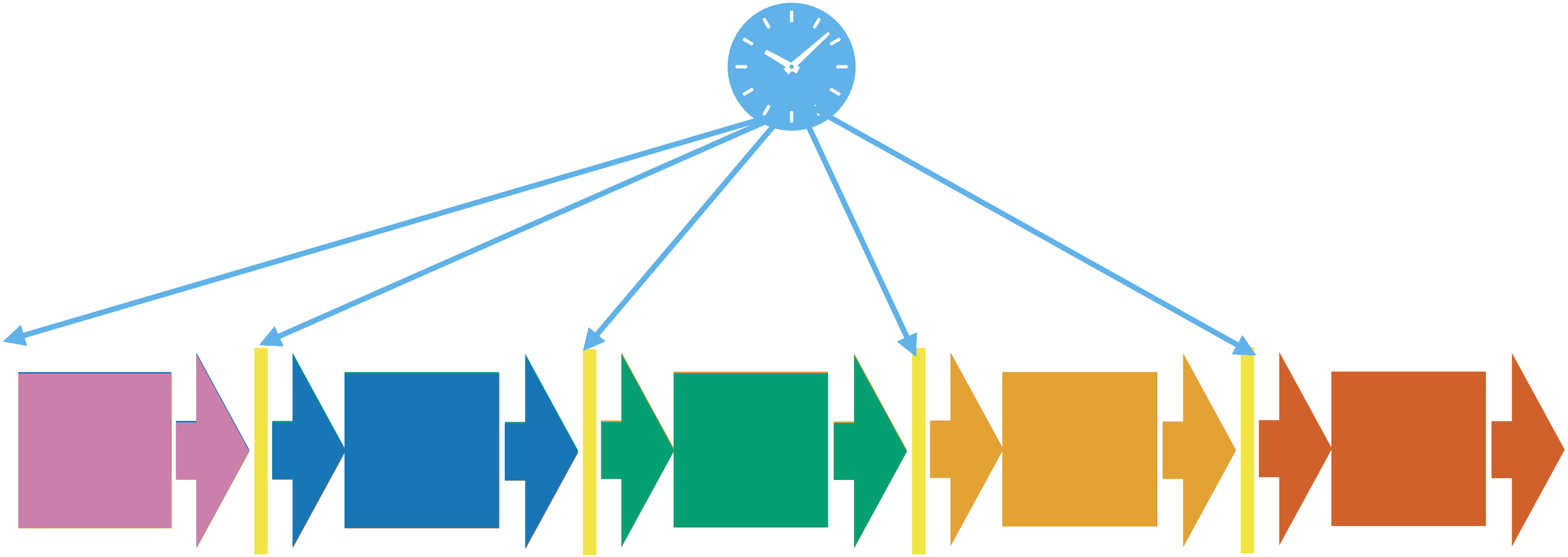
Data Hazards & Dynamic Instruction Scheduling (I)

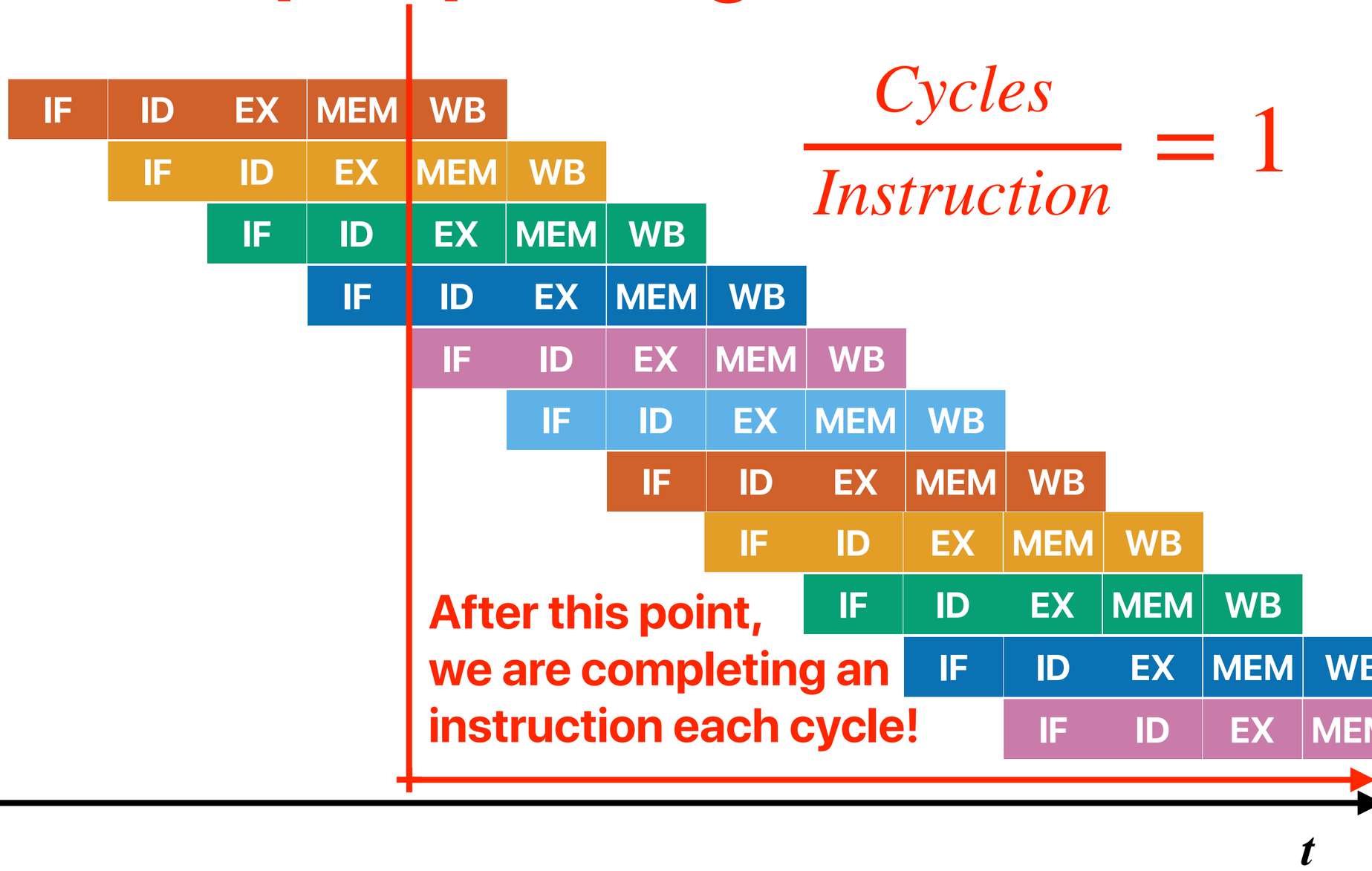
Hung-Wei Tseng

Recap: Pipelining



Recap: Pipelining

```
add x1, x2, x3
ld  x4, 0(x5)
sub x6, x7, x8
sub x9,x10,x11
sd  x1, 0(x12)
xor x13,x14,x15
and x16,x17,x18
add x19,x20,x21
sub x22,x23,x24
ld  x25, 4(x26)
sd  x27, 0(x28)
```



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

Recap: Three 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

Recap: addressing hazards

- Structural hazards
  - Stall
  - Modify hardware design
- Control hazards
  - Stall
  - Static prediction
  - Dynamic prediction

Outline

- Data hazards
  - Data forwarding
- SuperScalar
- Out-of-order, Dynamic instruction scheduling

Data hazards

Data hazards

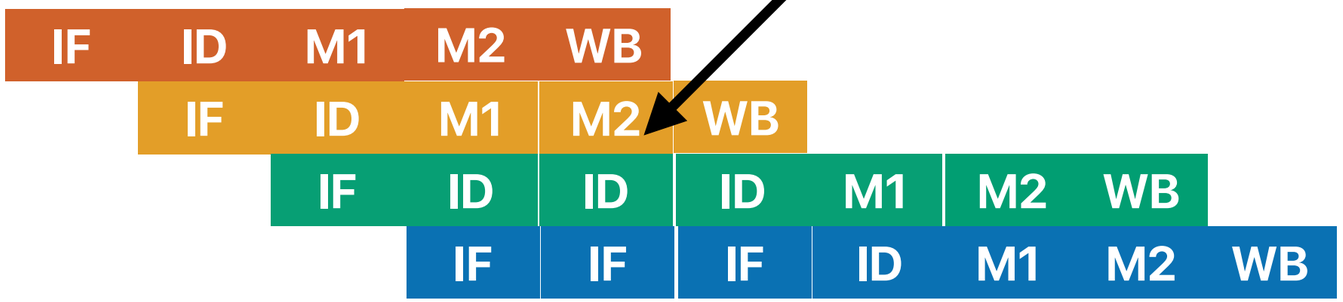
- An instruction currently in the pipeline cannot receive the "logically" correct value for execution
- Data dependencies
  - The output of an instruction is the input of a later instruction
  - May result in data hazard if the later instruction that consumes the result is still in the pipeline

Solution 1: Let's try "stall" again

- Whenever the input is not ready when the consumer is decoding, just stall — the consumer stays at ID.

we have the value for %edx already! Why another cycle?

