Modern Processor Design

Hung-Wei Tseng

Recap: Why adding a sort makes it faster

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

```cpp
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 ++;
    }
}
```
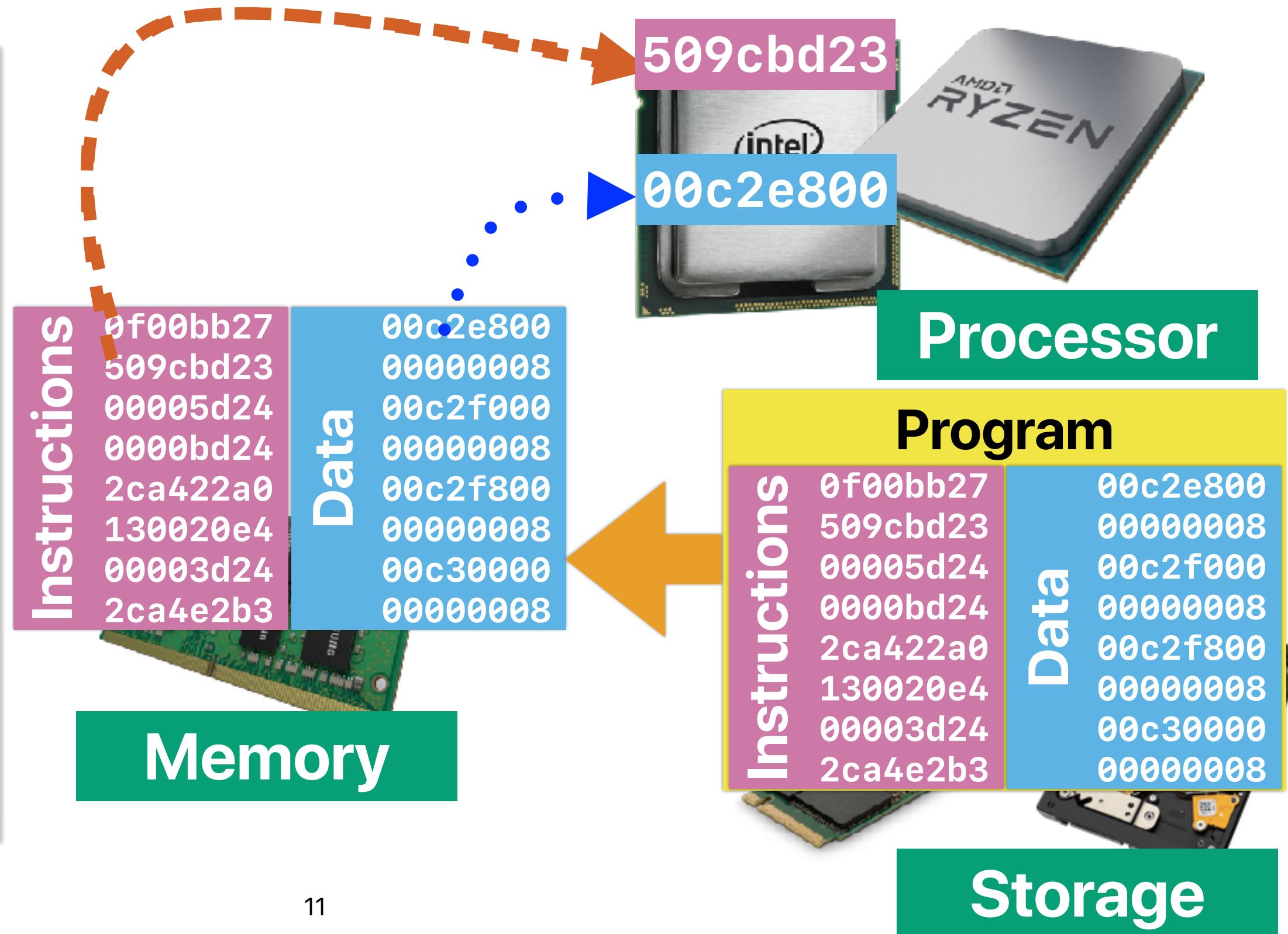
Outline

- Pipelined Processor

- Pipeline Hazards

  - Structural Hazards

  - Control Hazards

  - Data Hazards

Basic Processor Design

von Neumman Architecture

Processor

Program

Memory

Storage

509cbd23

00c2e800

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

Instructions	Data
0f00bb27	00c2e800
509cbd23	00000008
00005d24	00c2f000
0000bd24	00000008
2ca422a0	00c2f800
130020e4	00000008
00003d24	00c30000
2ca4e2b3	00000008

11

Recap: Microprocessor — a collection of functional units

Instructions

Instruction Set Architecture

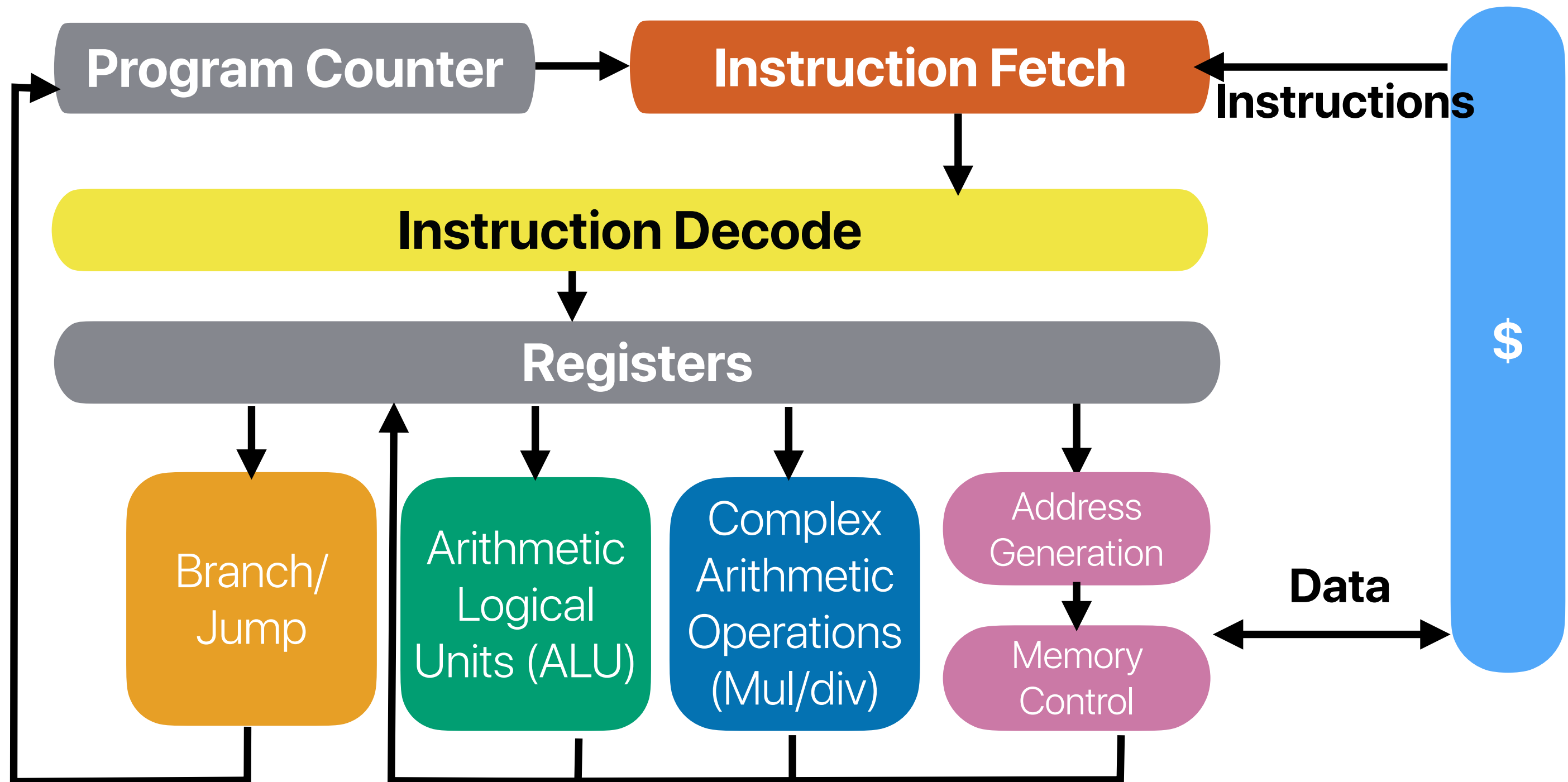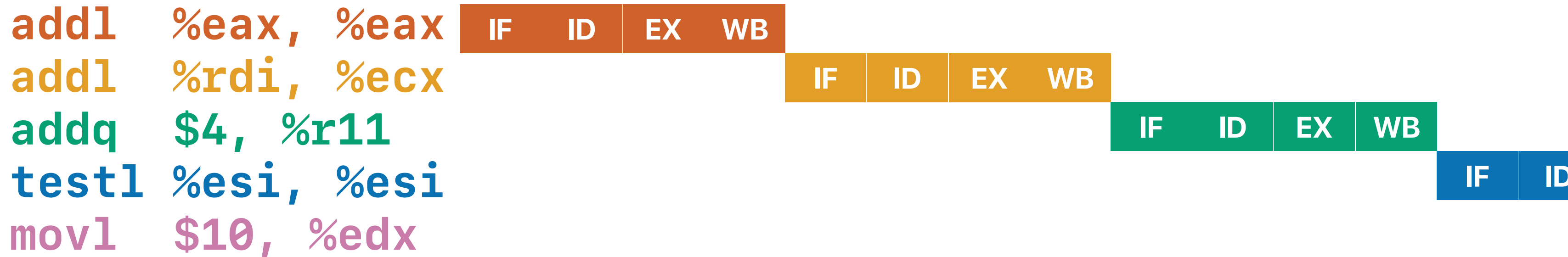| Logical operations | Simple Arithmetic Operations (Add/Sub) | Complex Arithmetic Operations (Mul/div) | Branch/ Jump | Memory Operations |

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

Program Counter → Instruction Fetch ← Instructions

Instruction Decode

Registers

Branch/Jump

Arithmetic Logical Units (ALU)

