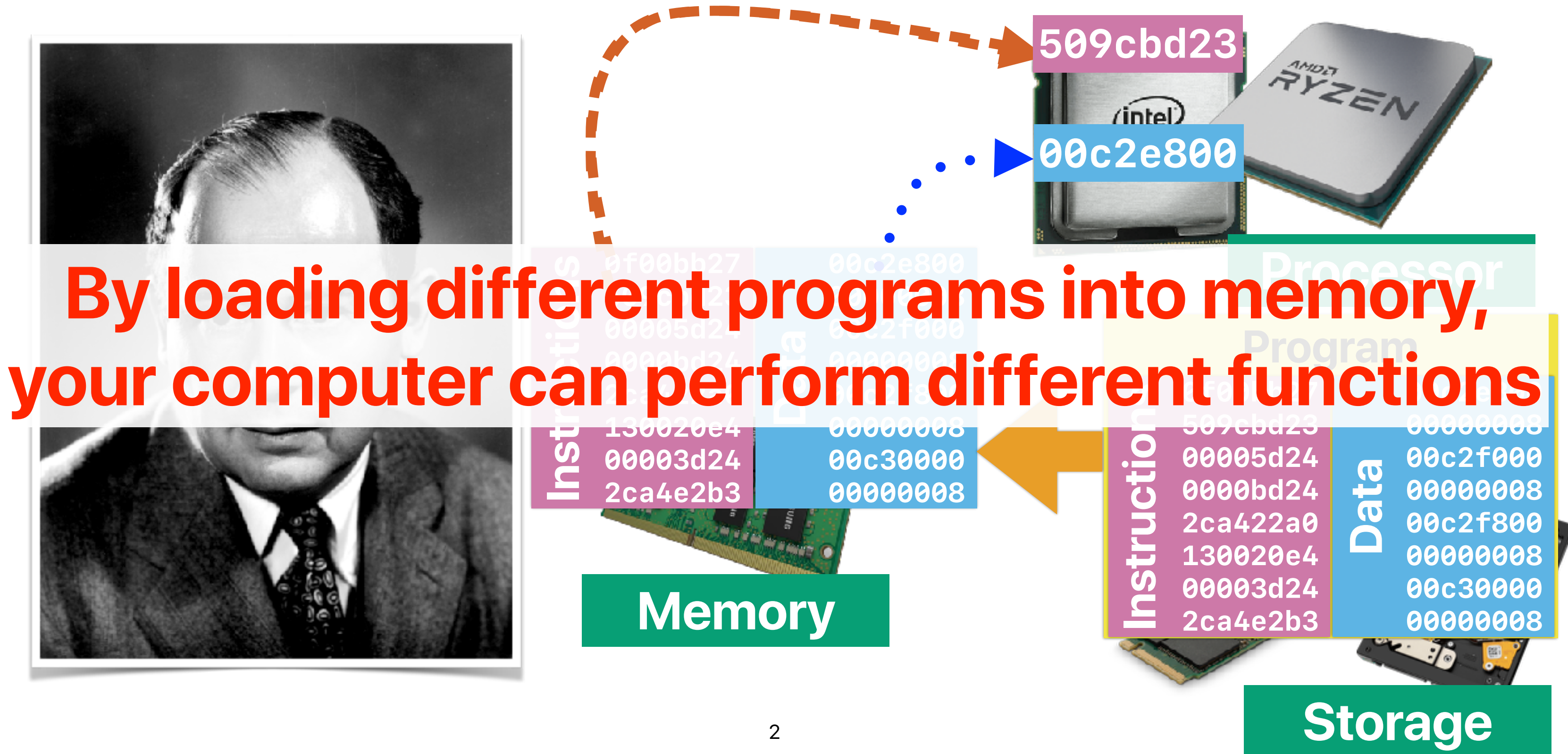


Final Review

Hung-Wei Tseng

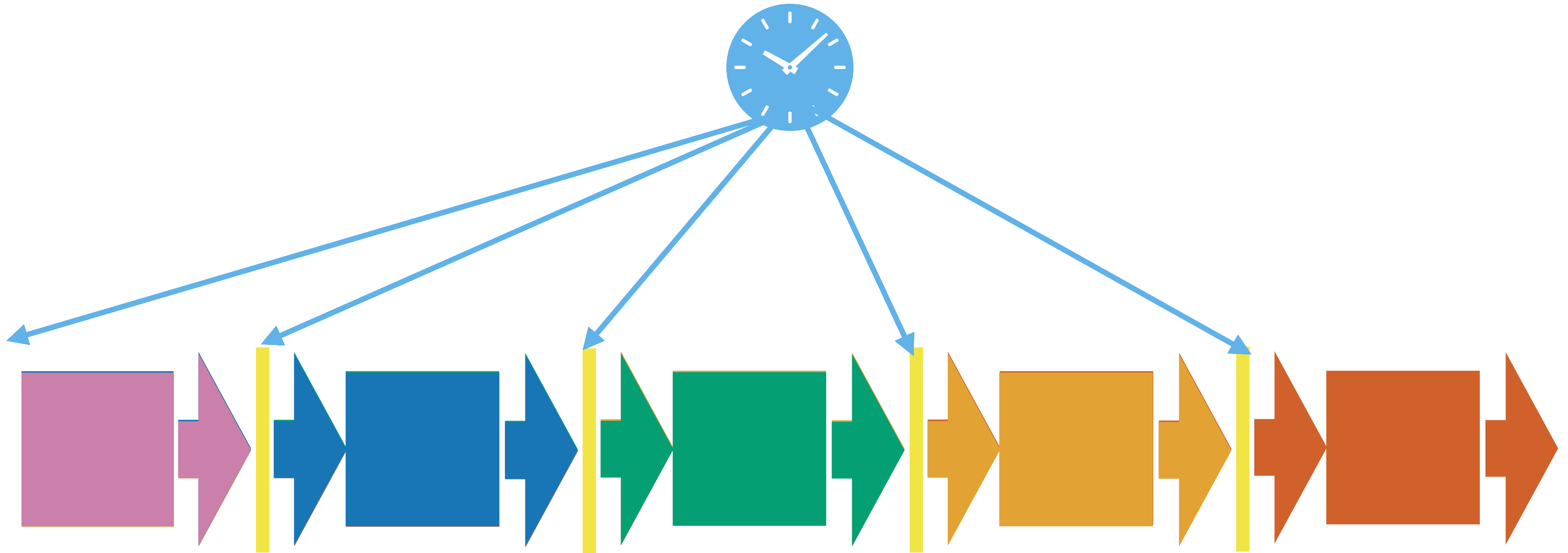
von Neumann Architecture



Tasks in processors supporting RISC-V ISA

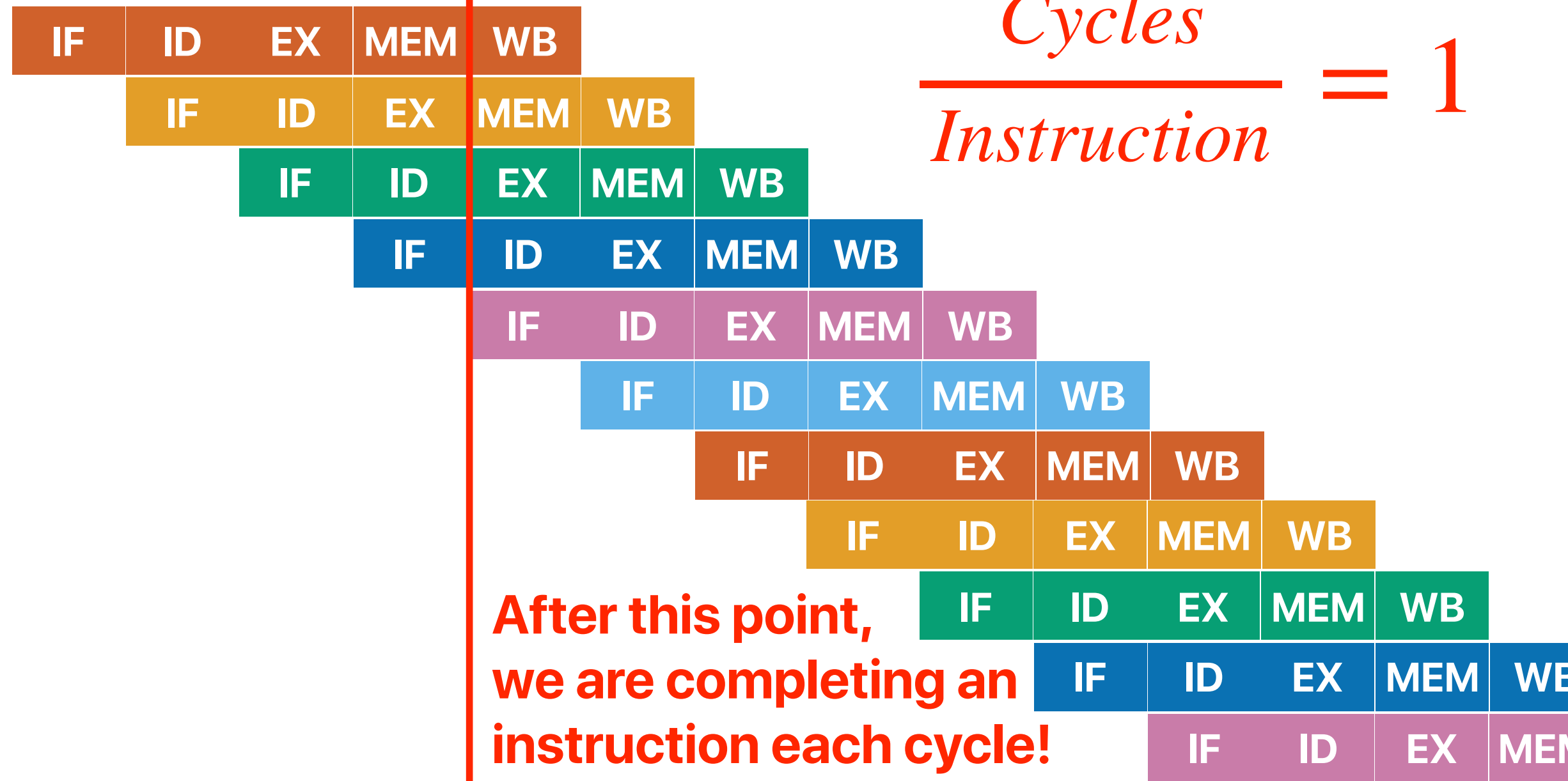
- Instruction Fetch (**IF**) — fetch the instruction from memory
- Instruction Decode (**ID**)
 - Decode the instruction for the desired operation and operands