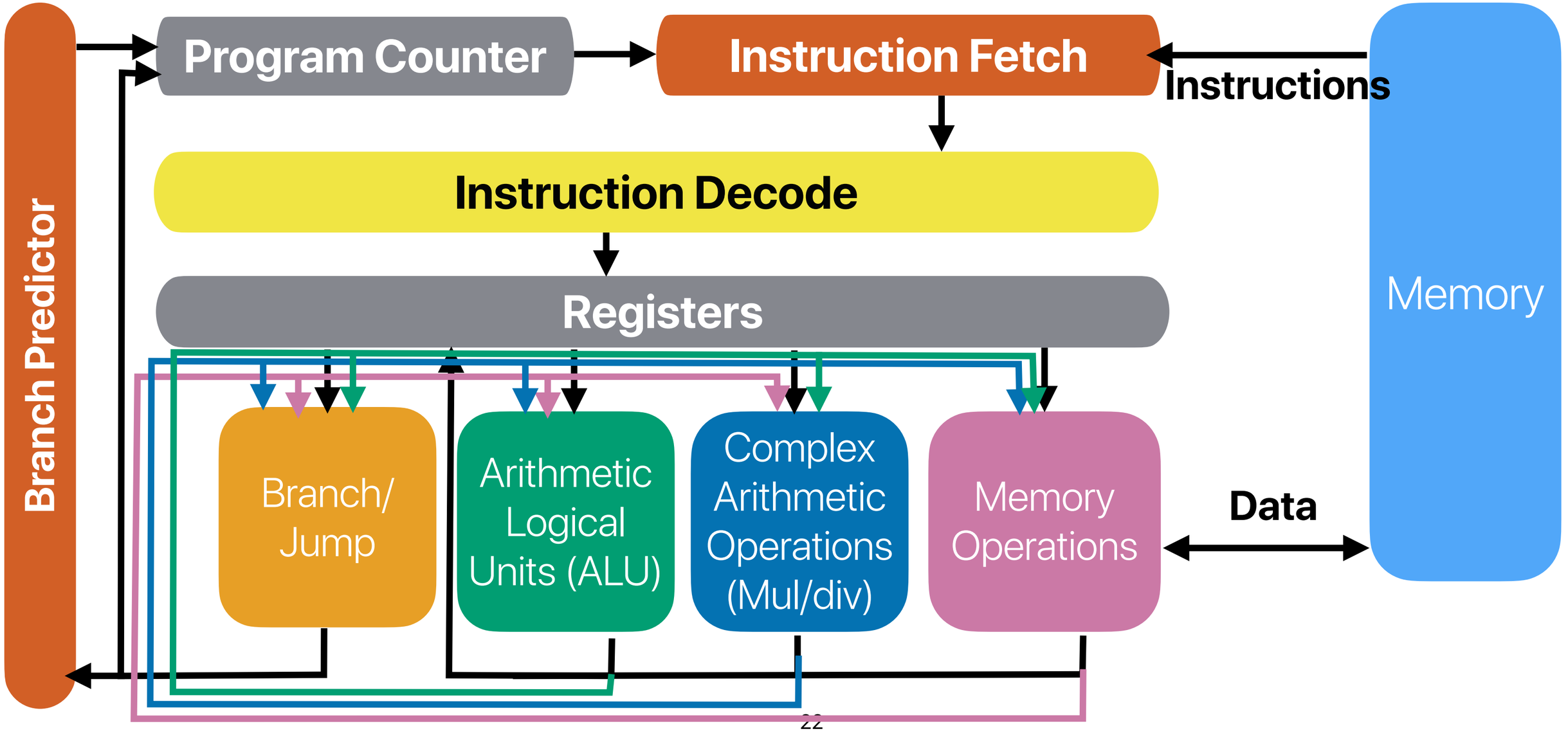
```
movl    (%rdi), %eax
movl    (%rsi), %edx
movl    %edx, (%rdi)
movl    %eax, (%rsi)
```



Solution 2: Data forwarding

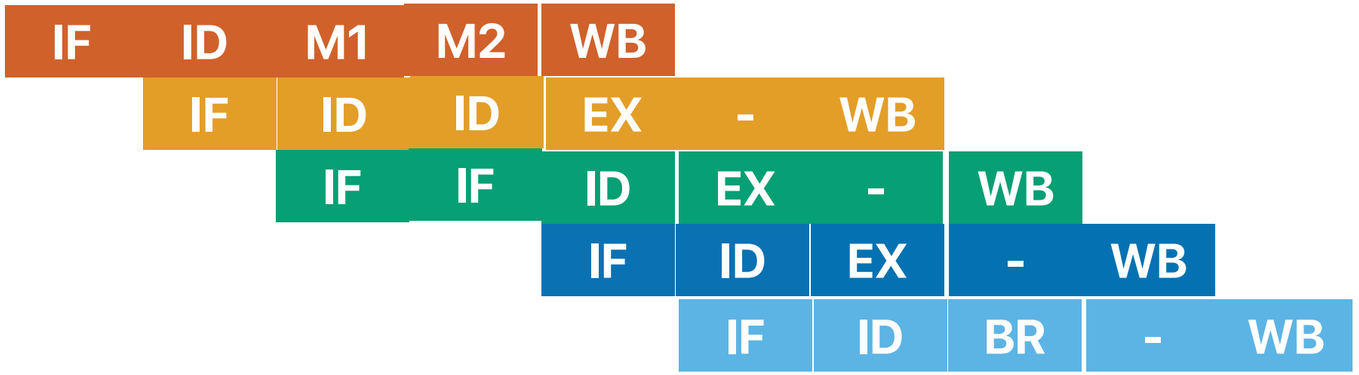
- Add logics/wires to forward the desired values to the demanding instructions

Data “forwarding”



Single pipeline

```
    for(i = 0; i < count; i++) {
        s += a[i];
    }
.L3:
        movl    (%rdi), %ecx
        addl    %ecx, %eax
        addq    $4, %rdi
        cmpq    %rdx, %rdi
        jne     .L3
        ret
```



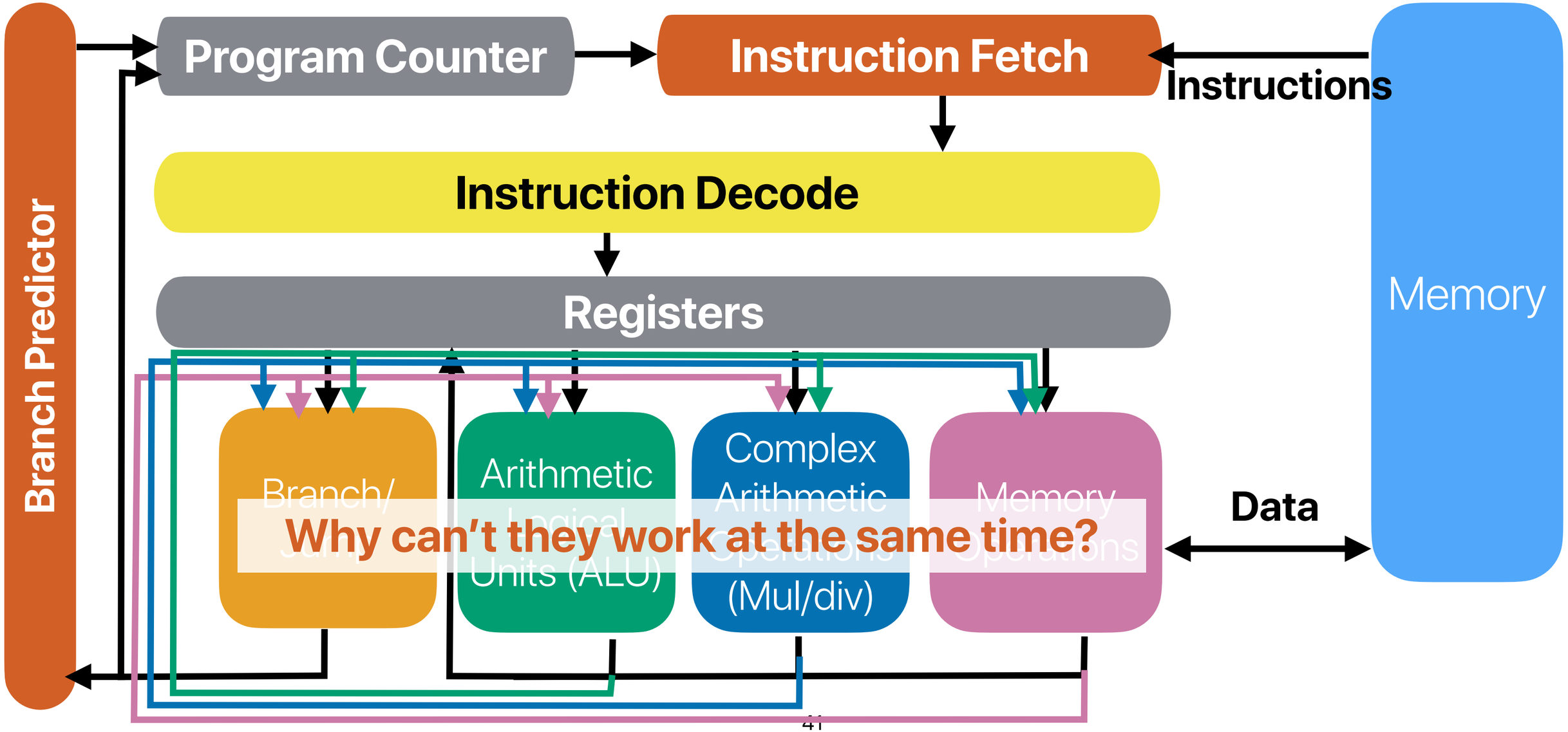
Outline

- Make CPI go below 1 — Super Scalar
- Dynamic Out of Order execution
- Modern processor design

Outline

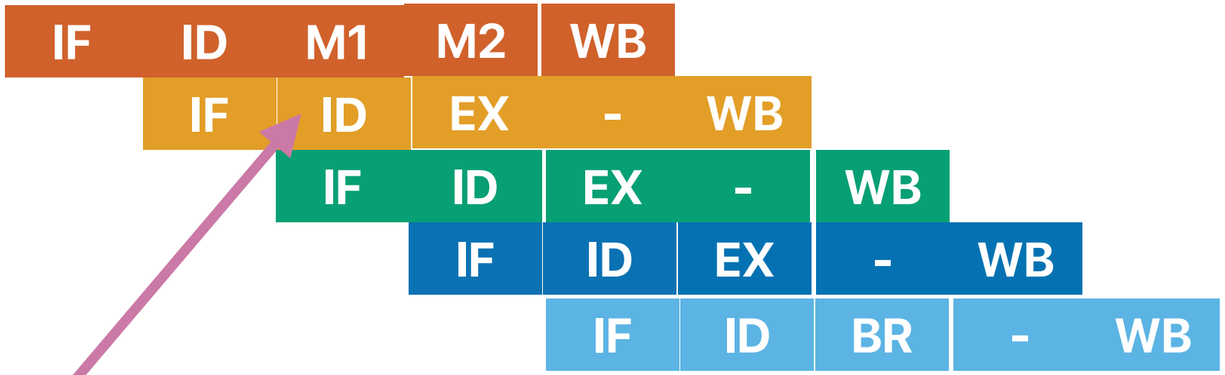
- Make CPI go below 1 — Super Scalar
- Dynamic Out of Order execution
- Modern processor design

Data “forwarding”