Complex Arithmetic Operations (Mul/div)

Address Generation

Memory Control

Data

$

Simple implementation w/o branch

```
addl   %eax, %eax
addl   %rdi, %ecx
addq   $4, %r11
testl  %esi, %esi
movl   $10, %edx
```


15

Pipelining

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

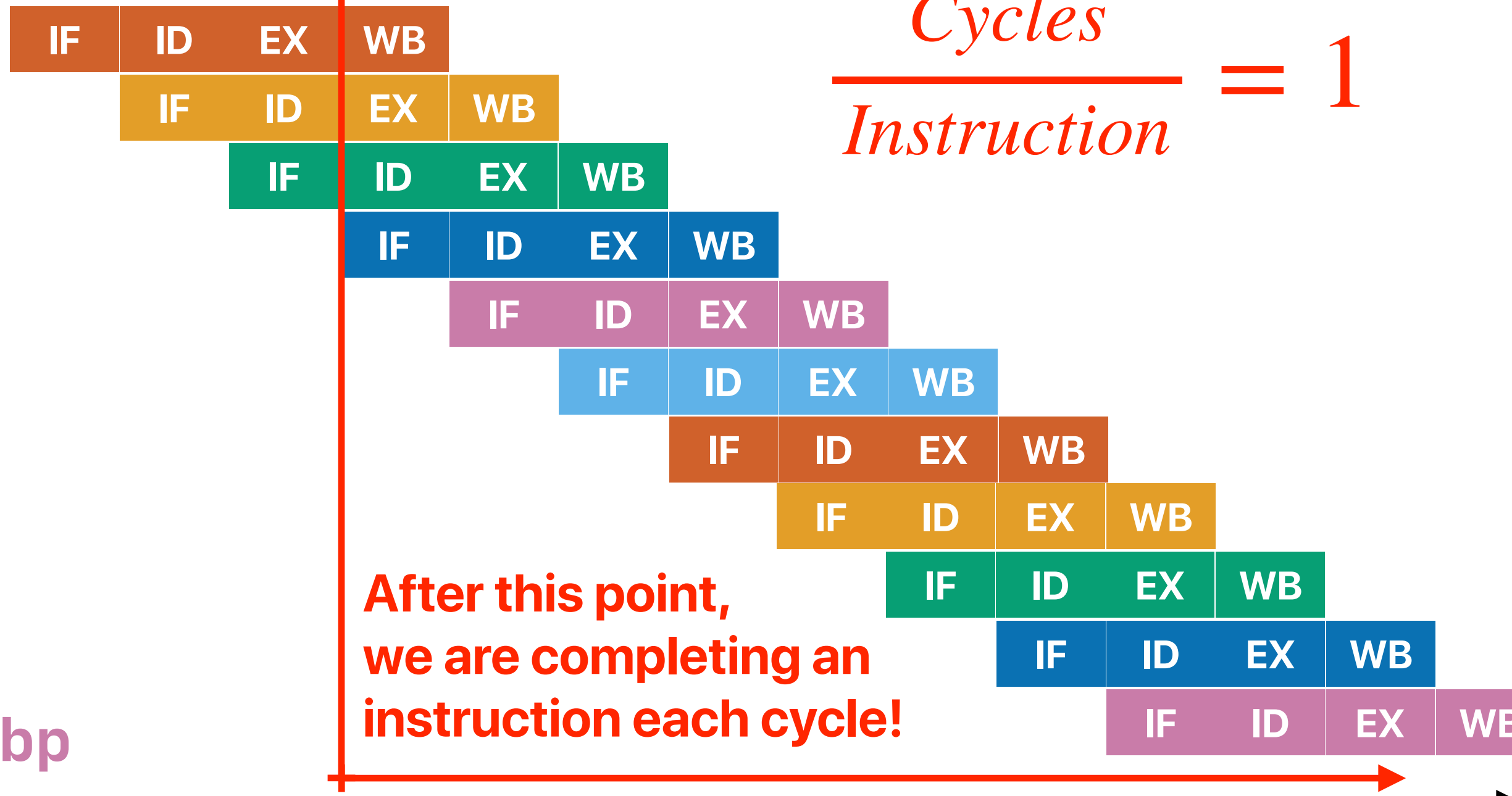# Pipelining
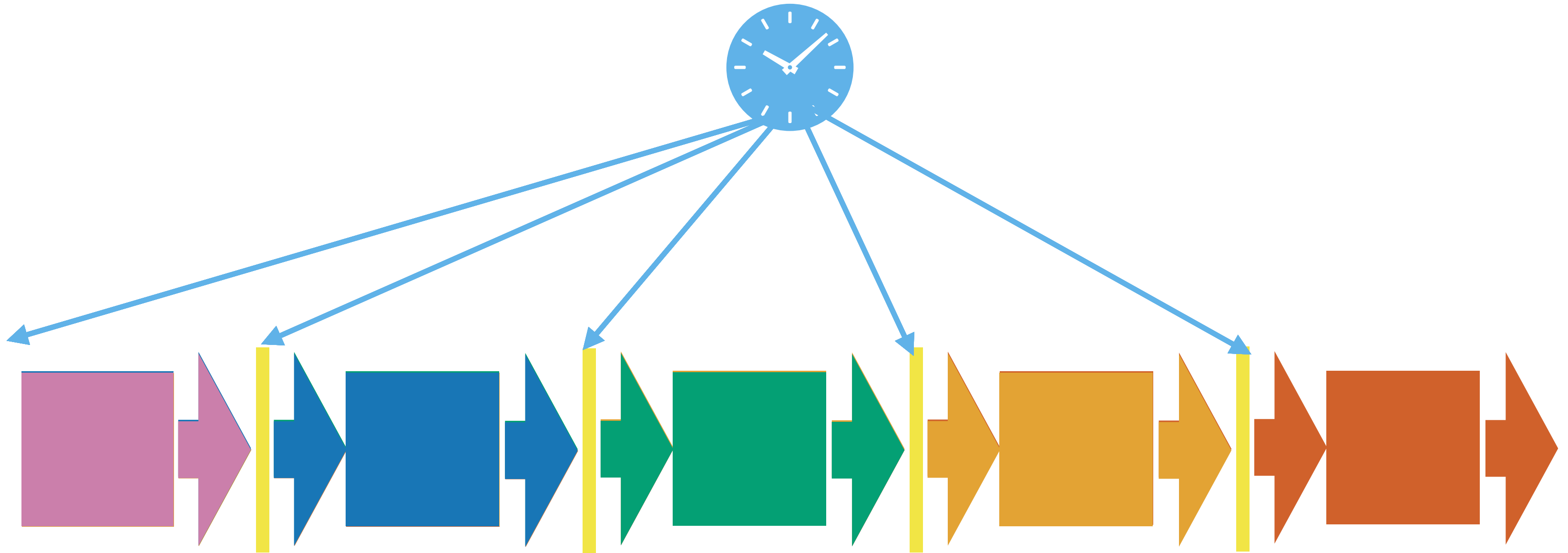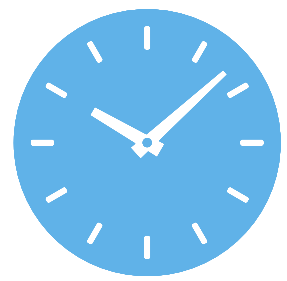
"Pipeline" the processor!
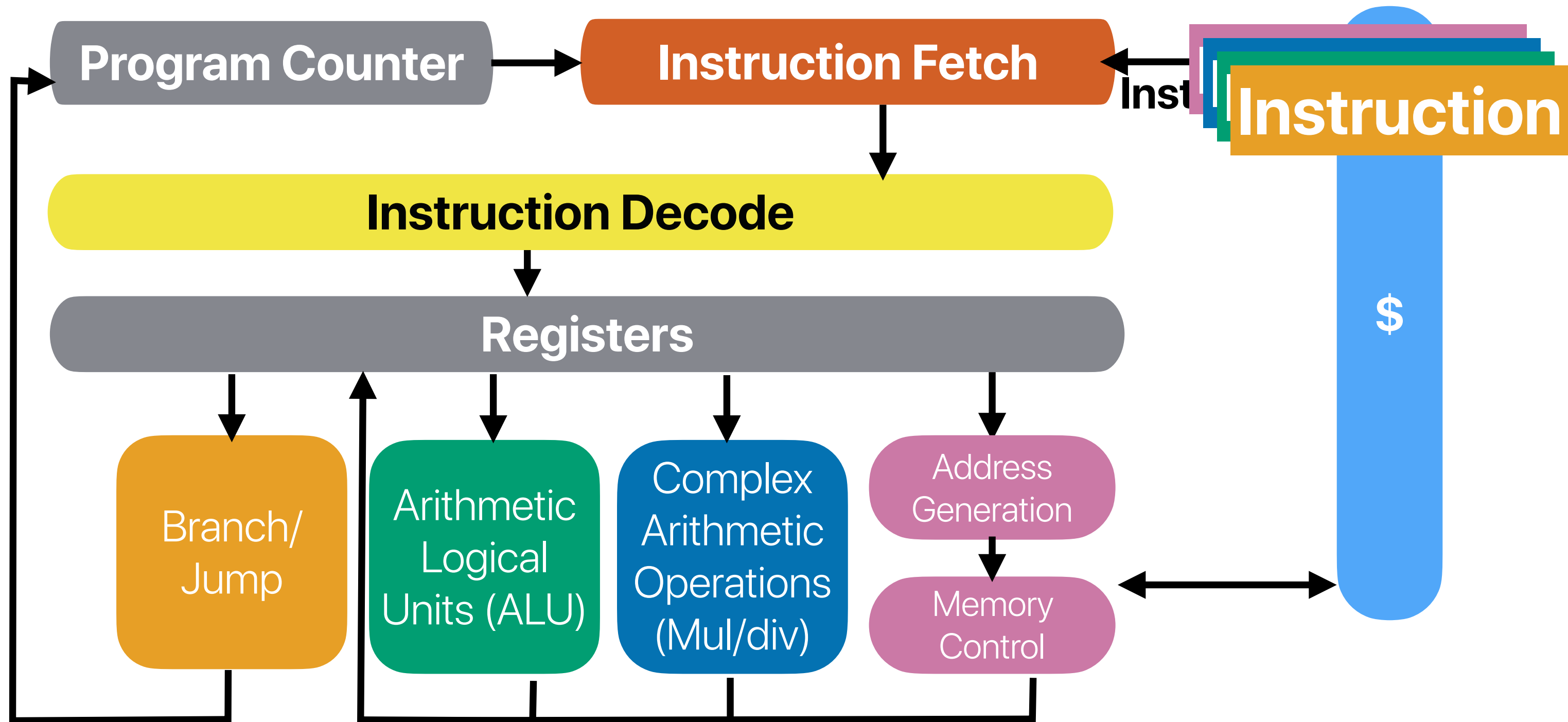
Pipelining

```
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
```

