2		
MUL x1, x2, x4	IF3	D3	DIS3					EX3	WB3	
SUB x8, x2, x6		IF4	D4	DIS4				EX4	WB4	

Register Status

x1	Mul/Div
x2	Load2
x3	
x4	
x5	
x6	Load1
x7	
x8	
x9	

Reservation Stations

	Busy	Op	Vj	Vk	Qj	Qk	A
Load 1	Yes	Load					32 + Regs[x9]
Load 2	Yes	Load					44 +Regs[x10]
ALU	No						
MUL/DIV	Yes	MUL			Load2	Regs[x4]	

Example of Reservation Station Utilization

LW x6, 32(x9)	IF1	D1	DIS1	EX1	EX2	WB1				
LW x2, 44(x10)	IF2	D2	DIS2			EX1	EX2	WB2		
MUL x1, x2, x4	IF3	D3	DIS3					EX3	WB3	
SUB x8, x2, x6		IF4	D4	DIS4				EX4	WB4	

Register Status

x1	Mul/Div
x2	Load2
x3	
x4	
x5	
x6	updated
x7	
x8	ALU 1
x9	

Reservation Stations

	Busy	Op	Vj	Vk	Qj	Qk	A
Load 1	No						X
Load 2	Yes	Load					44 +Regs[x10]
ALU 1	Yes	SUB			Load2	Regs[x6]	
MUL/DIV	Yes	MUL			Load2	Regs[x4]	

Example of Reservation Station Utilization

LW x6, 32(x9)	IF1	D1	DIS1	EX1	EX2	WB1				
LW x2, 44(x10)	IF2	D2	DIS2			EX1	EX2	WB2		
MUL x1, x2, x4	IF3	D3	DIS3					EX3	WB3	
SUB x8, x2, x6		IF4	D4	DIS4				EX4	WB4	

Register Status

x1	Mul/Div
x2	updated
x3	
x4	
x5	
x6	
x7	
x8	ALU 1
x9	

Reservation Stations

	Busy	Op	Vj	Vk	Qj	Qk	A
Load 1	No						
Load 2	No						
ALU 1	Yes	SUB			Regs[x2]	Regs[x6]	
MUL/DIV	Yes	MUL			Load2	Regs[x4]	

Example of Reservation Station Utilization

LW x6, 32(x9)	IF1	D1	DIS1	EX1	EX2	WB1				
LW x2, 44(x10)	IF2	D2	DIS2			EX1	EX2	WB2		
MUL x1, x2, x4	IF3	D3	DIS3					EX3	WB3	
SUB x8, x2, x6		IF4	D4	DIS4				EX4	WB4	

Register Status

x1	
x2	
x3	
x4	
x5	
x6	
x7	
x8	
x9	

Reservation Stations

	Busy	Op	Vj	Vk	Qj	Qk	A
Load 1	No						
Load 2	No						
ALU 1	No						
MUL/DIV	No						

Example 2 of Reservation Station Utilization

`LW x6, 32(x9)`										
`LW x2, 44(x10)`										
`MUL x1, x2, x4`										
`SUB x8, x2, x6`										
`DIV x1, x1, x6`										
`Add x6, x8, x2`										

Register Status

x1	
x2	
x3	
x4	
x5	
x6	
x7	
x8	
x9	

Reservation Stations

	Busy	Op	Vj	Vk	Qj	Qk	A
Load 1							
Load 2							
ALU 1							
ALU 2							
MUL/DIV 1							
MUL/DIV 2							