Compiler optimization

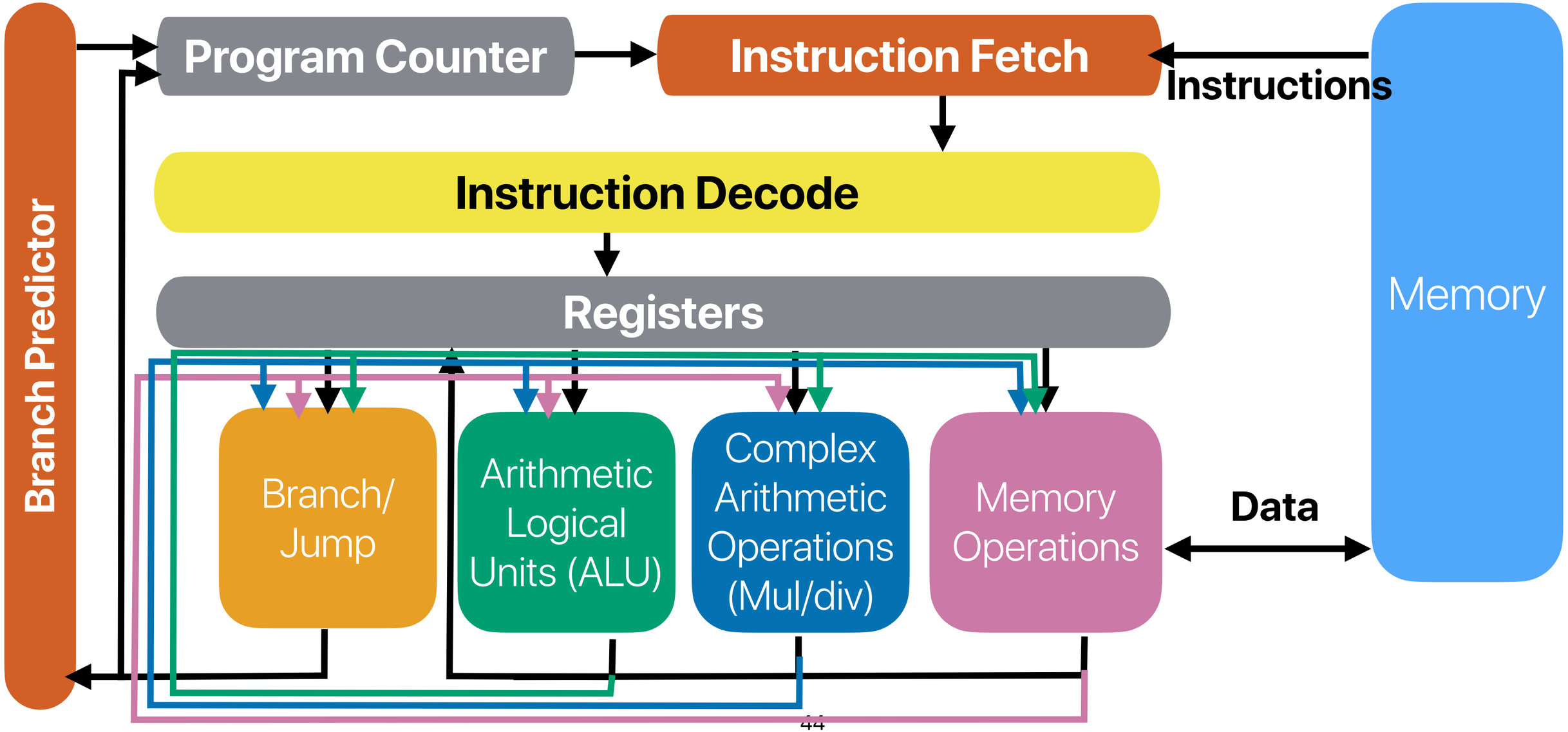
```
    for(i = 0; i < count; i++) {
        s += a[i];
    }
.L3:
        movl    (%rdi), %ecx
        addq    $4, %rdi
        addl    %ecx, %eax
        cmpq    %rdx, %rdi
        jne     .L3
        ret
```



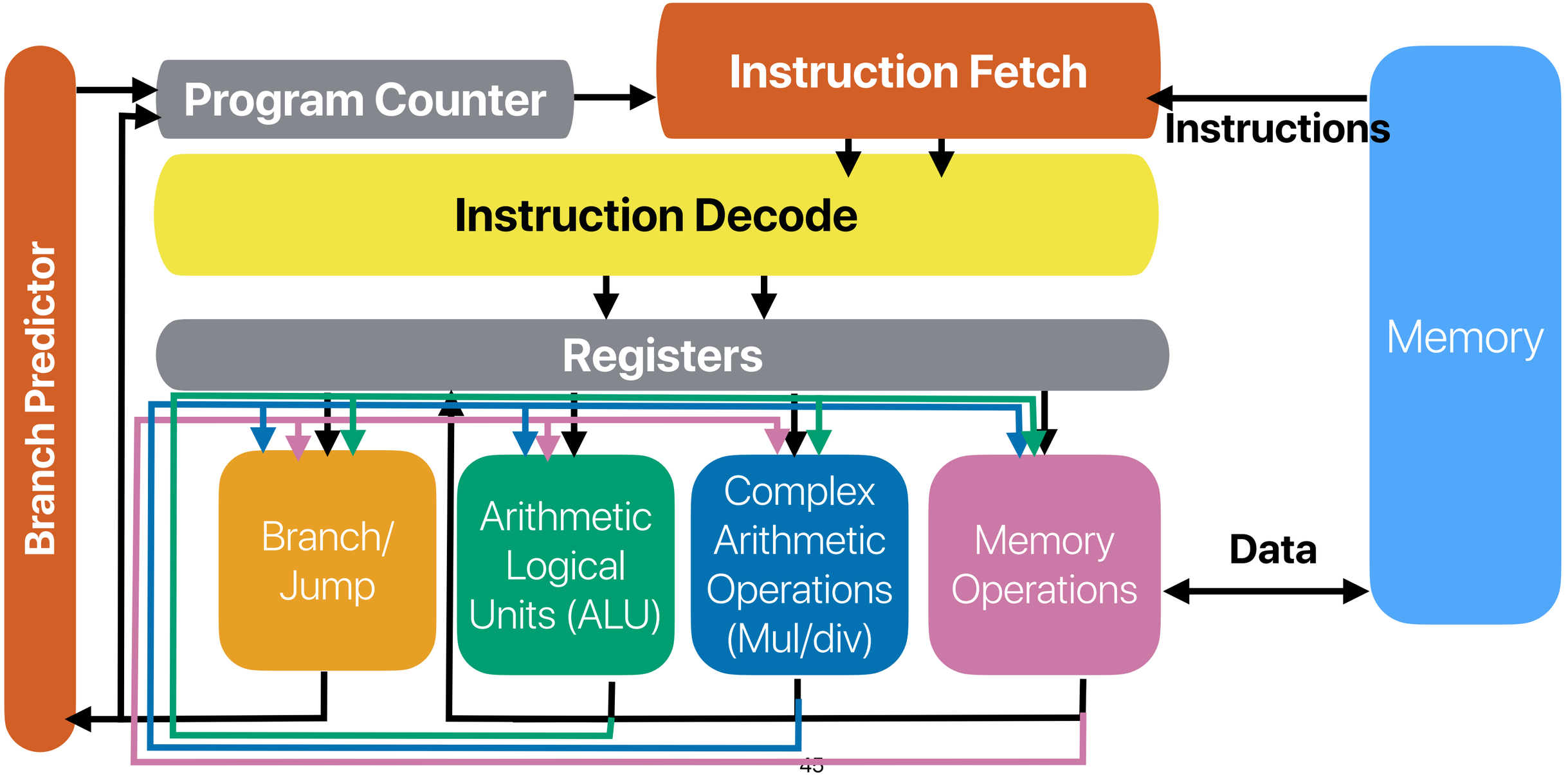
addq is not depending on movl and ALU is free! can we execute them together?

If CPI==1 the limitation?

Data “forwarding”



Super Scalar



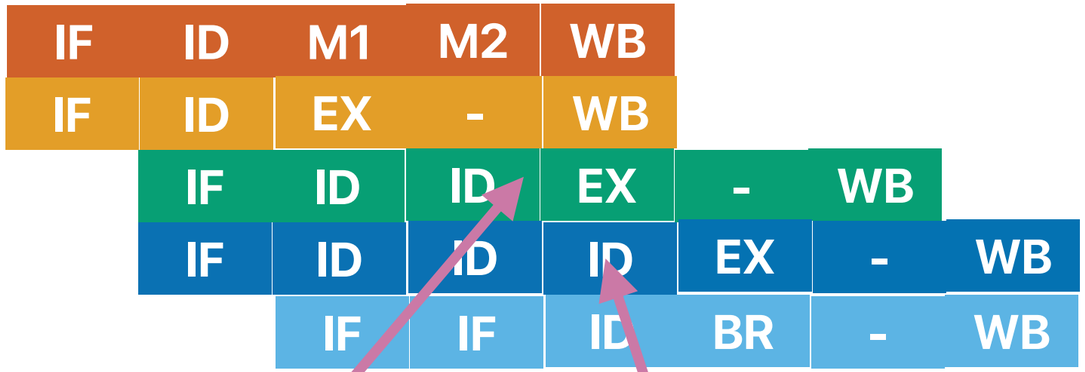
Super Scalar

Superscalar

- Since we have many functional units now, we should fetch/decode more instructions each cycle so that we can have more instructions to issue!
- Super-scalar: fetch/decode/issue more than one instruction each cycle
  - **Fetch width:** how many instructions can the processor fetch/decode each cycle
  - **Issue width**: how many instructions can the processor issue each cycle

Superscalar: fetch/issue width == 2

```
    for(i = 0; i < count; i++) {
        s += a[i];
    }
.L3:
        movl    (%rdi), %ecx
        addq    $4, %rdi
        addl    %ecx, %eax
        cmpq    %rdx, %rdi
        jne     .L3
        ret
```

