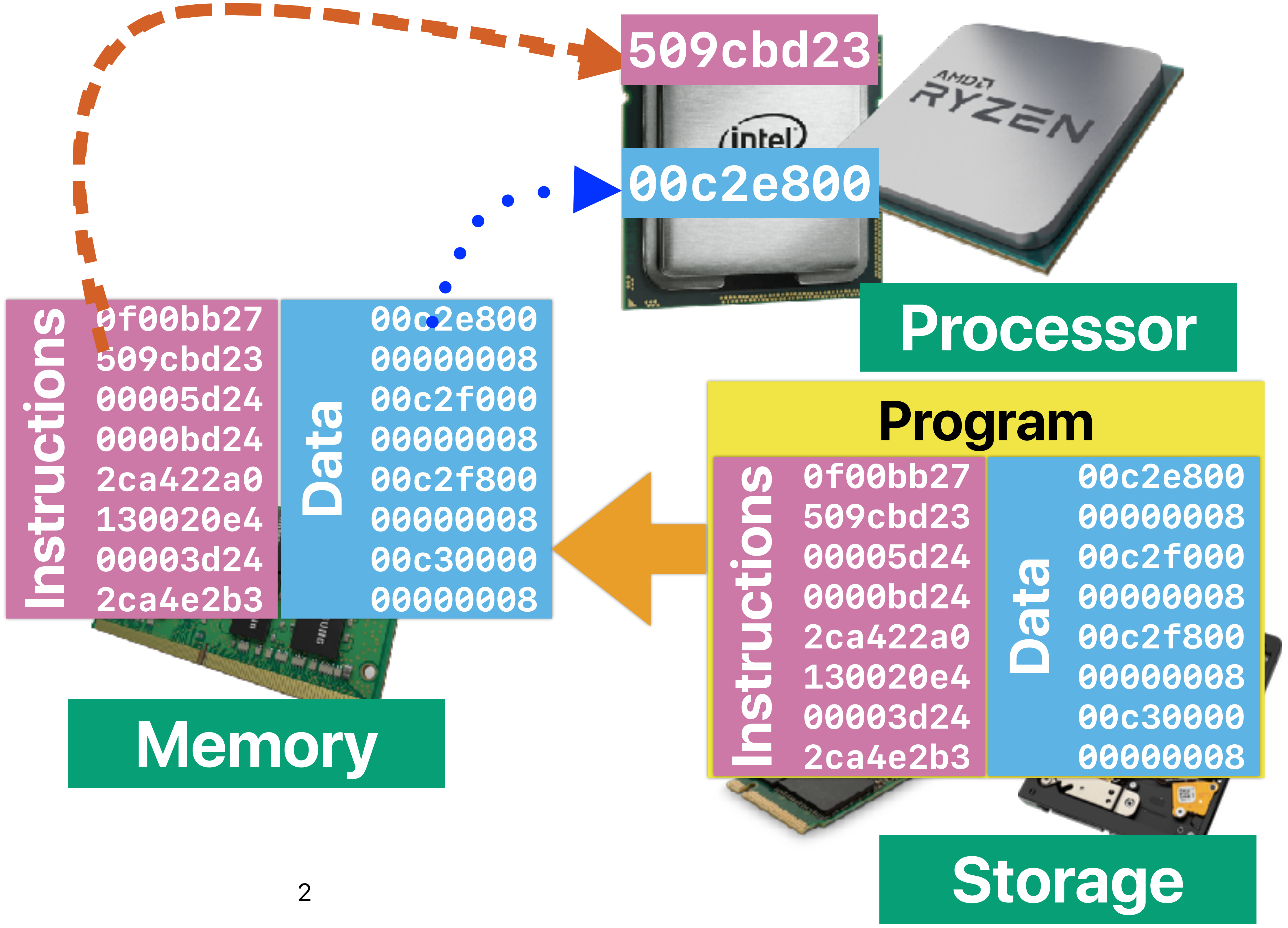
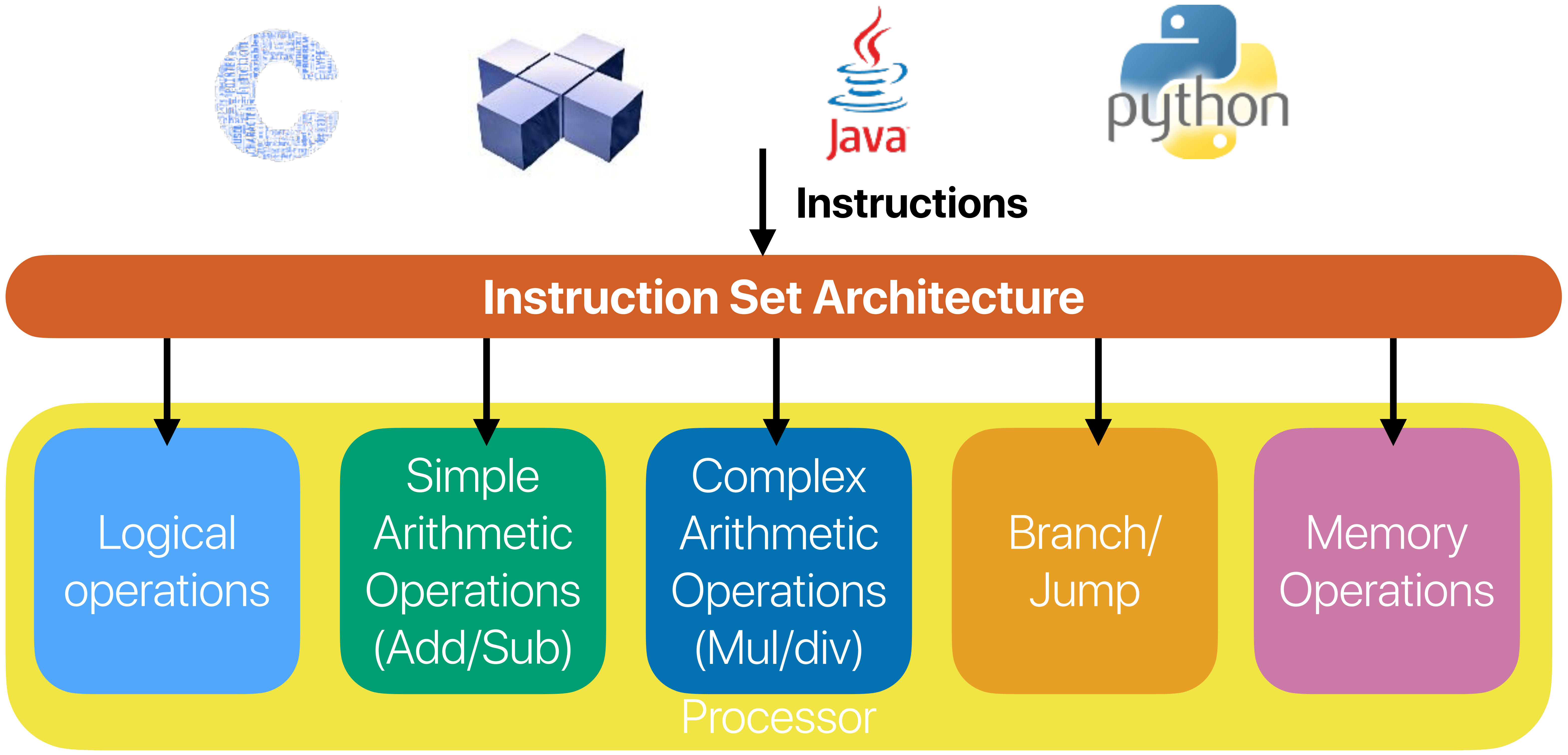
Modern Processor Design (I): in the pipeline

Hung-Wei Tseng

von Neumman Architecture



Recap: Microprocessor — a collection of functional units

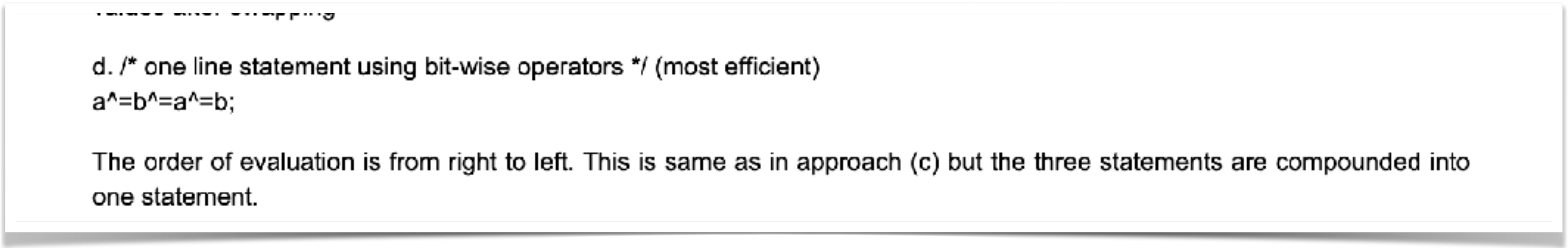

Java
python
Instructions
Instruction Set Architecture
Logical operations
Simple Arithmetic Operations (Add/Sub)
Complex Arithmetic Operations (Mul/div)
Branch/ Jump
Memory Operations
Processor

Tricky C/C++ programming questions?

- Give a fastest way to multiply any number by 9
- 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
- How to swap between first & 2nd byte of an integer in one line statement?
- What is the efficient way to divide a no. by 4?
- Suggest an efficient method to count the no. of 1's in a 32 bit no. Remember without using loop & testing each bit.
- Test whether a no. is power of 2 or not.
- How to check endianness of the computer.
- 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/

Recap: Demo (3) — Bitwise operations?



d. /* one line statement using bit-wise operators */ (most efficient)
a^=b^=a^=b;

The order of evaluation is from right to left. This is same as in approach (c) but the three statements are compounded into one statement.

A

```
void regswap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

B

```
void xorswap(int* a, int* b) {
    *a ^= *b ^= *a = *b;
}
```

Recap: Leveraging more “bit-wise” operations in C code will make the program significantly faster

✖

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 ++;
    }
}
```

Outline

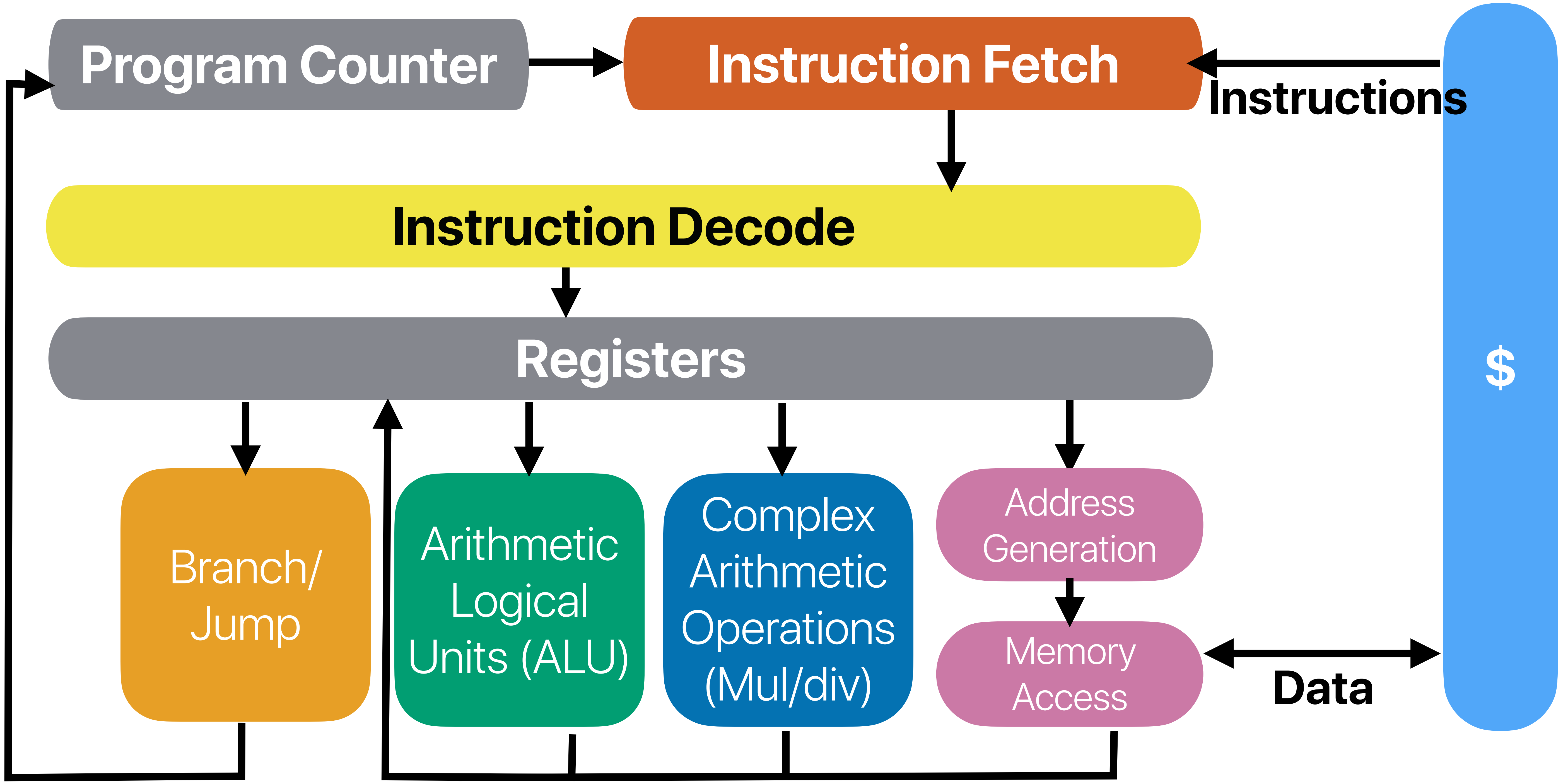
- Recap: the concept of a processor
- Pipelined Processor
- Pipeline Hazards
  - Structural Hazards
  - Control Hazards
  - Data Hazards