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

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

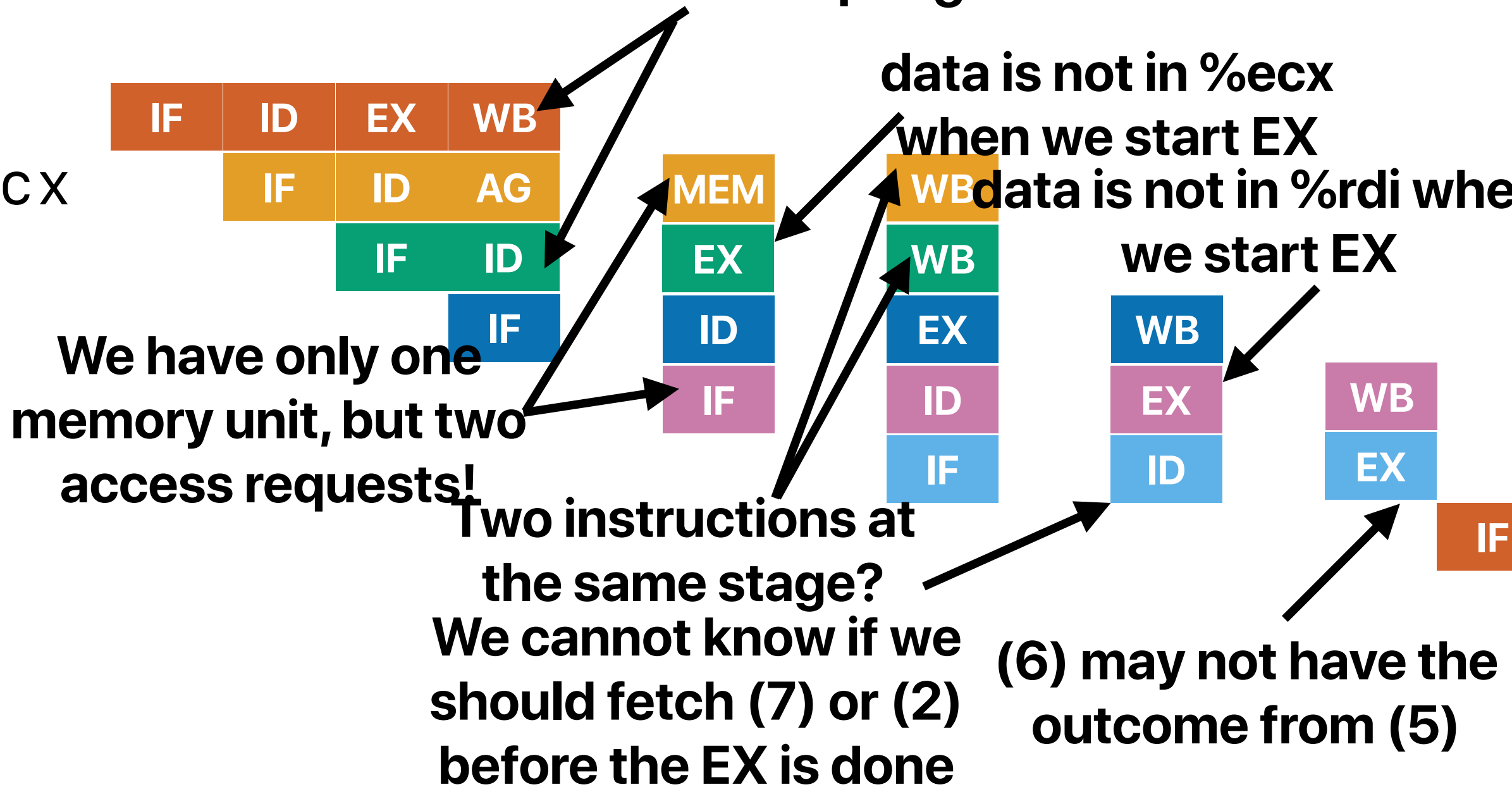*t*

21

How well can we pipeline?

- With a pipelined design, the processor is supposed to deliver the outcome of an instruction each cycle. For the following code snippet, how many pairs of instructions are preventing the pipeline from generating results in back-to-back cycles?

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

```
for(i = 0; i < count; i++) {
    s += a[i];
}
return s;
```

A. 1

B. 2

C. 3

D. 4

E. 5+

Pipelining

Both (1) and (3) are attempting to access %eax

data is not in %ecx when we start EX

data is not in %rdi when we start EX

```
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
```

IF	ID	EX	WB

	IF	ID	AG	MEM	WB

		IF	ID	EX	WB

			IF	ID	EX	WB

We have only one memory unit, but two access requests!

| | | | | IF | ID | EX | WB |

| | | | | | IF | ID | EX |

| | | | | | | IF |

Two instructions at the same stage? We cannot know if we should fetch (7) or (2) before the EX is done

(6) may not have the outcome from (5)

27

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

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`
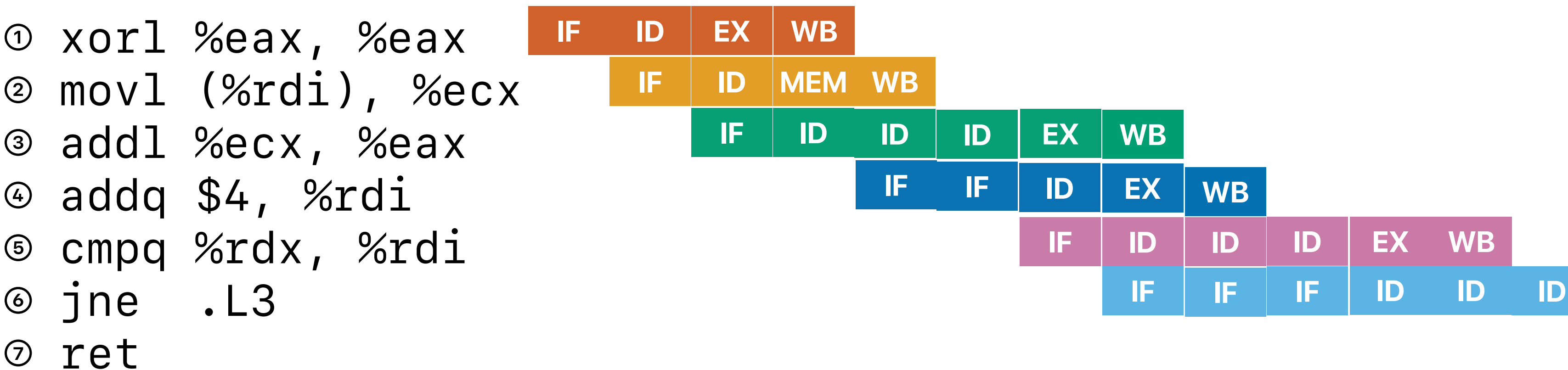
Slow! — 5 additional cycles

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!

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

How to with the conflicts between MEM and IF?

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

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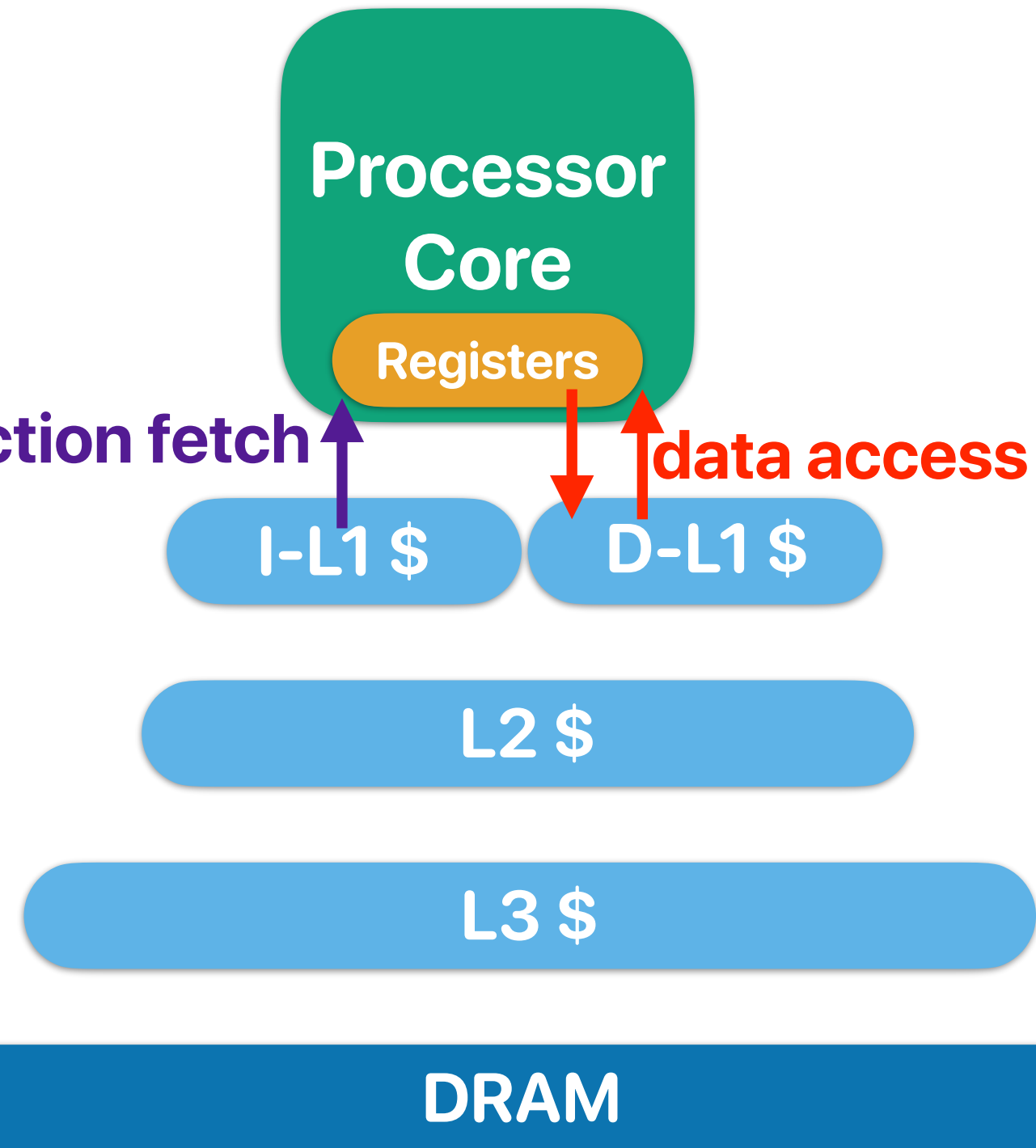② `movl (%rdi), %ecx`
③ `addl %ecx, %eax`
④ `addq $4, %rdi`

