

# Modern Processor Design

Hung-Wei Tseng

# Outline

- Pipelined Processor
- Pipeline Hazards
  - Structural Hazards
  - Control Hazards
  - Data Hazards
- Dynamic Branch Predictions

# Recap: Why adding a sort makes it faster

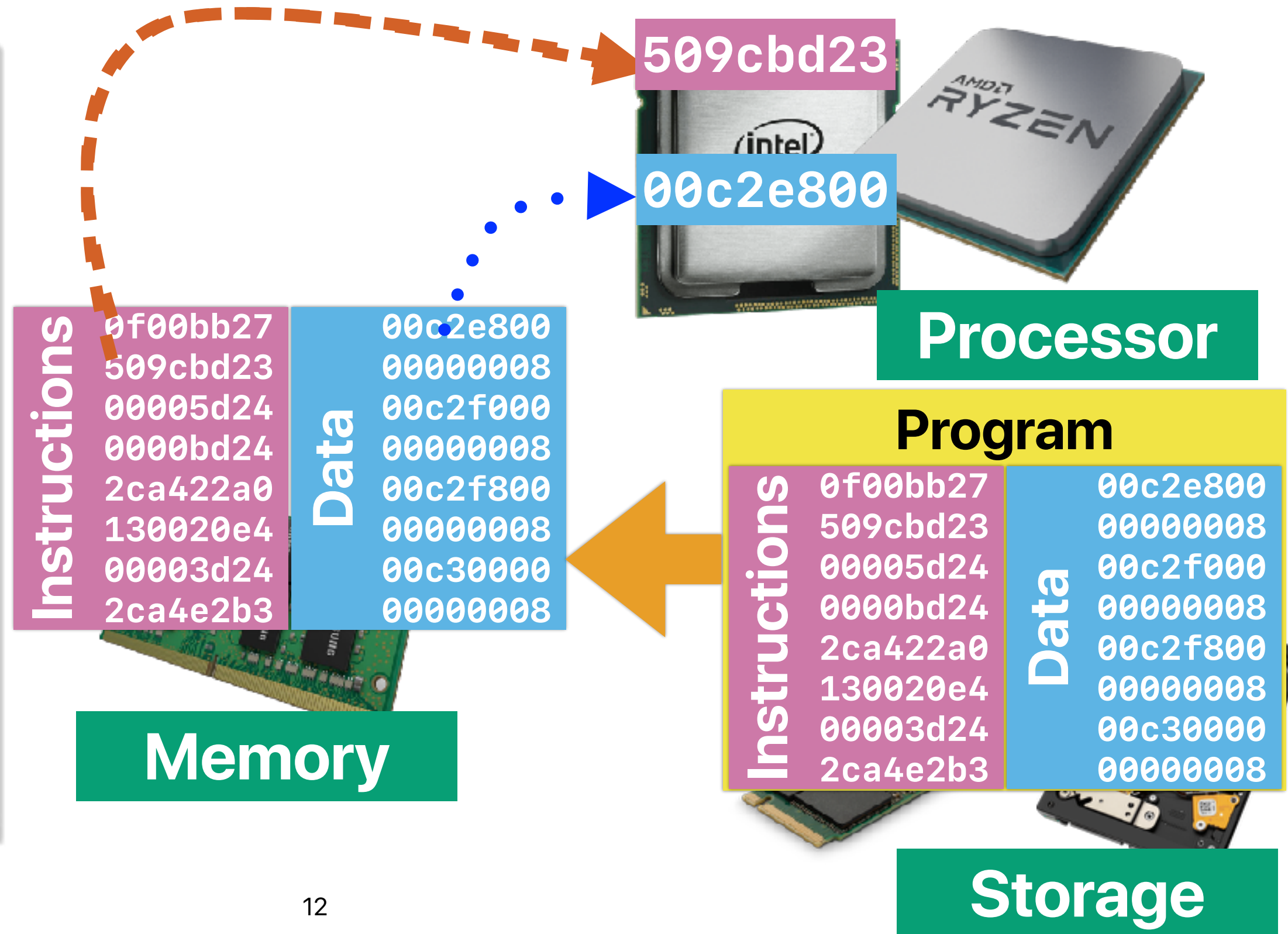
- Why the sorting the array speed up the code despite the increased instruction count?

```
if(option)
    std::sort(data, data + arraySize);

for (unsigned i = 0; i < 100000; ++i) {
    int threshold = std::rand();
    for (unsigned i = 0; i < arraySize; ++i) {
        if (data[i] >= threshold)
            sum ++;
    }
}
```

# Basic Processor Design

# von Neumann Architecture



# Recap: Microprocessor — a collection of functional units



Instructions

Instruction Set Architecture

Logical  
operations

Simple  
Arithmetic  
Operations  
(Add/Sub)

Complex  
Arithmetic  
Operations  
(Mul/div)

Branch/  
Jump

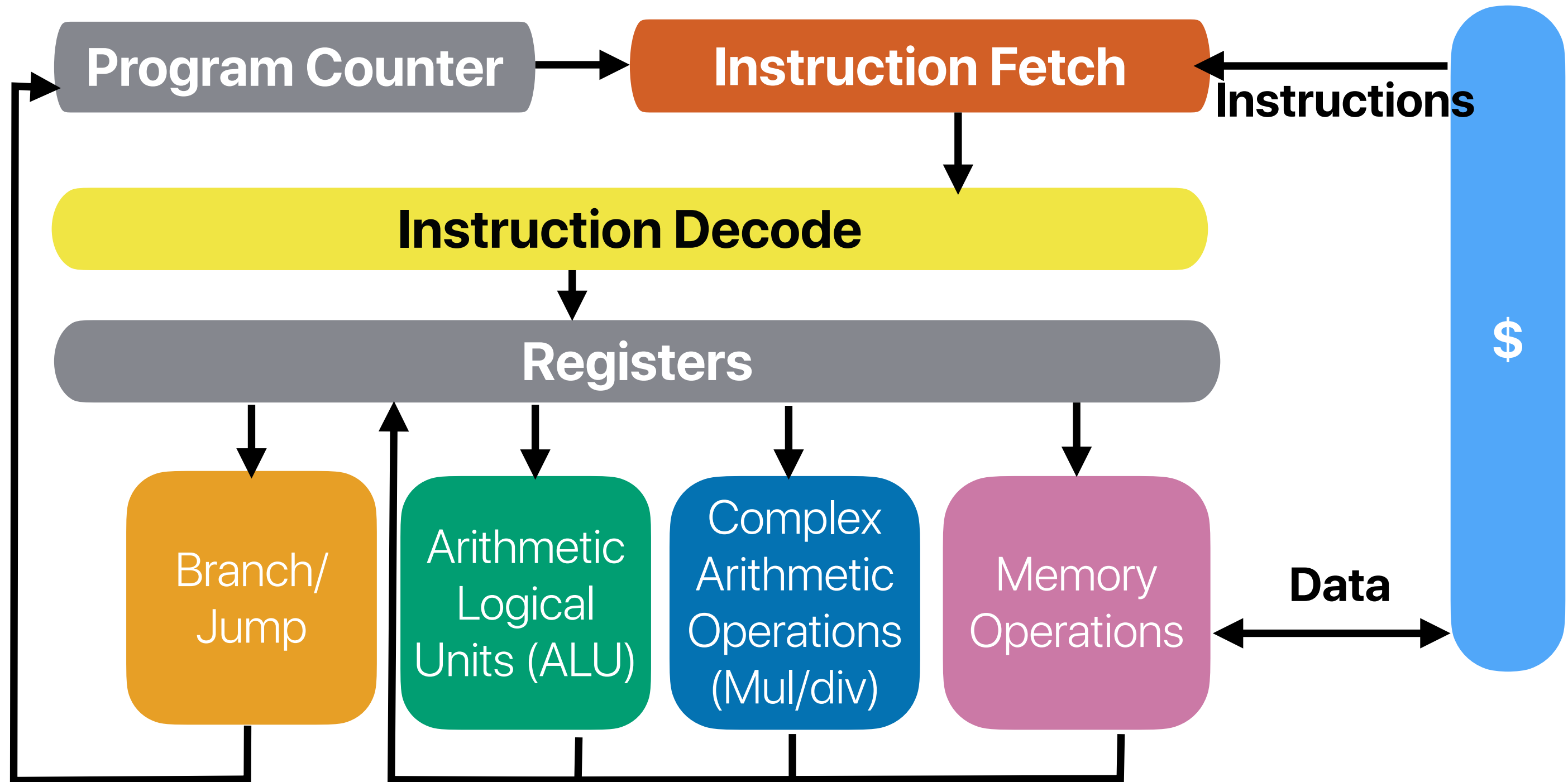
Memory  
Operations

Processor

# The “life” of an instruction

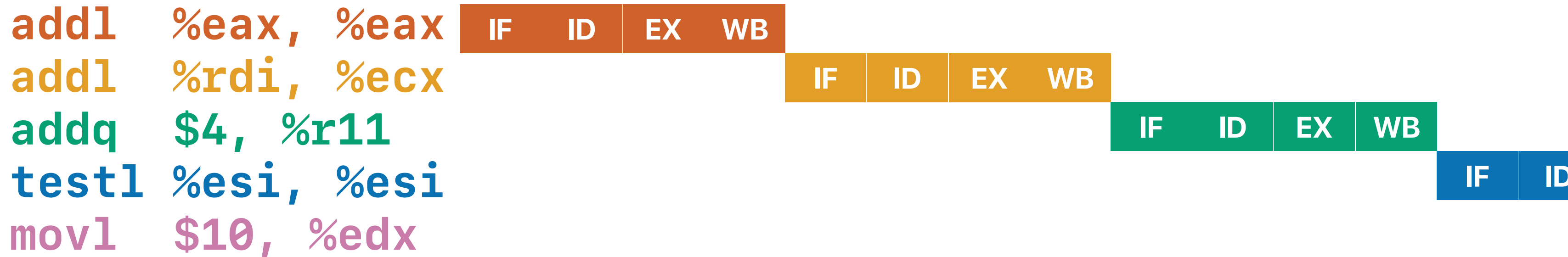
- Instruction Fetch (**IF**) — fetch the instruction from memory
- Instruction Decode (**ID**)
  - Decode the instruction for the desired operation and operands
  - Reading source register values
- Execution (**EX**)
  - ALU instructions: Perform ALU operations
  - Conditional Branch: Determine the branch outcome (taken/not taken)
  - Memory instructions: Determine the effective address for data memory access
- Data Memory Access (**MEM**) — Read/write memory
- Write Back (**WB**) — Present ALU result/read value in the target register
- Update PC
  - If the branch is taken — set to the branch target address
  - Otherwise — advance to the next instruction — current PC + 4