 - Reading source register values
- Execution (**EX**)
 - ALU instructions: Perform ALU operations
 - Conditional Branch: Determine the branch 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

Pipelining



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)
```



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

Structural Hazards

Cache & Performance

- Application: 80% ALU, 20% Loads
- Assume the 1-cycle L1 hit time allows the CPI to be 1
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle
- L2 U-Cache miss rate: 20%, hit time: 10 cycles
- Main memory hit time: 100 cycles
- What's the average CPI?

$$\text{CPI}_{\text{Average}} = \text{CPI}_{\text{base}} + \text{miss_rate} * \text{miss_penalty}$$

$$= 1 + 100\% * (5\% * (10 + 20\% * (1 * 100)))$$

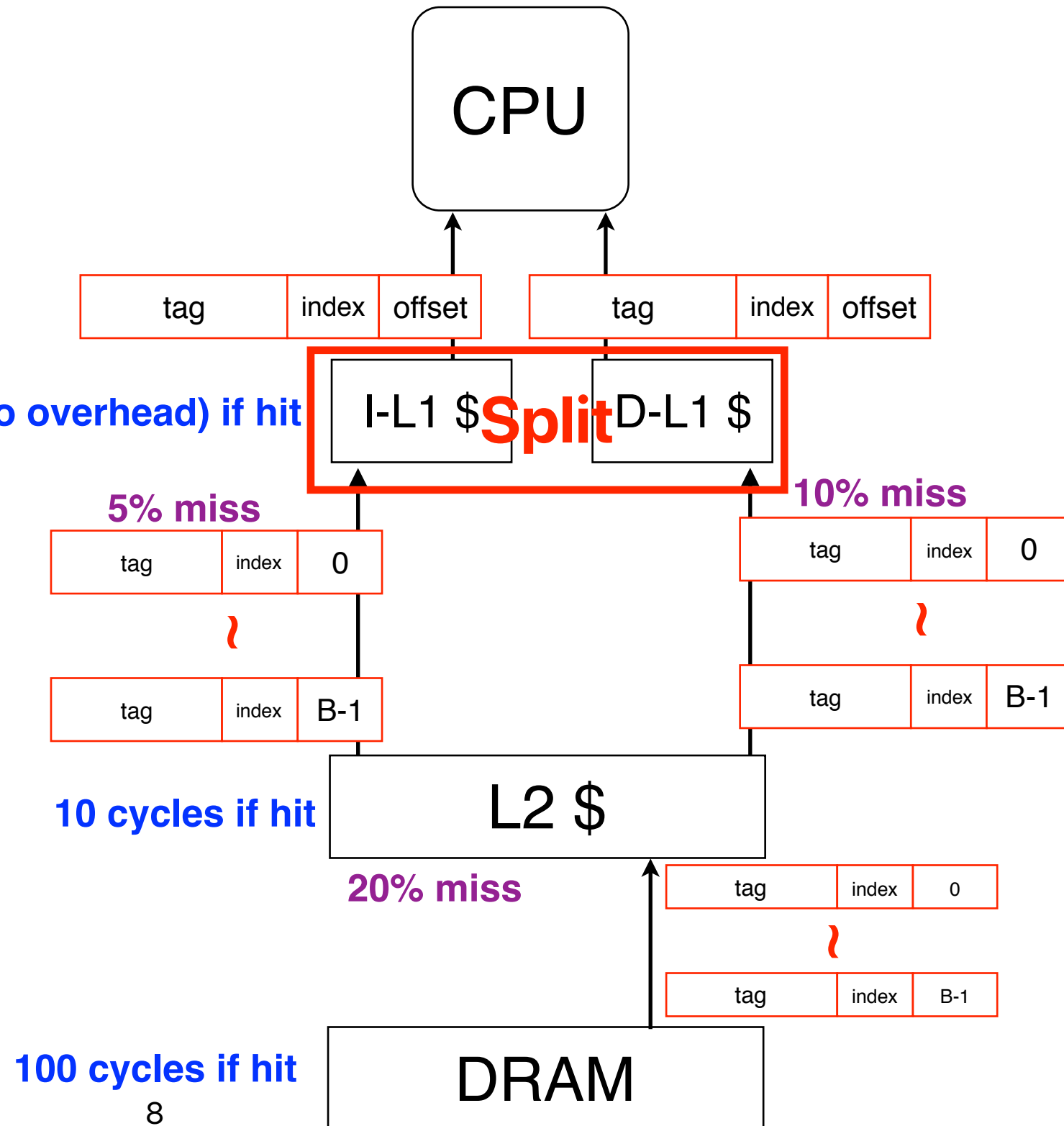
Fetch Instructions

$$+ 20\% * (10\% * (1) * (10 + 20\% * ((1) * 100)))$$

$$= 3.1$$

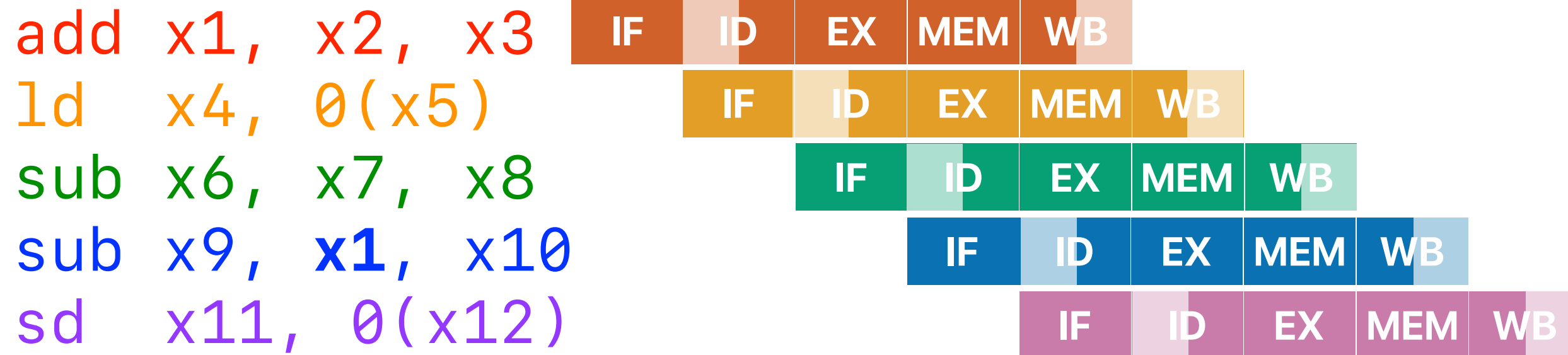
Access Data

1 cycle (no overhead) if hit



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!