IF	ID	EX	WB

IF	ID	MEM

IF	ID

IF
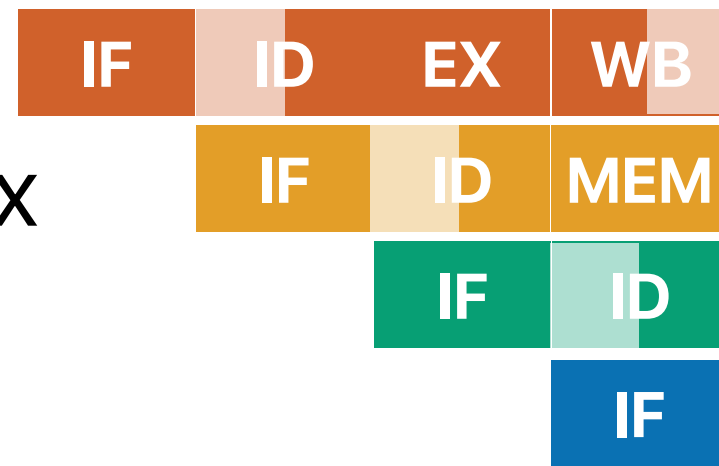
instruction fetch

data access
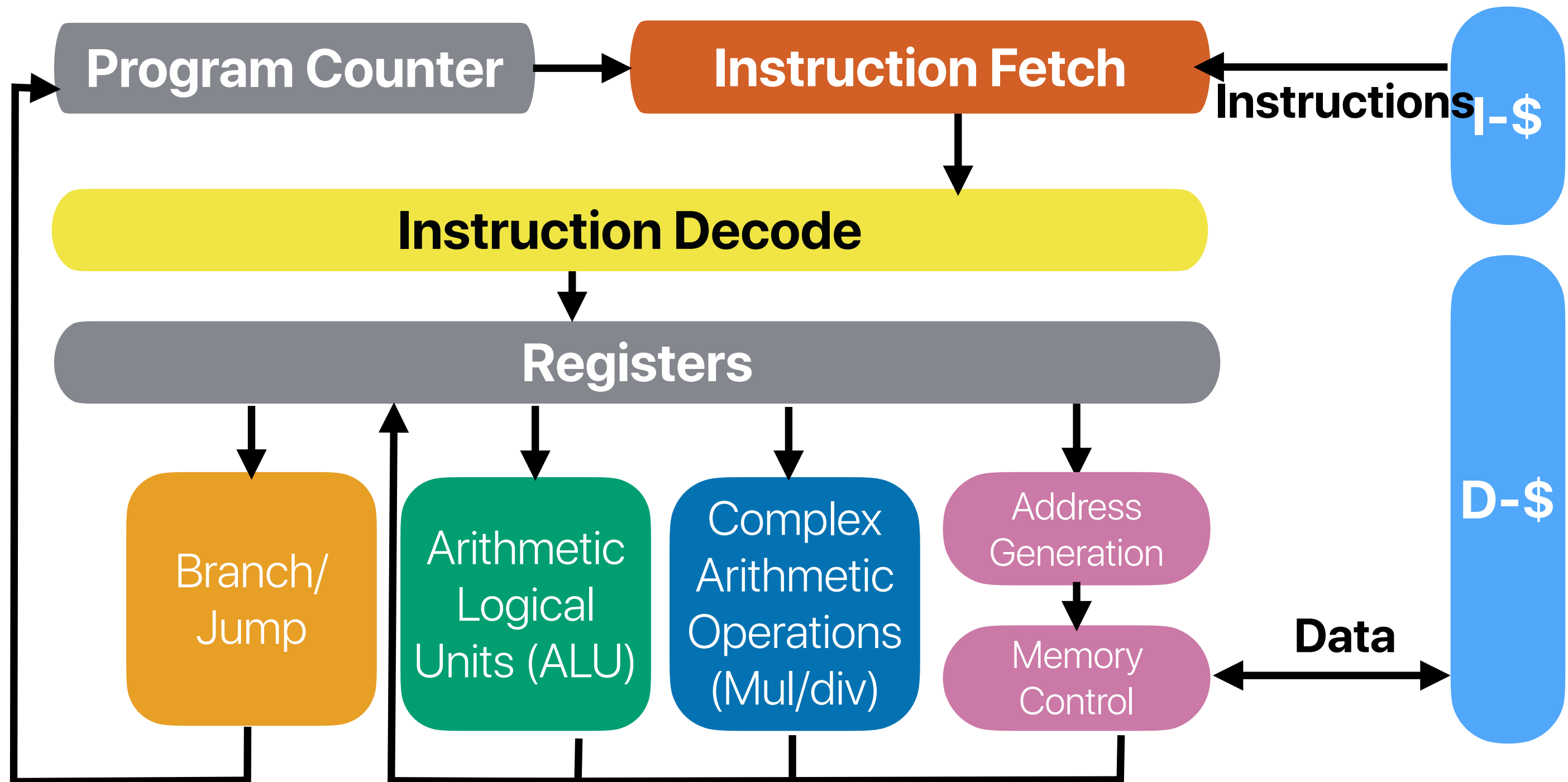
Processor Core

Registers

I-L1 $ **D-L1 $**

L2 $

L3 $

DRAM

"Split L1" cache!

Split L1-$

What if the memory instruction needs more time?
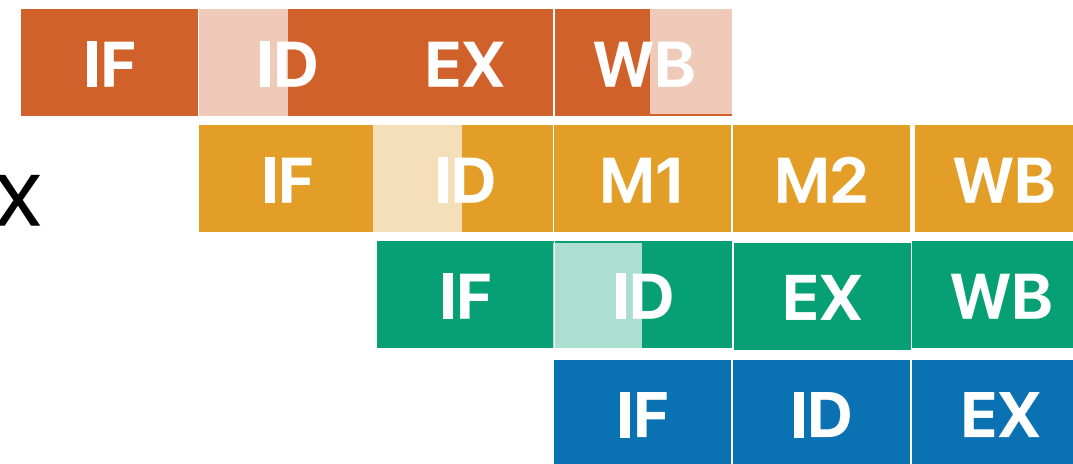
Both (2) and (3) are attempting to "WB"

① `xorl %eax, %eax`
② `movl (%rdi), %ecx`
③ `addl %ecx, %eax`
④ `addq $4, %rdi`
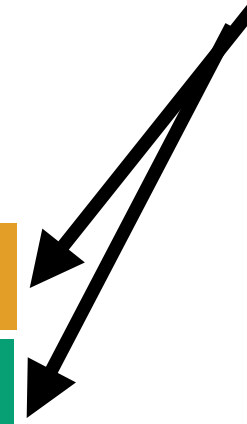
What if the memory instruction needs more time?