# Functional Units of a Microprocessor





# Simple implementation w/o branch



# Pipelining

 x83  x05  x05  x04

004861790

 163



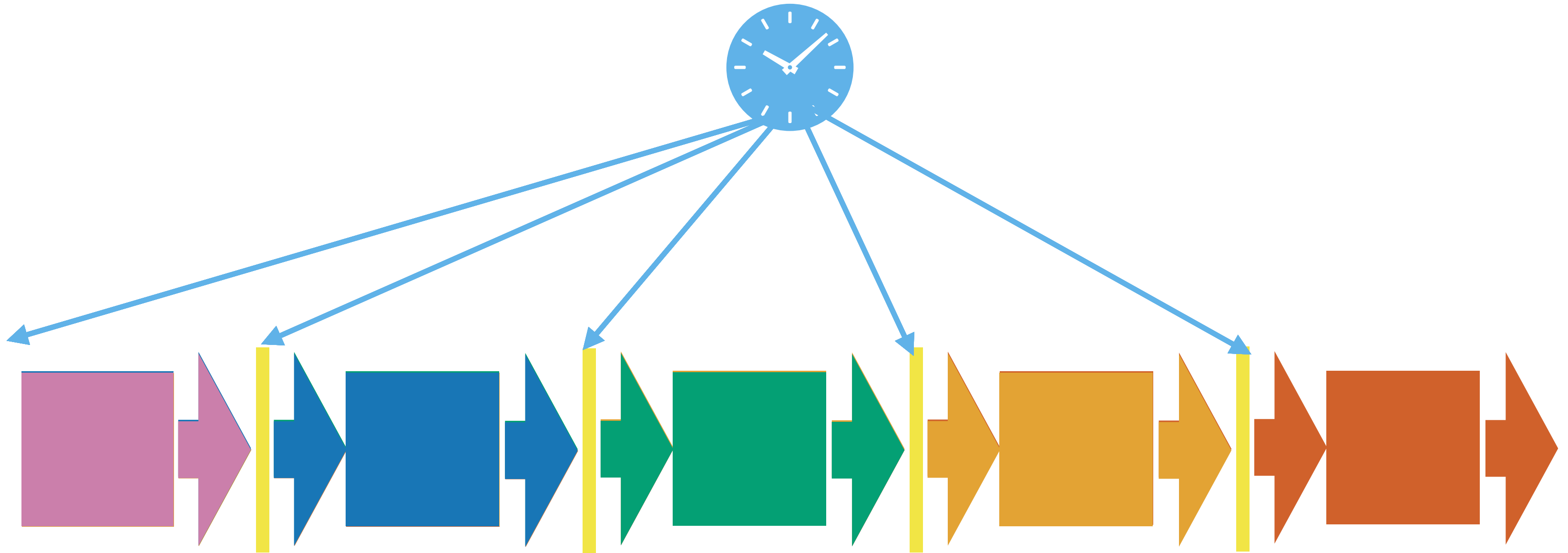
 41



# Pipelining

- Different parts of the processor works on different instructions simultaneously
- A processor is now working on multiple instructions from the same program (though on different stages) simultaneously.
  - **ILP: Instruction-level parallelism**
- A **clock** signal controls and synchronize the beginning and the end of each part of the work
- A **pipeline register** between different parts of the processor to keep intermediate results necessary for the upcoming work

# Pipelining

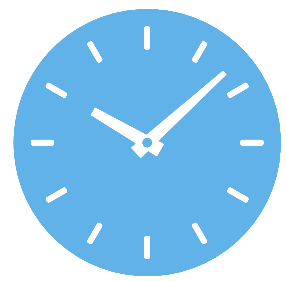




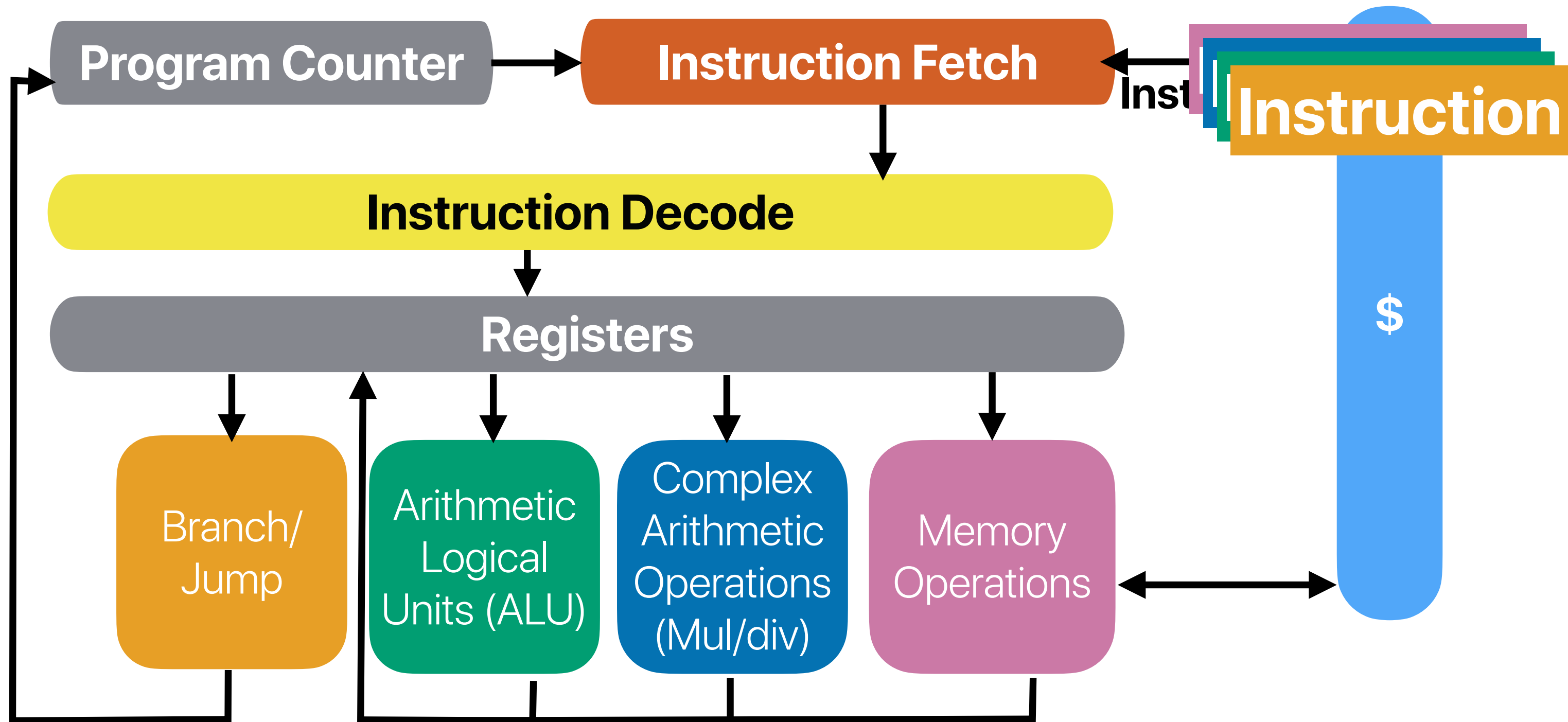
# Pipelining





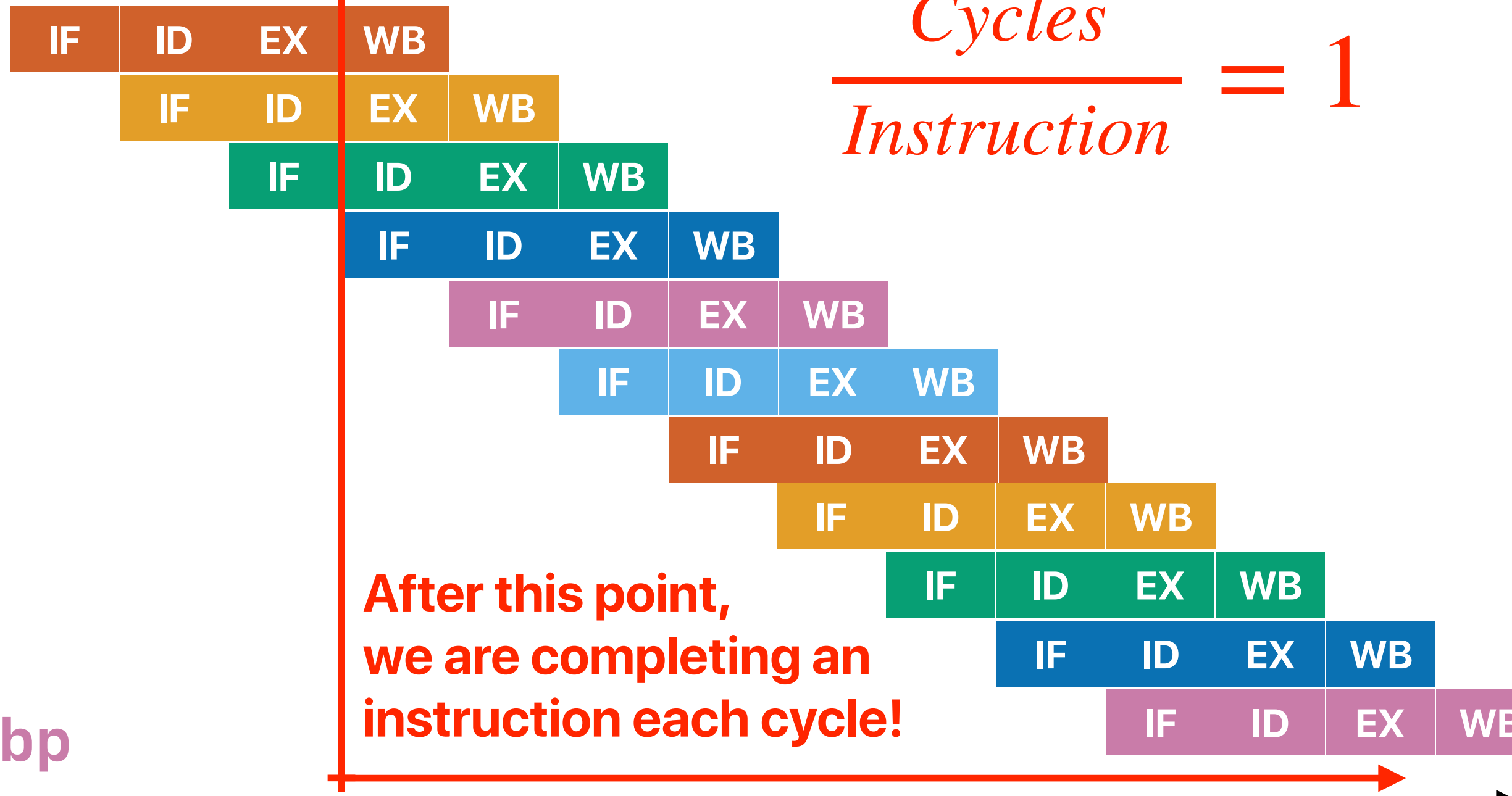


# "Pipeline" the processor!



# Pipelining

```
addl    %eax, %eax
addl    %rdi, %ecx
addq    $4, %r11
testl   %esi, %esi
movl    $10, %edx
pushq   %r12
pushq   %rbp
pushq   %rbx
subq    $8, %rsp
addl    %rsi, %rdi
movslq  %eax, %rbp
```