Structural Hazards

- What pair of instructions will be problematic if we allow ALU instructions to skip the "MEM" stage?

a: ld x1, 0(x2)



b: add x3, x4, x5



c: sub x6, x7, x8



d: sub x9, x10, x11



e: sd x1, 0(x12)



A. a & b

B. a & c

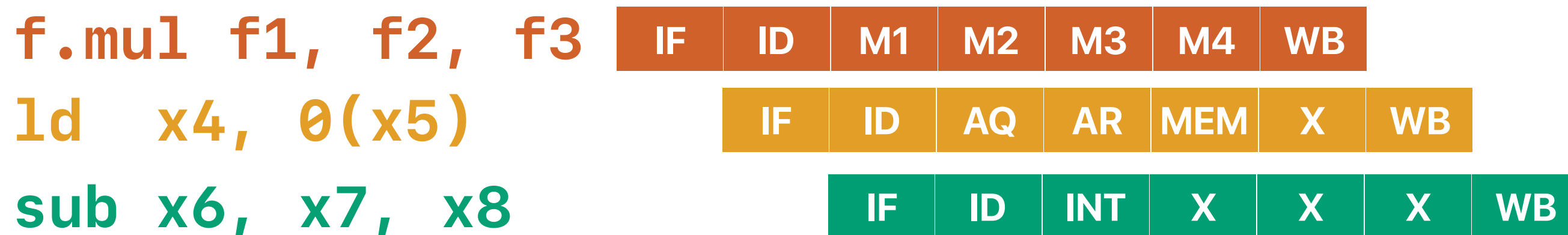
C. b & e

D. c & e

E. None

What if we need to support more complex or longer instructions?

- If we need to support “floating point” arithmetics and it takes 4 stages for floating-point ALU execution



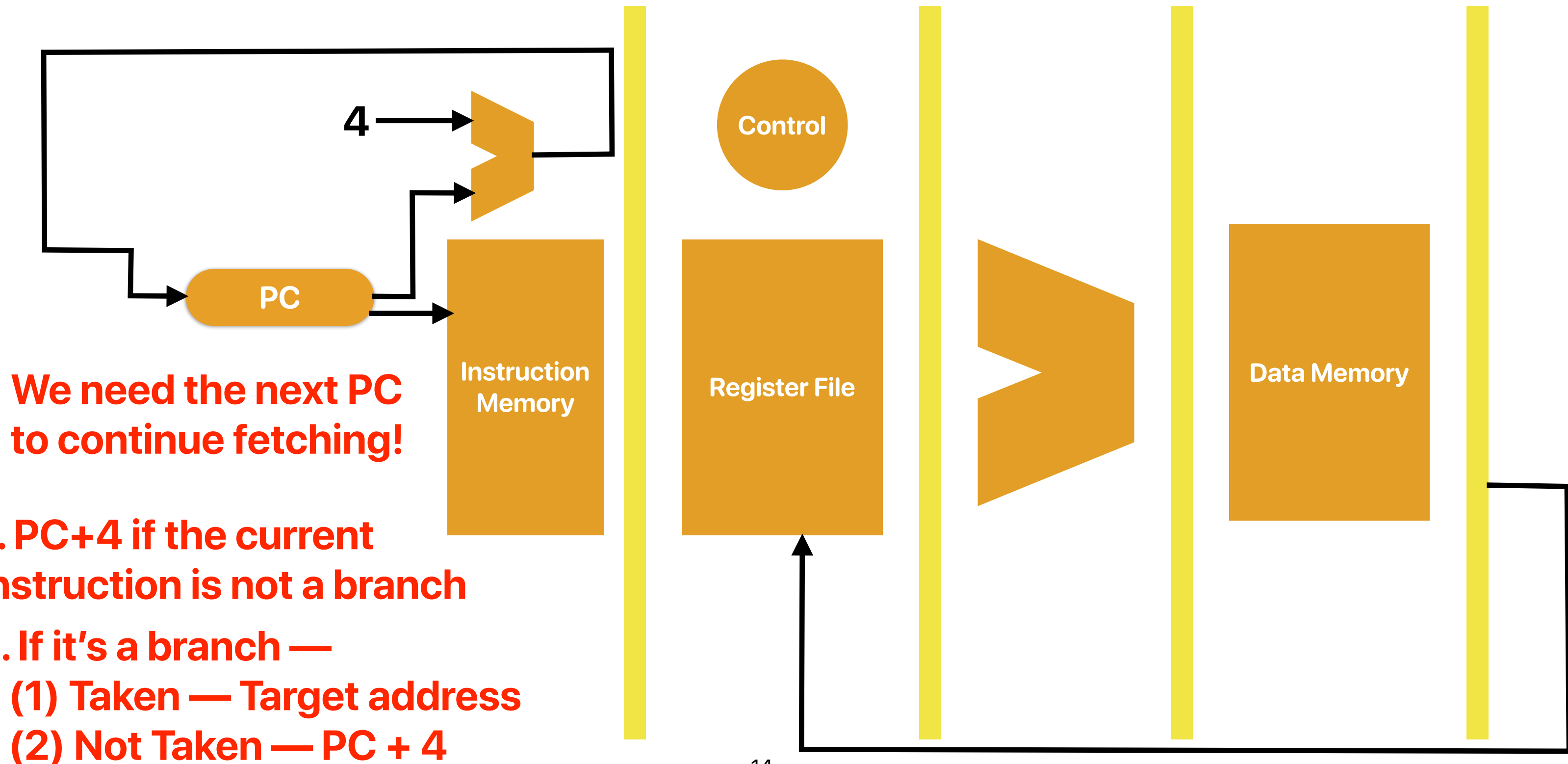
- All instructions have to be the same length in pipeline to avoid structural hazards

Solutions/work-around of pipeline hazards

- Structural
 - Stall
 - More read/write ports
 - Split hardware units (e.g., instruction/data caches)

Control Hazards

A basic dynamic branch predictor



Let's stall whenever it's a branch