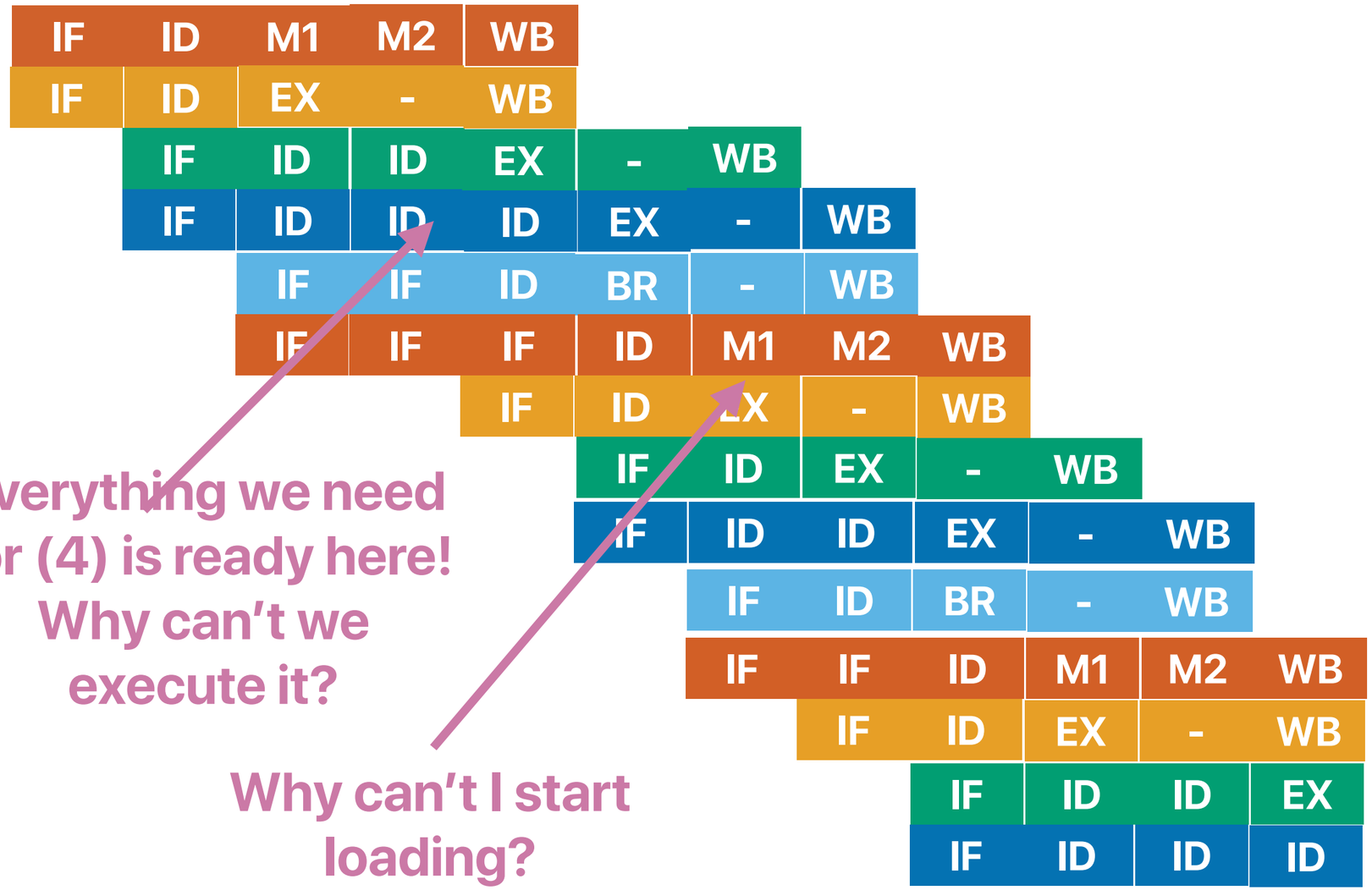


Stall because %ecx is not ready

Stall because we have only one ALU (structural hazard)

If we loop many times (assume perfect predictor)

```
① movl    (%rdi), %ecx
② addq    $4, %rdi
③ addl    %ecx, %eax
④ cmpq    %rdx, %rdi
⑤ jne     .L3
⑥ movl    (%rdi), %ecx
⑦ addq    $4, %rdi
⑧ addl    %ecx, %eax
⑨ cmpq    %rdx, %rdi
⑩ jne     .L3
⑪ movl    (%rdi), %ecx
⑫ addq    $4, %rdi
⑬ addl    %ecx, %eax
⑭ cmpq    %rdx, %rdi
⑮ jne     .L3
```



Limitations of Compiler Optimizations

- If the hardware (e.g., pipeline changes), the same compiler optimization may not be that helpful
- The compiler can only optimize on static instructions, but cannot optimize dynamic instructions

What do you need to execution an instruction?

- Whenever the instruction is decoded — put decoded instruction somewhere
- Whenever the inputs are ready — **all data dependencies are resolved**
- Whenever the target functional unit is available

Dynamic instruction scheduling/ Out-of-order (OoO) execution

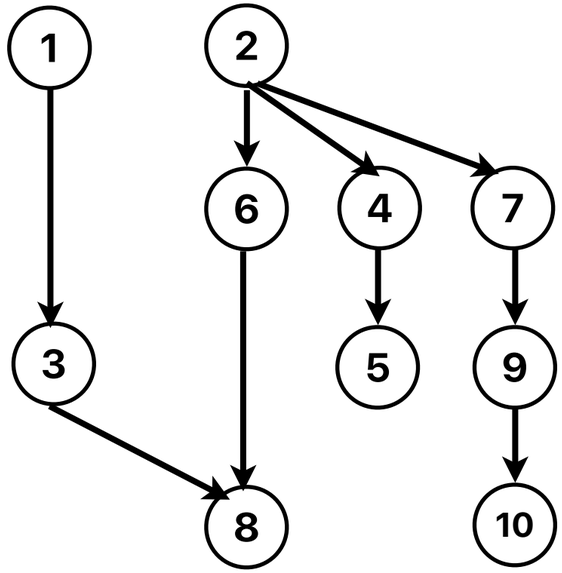
What do you need to execution an instruction?

- Whenever the instruction is decoded — put decoded instruction somewhere
- Whenever the inputs are ready — **all data dependencies are resolved**
- Whenever the target functional unit is available

Scheduling instructions: based on data dependencies

- Draw the data dependency graph, put an arrow if an instruction depends on the other.

```
① movl    (%rdi), %ecx
② addq    $4, %rdi
③ addl    %ecx, %eax
④ cmpq    %rdx, %rdi
⑤ jne     .L3
⑥ movl    (%rdi), %ecx
⑦ addq    $4, %rdi
⑧ addl    %ecx, %eax
⑨ cmpq    %rdx, %rdi
⑩ jne     .L3
```

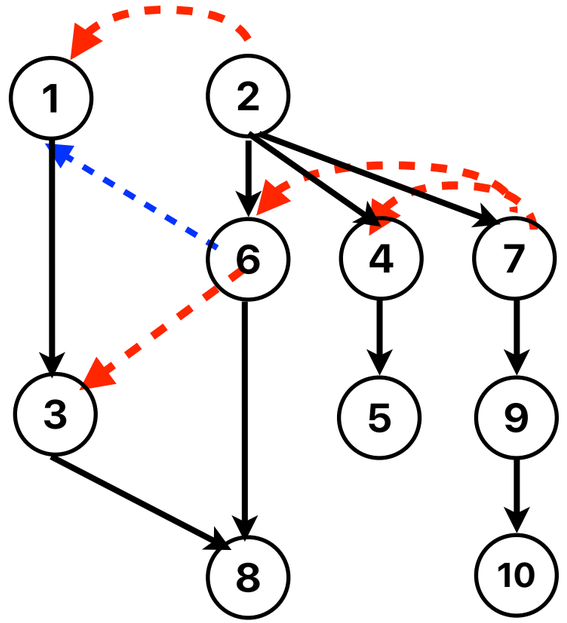


- **In theory**, instructions without dependencies can be executed in parallel or out-of-order
- Instructions with dependencies can never be reordered

False dependencies

- We are still limited by **false dependencies**
- They are not "true" dependencies because they don't have an arrow in data dependency graph
  - WAR (Write After Read): a later instruction overwrites the source of an earlier one
    - 2 and 1, 6 and 3, 7 and 4, 7 and 6
  - WAW (Write After Write): a later instruction overwrites the output of an earlier one
    - 6 and 1

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```



What if we can use more registers...

```
① movl    (%rdi), %ecx
② addq    $4, %rdi
③ addl    %ecx, %eax
④ cmpq    %rdx, %rdi
⑤ jne     .L3
⑥ movl    (%rdi), %ecx
⑦ addq    $4, %rdi
⑧ addl    %ecx, %eax
⑨ cmpq    %rdx, %rdi
⑩ jne     .L3
```

```
① movl    (%rdi), %ecx
② addq    $4, %rdi, %t0
③ addl    %ecx, %eax, %t1
④ cmpq    %rdx, %t0
⑤ jne     .L3
⑥ movl    (%t0), %t2
⑦ addq    $4, %t0, %t3
⑧ addl    %t1, %t2, %t4
⑨ cmpq    %rdx, %t3
⑩ jne     .L3
```

All false dependencies are gone!!!

Limitations of Compiler Optimizations

- If the hardware (e.g., pipeline changes), the same compiler optimization may not be that helpful
- The compiler can only optimize on static instructions, but cannot optimize dynamic instructions
- Compilers are limited by the registers an ISA provides

Register renaming + speculative execution

- K. C. Yeager, “The Mips R10000 superscalar microprocessor,” in IEEE Micro, vol. 16, no. 2, pp. 28-41, April 1996.
- R. E. Kessler, “The Alpha 21264 microprocessor,” in IEEE Micro, vol. 19, no. 2, pp. 24-36, March-April 1999.