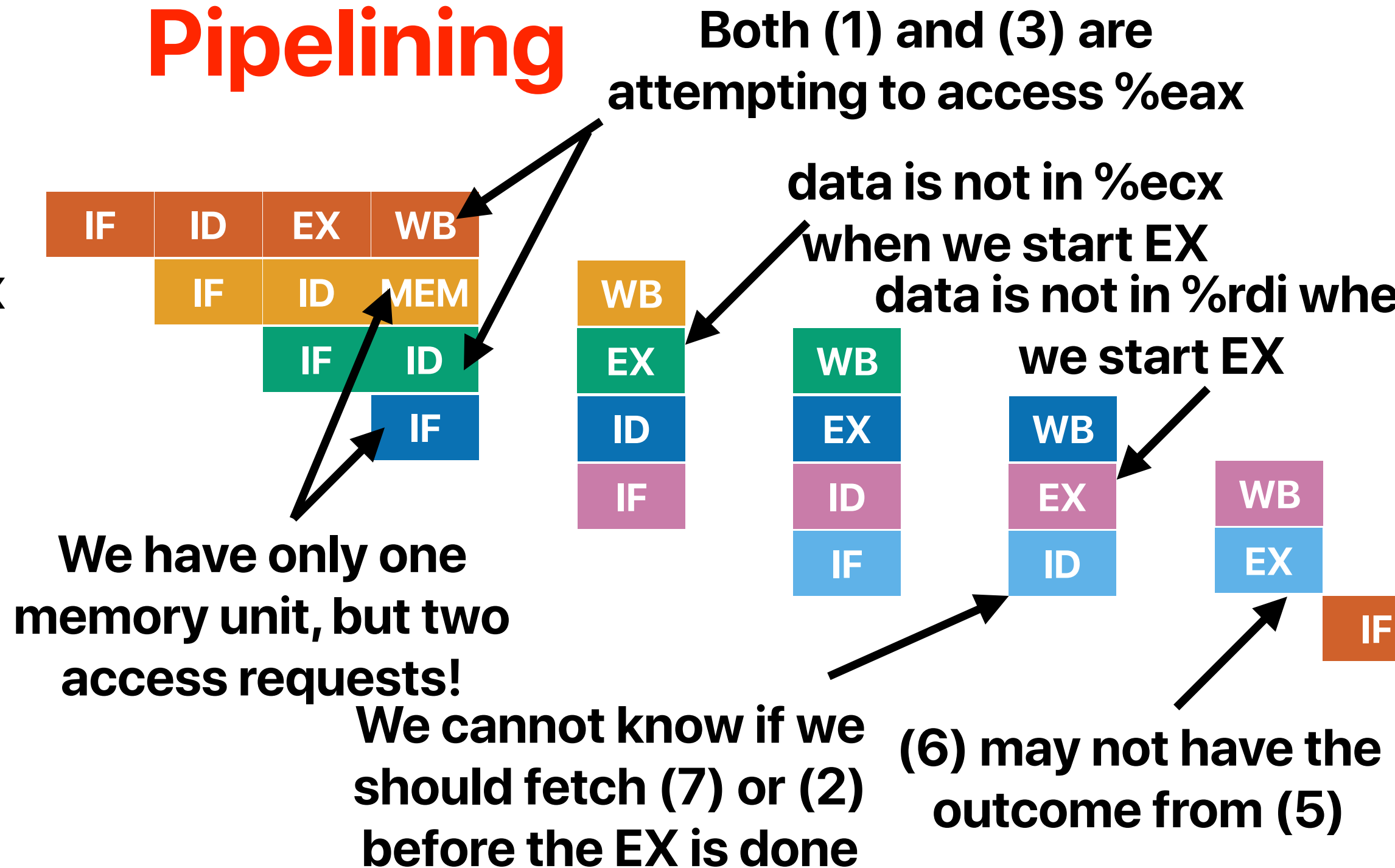
$$\frac{\text{Cycles}}{\text{Instruction}} = 1$$

After this point,  
we are completing an  
instruction each cycle!



# Pipelining

```
① xorl %eax, %eax
② movl (%rdi), %ecx
③ addl %ecx, %eax
④ addq $4, %rdi
⑤ cmpq %rdx, %rdi
⑥ jne .L3
⑦ ret
```



# Pipeline hazards

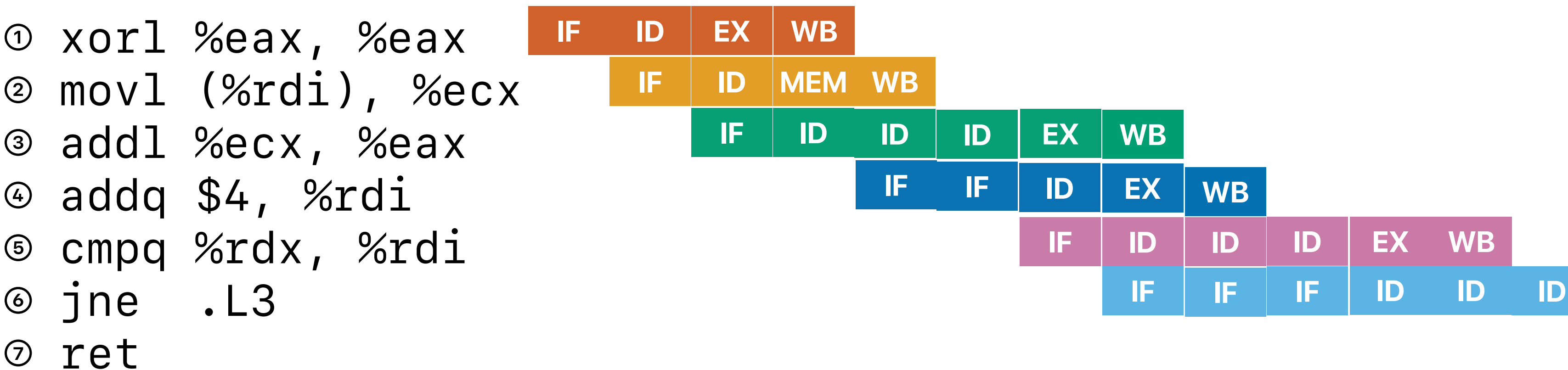
# Three types of pipeline hazards

- Structural hazards — resource conflicts cannot support simultaneous execution of instructions in the pipeline
- Control hazards — the PC can be changed by an instruction in the pipeline
- Data hazards — an instruction depending on a the result that's not yet generated or propagated when the instruction needs that

**Stall — the universal solution to  
pipeline hazards**

# Stall whenever we have a hazard

- Stall: the hardware allows the earlier instruction to proceed, all later instructions stay at the same stage



**Slow! — 5 additional cycles**

# Structural Hazards

# Dealing with the conflicts between ID/WB

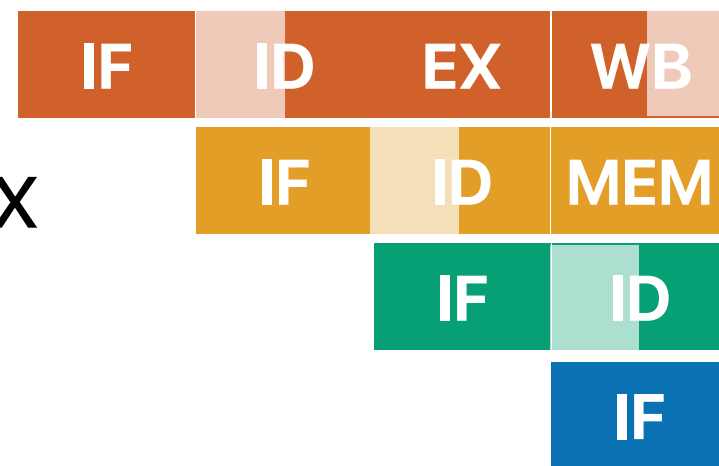
- The same register cannot be read/written at the same cycle
- Better solution: write early, read late
  - Writes occur at the clock edge and complete long enough before the end of the clock cycle.
  - This leaves enough time for outputs to settle for reads
  - The revised register file is the default one from now!