- Assuming that we have an application with 20% of branch instructions and the instruction stream incurs no data hazards. When there is a branch, we disable the instruction fetch and insert no-ops until we can determine the PC. What's the average CPI if we execute this program on the 5-stage RISC-V pipeline?

A. 1

B. 1.2

C. 1.4

D. 1.6

E. 1.8

add x1, x2, x3

ld x4, 0(x5)

bne x0, x7, L

add x0, x0, x0

add x0, x0, x0

sub x9, x10, x11

sd x1, 0(x12)

IF	ID	EX	MEM	WB
----	----	----	-----	----

IF	ID	EX	MEM	WB
----	----	----	-----	----

IF	ID	EX	MEM	WB
----	----	----	-----	----

IF	ID	EX	MEM	WB
----	----	----	-----	----

IF	ID	EX	MEM	WB
----	----	----	-----	----

IF	ID	EX	MEM	WB
----	----	----	-----	----

IF	ID	EX	MEM	WB
----	----	----	-----	----

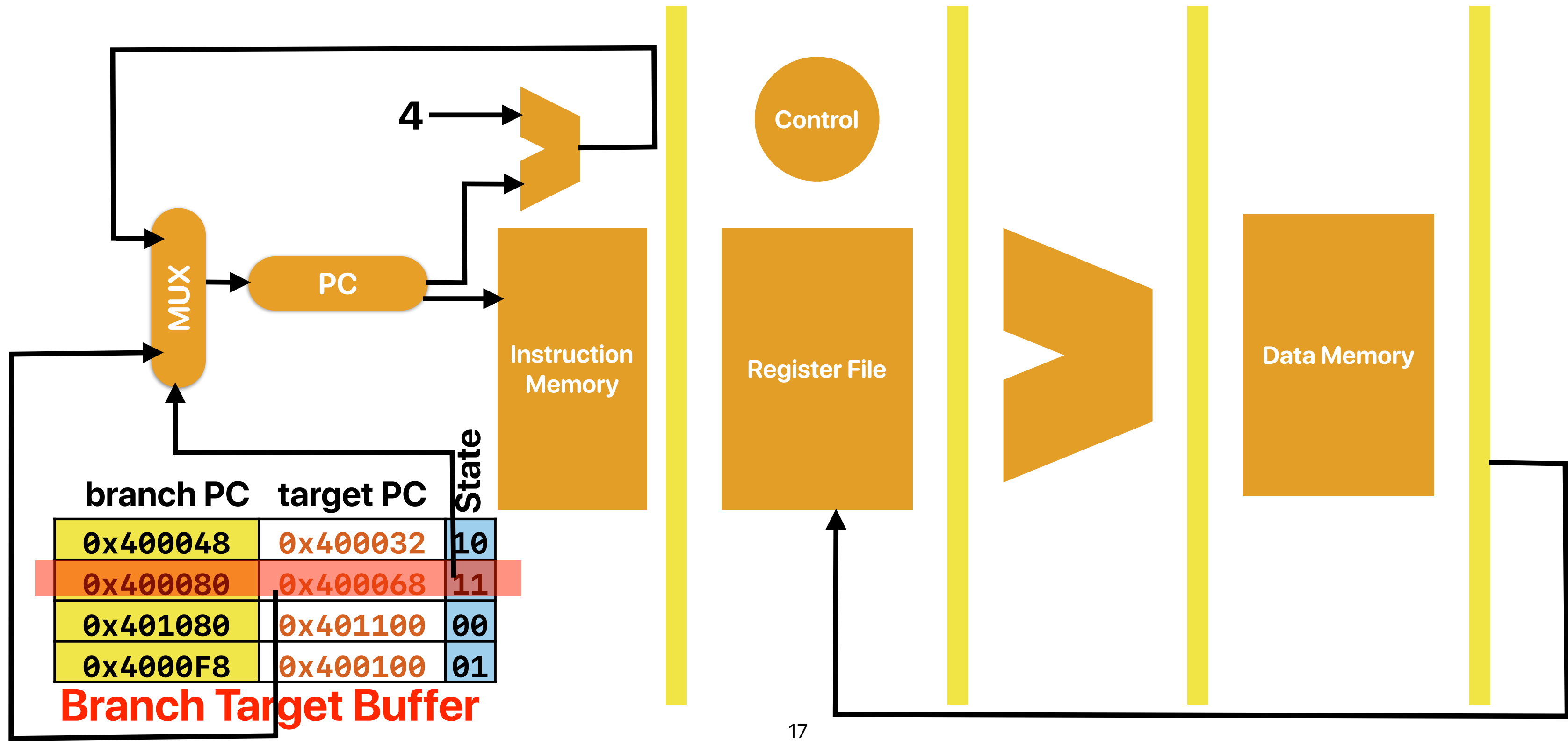
$$1 + 20\% \times 2 = 1.4$$

The frequency of branch

Program	Loads	Stores	Branches	Jumps	ALU operations
astar	28%	6%	18%	2%	46%
bzip	20%	7%	11%	1%	54%
gcc	17%	23%	20%	4%	36%
gobmk	21%	12%	14%	2%	50%
h264ref	33%	14%	5%	2%	45%
hmmer	28%	9%	17%	0%	46%
libquantum	16%	6%	29%	0%	48%
mcf	35%	11%	24%	1%	29%
omnetpp	23%	15%	17%	7%	31%
perlbench	25%	14%	15%	7%	39%
sjeng	19%	7%	15%	3%	56%
xalancbmk	30%	8%	27%	3%	31%

Figure A.29 RISC-V dynamic instruction mix for the SPECint2006 programs. Omnetpp includes 7% of the instructions that are floating point loads, stores, operations, or compares; no other program includes even 1% of other instruction types. A change in gcc in SPECint2006, creates an anomaly in behavior. Typical integer programs have load frequencies that are 1/5 to 3x the store frequency. In gcc, the store frequency is actually higher than the load frequency! This arises because a large fraction of the execution time is spent in a loop that clears memory by storing x0 (not where a compiler like gcc would usually spend most of its execution time!). A store instruction that stores a register pair, which some other RISC ISAs have included, would address this issue.