Basic Processor Design

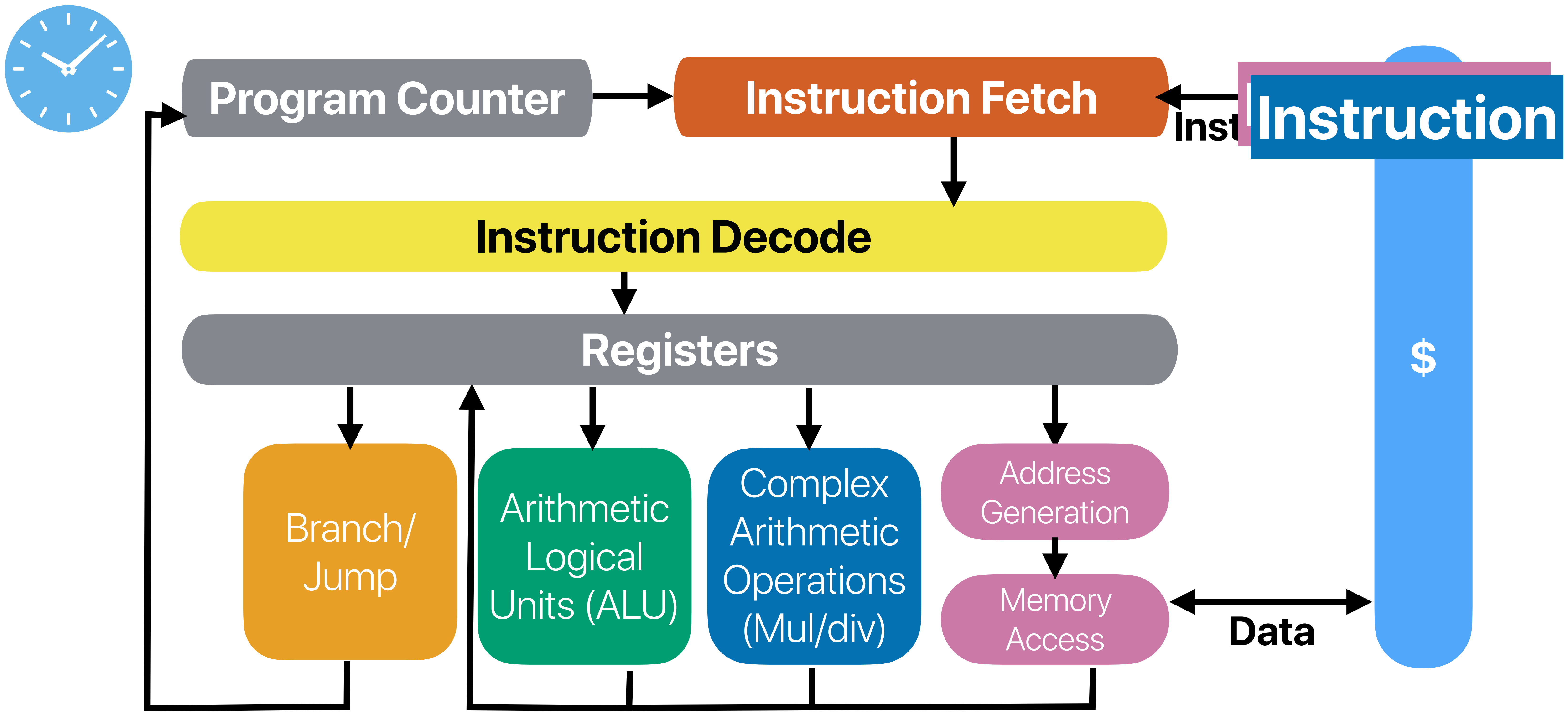
The “life” of an instruction

- 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

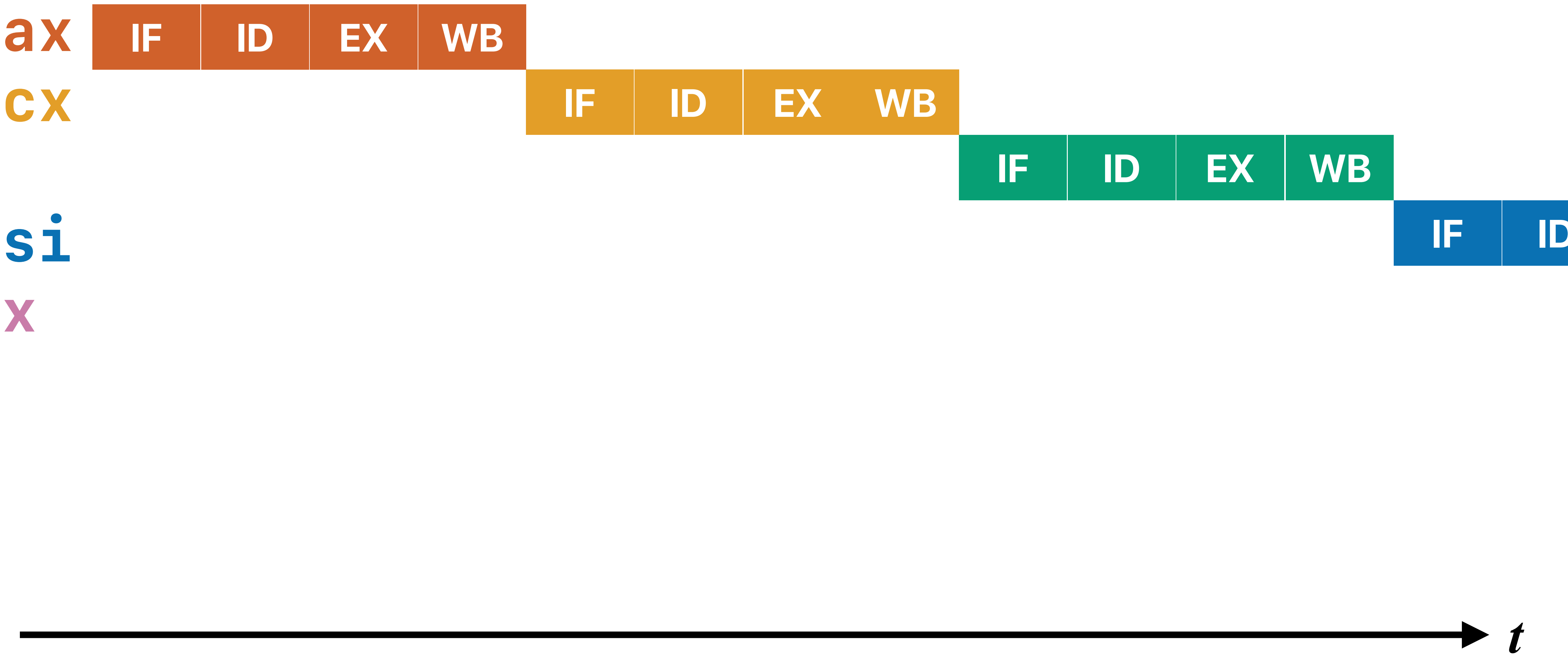


If we want to perform one instruction each cycle...



Simple implementation w/o branch

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

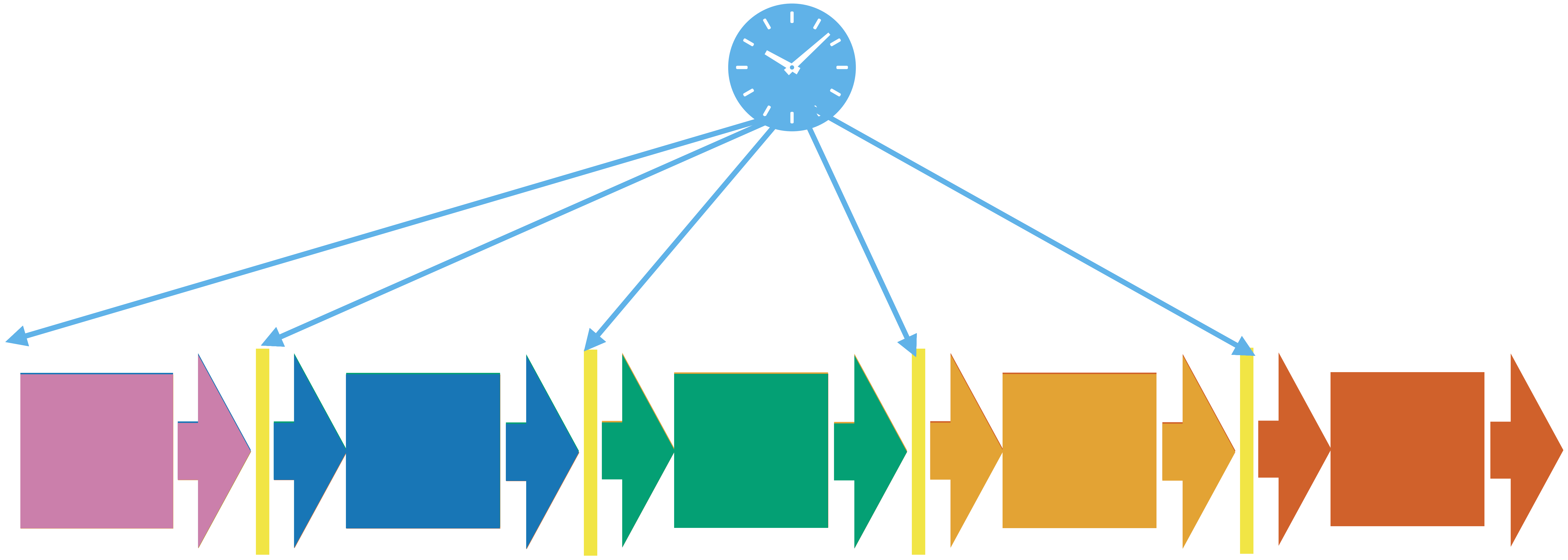


Pipelining

Pipelining

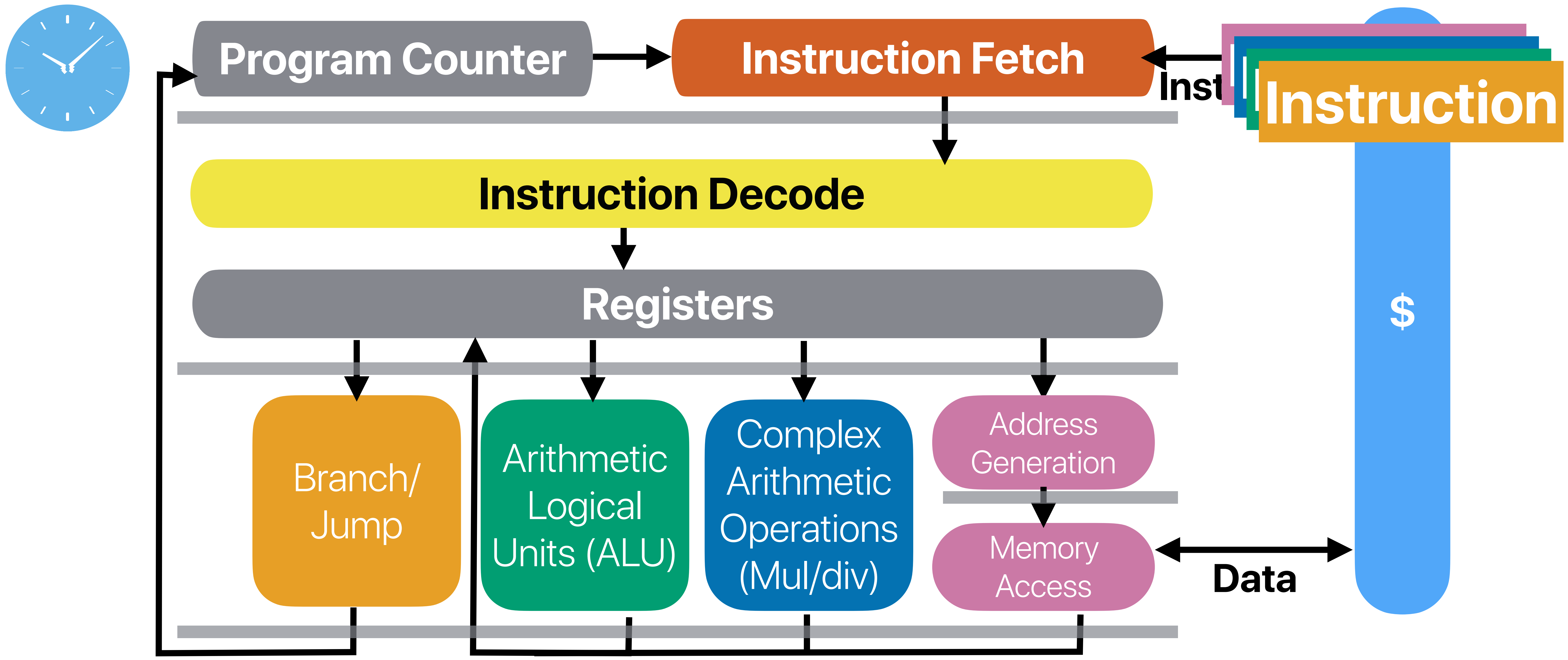
- Different parts of the processor works on different instructions simultaneously
- A processor is now working on multiple instructions from the same program (though on different stages) simultaneously.
  - ILP: **Instruction-level parallelism**
- A **clock** signal controls and synchronize the beginning 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