# How to with the conflicts between MEM and IF?

- The memory unit can only accept/perform one request each cycle

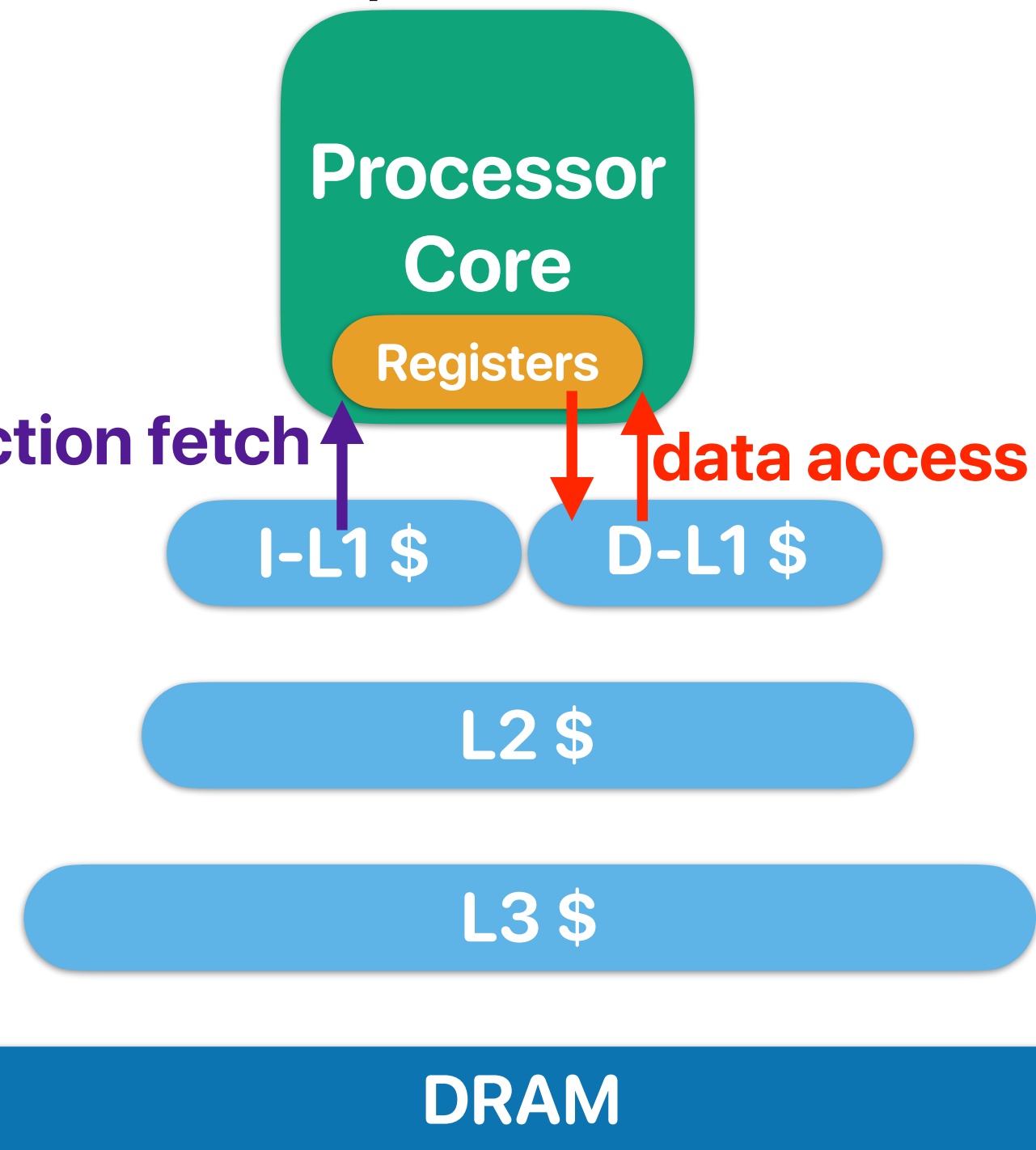
① `xorl %eax, %eax`  
② `movl (%rdi), %ecx`  
③ `addl %ecx, %eax`  
④ `addq $4, %rdi`



instruction fetch

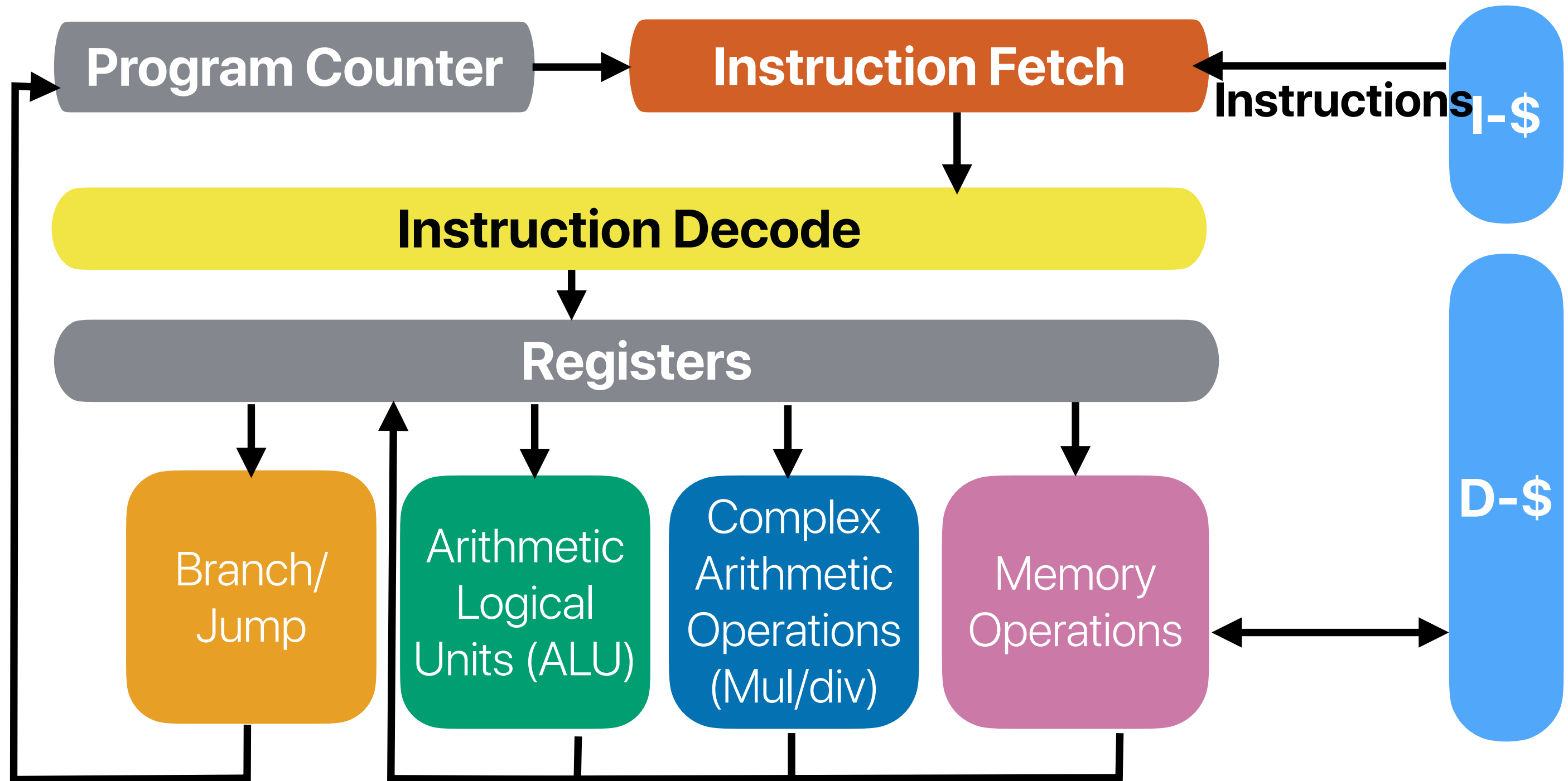
data access

**"Split L1" cache!**



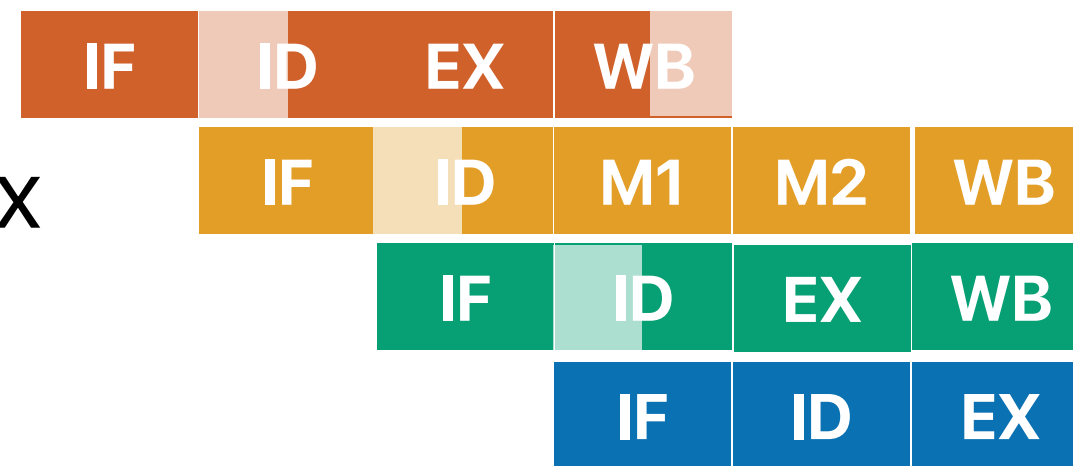


# Split L1-\$

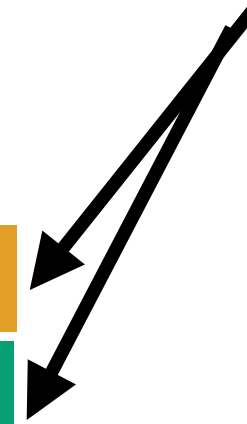


# What if the memory instruction needs more time?

① `xorl %eax, %eax`  
② `movl (%rdi), %ecx`  
③ `addl %ecx, %eax`  
④ `addq $4, %rdi`

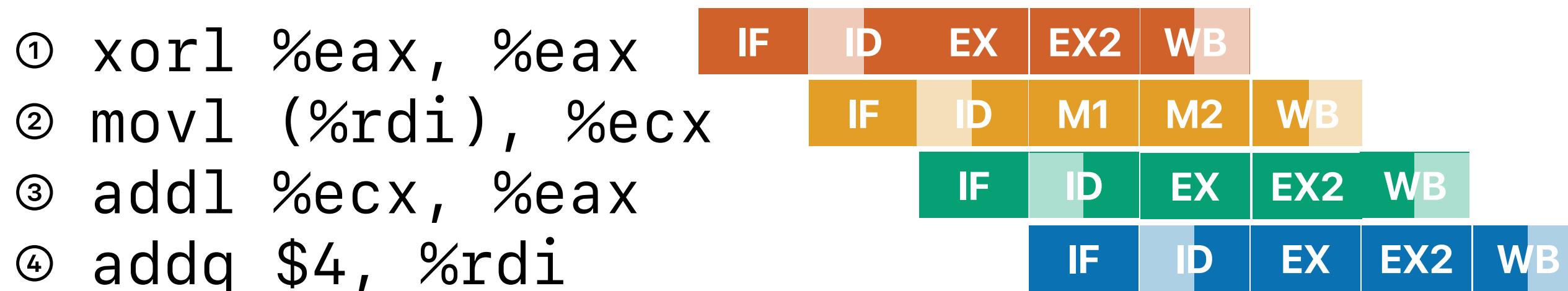


Both (2) and (3) are attempting to "WB"