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

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

IF	ID	EX	EX2	WB

IF	ID	M1	M2	WB

IF	ID	EX	EX2	WB

IF	ID	EX	EX2	WB

Blocking cache

fetch block
0xDEADBE

return block
0xDEADBE

fetch block
0xDEAEBE

return block
0xDEAEBE

RAM   RAM   RAM   RAM
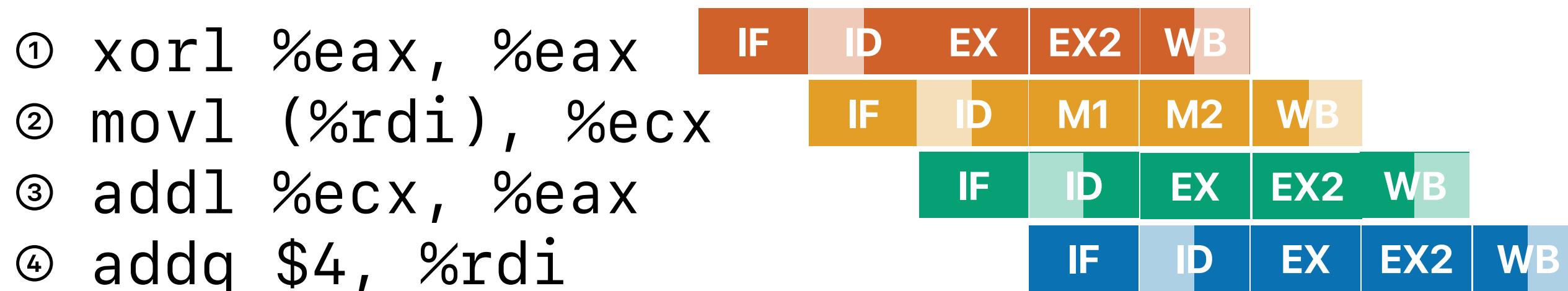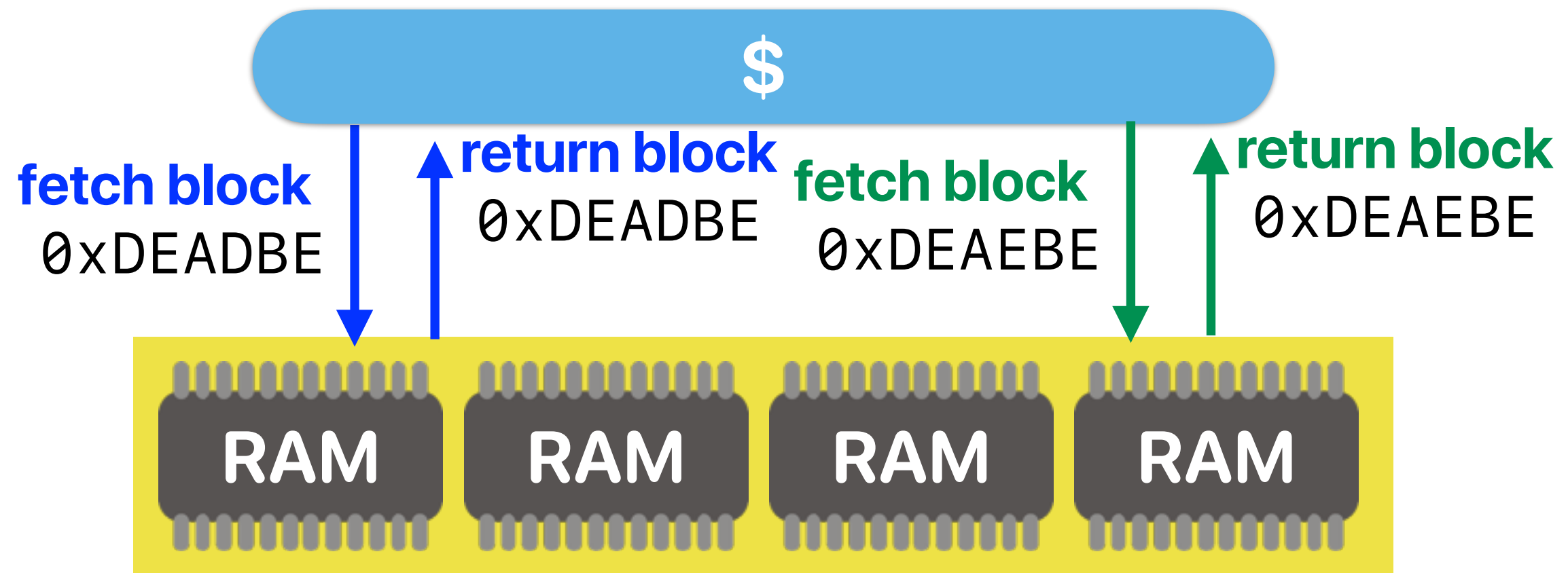
Multibanks & non-blocking caches
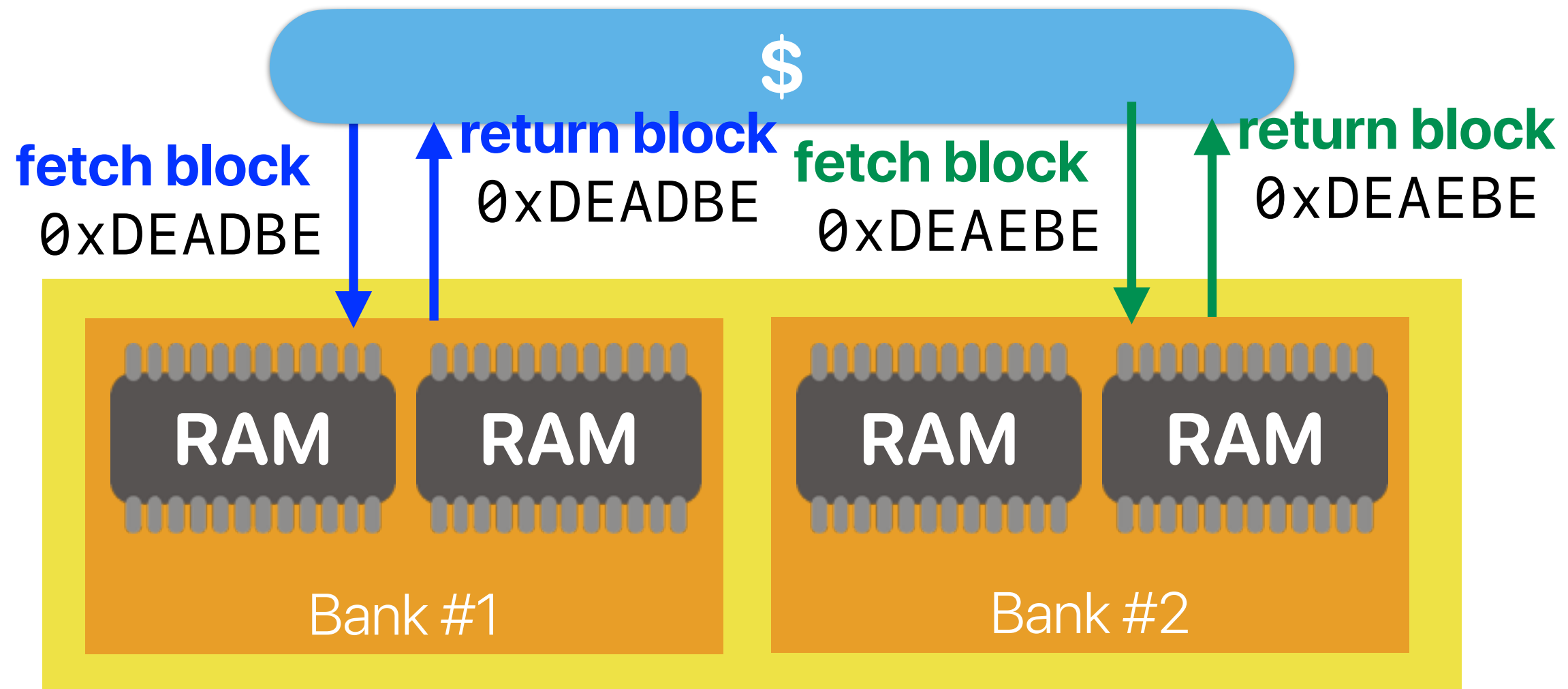
Pipelined access and multi-banked caches