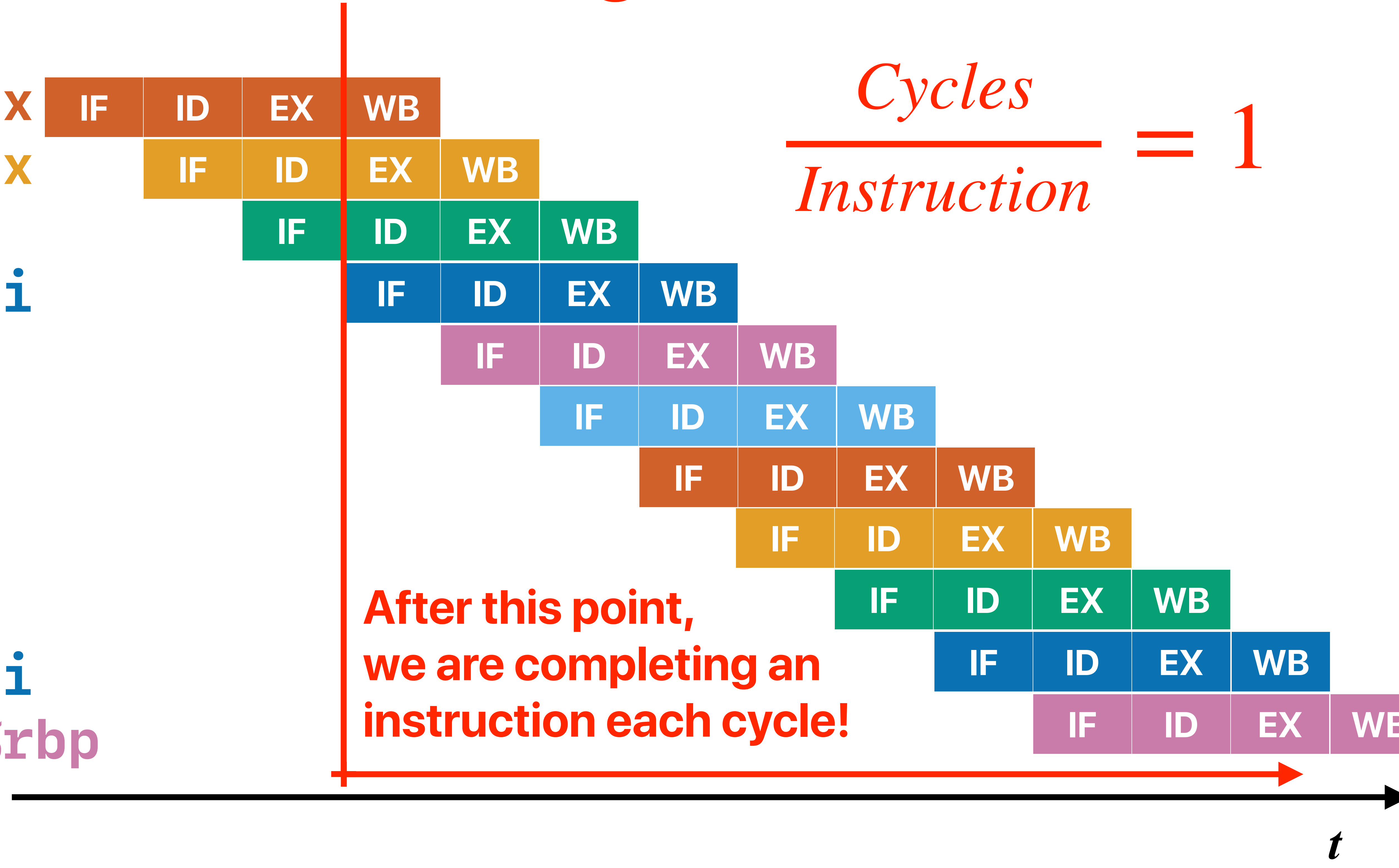
Pipelining

Pipelined execution



Pipelining

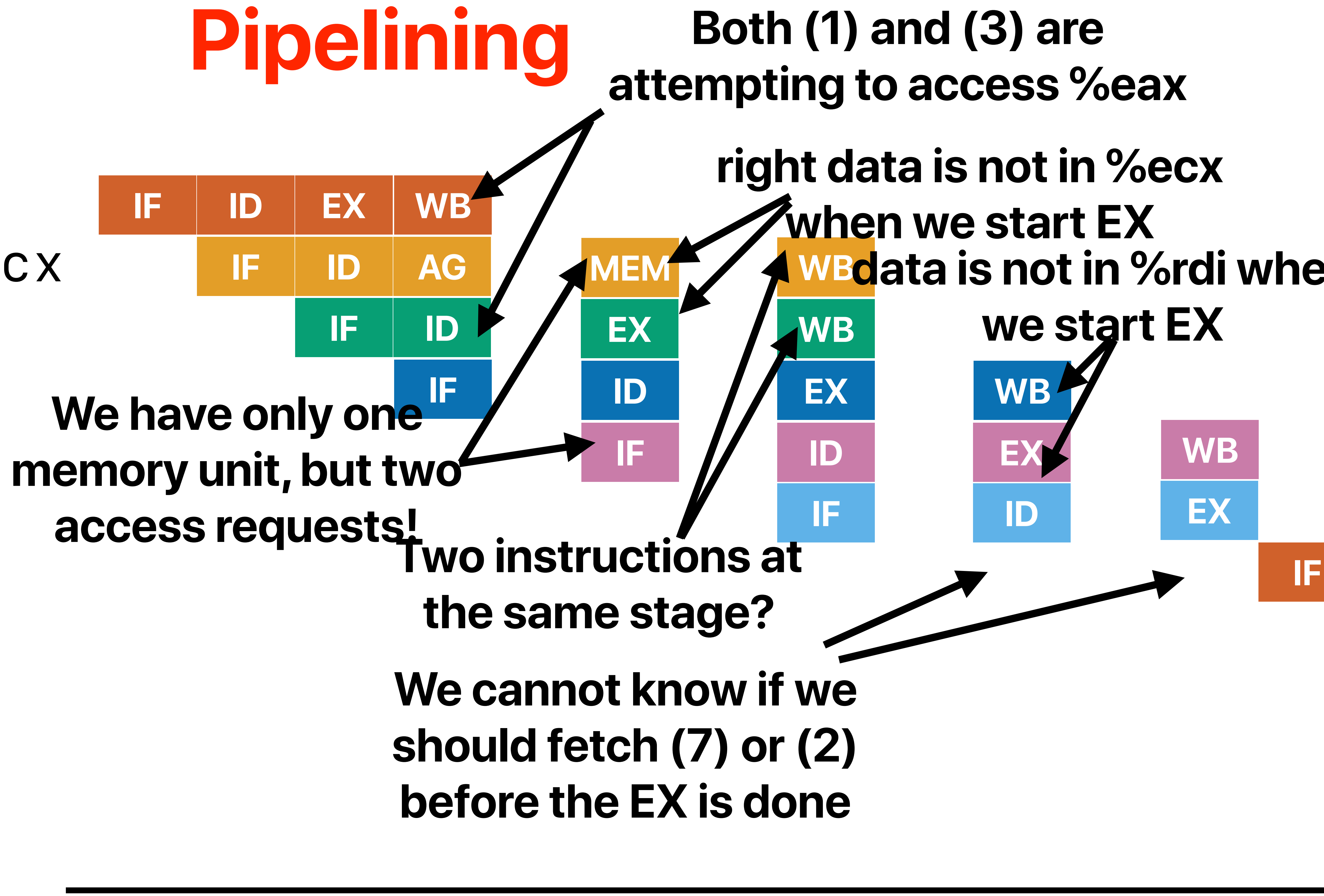
```
addl   %eax, %eax
addl   %rdi, %ecx
addq   $4, %r11
testl %esi, %esi
movl   $10, %edx
pushq %r12
pushq %rbp
pushq %rbx
subq   $8, %rsp
addl   %rsi, %rdi
movslq  %eax, %rbp
```


IF
ID
EX
WB
After this point, we are completing an instruction each cycle!
t

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

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
```



IF
ID
EX
WB
AG
MEM
Both (1) and (3) are attempting to access %eax
right data is not in %ecx when we start EX
data is not in %rdi whe
we start EX
We have only one memory unit, but two access requests!
Two instructions at the same stage?
We cannot know if we should fetch (7) or (2) before the EX is done

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]

Takeaways: pipeline processors

- Pipelining helps to improve the throughput of processors
  - Allowing shorter cycle time as each cycle only make progress for part of an instruction
  - Different pipeline stages work on different instructions concurrently
  - Theoretical CPI remains the same as single-cycle design and the throughput/speedup is in proportion to the speedup of cycle time

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

Takeaways: pipeline processors

- Pipelining helps to improve the throughput of processors
  - Allowing shorter cycle time as each cycle only make progress for part of an instruction
  - Different pipeline stages work on different instructions concurrently
  - Theoretical CPI remains the same as single-cycle design and the throughput/speedup is in proportion to the speedup of cycle time
- Pipeline hazards prevent us from reaching the theoretical CPI
  - Structural hazards
  - Control hazards
  - Data hazards

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
- Disable the pipeline register update for later instructions
- The stalled instructions still have the same input from the pipeline registers

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

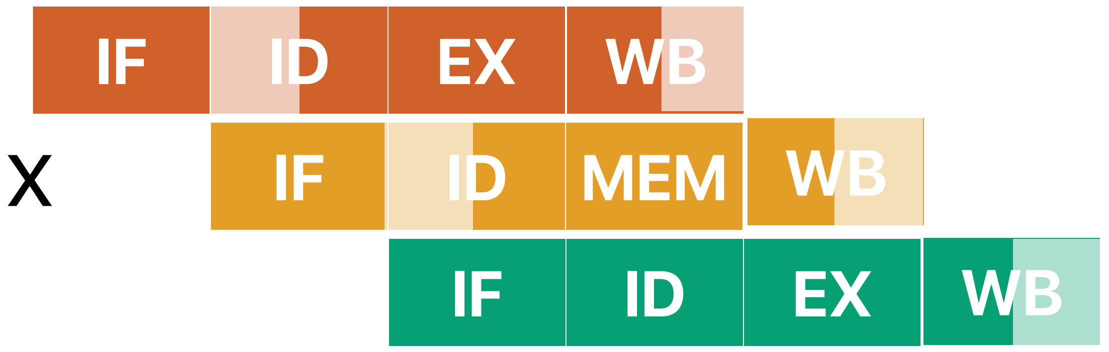
Slow! — 4 additional cycles

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
```