# What if the memory instruction needs more time?

- Every instruction needs to go through exactly the same number of stages



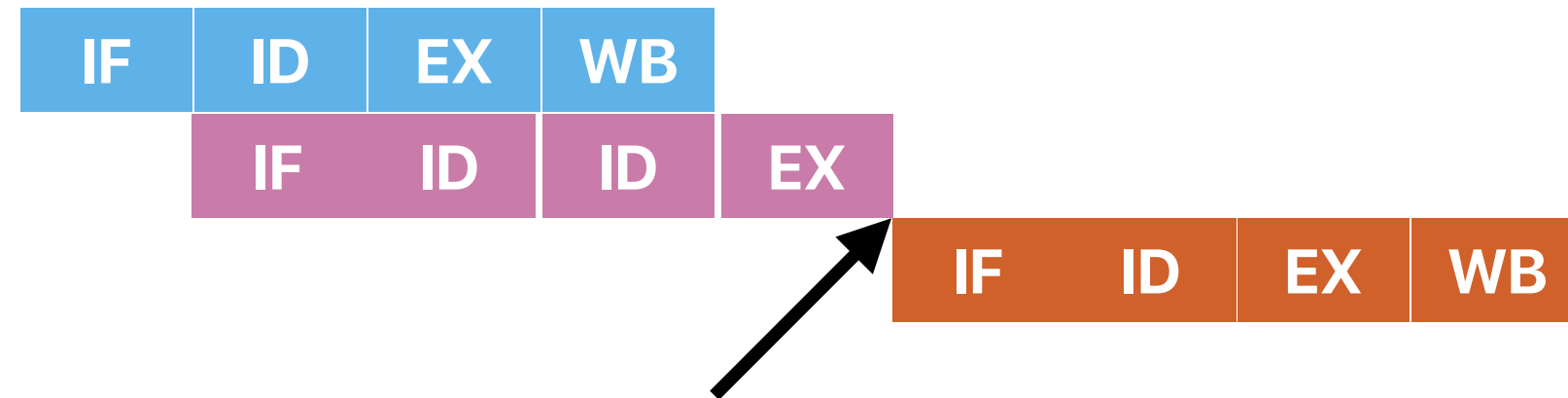
# Structural Hazards

- Stall can address the issue — but slow
- Improve the pipeline unit design to allow parallel execution
  - Write-first, read later register files
  - Split L1-Cache
  - Force all instructions go through exactly the same number of stages

# Control Hazards

# Control Hazard

① `cmpq %rdx, %rdi`  
② `jne .L3`  
③ `ret`



**We cannot know if we  
should fetch (7) or (2)  
before the EX is done**

# How does the code look like?

```
for (j = 0; j < reps; ++j) {  
    for (unsigned i = 0; i < size; ++i) {  
        if (data[i] >= threshold)  
            sum++;  
    }  
}
```

**We skip the following code block if  $\text{data}[i] < \text{threshold}$**

**We use "backward" branches (taking if going back) to implement loops**

```
loop0:  
.LFB0:  
.cfi_startproc  
endbr64  
pushq %rbp  
.cfi_def_cfa_offset 16  
.cfi_offset 6, -16  
movq %rsp, %rbp  
.cfi_def_cfa_register 6  
movq %rdi, -24(%rbp)  
movl %esi, -28(%rbp)  
movl %edx, -32(%rbp)  
movl %ecx, -36(%rbp)  
movl $0, -8(%rbp)  
movl $0, -12(%rbp)  
jmp .L2
```

```
.L6:  
    movl $0, -4(%rbp)  
    jmp .L3  
.L5:  
    movl -4(%rbp), %eax  
    leaq 0(,%rax,4), %rdx  
    movq -24(%rbp), %rax  
    addq %rdx, %rax  
    movl (%rax), %eax  
    cmpl %eax, -32(%rbp)  
    jg .L4  
    addl $1, -8(%rbp)  
.L4:  
    addl $1, -4(%rbp)  
.L3:  
    movl -28(%rbp), %eax
```

```
    cmpl %eax, -4(%rbp)  
    jb .L5  
    addl $1, -12(%rbp)  
.L2:  
    movl -12(%rbp), %eax  
    cmpl -36(%rbp), %eax  
    jl .L6  
    movl -8(%rbp), %eax  
    popq %rbp  
.cfi_def_cfa 7, 8  
ret
```

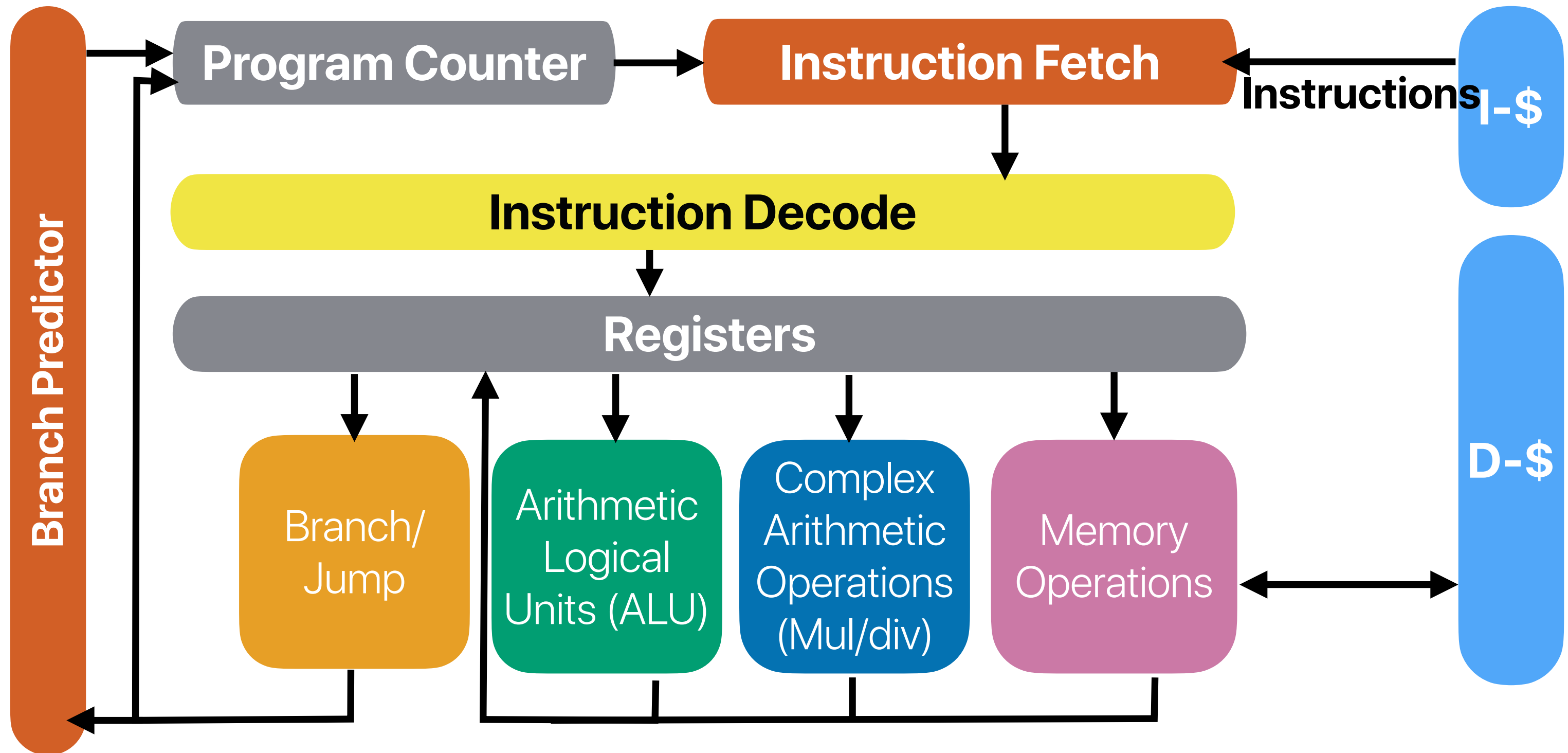
# Why can't we proceed without stalls/no-ops?

- How many of the following statements are true regarding why we have to stall for each branch in the current pipeline processor
    - ① ☒ The target address when branch is taken is not available for instruction fetch stage of the next cycle **You need a cheatsheet for that — branch target buffer**
    - ② The target address when branch is not-taken is not available for instruction fetch stage of the next cycle
    - ③ ☒ The branch outcome cannot be decided until the comparison result of ALU is not out **You need to predict that — history/states**
    - ④ The next instruction needs the branch instruction to write back its result
- A. 0  
B. 1  
**C. 2**  
D. 3  
E. 4