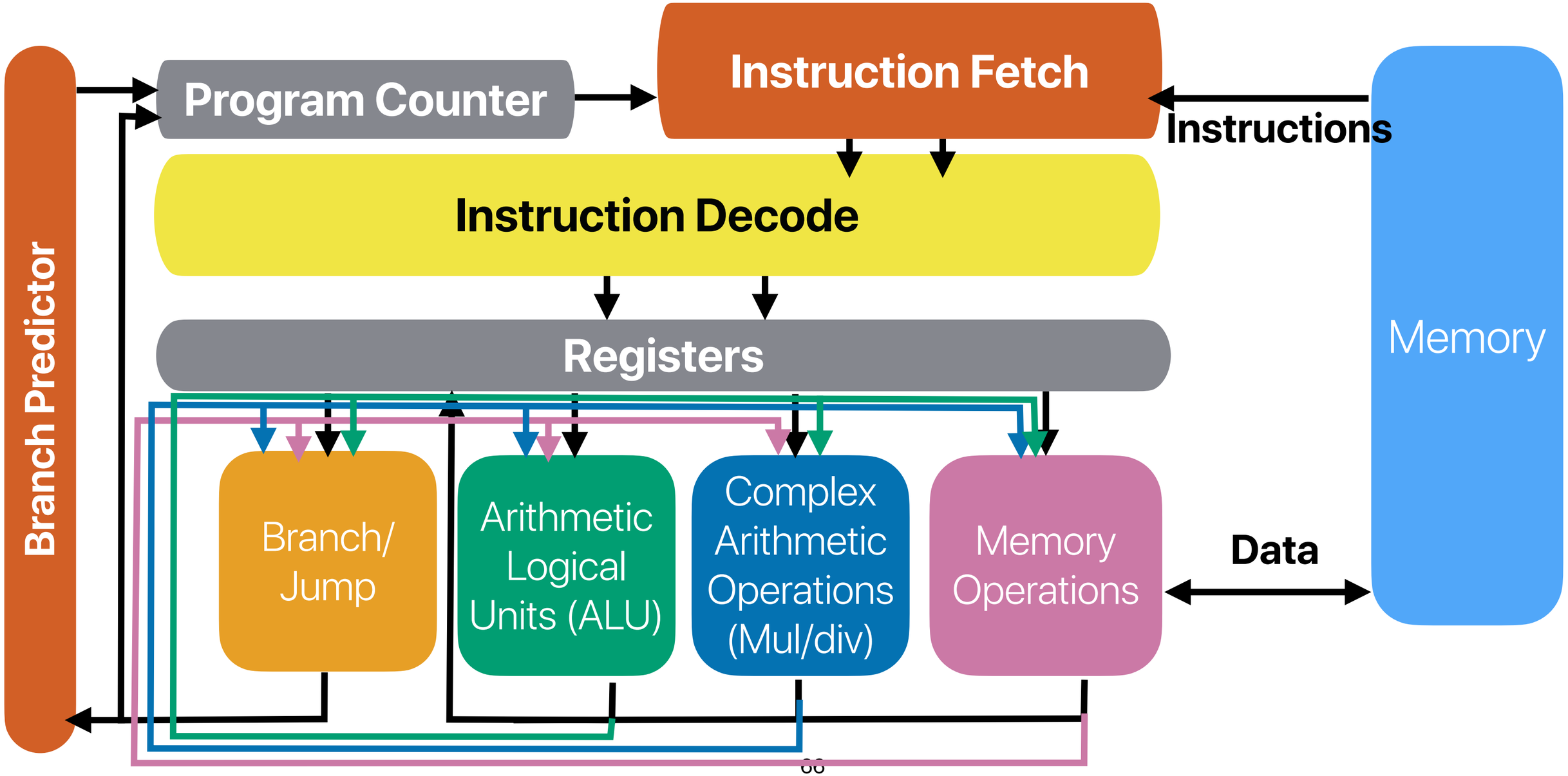
Register renaming

- Provide a set of **physical registers** and a mapping table mapping **architectural registers** to physical registers
  - Architectural registers are virtual registers that software can see/use
- Allocate a physical register for a new output
- Stages
  - Dispatch/Rename (REN) — allocate a "physical register" for the output of a decoded instruction
  - Execute (EX, M1/M2, BR) — send the instruction to its corresponding pipeline if no structural hazards
  - Write Back (WB) — broadcast the result through CDB

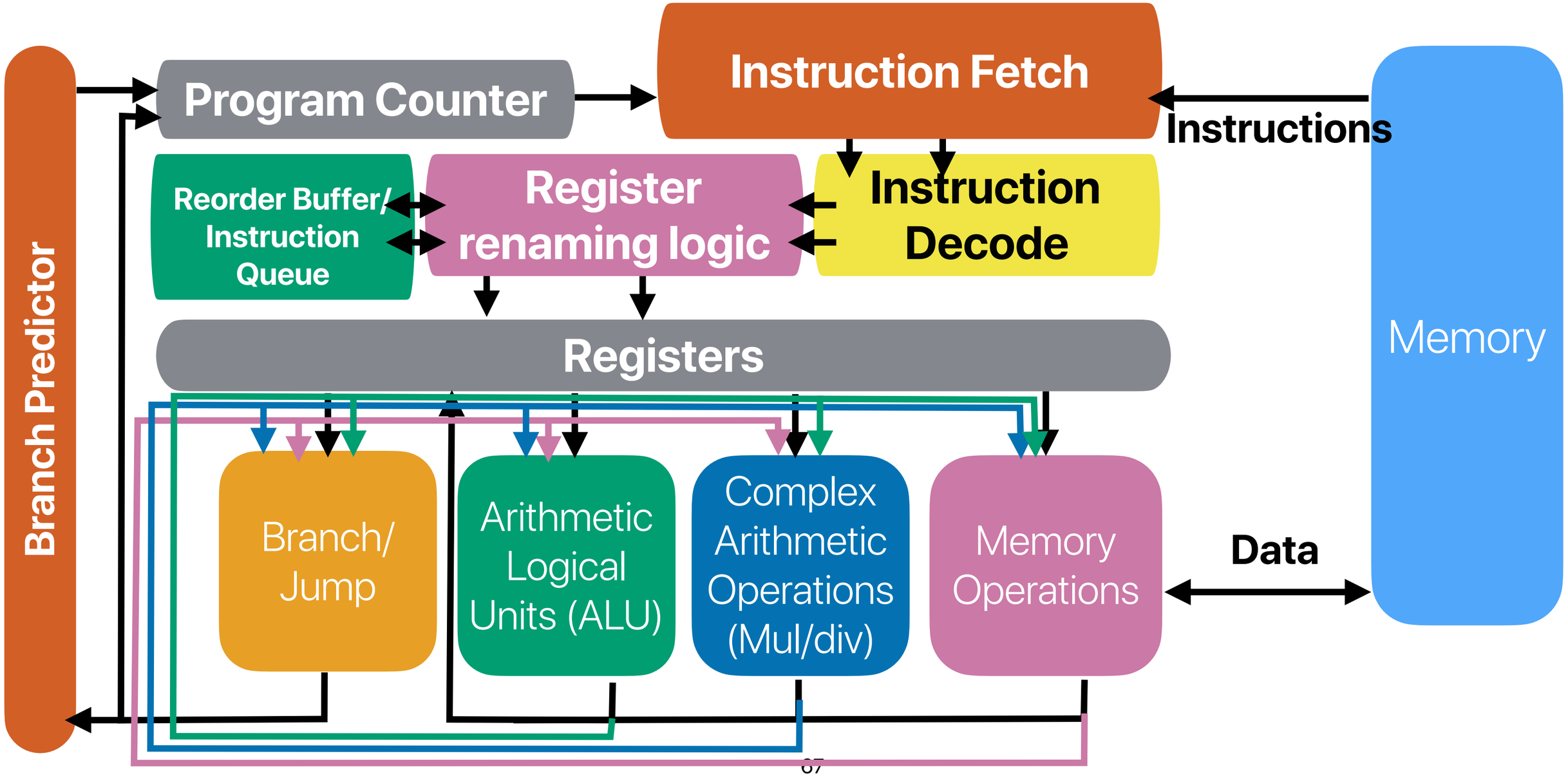
Speculative Execution

- Exceptions (e.g. divided by 0, page fault) may occur anytime
  - A later instruction cannot write back its own result otherwise the architectural states won't be correct
- Hardware can schedule instruction across branch instructions with the help of branch prediction
  - Fetch instructions according to the branch prediction
  - However, branch predictor can never be perfect
- Execute instructions across branches
  - Speculative execution: execute an instruction before the processor know if we need to execute or not
  - Execute an instruction all operands are ready (the values of depending physical registers are generated)
  - Store results in **reorder buffer** before the processor knows if the instruction is going to be executed or not.

Super Scalar



Register renaming



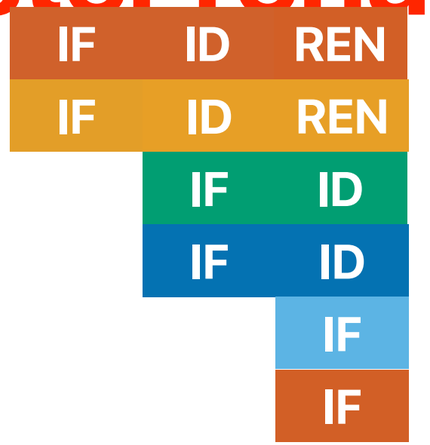
Register renaming

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```

	IF	ID	REN	M1	M2	EX	-	BR	-	WB
1	(1) (2)									
2	(3)(4)	(1) (2)								
3	(5)(6)	(3)(4)	(1) (2)							
4	(7)(8)	(5)(6)	(3)(4)	(1)		(2)				
5	(9)(10)	(7)(8)	(5)(6)		(1)	(4)				
6		(9)(10)	(7)(8)			(3)	(4)	(5)		(1)(2)
7			(9)(10)	(6)		(7)	(3)		(5)	
8					(6)	(9)				(3)(4)
9						(8)		(10)		(5)(6)
10										(7)(8)
11										(9)(10)

Register renaming in motion

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```



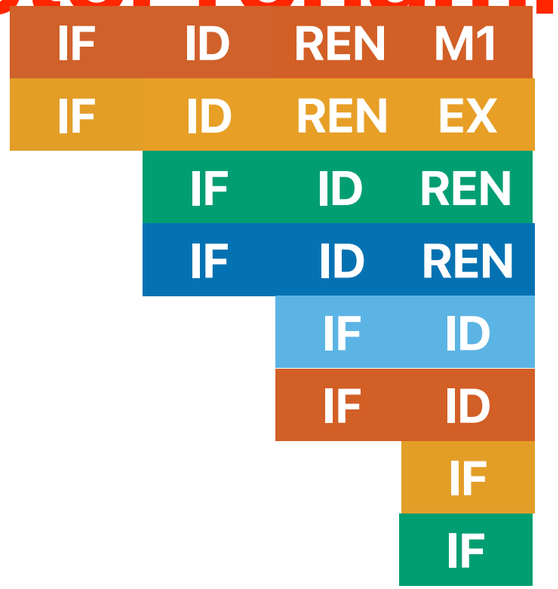
	Renamed instruction
1	movl (%rdi), P1
2	addq 4, P2
3	
4	
5	
6	
7	
8	
9	
10	

	Physical Register
eax	
ecx	P1
rdi	P2
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	0		1	P6			
P2	0		1	P7			
P3				P8			
P4				P9			
P5				P10			

Register renaming in motion

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```



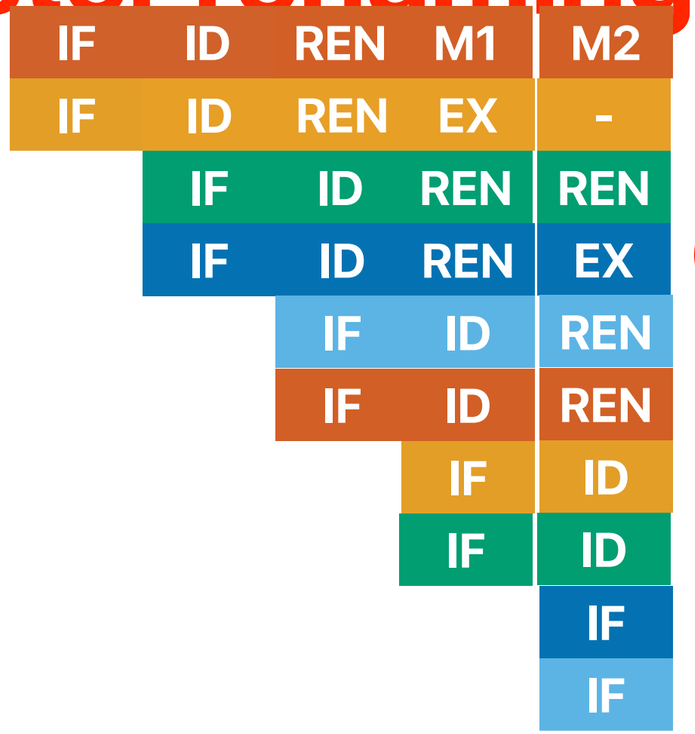
	Renamed instruction
1	movl (%rdi), P1
2	addq $4,%rdi, P2
3	addl P1, %eax, P3
4	cmpq %rdx, P2
5	
6	
7	
8	
9	
10	

	Physical Register
eax	P3
ecx	P1
rdi	P2
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	0		1	P6			
P2	0		1	P7			
P3	0		1	P8			
P4				P9			
P5				P10			

Register renaming in motion

```
① movl    (%rdi), %ecx
② addq    $4, %rdi
③ addl    %ecx, %eax
④ cmpq    %rdx, %rdi
⑤ jne     .L3
⑥ movl    (%rdi), %ecx
⑦ addq    $4, %rdi
⑧ addl    %ecx, %eax
⑨ cmpq    %rdx, %rdi
⑩ jne     .L3
```



(4) is now executing before (3)!