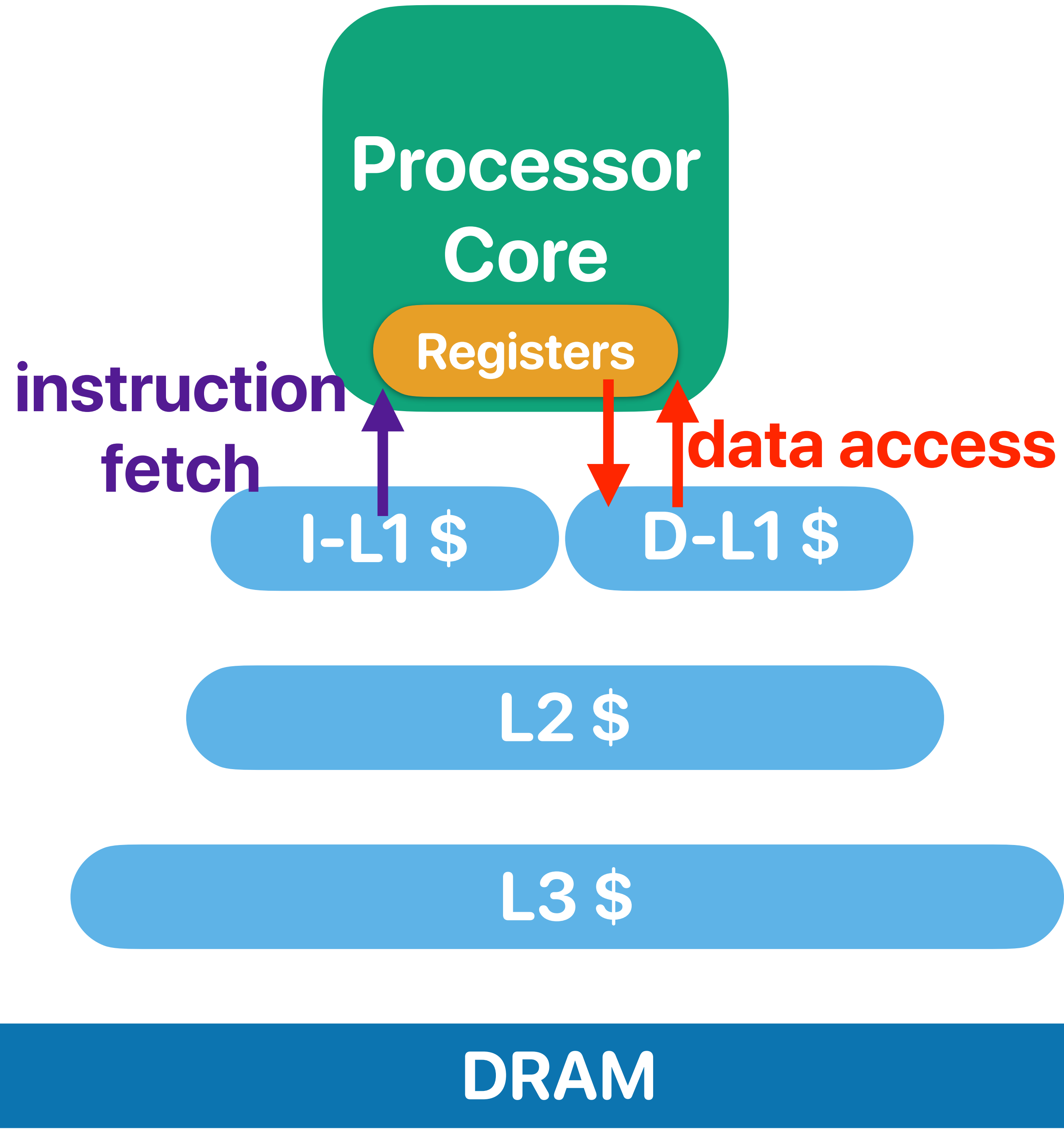


How to with the conflicts between MEM and IF?

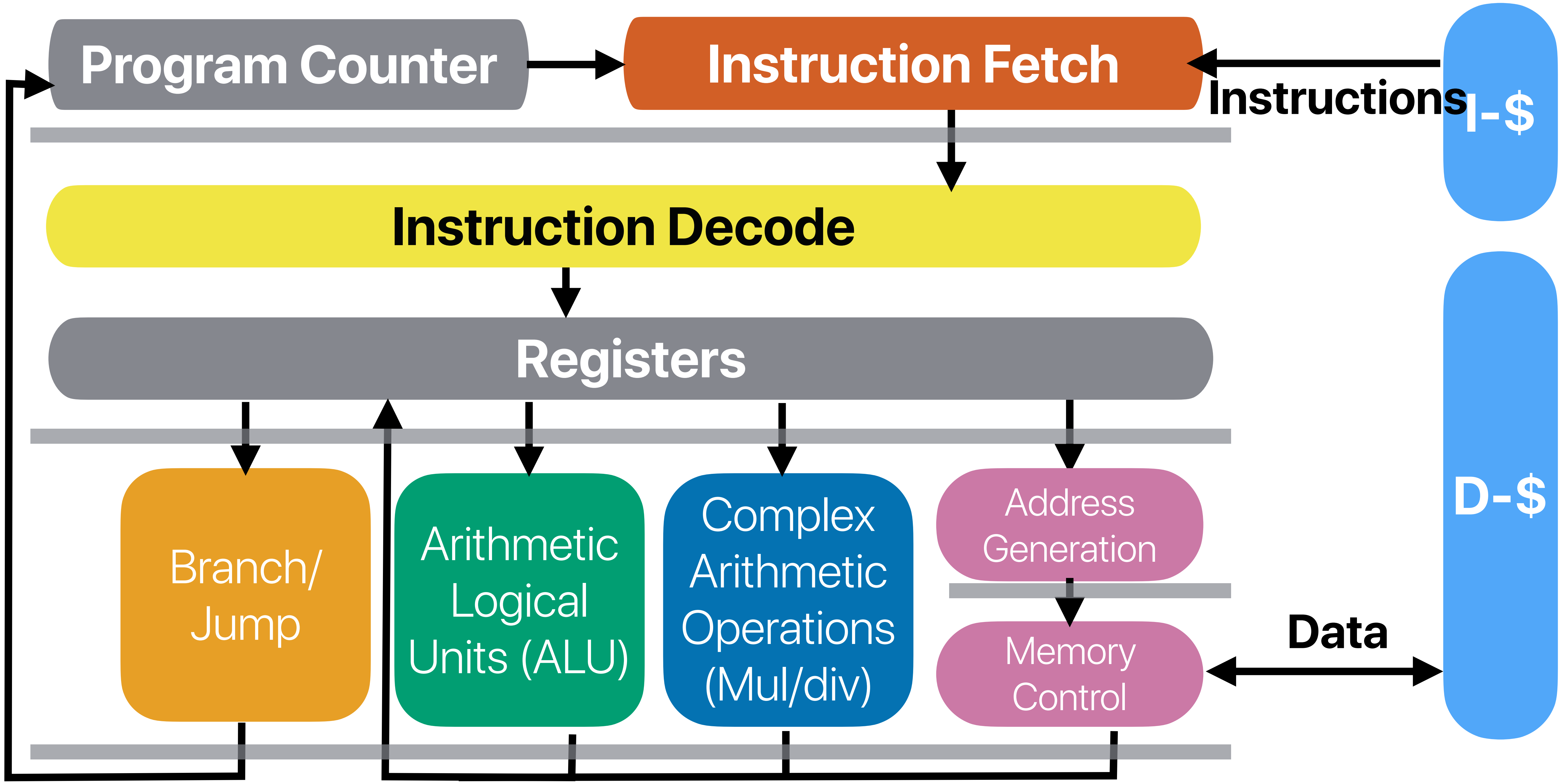
- The memory unit can only accept/perform one request each cycle

① `xorl %eax, %eax`	IF	ID	EX	WB	
② `movl (%rdi), %ecx`		IF	ID	AG	MEM
③ `addl %ecx, %eax`			IF	ID	EX
④ `addq $4, %rdi`				IF	ID
⑤ `cmpq %rdx, %rdi`					IF

"Split L1" cache!



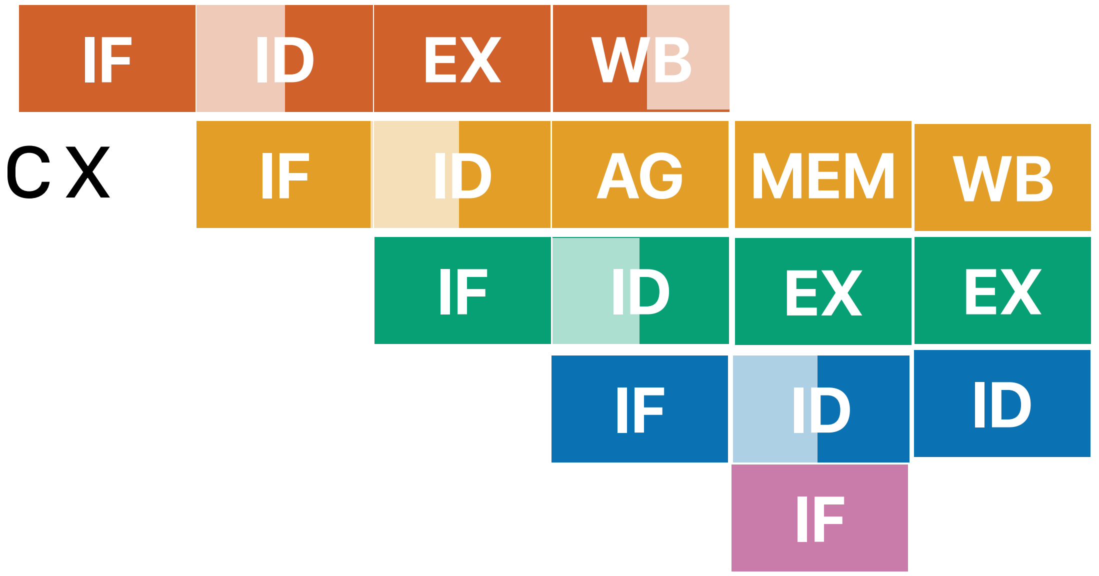
Split L1-$


Program Counter
Instruction Fetch
Instructions
I-$
Instruction Decode
Registers
D-$
Branch/ Jump
Arithmetic Logical Units (ALU)
Complex Arithmetic Operations (Mul/div)
Address Generation
Memory Control
Data

Both (2) and (3) want to "WB"

- 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
⑤ cmpq %rdx, %rdi
```



(3) has to stall

Structural Hazards

- Force later instructions to stall
- Improve the pipeline unit design to allow parallel execution
  - Write-first, read later register files
  - Split L1-Cache

Takeaways: pipeline processors

- Pipelining helps to improve the throughput of processors
    - Allowing shorter cycle time as each cycle only make progress for part of an instruction
    - Different pipeline stages work on different instructions concurrently
    - Theoretical CPI remains the same as single-cycle design and the throughput/speedup is in proportion to the speedup of cycle time
- Pipeline hazards prevent us from reaching the theoretical CPI
    - Structural hazards
    - Control hazards
    - Data hazards
- The most efficient approach to address structural hazards is to make the hardware available to support concurrent execution
    - Register file
    - Split caches

Control Hazards

Outline

- Dynamic Branch Predictions

How does the code look like?

```
for (unsigned i = 0; i < size; ++i) { // taken when true
     if (data[i] < threshold)         // taken when true
        call_when_true(&a[i]);
     else
        call_when_false(&a[i]);
 }
```

Branch taken simply means we are using branch target address as the next address

```
.LFB16:
   endbr64
   testl %esi, %esi
   jle    .L10
   movslq   %esi, %rsi
   pushq %r12
   leaq  (%rdi,%rsi,8), %r12
   pushq %rbp
   movslq   %edx, %rbp
   pushq %rbx
   movq  %rdi, %rbx
   jmp    .L5
   .p2align 4,,10
   .p2align 3
.L15:
   call  call_when_true@PLT
   addq  $8, %rbx
```

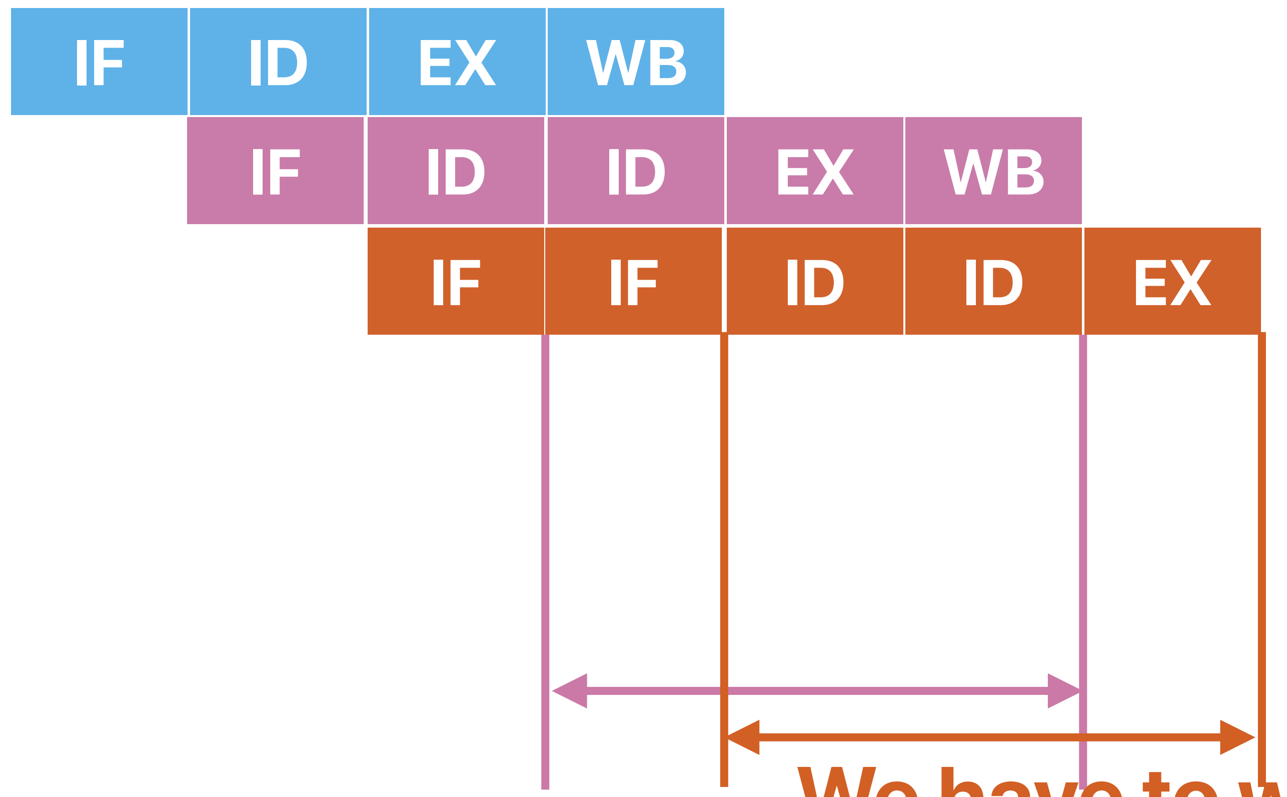
```
   cmpq  %r12, %rbx
   je .L14
.L5:
   movq  %rbx, %rdi
   cmpq  %rbp, (%rbx)
   jl .L15
   call  call_when_false@PLT
   addq  $8, %rbx
   cmpq  %r12, %rbx
   jne    .L5
.L14:
   popq  %rbx
   xorl  %eax, %eax
   popq  %rbp
   popq  %r12
   ret
```

Branch taken

Branch taken

Why is “branch” problematic in performance?