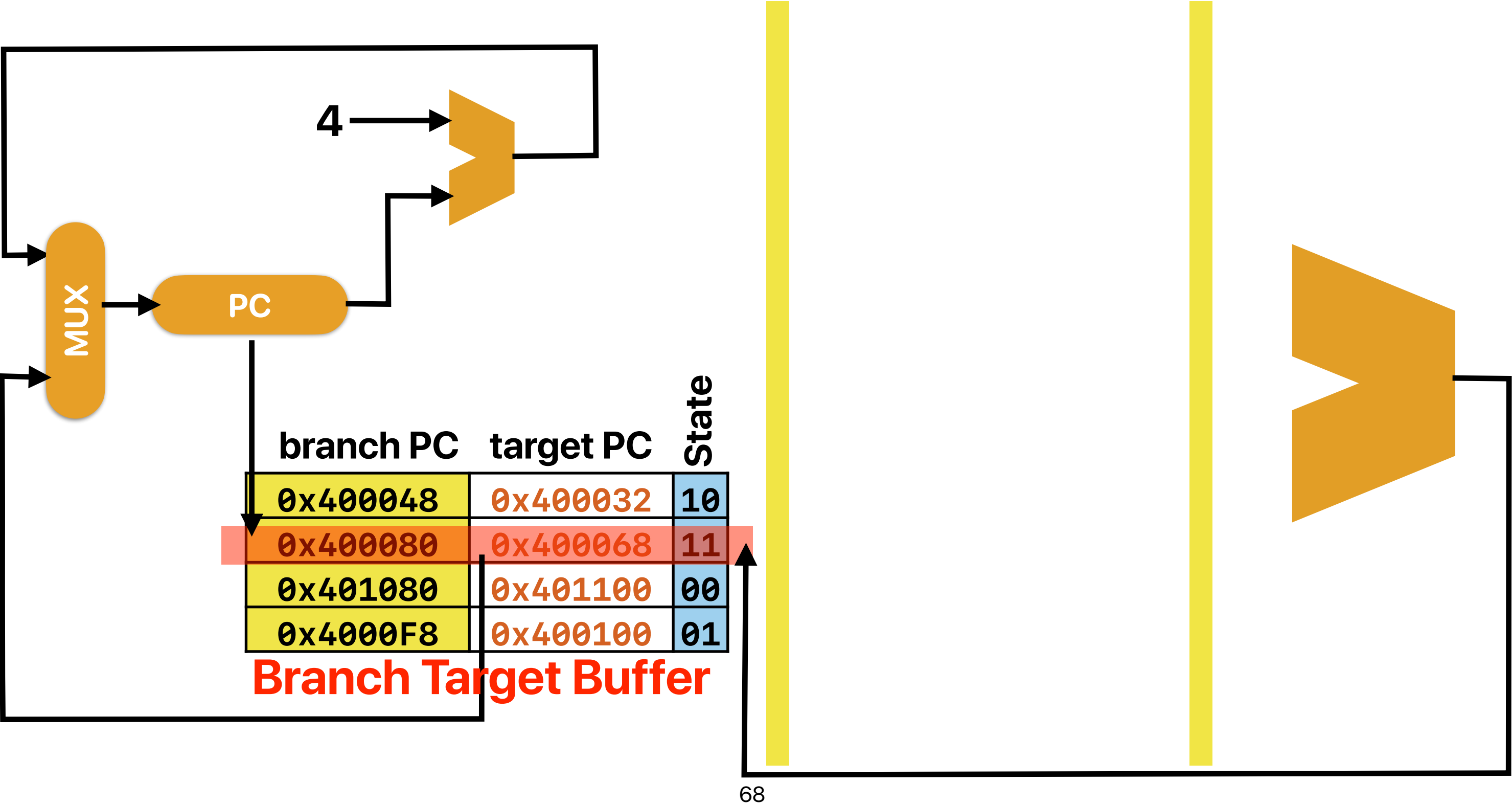


# Dynamic Branch Prediction

# Microprocessor with a "branch predictor"



# Detail of a basic dynamic branch predictor

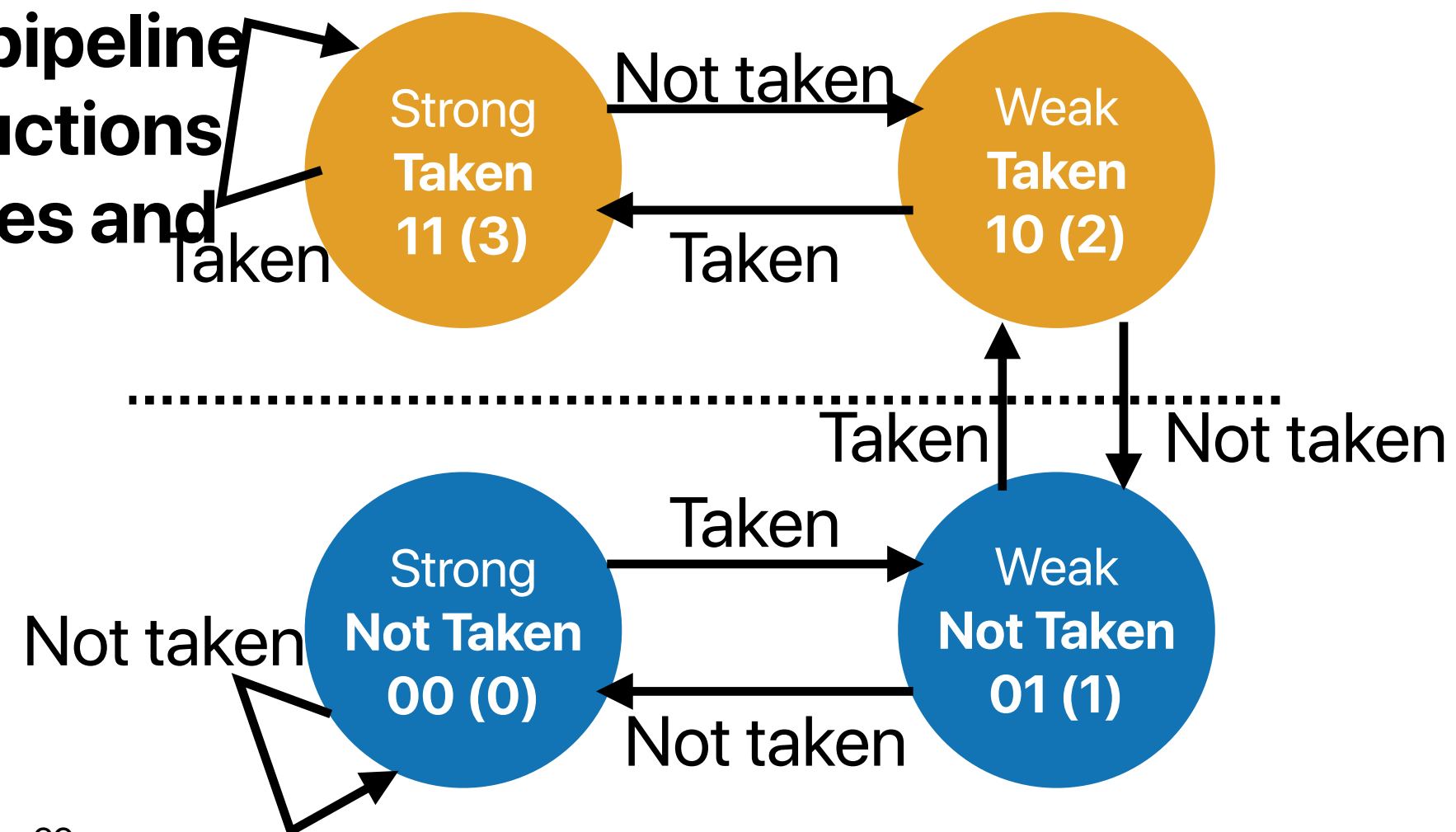


# 2-bit/Bimodal local predictor

- Local predictor — every branch instruction has its own state
- 2-bit — each state is described using 2 bits
- Change the state based on **actual** outcome
- If we guess right — no penalty
- **If we guess wrong — flush (clear pipeline registers) for mis-predicted instructions that are currently in IF and ID stages and reset the PC**

branch PC	target PC	State
0x400048	0x400032	10
0x400080	0x400068	11
0x401080	0x401100	00
0x4000F8	0x400100	01

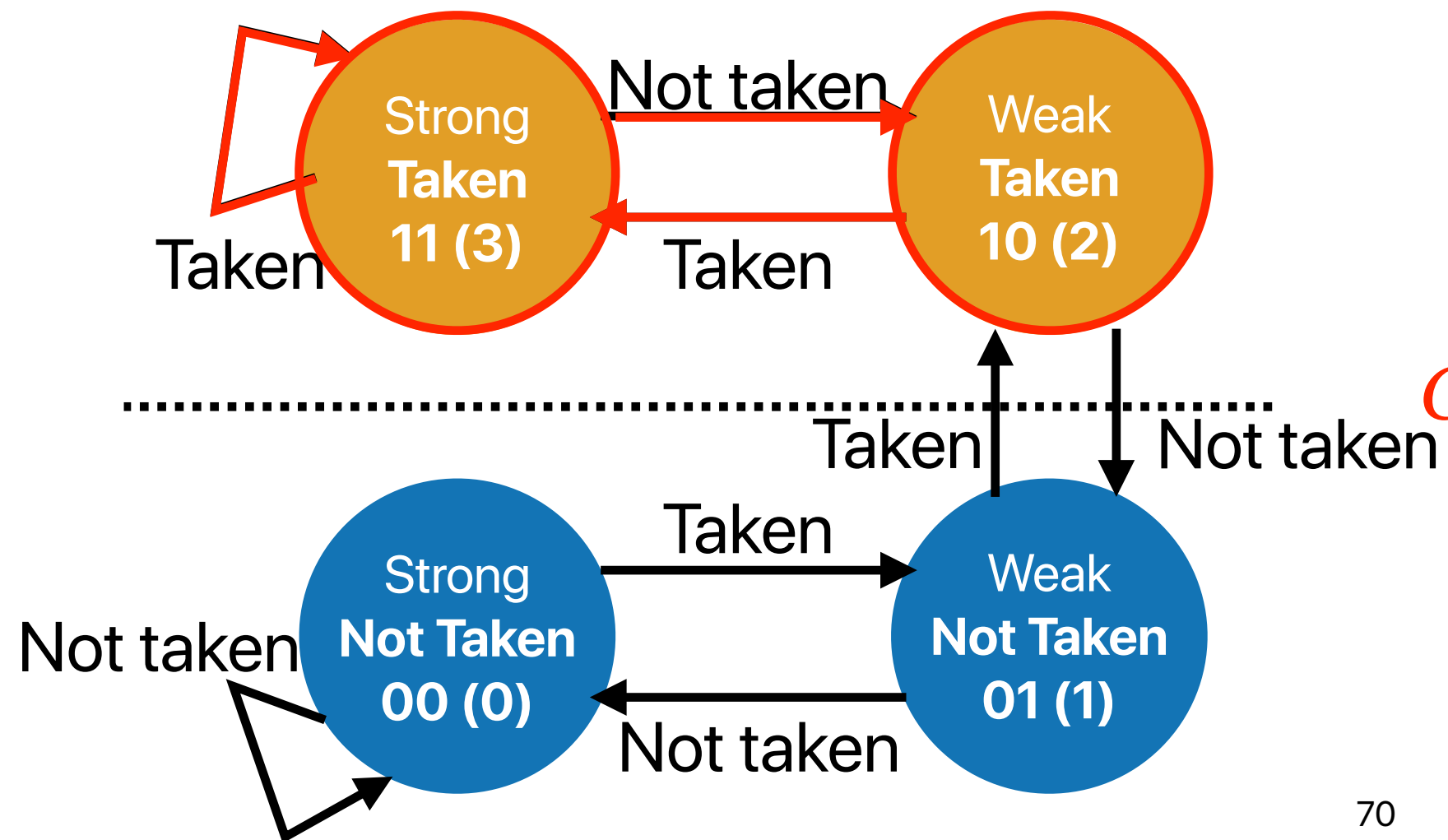
Predict Taken



# 2-bit local predictor

```
i = 0;  
do {  
    sum += a[i];  
} while(++i < 10);
```

i	state	predict	actual
1	10	T	T
2	11	T	T
3	11	T	T
4-9	11	T	T
10	11	T	NT



**90% accuracy!**

$$CPI_{average} = 1 + 20\% \times 10\% \times 2 = 1.04$$

# Two-level global predictor

Marius Evers, Sanjay J. Patel, Robert S. Chappell, and Yale N. Patt. 1998. An analysis of correlation and predictability: what makes two-level branch predictors work. In Proceedings of the 25th annual international symposium on Computer architecture (ISCA '98).