Baseline

Request #1 | Request #2 | Request #3

Memory | Memory | Memory

Multi-banked

Request #1 — Bank #1

Request #2 — Bank #2

Request #3 — Bank #3

Request #4 — Bank #4

51
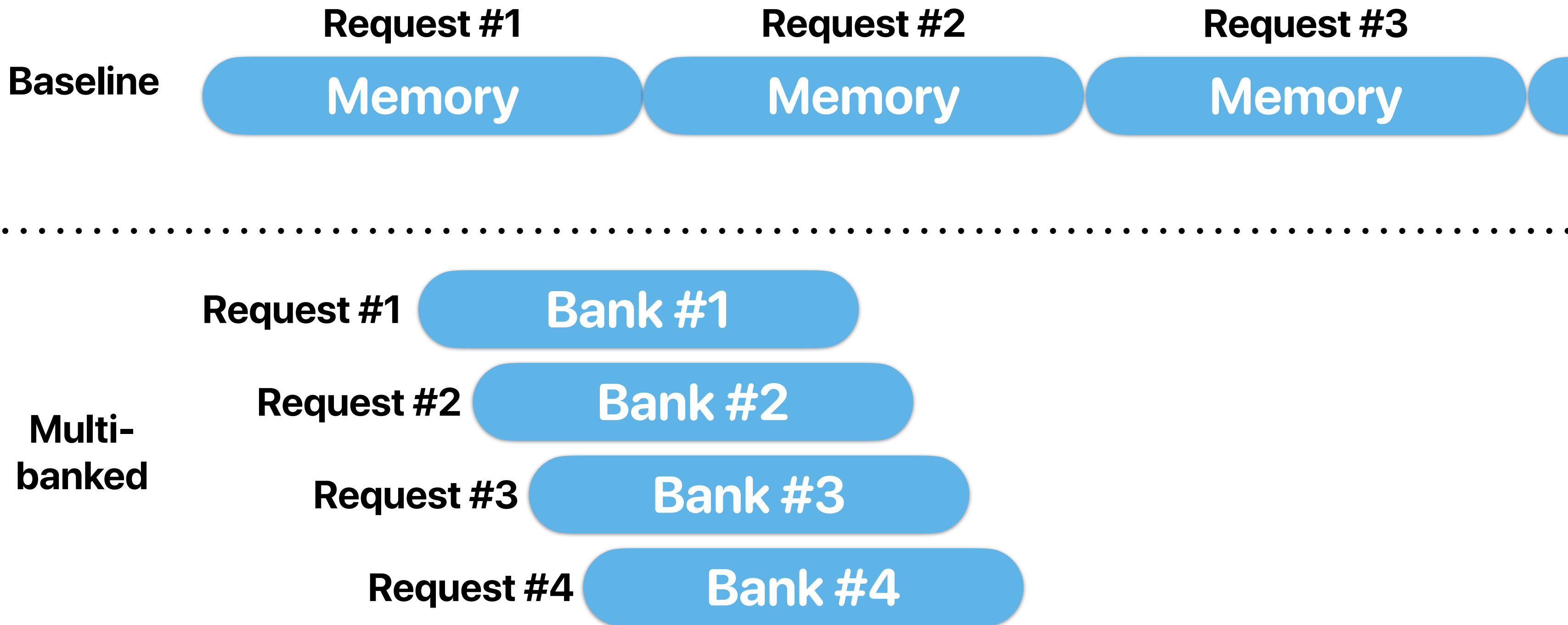
Structural Hazards

- 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`

IF	ID	EX	WB

IF	ID	ID	EX

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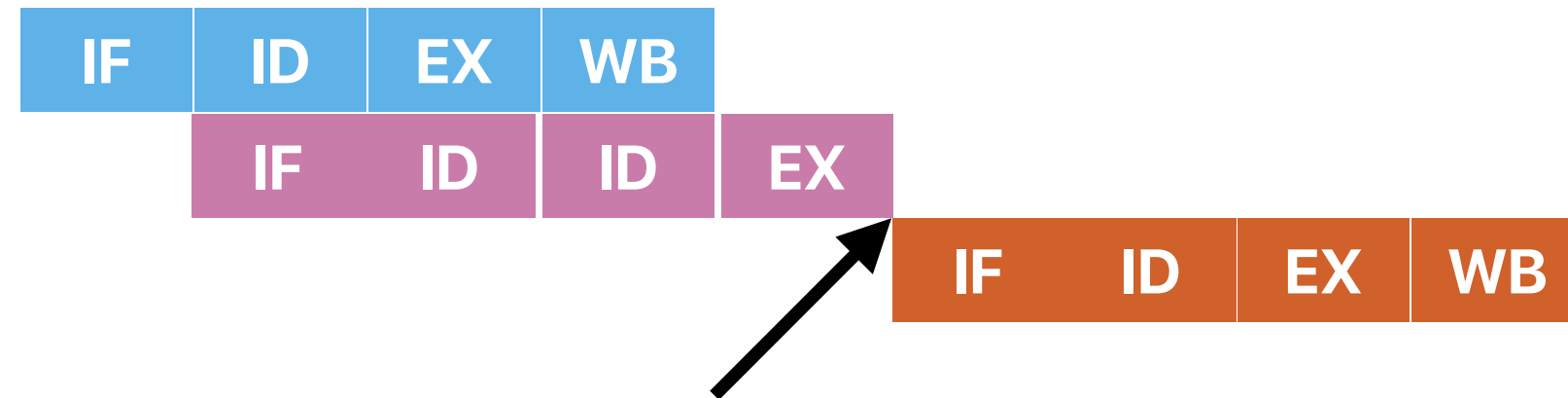
We cannot know if we should fetch (7) or (2) before the EX is done

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 data[i]< 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
```

55

Dynamic Branch Prediction

Microprocessor with a "branch predictor"

Branch Predictor

Program Counter → **Instruction Fetch** ← **Instructions** **I-$**

Instruction Decode

Registers

Branch/ Jump | Arithmetic Logical Units (ALU) | Complex Arithmetic Operations (Mul/div) | Address Generation → Memory Control ↔ **D-$**

68

Detail of a basic dynamic branch predictor

Next PC

MUX

Program Counter

Instruction Fetch

I-$

Instructions

Instruction Decode

Registers

D-$

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

Branch Target Buffer

Branch/ Jump

Arithmetic Logical Units (ALU)

Complex Arithmetic Operations (Mul/div)

**Address Generation**

Memory Control

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

70
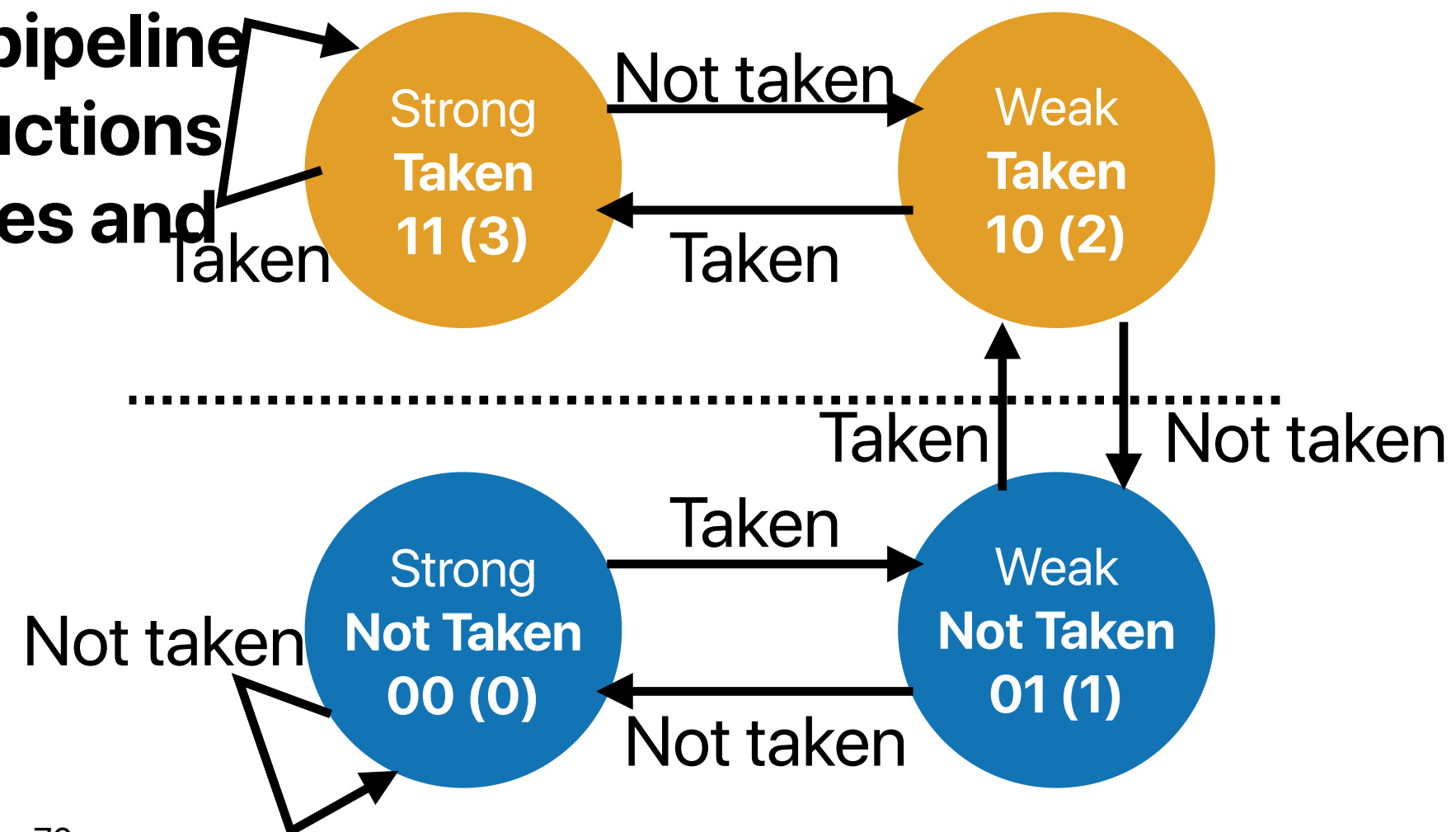
2-bit local predictor

```
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
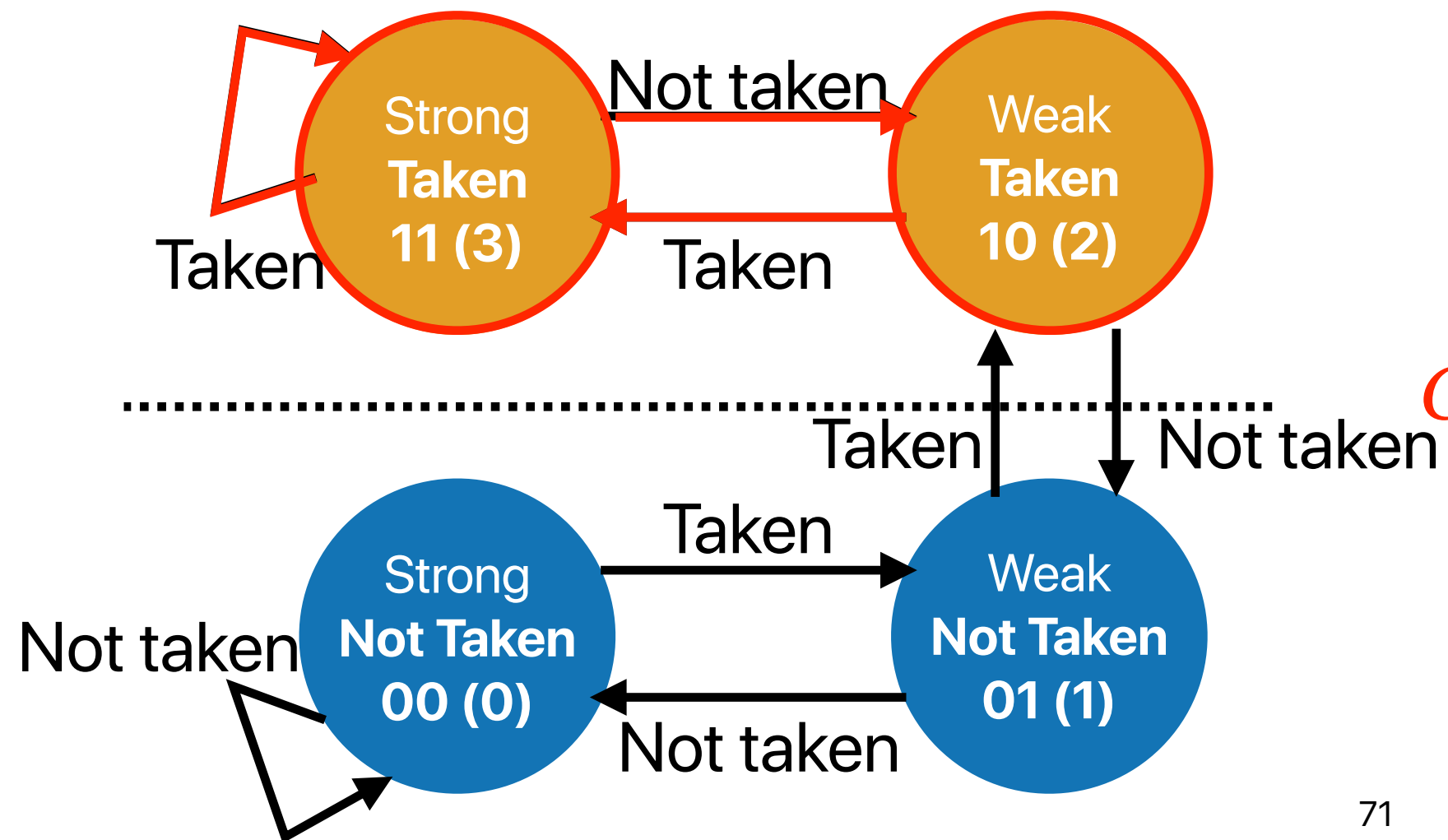
90% accuracy!