```
① addq $8, %rbx
② cmpq %r12, %rbx
③ jne  .L5
```

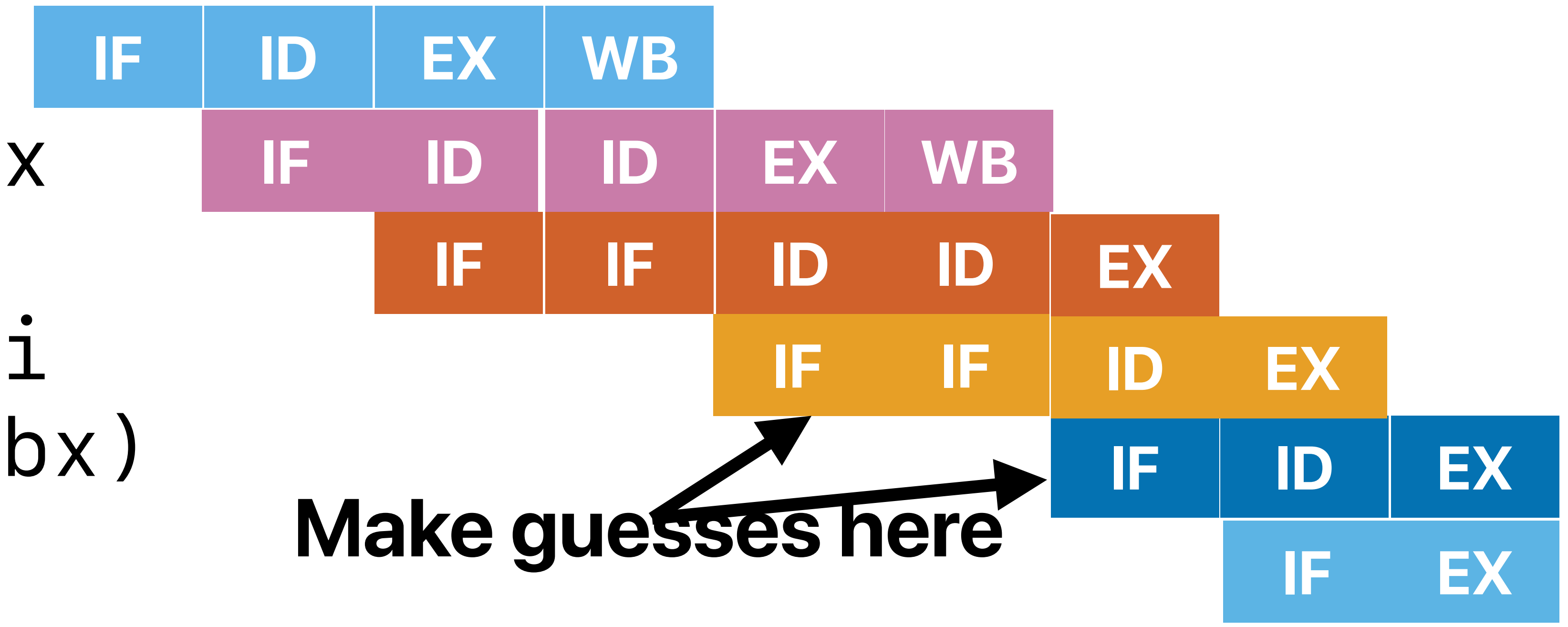


The latency of executing the cmpq instruction

We have to wait almost as long as the latency of the previous instruction to make a decision — we cannot fetch anything before that

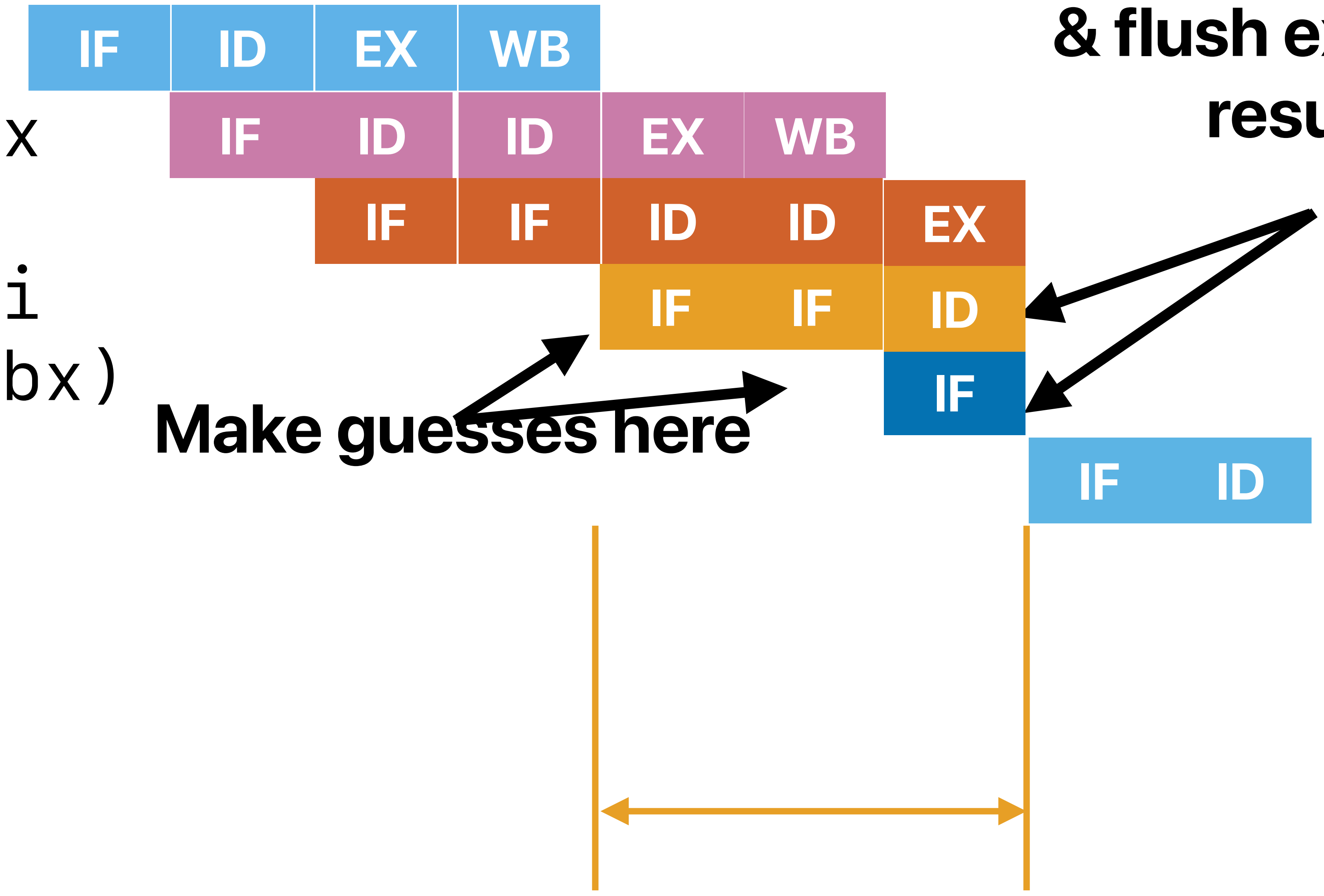
Prediction: What if we guessed right?