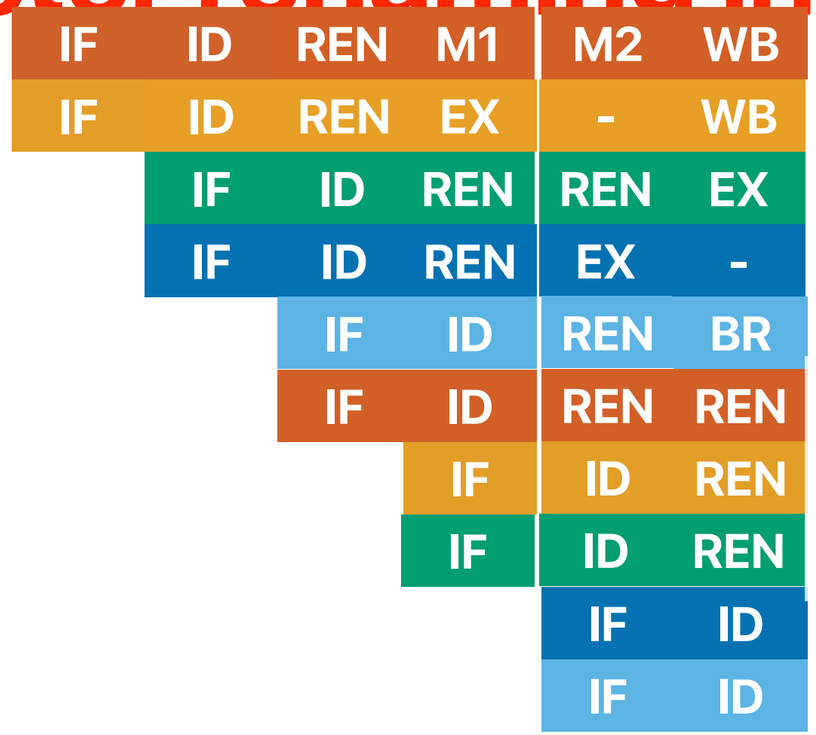
	Renamed instruction
1	movl (%rdi), P1
2	addq $4,%rdi, P2
3	addl P1, %eax, P3
4	cmpq %rdx, P2
5	jne .L3
6	movl (P2), P4
7	
8	
9	
10	

	Physical Register
eax	P3
ecx	P4
rdi	P2
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	0		1	P6			
P2	1		1	P7			
P3	0		1	P8			
P4	0		1	P9			
P5				P10			

Register renaming in motion

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```



Assume issue width == 2, can only put 2 instructions into execution

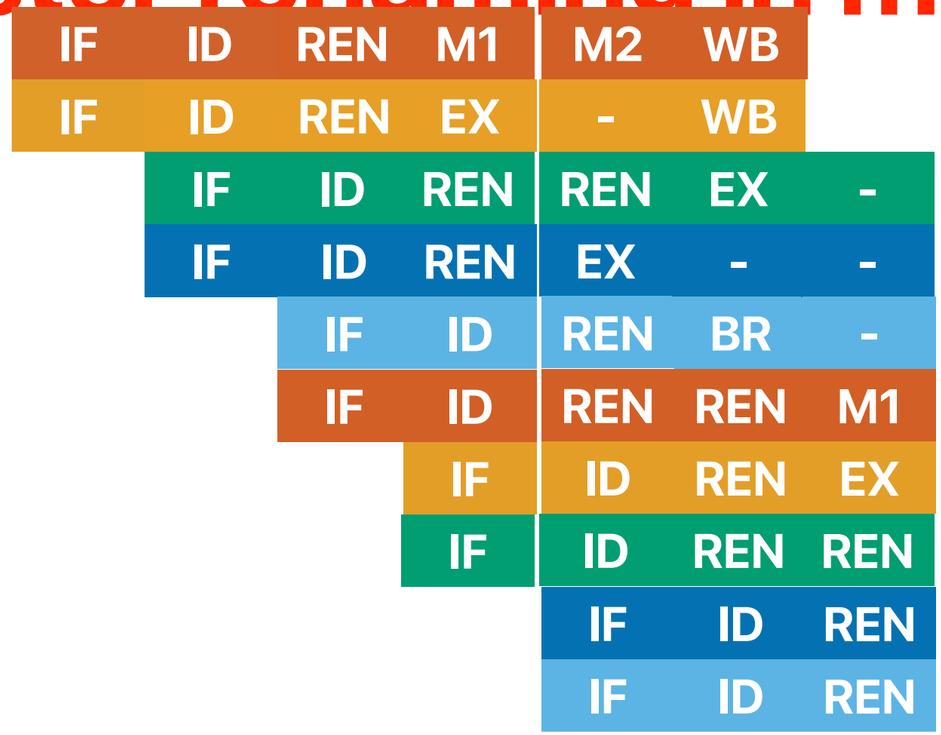
	Renamed instruction
1	movl (%rdi), P1
2	addq $4,%rdi, P2
3	addl P1, %eax, P3
4	cmpq %rdx, P2
5	jne .L3
6	movl (P2), P4
7	addq $4, P2, P5
8	addl P4, P3, P6
9	
10	

	Physical Register
eax	P3
ecx	P4
rdi	P5
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	1		1	P6	0		1
P2	1		1	P7			
P3	0		1	P8			
P4	0		1	P9			
P5	0		1	P10			

Register renaming in motion

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```



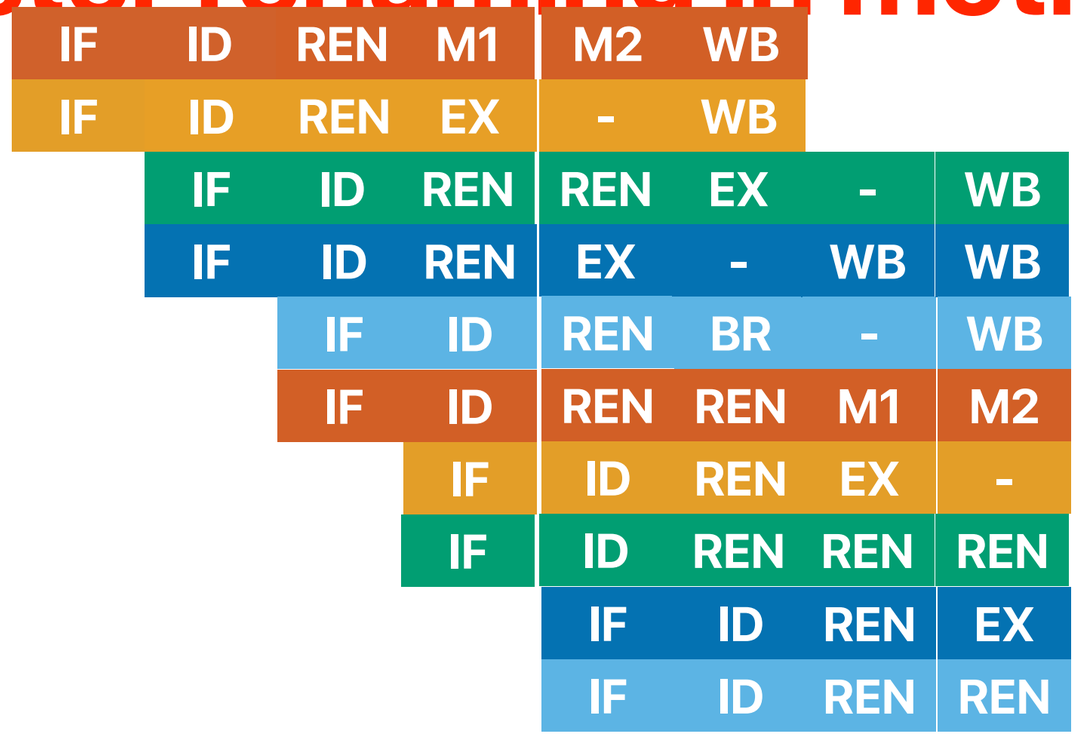
	Renamed instruction
1	~~movl (%rdi), P1~~
2	~~addq $4,%rdi, P2~~
3	addl P1, %eax, P3
4	cmpq %rdx, P2
5	jne .L3
6	movl (P2), P4
7	addq $4, P2, P5
8	addl P4, P3, P6
9	cmpq %rdx, P5
10	jne .L3

	Physical Register
eax	P3
ecx	P4
rdi	P5
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	1		1	P6	0		1
P2	1		1	P7			
P3	0		1	P8			
P4	0		1	P9			
P5	0		1	P10			

Register renaming in motion

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```



	Renamed instruction
1	~~movl (%rdi), P1~~
2	~~addq $4,%rdi, P2~~
3	~~addl P1, %eax, P3~~
4	~~cmpq %rdx, P2~~
5	jne .L3
6	movl (P2), P4
7	addq $4, P2, P5
8	addl P4, P3, P6
9	cmpq %rdx, P5
10	jne .L3

	Physical Register
eax	P3
ecx	P4
rdi	P5
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	1		1	P6	0		1
P2	1		1	P7			
P3	0		1	P8			
P4	0		1	P9			
P5	1		1	P10			

Register renaming in motion

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```

	IF	ID	REN	M1	M2	WB						
	IF	ID	REN	EX	-	WB						
		IF	ID	REN	REN	EX	-	WB				
		IF	ID	REN	EX	-	WB	WB				
			IF	ID	REN	BR	-	WB	WB			
			IF	ID	REN	REN	M1	M2	WB			
				IF	ID	REN	EX	-	WB			
				IF	ID	REN	REN	REN	EX			
					IF	ID	REN	EX	-			
					IF	ID	REN	REN	BR			