# 2-bit local predictor

- What's the overall branch prediction (include both branches) accuracy for this nested for loop?

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch Y
```

(assume all states started with 00)

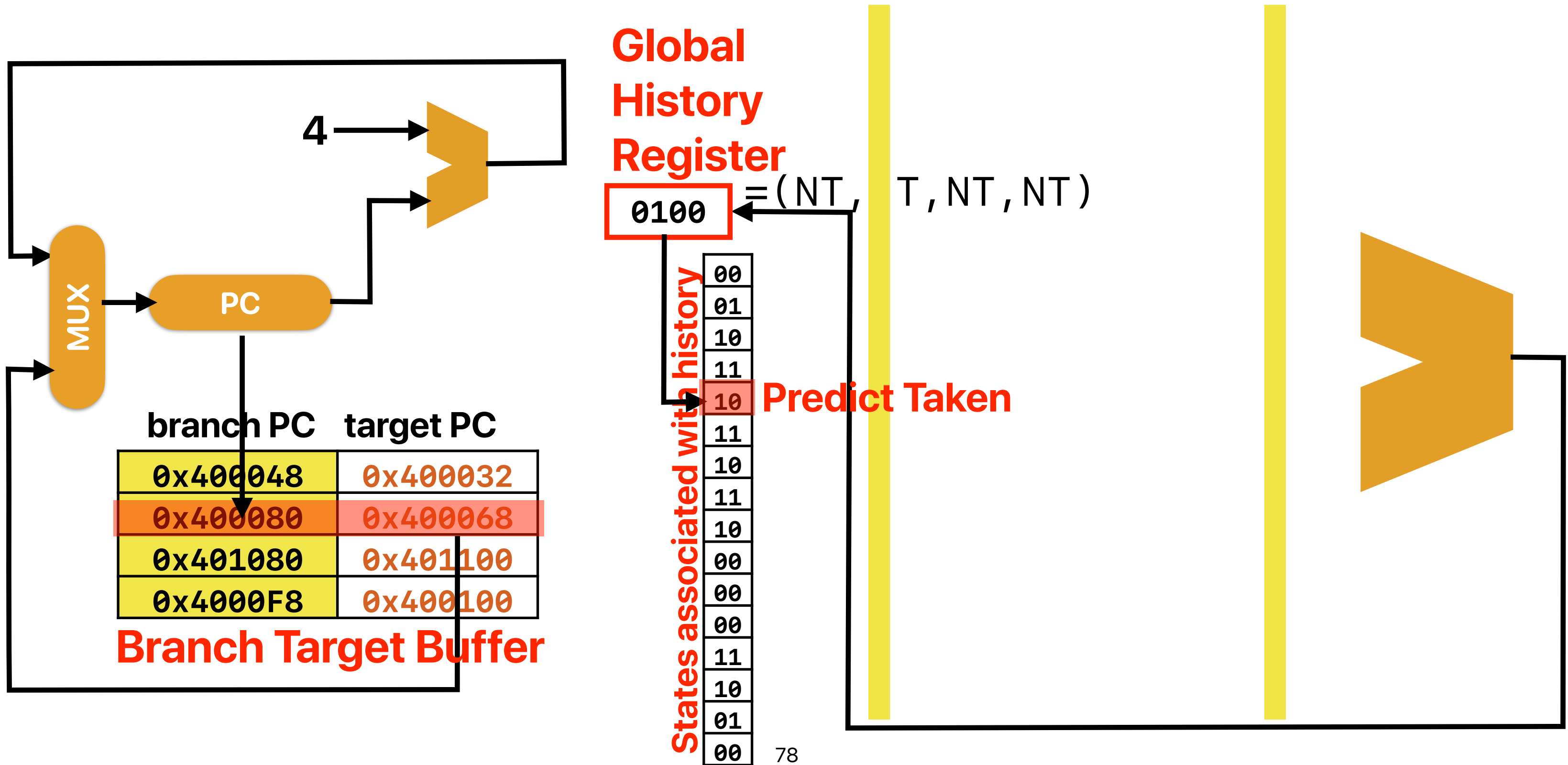
- A. ~25%
- B. ~33%
- C. ~50%
- D. ~67%
- E. ~75%**

**This pattern  
repeats all the time!**

For branch Y, almost 100%,  
For branch X, only 50%

i	branch?	state	prediction	actual
0	X	00	NT	T
0	Y	00	NT	T
1	X	01	NT	NT
1	Y	01	NT	T
2	X	00	NT	T
2	Y	10	T	T
3	X	01	NT	NT
3	Y	11	T	T
4	X	00	NT	T
4	Y	11	T	T
5	X	01	NT	NT
5	Y	11	T	T
6	X	00	NT	T
6	Y	11	T	T

# Global history (GH) predictor





# Performance of GH predictor