```
①        addq  $8, %rbx
②        cmpq  %r12, %rbx
③        jne   .L5
④ .L5:   movq  %rbx, %rdi
⑤        cmpq  %rbp, (%rbx)
```



Prediction: What if we are wrong?

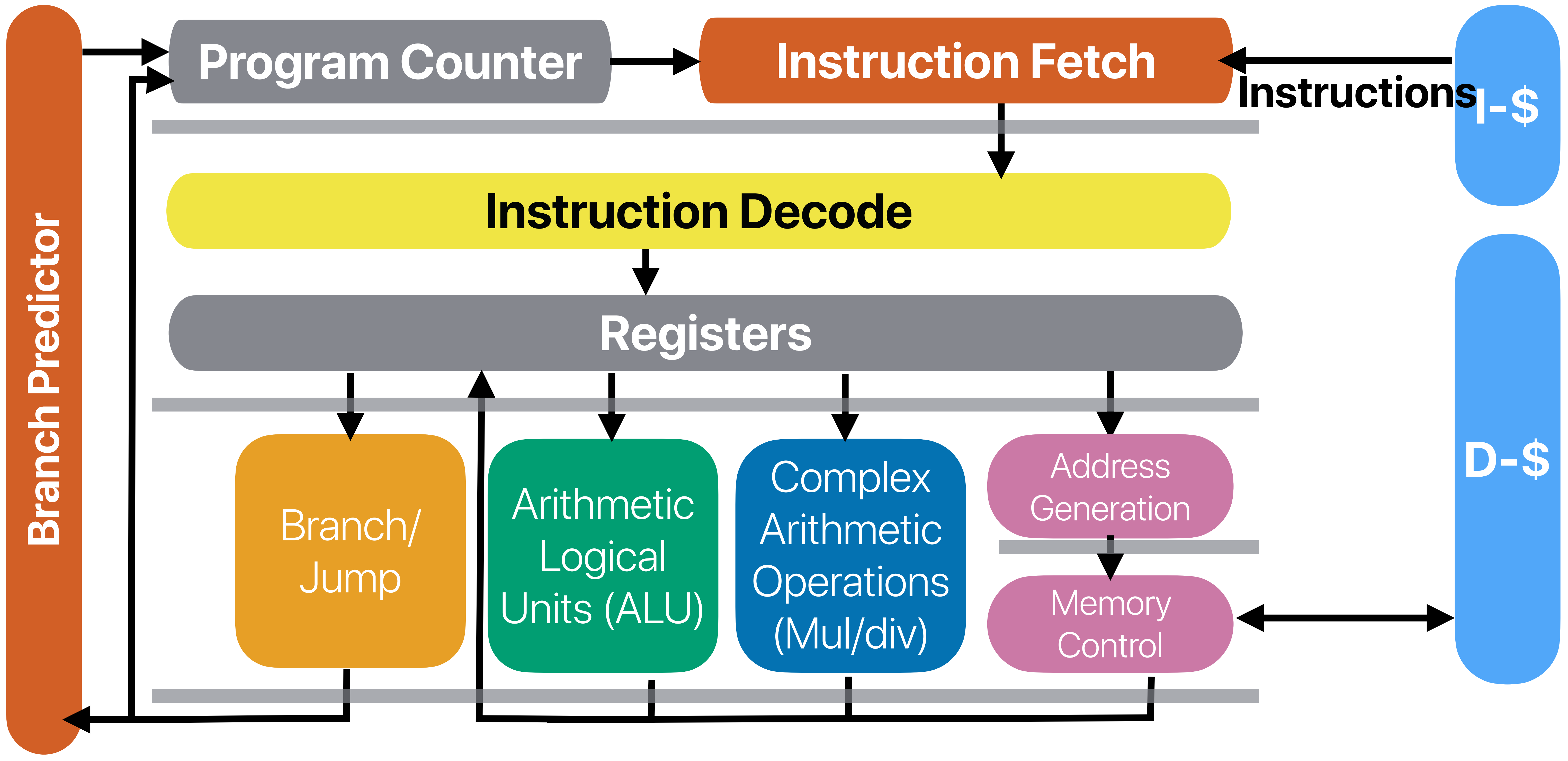
```
① .        addq  $8, %rbx
② .        cmpq  %r12, %rbx
③ .        jne   .L5
④ .L5:     movq  %rbx, %rdi
⑤ .        cmpq  %rbp, (%rbx)
⑥ .L14:    popq  %rbx
```



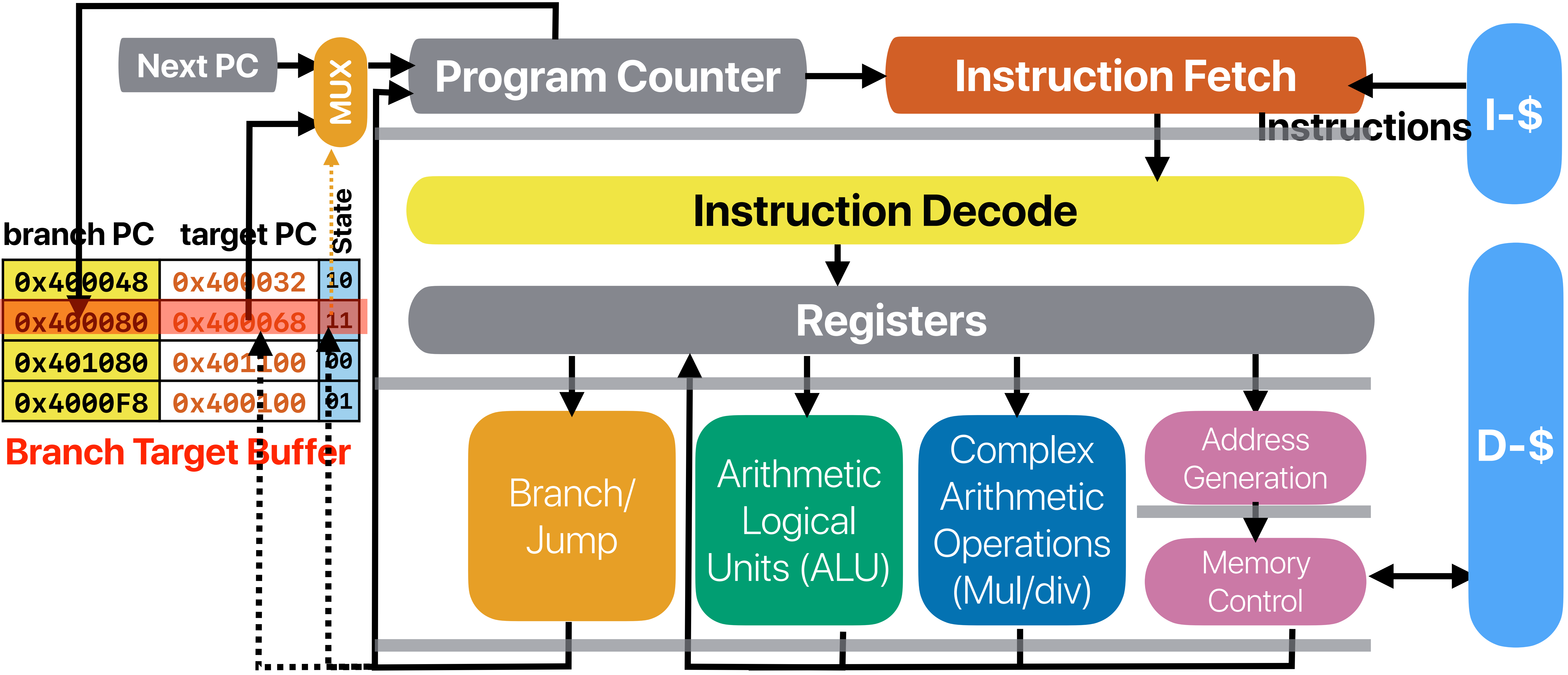
Bubble the rest stages & flush executed results

We still only wasted three cycles — same as not doing anything

Microprocessor with a “branch predictor”



Detail of a basic dynamic branch predictor



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

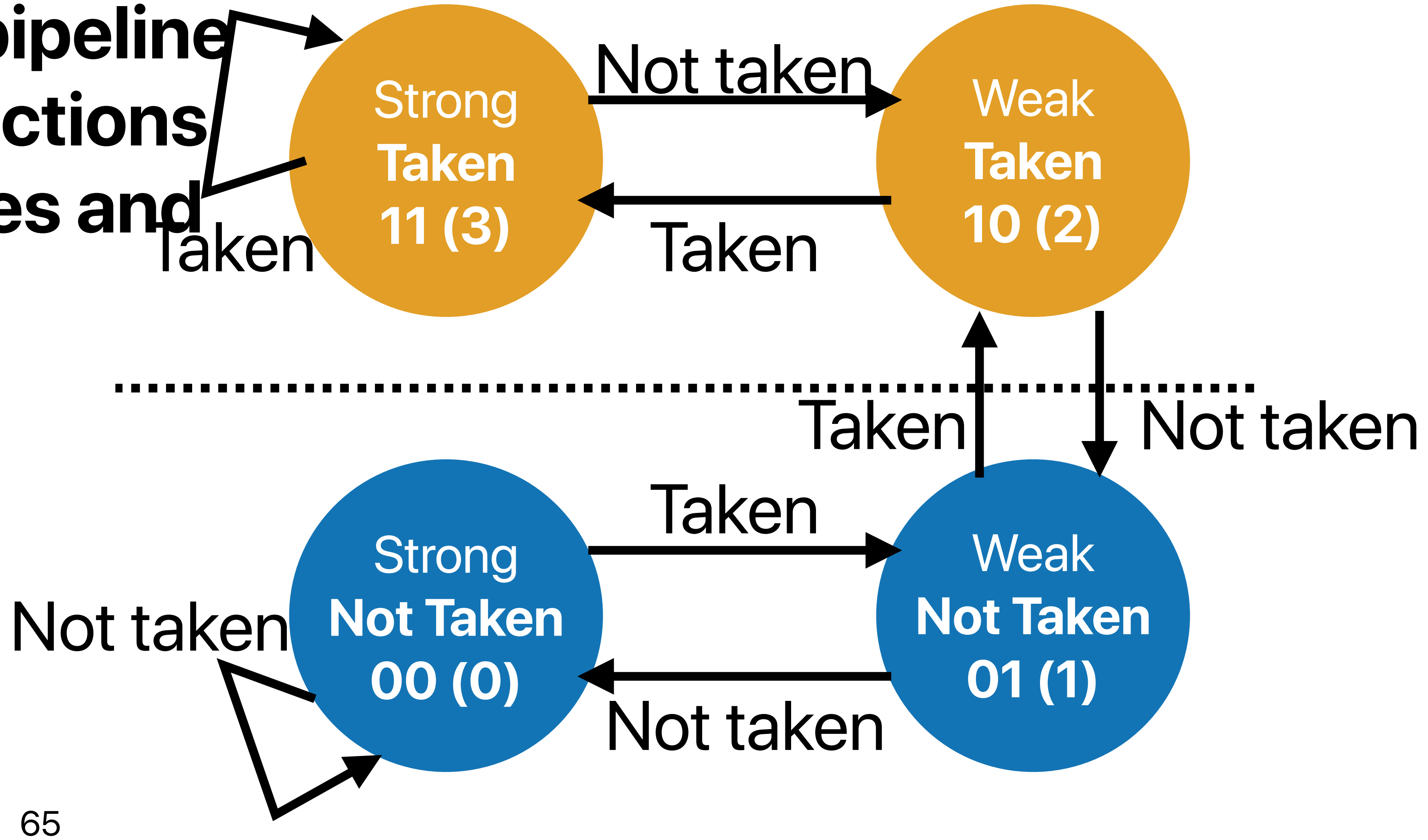
Branch Target Buffer

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**

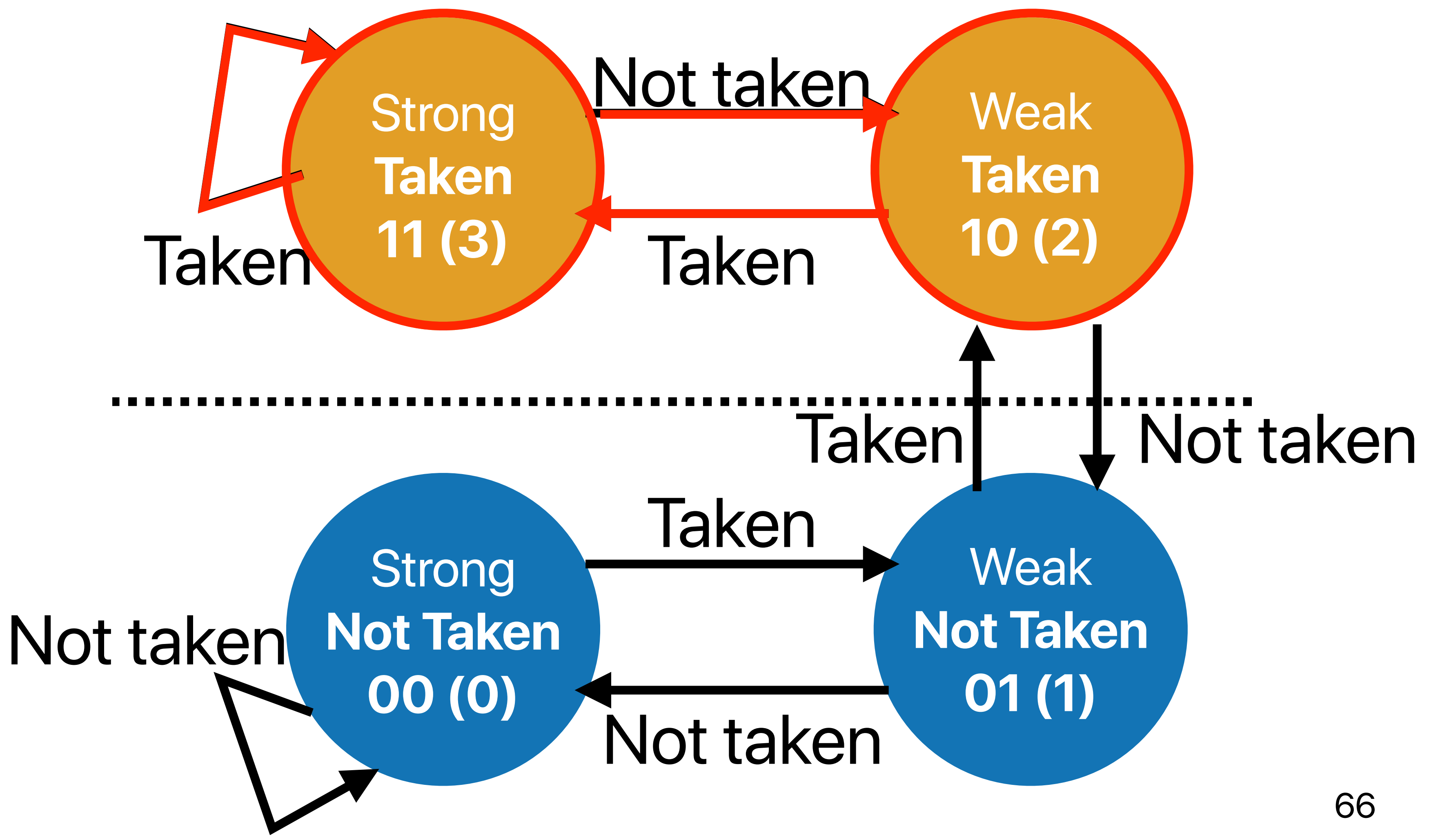
branch PC	target PC	State
0x400048	0x400032	10
0x400080	0x400068	11
0x401080	0x401100	00
0x4000F8	0x400100	01

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!

Demo revisited: evaluating the cost of mis-predicted branches

- Compare the number of mis-predictions
- Calculate the difference of cycles
- We can get the “average CPI” of a mis-prediction!

34 cycles!!!

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)

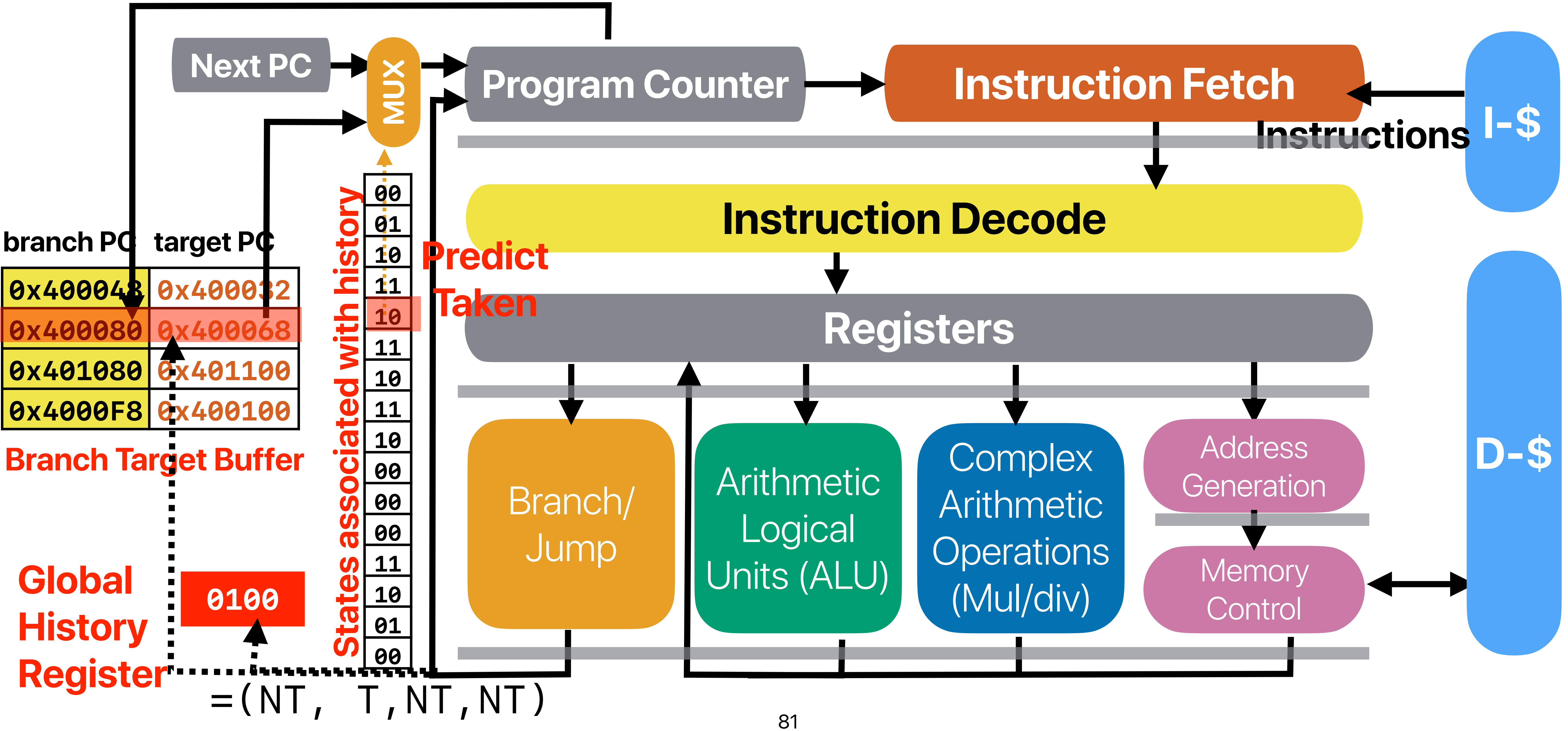
A. ~25%
B. ~33%
C. ~50%
D. ~67%
E. ~75% [✓]

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

This pattern repeats all the time!

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