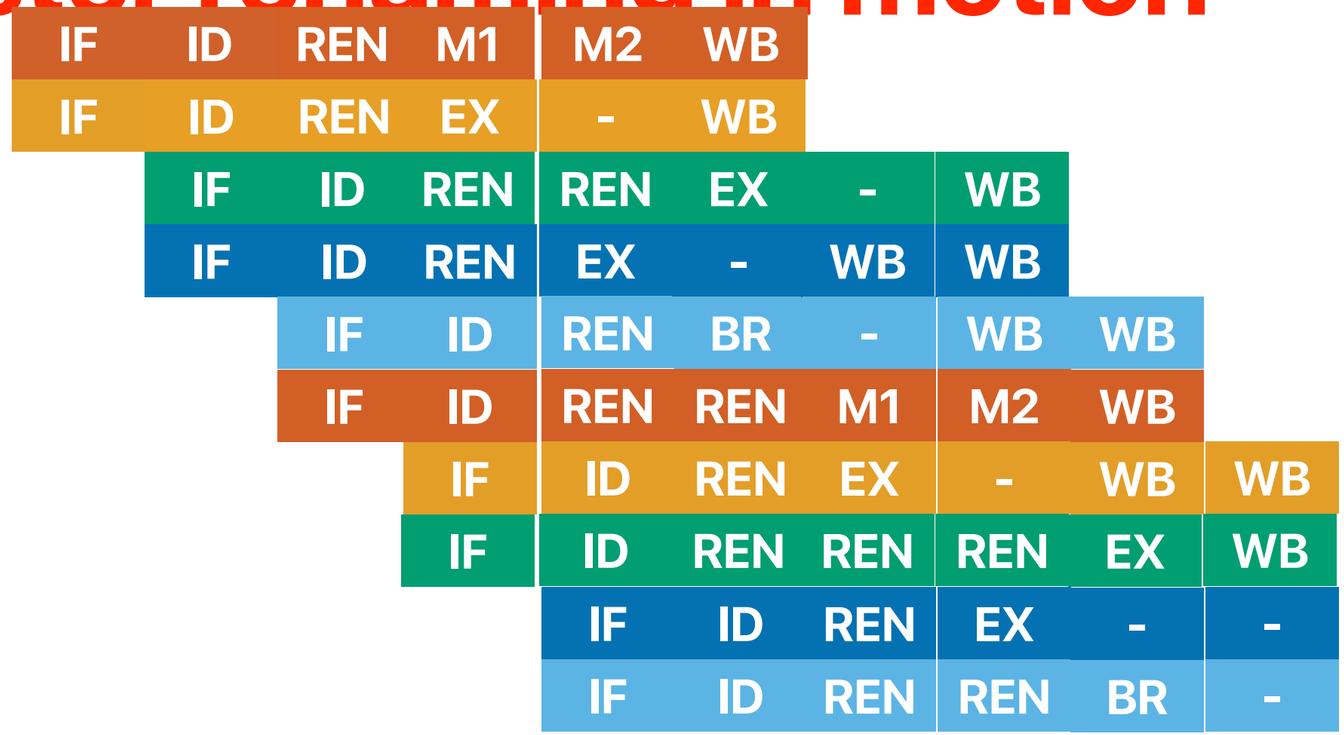
	Renamed instruction
1	~~movl (%rdi), P1~~
2	~~addq $4,%rdi, P2~~
3	~~addl P1, %eax, P3~~
4	~~cmpq %rdx, P2~~
5	jne .L3
6	movl (P2), P4
7	addq $4, P2, P5
8	addl P4, P3, P6
9	cmpq %rdx, P5
10	jne .L3

	Physical Register
eax	P3
ecx	P4
rdi	P5
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	1		1	P6	1		1
P2	1		1	P7			
P3	0		1	P8			
P4	1		1	P9			
P5	1		1	P10			

Register renaming in motion

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```



	Renamed instruction
1	~~movl (%rdi), P1~~
2	~~addq $4,%rdi, P2~~
3	~~addl P1, %eax, P3~~
4	~~cmpq %rdx, P2~~
5	~~jne .L3~~
6	~~movl (P2), P4~~
7	~~addq $4, P2, P5~~
8	~~addl P4, P3, P6~~
9	cmpq %rdx, P5
10	jne .L3

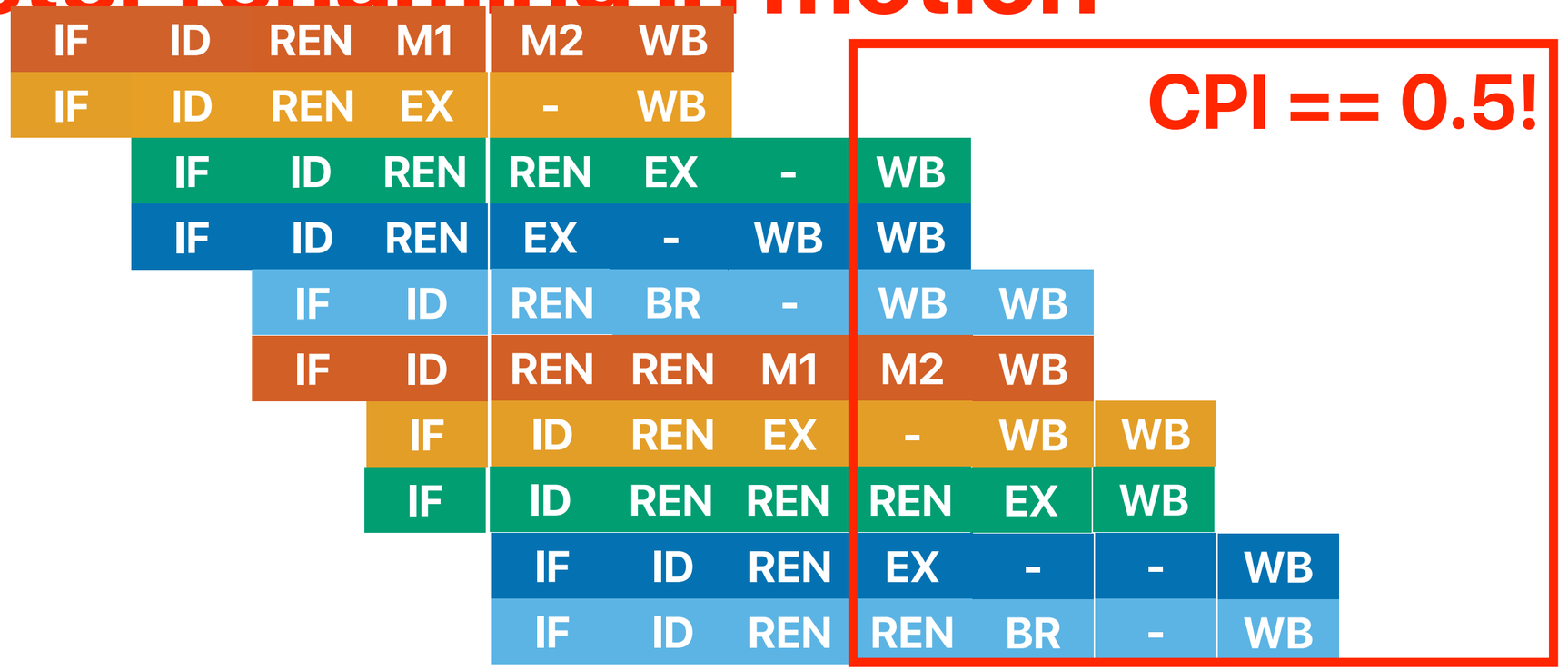
	Physical Register
eax	P3
ecx	P4
rdi	P5
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	1		1	P6	0		1
P2	1		1	P7			
P3	0		1	P8			
P4	1		1	P9			
P5	1		1	P10			

Register renaming in motion

```
①  movl    (%rdi), %ecx
②  addq    $4, %rdi
③  addl    %ecx, %eax
④  cmpq    %rdx, %rdi
⑤  jne     .L3
⑥  movl    (%rdi), %ecx
⑦  addq    $4, %rdi
⑧  addl    %ecx, %eax
⑨  cmpq    %rdx, %rdi
⑩  jne     .L3
```

	Renamed instruction
1	~~movl (%rdi), P1~~
2	~~addq $4,%rdi, P2~~
3	~~addl P1, %eax, P3~~
4	~~cmpq %rdx, P2~~
5	~~jne .L3~~
6	~~movl (P2), P4~~
7	~~addq $4, P2, P5~~
8	~~addl P4, P3, P6~~
9	~~cmpq %rdx, P5~~
10	~~jne .L3~~

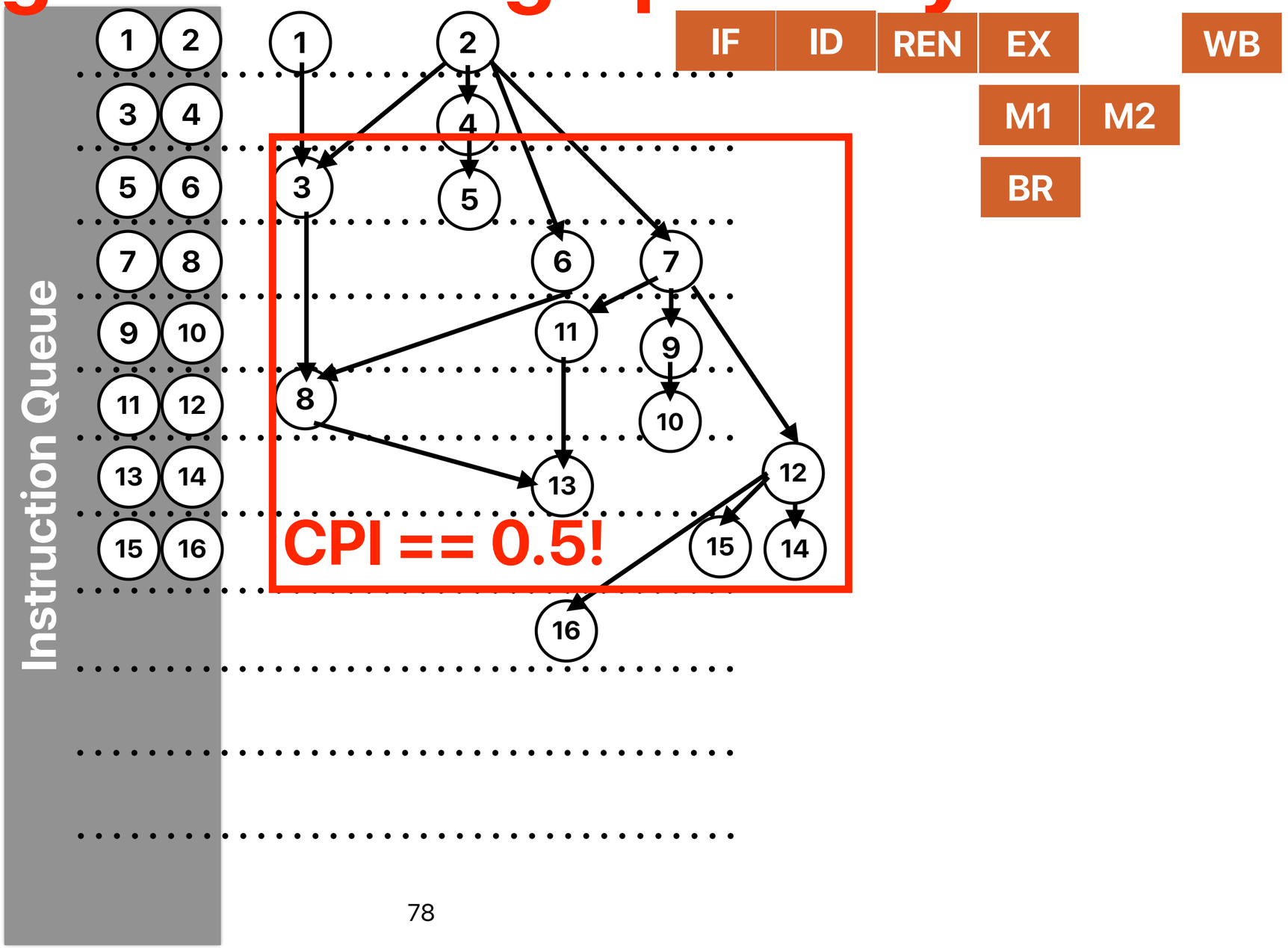


	Physical Register
eax	P3
ecx	P4
rdi	P5
rdx	

	Valid	Value	In use		Valid	Value	In use
P1	1		1	P6	0		1
P2	1		1	P7			
P3	0		1	P8			
P4	1		1	P9			
P5	1		1	P10			