```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100) // Branch Y
```

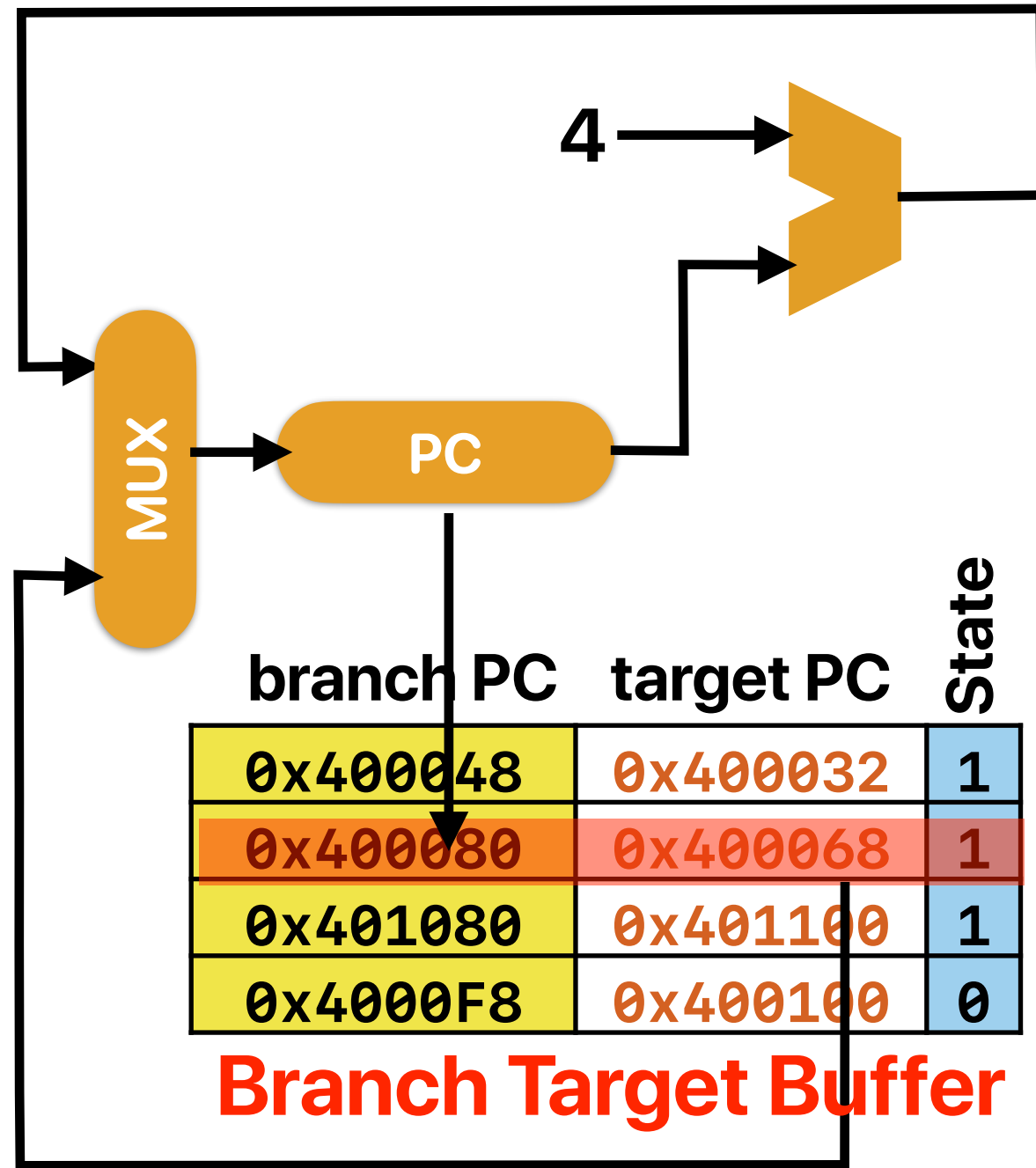
i	branch?	GHR	state	prediction	actual
0	X	000	00	NT	T
0	Y	001	00	NT	T
1	X	011	00	NT	NT
1	Y	110	00	NT	T
2	X	101	00	NT	T
2	Y	011	00	NT	T
3	X	111	00	NT	NT
3	Y	110	01	NT	T
4	X	101	01	NT	T
4	Y	011	01	NT	T
5	X	111	00	NT	NT
5	Y	110	10	T	T
6	X	101	10	T	T
6	Y	011	10	T	T
7	X	111	00	NT	NT
7	Y	110	11	T	T
8	X	101	11	T	T
8	Y	011	11	T	T
9	X	111	00	NT	NT
9	Y	110	11	T	T
10	X	101	11	T	T
10	Y	011	11	T	T

Near perfect after this



# Hybrid predictors

# Tournament Predictor



**Global  
History  
Register**

0100

States associated with history

00
01
10
11
10
11
10
11
10
11
10
00
00
00
00
00
11
10
01
00

**Local  
History  
Predictor**

branch PC local history

0x400048	1000
0x400080	0110
0x401080	1010
0x4000F8	0110

**Predict Taken**

States associated with history

00
01
10
11
10
11
10
11
10
11
10
00
00
00
00
00
11
10
01
00

# Tournament Predictor

- The state predicts “which predictor is better”
  - Local history
  - Global history
- The predicted predictor makes the prediction

# Perceptron

Jiménez, Daniel, and Calvin Lin. "Dynamic branch prediction with perceptrons." Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture. IEEE, 2001.

The following slides are excerpted from <https://www.jilp.org/cbp/Daniel-slides.PDF> by Daniel Jiménez

# Branch Prediction is Essentially an ML Problem

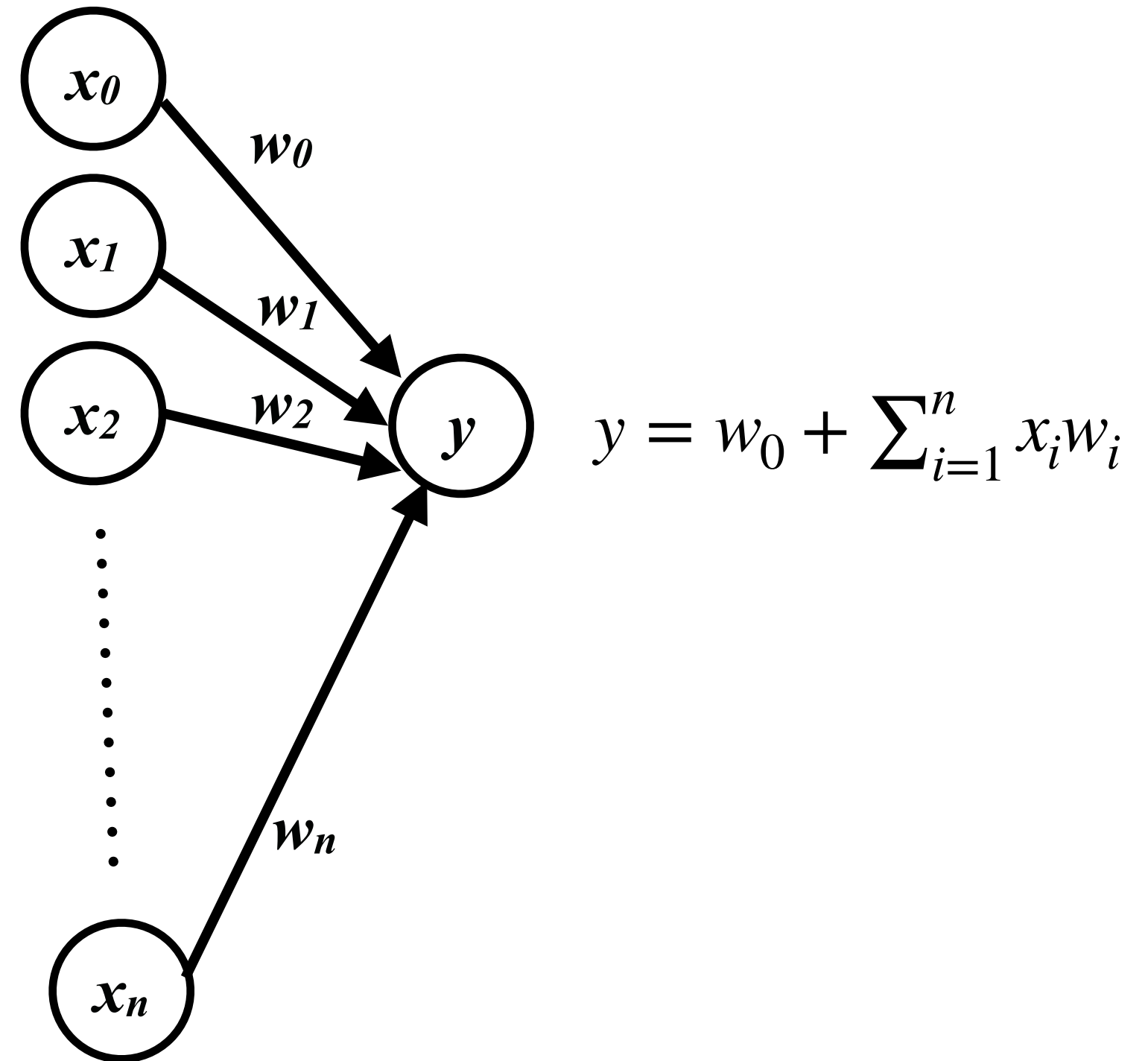
- The machine learns to predict conditional branches
- Artificial neural networks
  - Simple model of neural networks in brain cells
  - Learn to recognize and classify patterns

# Mapping Branch Prediction to NN

- The inputs to the perceptron are branch outcome histories
  - Just like in 2-level adaptive branch prediction
  - Can be global or local (per-branch) or both (alloyed)
  - Conceptually, branch outcomes are represented as
    - +1, for taken
    - -1, for not taken
- The output of the perceptron is
  - Non-negative, if the branch is predicted taken
  - Negative, if the branch is predicted not taken
- Ideally, each static branch is allocated its own perceptron

# Mapping Branch Prediction to NN (cont.)

- Inputs ( $x$ 's) are from branch history and are -1 or +1
- $n + 1$  small integer weights ( $w$ 's) learned by on-line training
- Output ( $y$ ) is dot product of  $x$ 's and  $w$ 's; predict taken if  $y = 0$
- Training finds correlations between history and outcome





# Training Algorithm

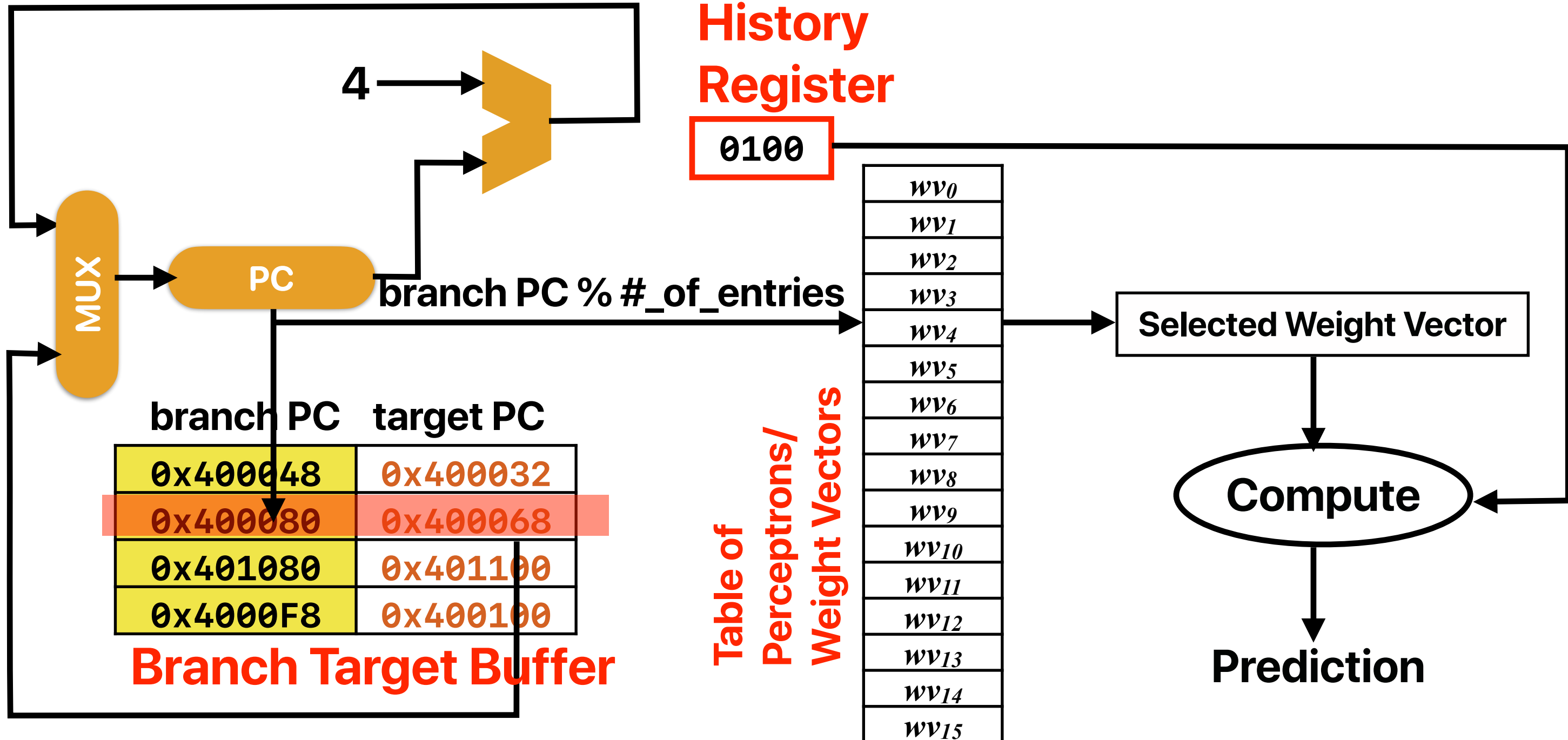
$x_{1..n}$  is the  $n$ -bit history register,  $x_0$  is 1.  
 $w_{0..n}$  is the weights vector.  
 $t$  is the Boolean branch outcome.  
 $\theta$  is the training threshold.

```
if  $|y| \leq \theta$  or  $((y \geq 0) \neq t)$  then
  for each  $0 \leq i \leq n$  in parallel
    if  $t = x_i$  then
       $w_i := w_i + 1$ 
    else
       $w_i := w_i - 1$ 
    end if
  end for
end if
```

# Predictor Organization

Global  
History  
Register

0100



# Branch predictors in processors

- The Intel Pentium MMX, Pentium II, and Pentium III have local branch predictors with a local 4-bit history and a local pattern history table with 16 entries for each conditional jump.
- Global branch prediction is used in Intel Pentium M, Core, Core 2, and Silvermont-based Atom processors.
- Tournament predictor is used in DEC Alpha, AMD Athlon processors
- The AMD Ryzen multi-core processor's Infinity Fabric and the Samsung Exynos processor include a perceptron based neural branch predictor.

# Computer Science & Engineering

142

つづく