Detail of a basic dynamic branch predictor



Performance of GH predictor

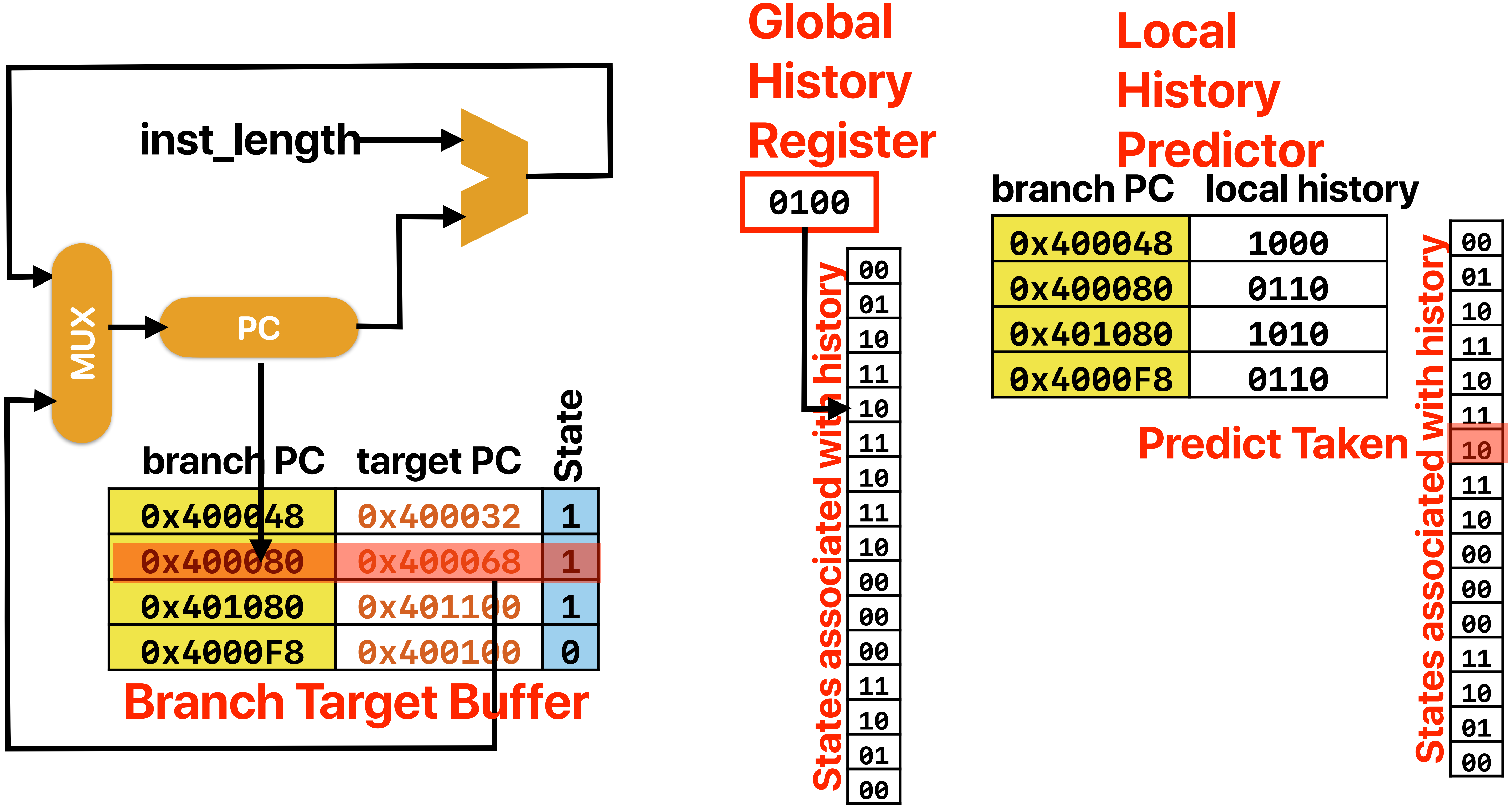
```
i = 0;
do {
    if( i % 2 != 0) // Branch X, taken if i % 2 == 0
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)// Branch Y
```

Near perfect after this

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

Hybrid predictors

Tournament Predictor



Branch Target Buffer

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

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

Predict Taken

States associated with history:

00
01
10
11
10
11
10
11
10
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
- Tournament predictor is a “hybrid predictor” as it takes both local & global information into account

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

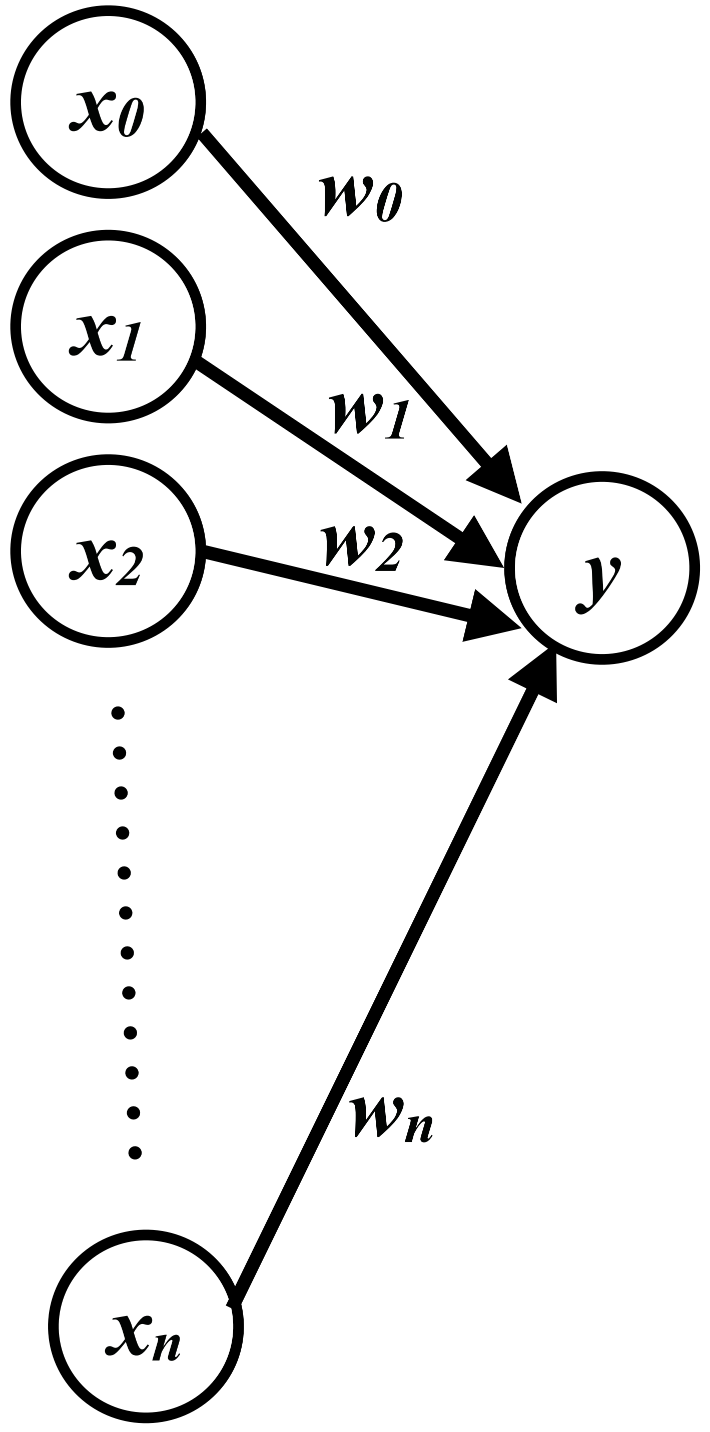
- The machine learns to predict conditional branches
- Artificial neural networks
  - Simple model of neural networks in brain cells
  - Learn to recognize and classify patterns

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

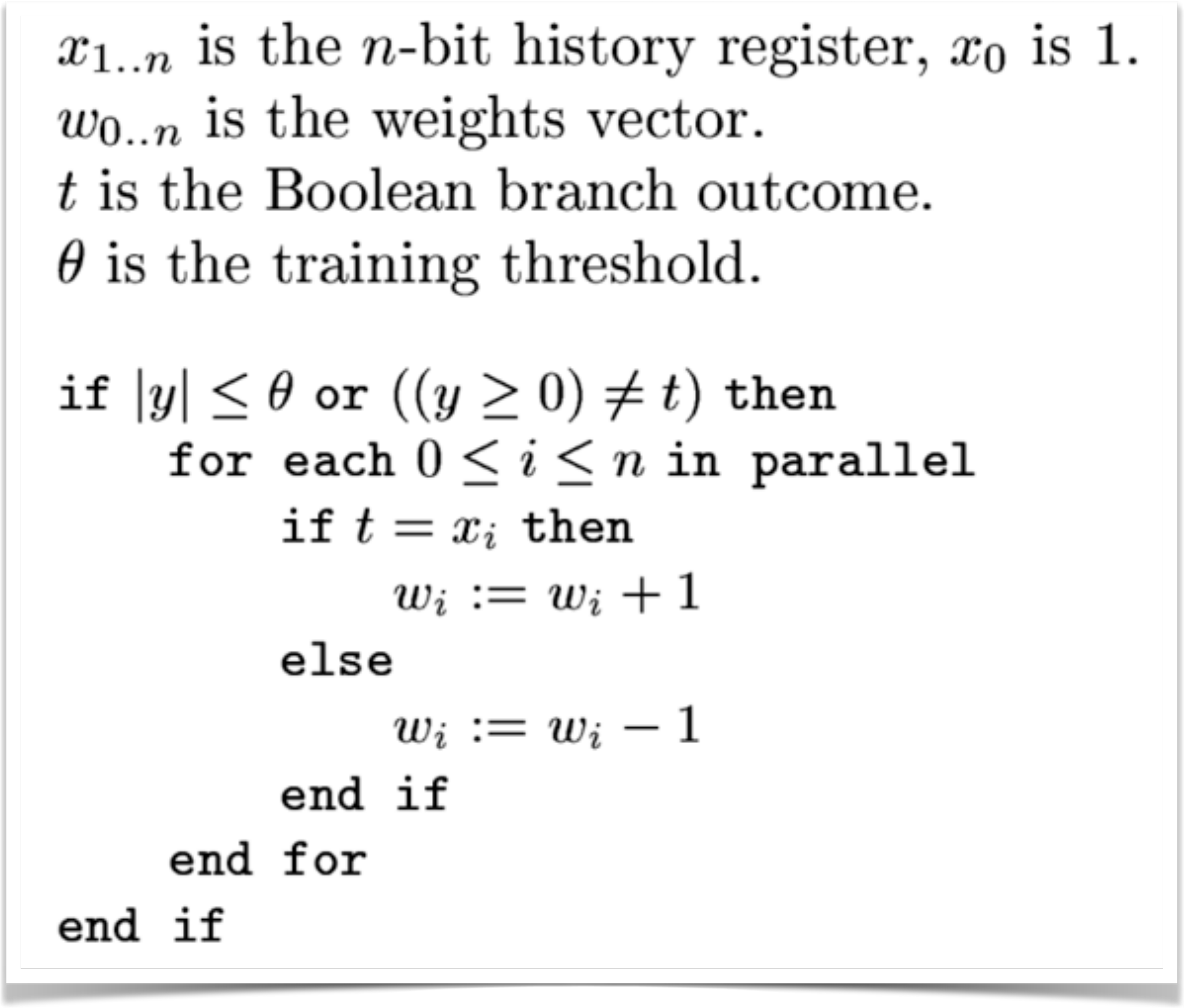
Mapping Branch Prediction to NN (cont.)

- 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



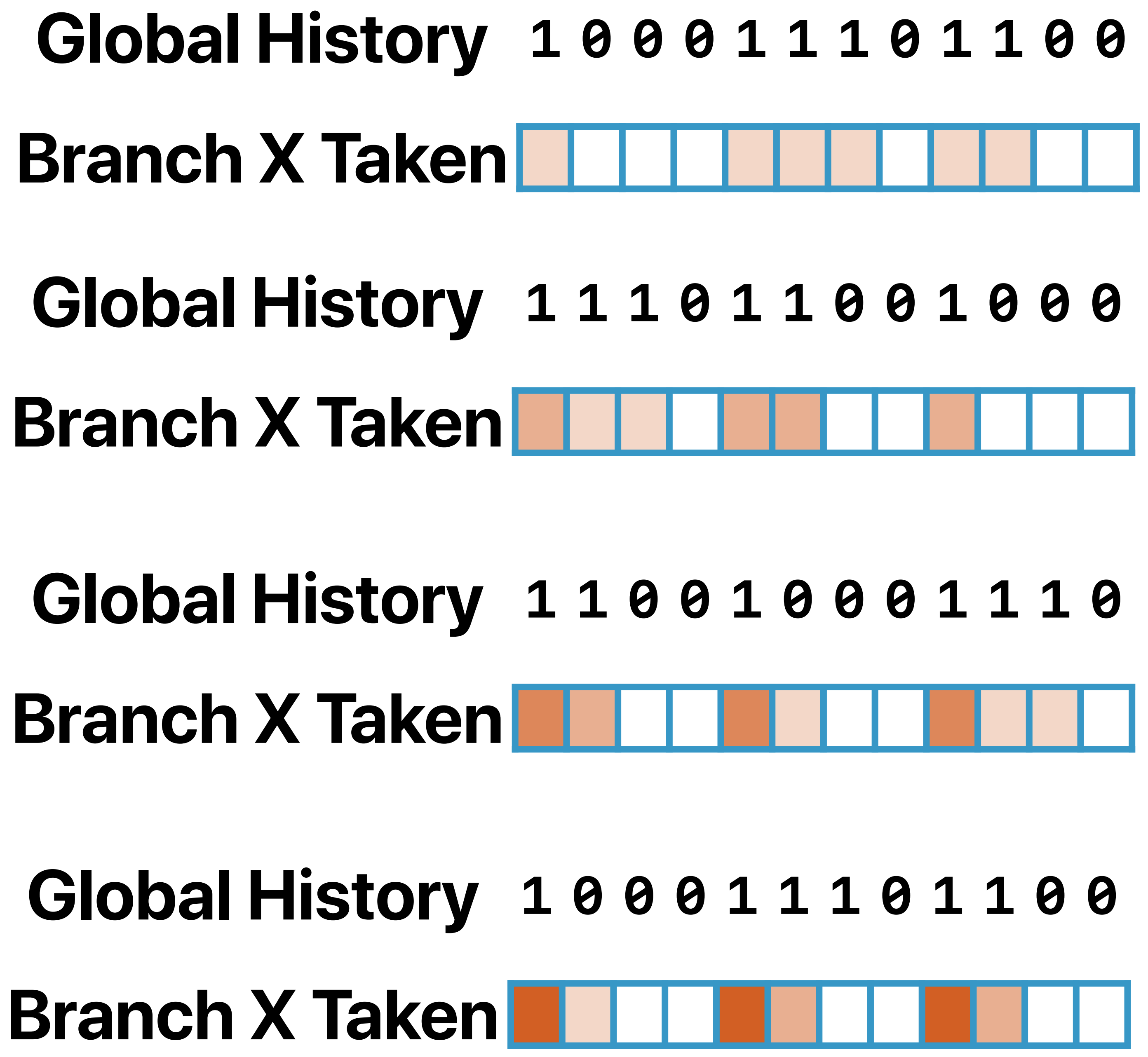
$$y = w_0 + \sum_{i=1}^{n} x_i w_i$$

Training Algorithm



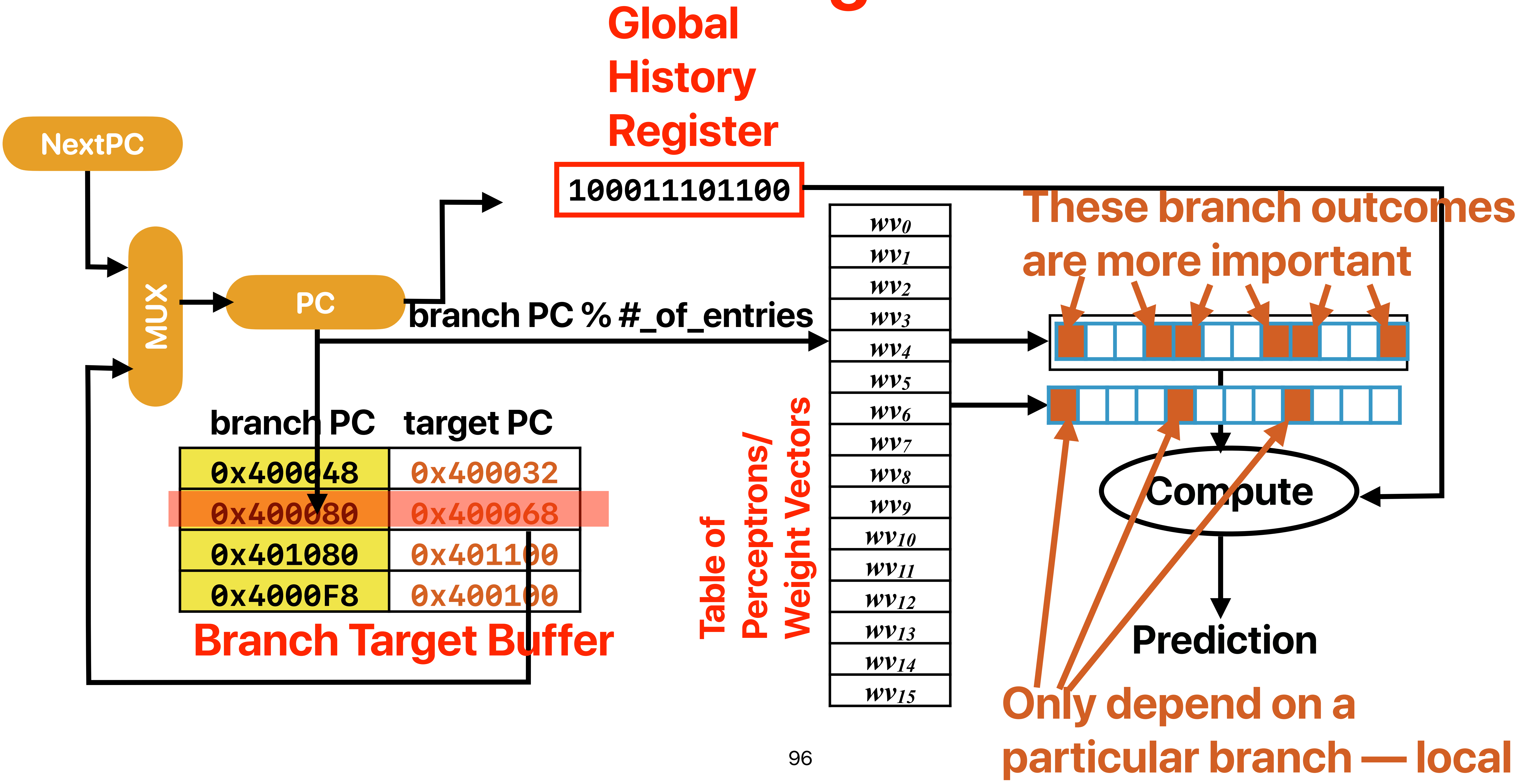
$x_{1..n}$ is the n-bit history register, x_0 is 1.
$w_{0..n}$ is the weights vector.
t is the Boolean branch outcome.
θ is the training threshold.