Through data flow graph analysis

```
①  movl (%rdi), %ecx
②  addq $4, %rdi
③  addl %ecx, %eax
④  cmpq %rdx, %rdi
⑤  jne  .L3
⑥  movl (%rdi), %ecx
⑦  addq $4, %rdi
⑧  addl %ecx, %eax
⑨  cmpq %rdx, %rdi
⑩  jne  .L3
⑪  movl (%rdi), %ecx
⑫  addq $4, %rdi
⑬  addl %ecx, %eax
⑭  cmpq %rdx, %rdi
⑮  jne  .L3
⑯  movl (%rdi), %ecx
```



Reorder Buffer (ROB)

Speculative Execution

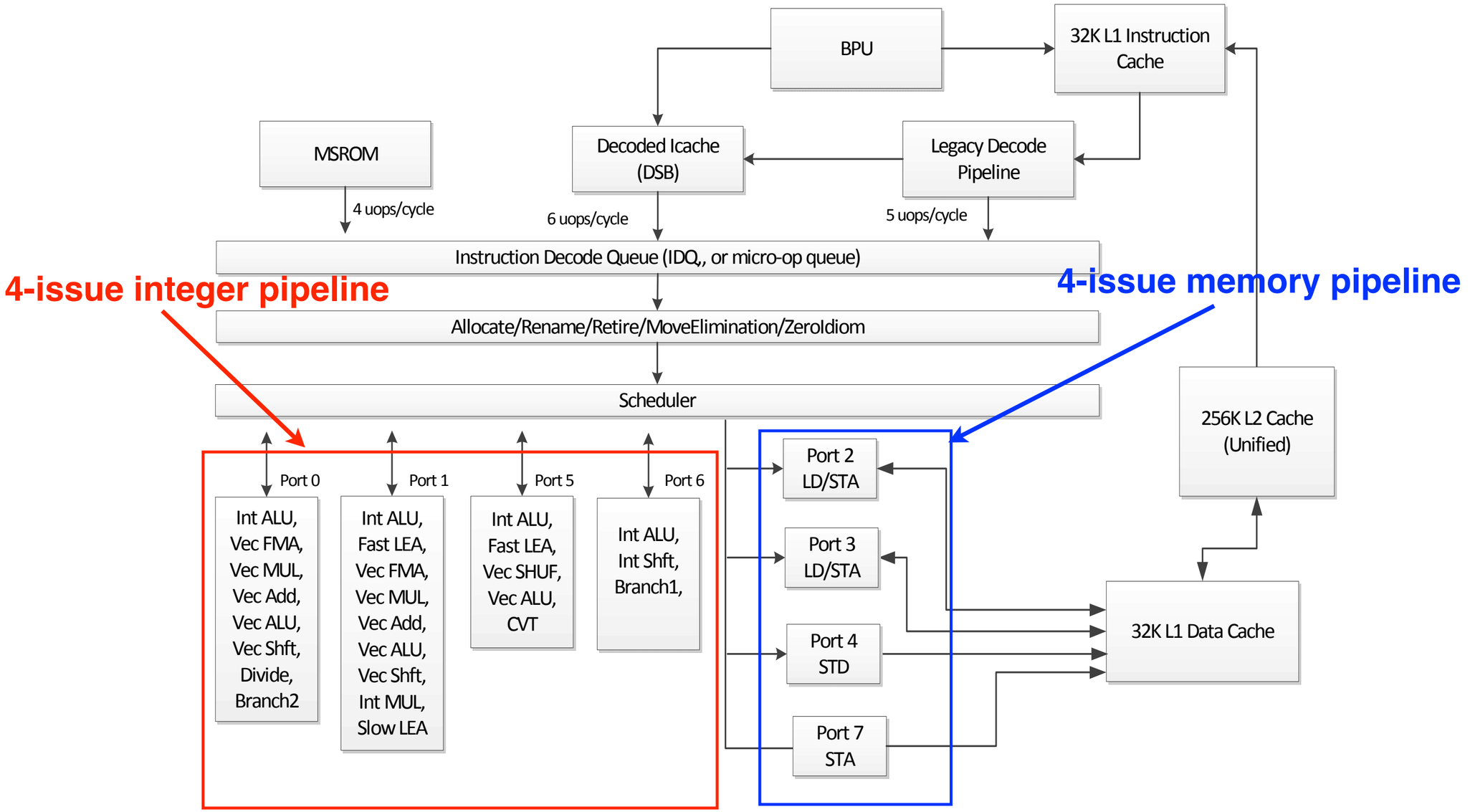
- Any execution of an instruction before a prior instruction finishes is considered as **speculative execution**
- Because it's speculative, we need to preserve the capability to restore to the states before it's executed
  - Branch mis-prediction
  - Exceptions

Reorder buffer/Commit stage

- Reorder buffer — a buffer keep track of the program order of instructions
  - Can be combined with IQ or physical registers — make either as a circular queue
- Commit stage — should the outcome of an instruction be realized
  - An instruction can only leave the pipeline if all it’s previous are committed
  - If any prior instruction failed to commit, the instruction should yield it’s ROB entry, restore all it’s architectural changes

The pipelines of Modern Processors

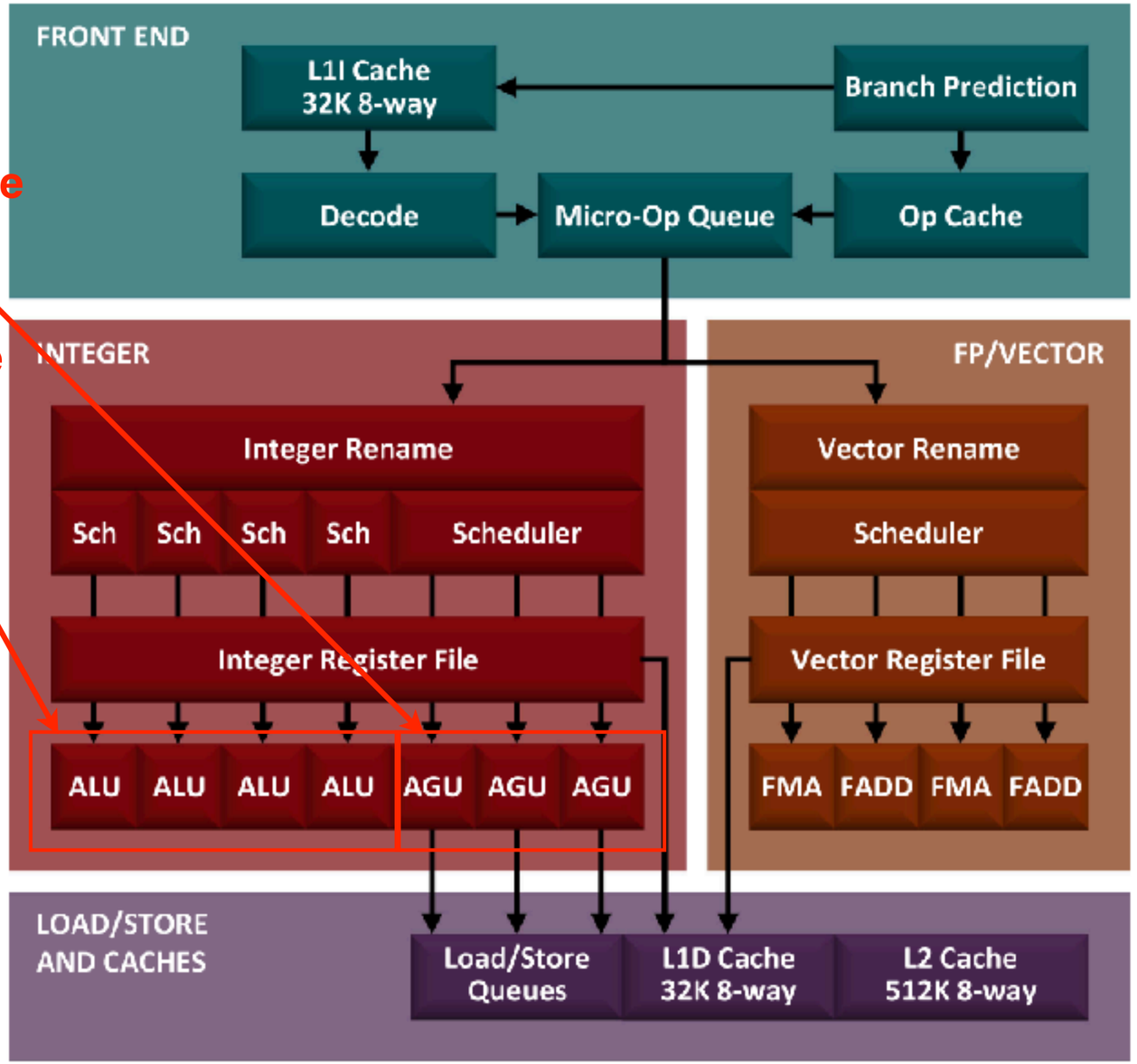
Intel Skylake



AMD Zen 2 (RyZen 3000 Series)

3-issue memory pipeline

4-issue integer pipeline



Demo: ILP within a program

- perf is a tool that captures performance counters of your processors and can generate results like branch mis-prediction rate, cache miss rates and ILP.

Announcements

- Assignment #3 due **Friday**
- Reading Quiz due next Monday