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

71

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%**

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

**This pattern
repeats all the time!**

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

Detail of a basic dynamic branch predictor

Next PC

MUX

Program Counter

Instruction Fetch

Instructions

I-$

branch PC   target PC

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

States associated with history

Predict Taken

branch PC	target PC
0x400048	0x400032
0x400080	0x400068
0x401080	0x401100
0x4000F8	0x400100

Branch Target Buffer

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

Instruction Decode

Registers

Global History Register

0100

Branch/ Jump

Arithmetic Logical Units (ALU)

Complex Arithmetic Operations (Mul/div)

Address Generation

Memory Control

D-$

79

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
```
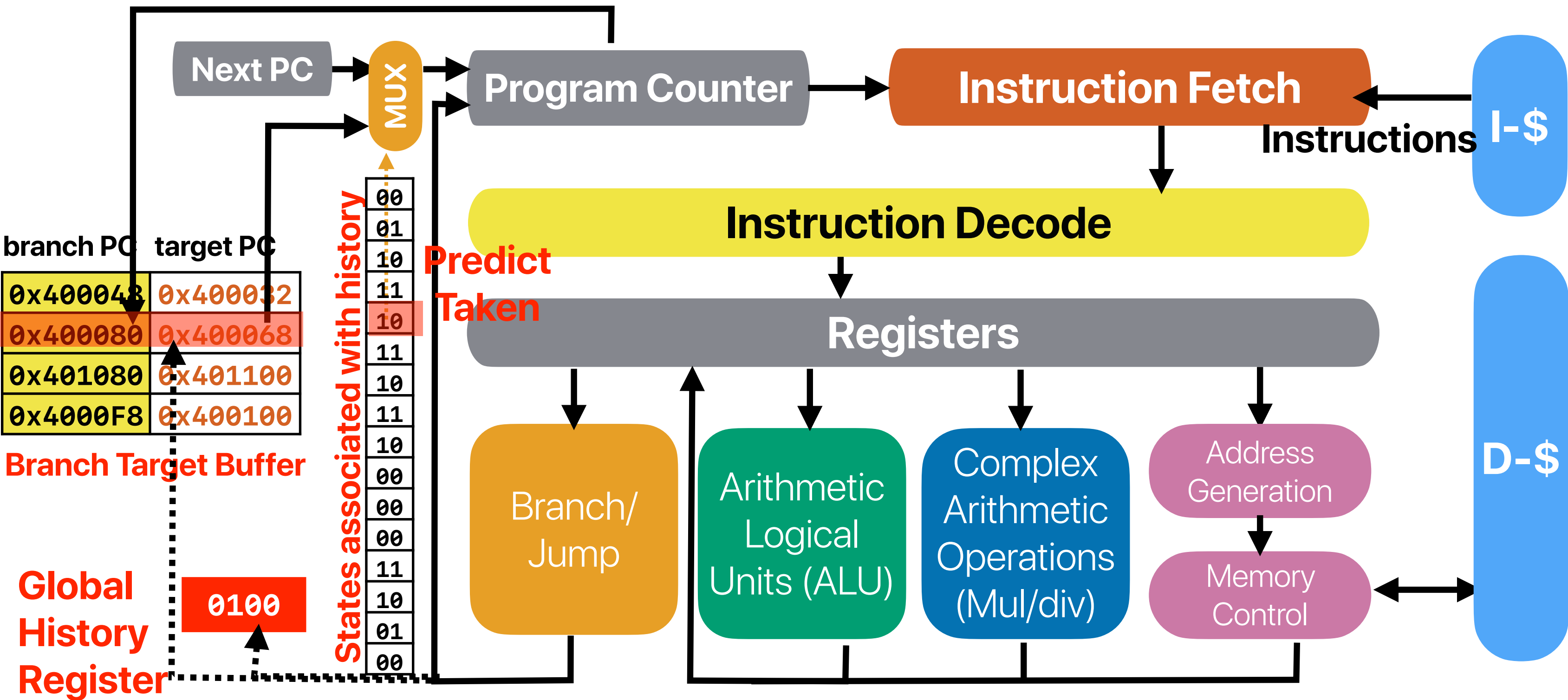
Near perfect after this

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

80

Hybrid predictors

Tournament Predictor
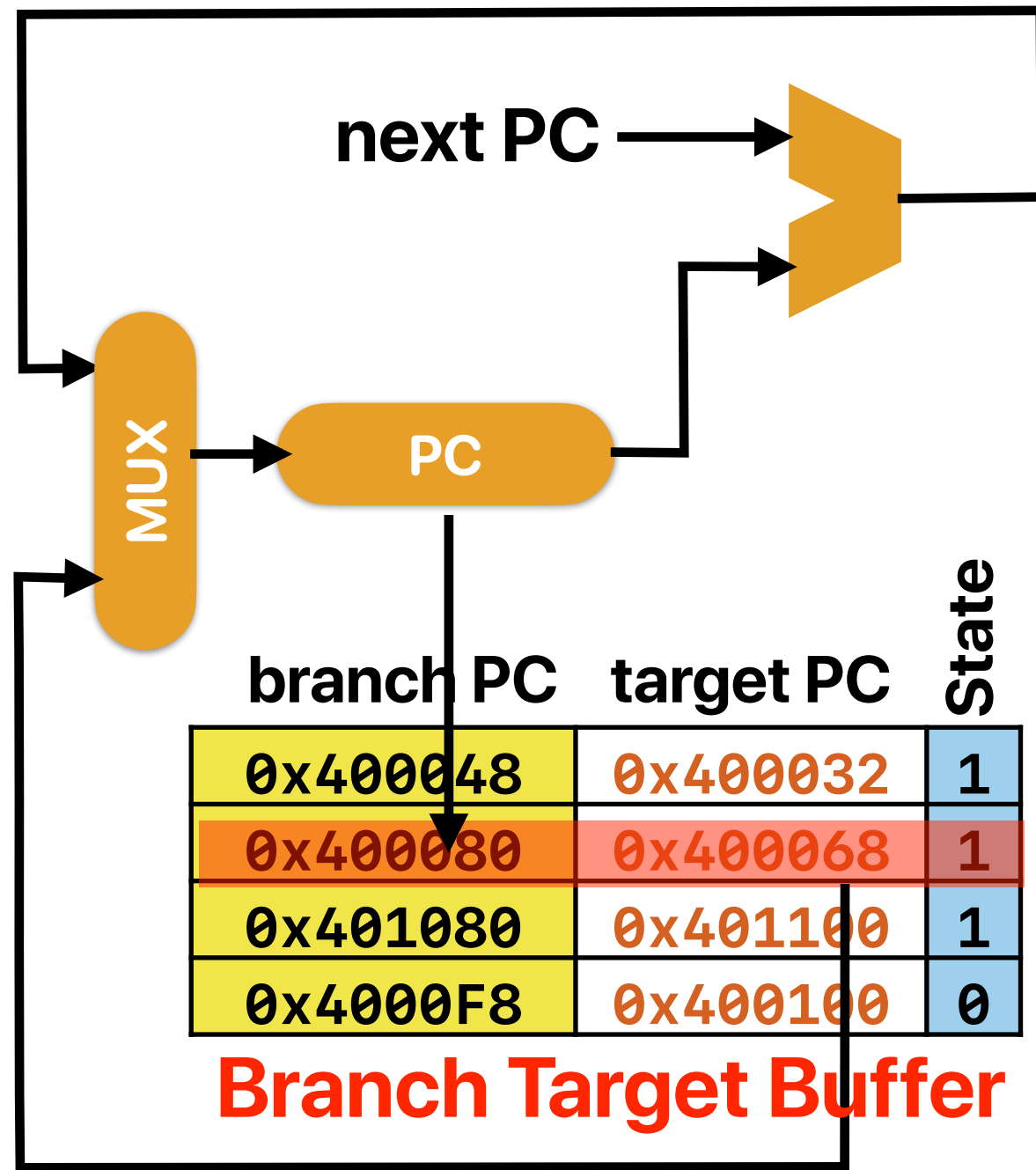
Global History Register

0100

States associated with history

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

Local History Predictor

branch PC	local history
0x400048	1000
0x400080	0110
0x401080	1010
0x4000F8	0110

States associated with history

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

Predict Taken

next PC

MUX

PC

branch PC	target PC	State
0x400048	0x400032	1
0x400080	0x400068	1
0x401080	0x401100	1
0x4000F8	0x400100	0

Branch Target Buffer

87

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

$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$
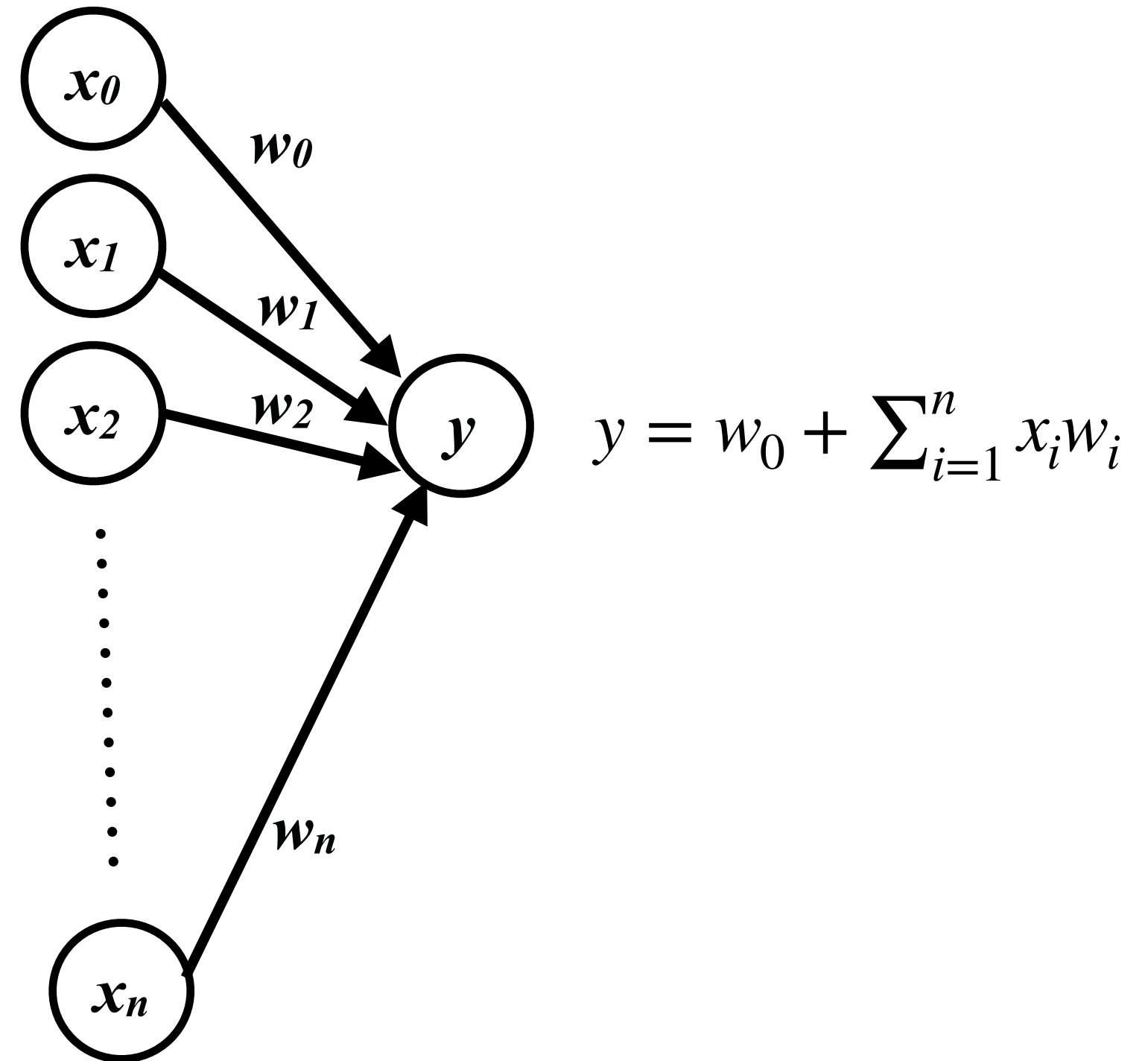
Training Algorithm

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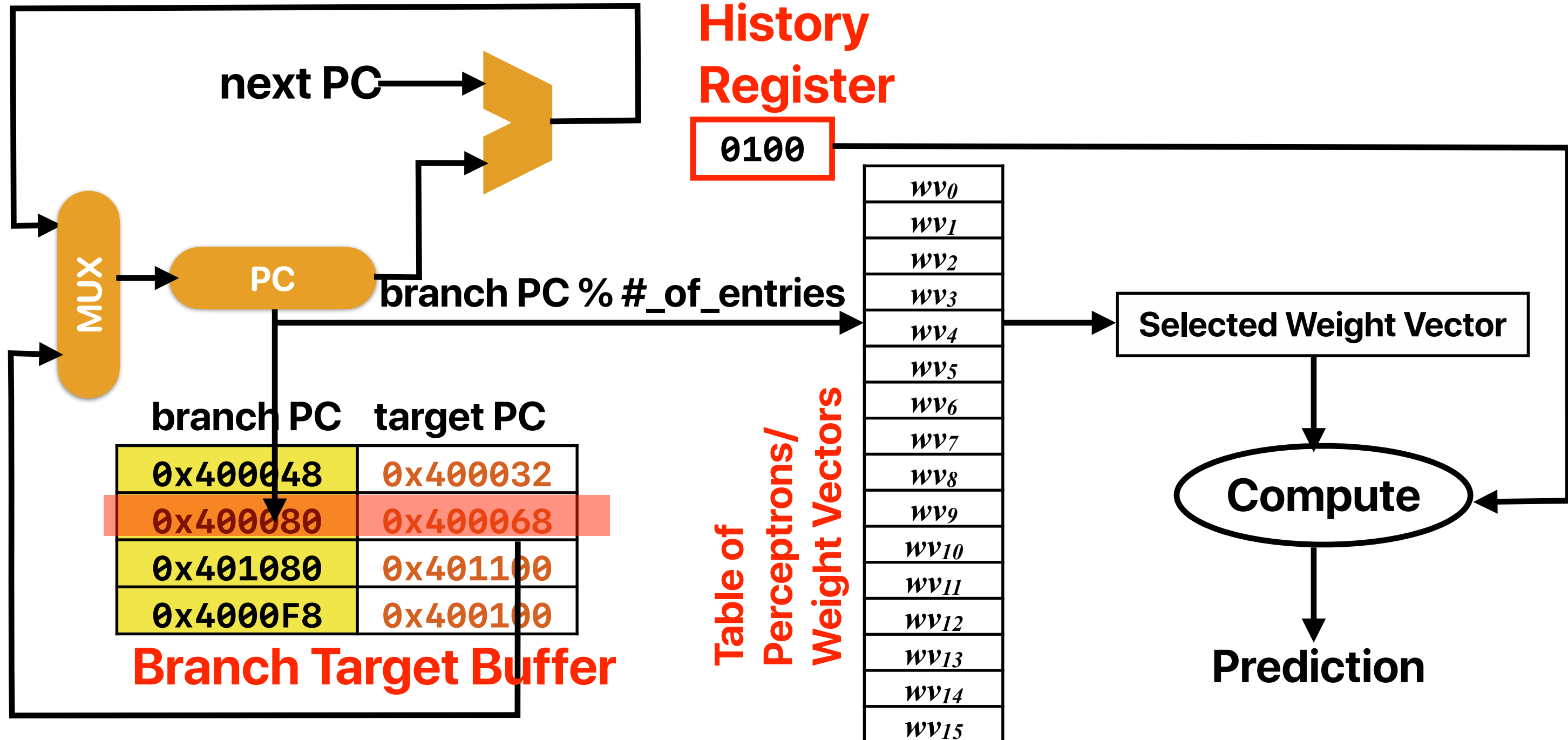$w_{0..n}$ is the weights vector.
t is the Boolean branch outcome.
θ is the training threshold.