```
if $|y| \leq \theta$ 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
```

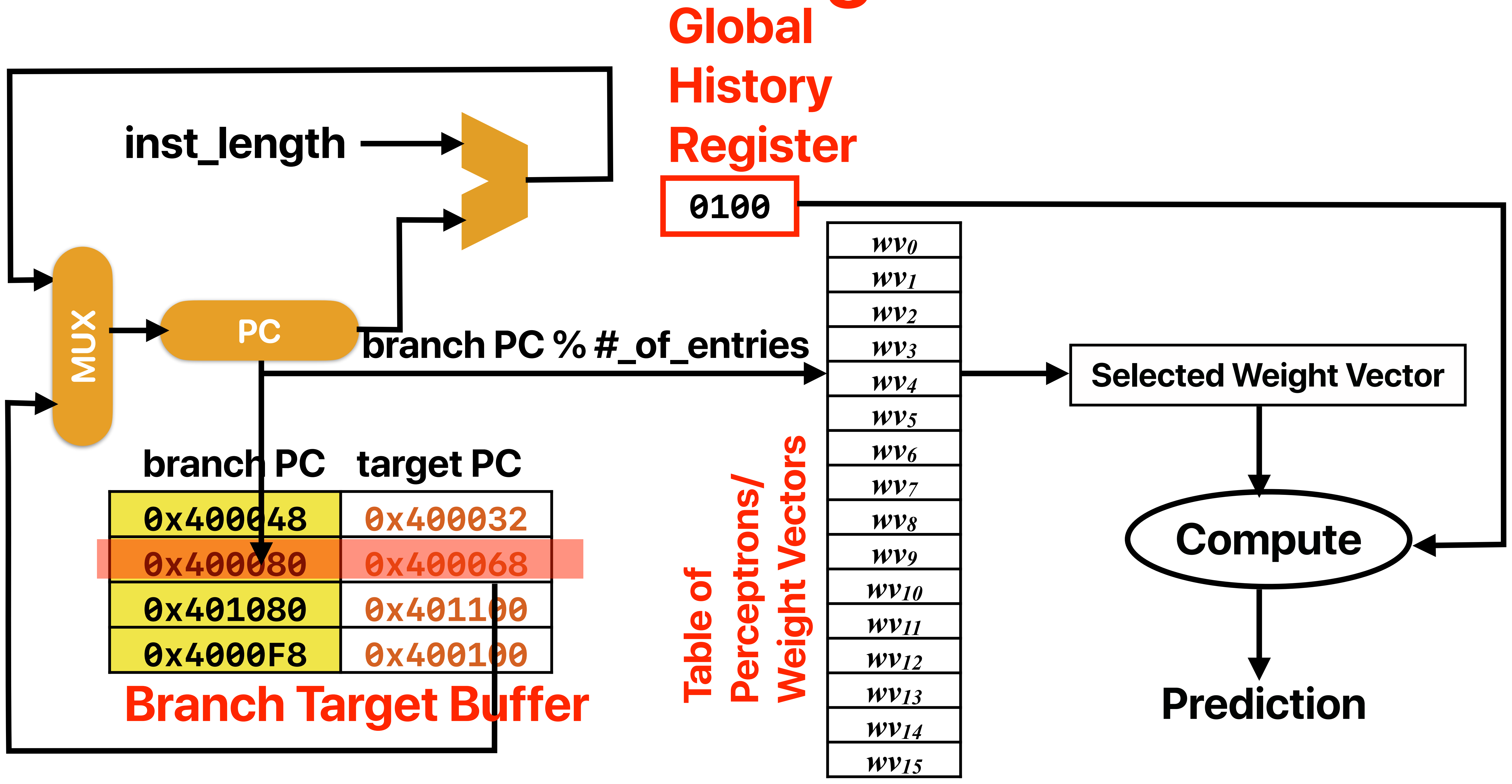


Global History 1 0 0 0 1 1 1 0 1 1 0 0
Branch X Taken
Global History 1 1 1 0 1 1 0 0 1 0 0 0
Branch X Taken
Global History 1 1 0 0 1 0 0 0 1 1 1 0
Branch X Taken
Global History 1 0 0 0 1 1 1 0 1 1 0 0
Branch X Taken

Predictor Organization



Predictor Organization



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

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.

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;
}
```

Computer
Science &
Engineering

142

つづく