```
if |y| ≤ θ or ((y ≥ 0) ≠ t) then
      for each 0 ≤ i ≤ n in parallel
            if t = xᵢ then
                  wᵢ := wᵢ + 1
            else
                  wᵢ := wᵢ − 1
            end if
      end for
end if
```

Predictor Organization

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.

Branch and programming

Demo revisited

```cpp
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 ++;
    }
}
```

option = 1 is faster!!!

SELECT count(*) FROM TABLE WHERE val < A and val >= B;

Demo: Popcount

- The population count (or popcount) of a specific value is the number of set bits (i.e., bits in 1s) in that value.
- Applications
  - Parity bits in error correction/detection code
  - Cryptography
  - Sparse matrix
  - Molecular Fingerprinting
  - Implementation of some succinct data structures like bit vectors and wavelet trees.

Demo: Popcount

- Given a 64-bit integer number, find the number of 1s in its binary representation.
- Example 1:
  Input: 59487
  Output: 9
  Explanation: 59487's binary representation is 0b1011001010000 1111

```c
int main(int argc, char *argv[]) {

    uint64_t key = 0xdeadbeef;

    int count = 1000000000;
    uint64_t sum = 0;

    for (int i=0; i < count; i++)
    {
        sum += popcount(RandLFSR(key));
    }
    printf("Result: %lu\n", sum);
    return sum;
}
```

Hardware acceleration

- Because popcount is important, both intel and AMD added a POPCNT instruction in their processors with SSE4.2 and SSE4a

- In C/C++, you may use the intrinsic "_mm_popcnt_u64" to get # of "1"s in an unsigned 64-bit number

  - You need to compile the program with -m64 -msse4.2 flags to enable these new features

```
#include <smmintrin.h>
inline int popcount(uint64_t x) {
    int c = _mm_popcnt_u64(x);
    return c;
}
```

Tricky C/C++ programming questions?

- Give a fastest way to multiply any number by 9 **— compilers can help**
- How to measure the size of any variable without "sizeof" operator?.
- How to measure the size of any variable without using "sizeof" operator?
- Write code snippets to swap two variables in five different ways **— other fancy ways are slower**
- How to swap between first & 2nd byte of an integer in one line statement? **— does not help performance**
- What is the efficient way to divide a no. by 4? **— compilers can help**
- Suggest an efficient method to count the no. of 1's in a 32 bit no. Remember without using loop & testing each bit. **— programmers can help**
- Test whether a no. is power of 2 or not. **— programmers can help**
- How to check endianness of the computer. **— other fancy ways are slower**
- Write a C-program which does the addition of two integers without using '+' operator.
- Write a C-program to find the smallest of three integers without using any of the comparision operators.
- Find the maximum & minimum of two numbers in a single line without using any condition & loop.
- What "condition" expression can be used so that the following code snippet will print Hello world.
- How to print number from 1 to 100 without using conditional operators.
- WAP to print 100 times "Hello" without using loop & goto statement.
- Write the equivalent expression for x%8.

https://www.emblogic.com/blog/12/tricky-c-interview-questions/