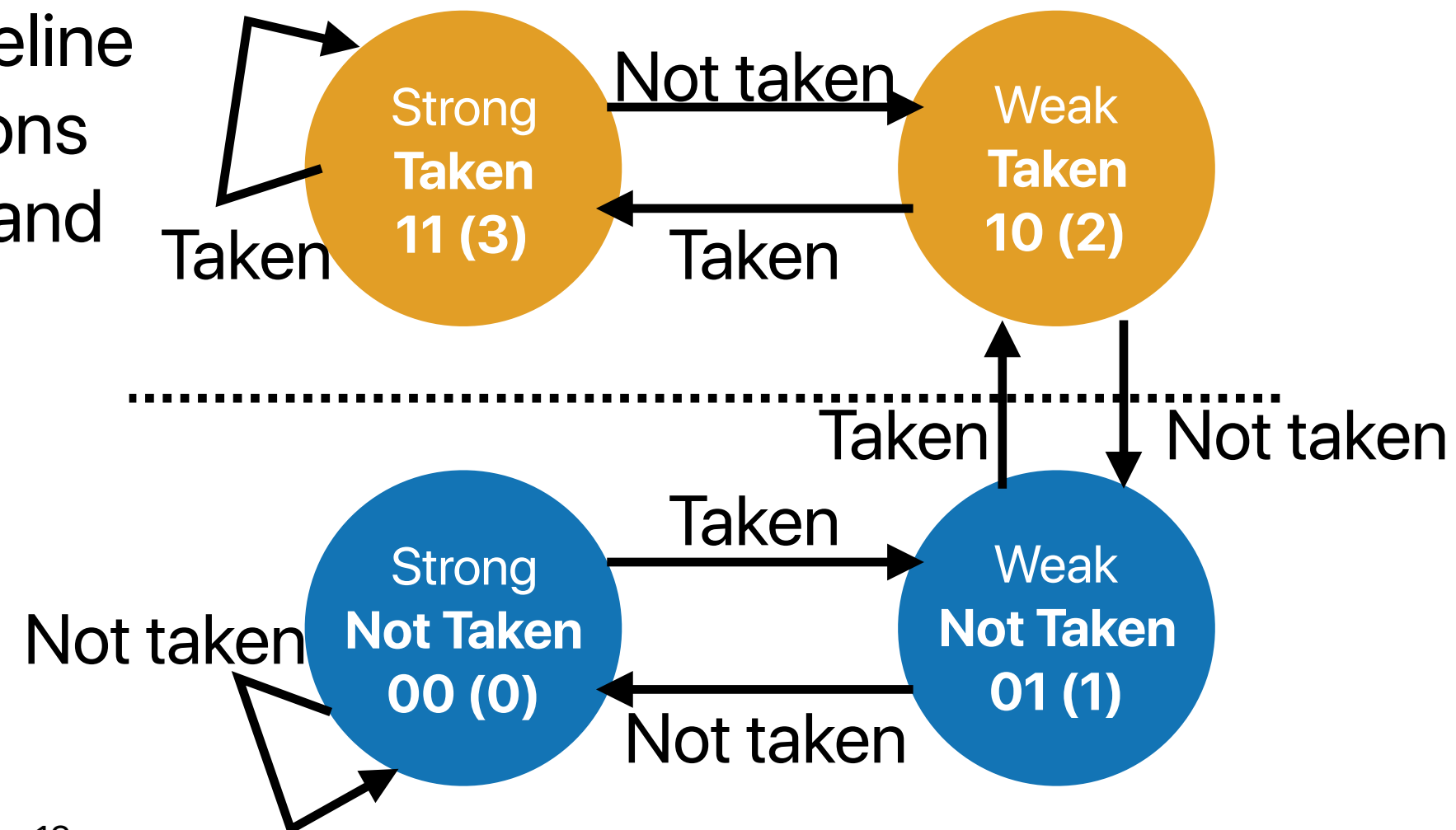
A basic dynamic branch predictor



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

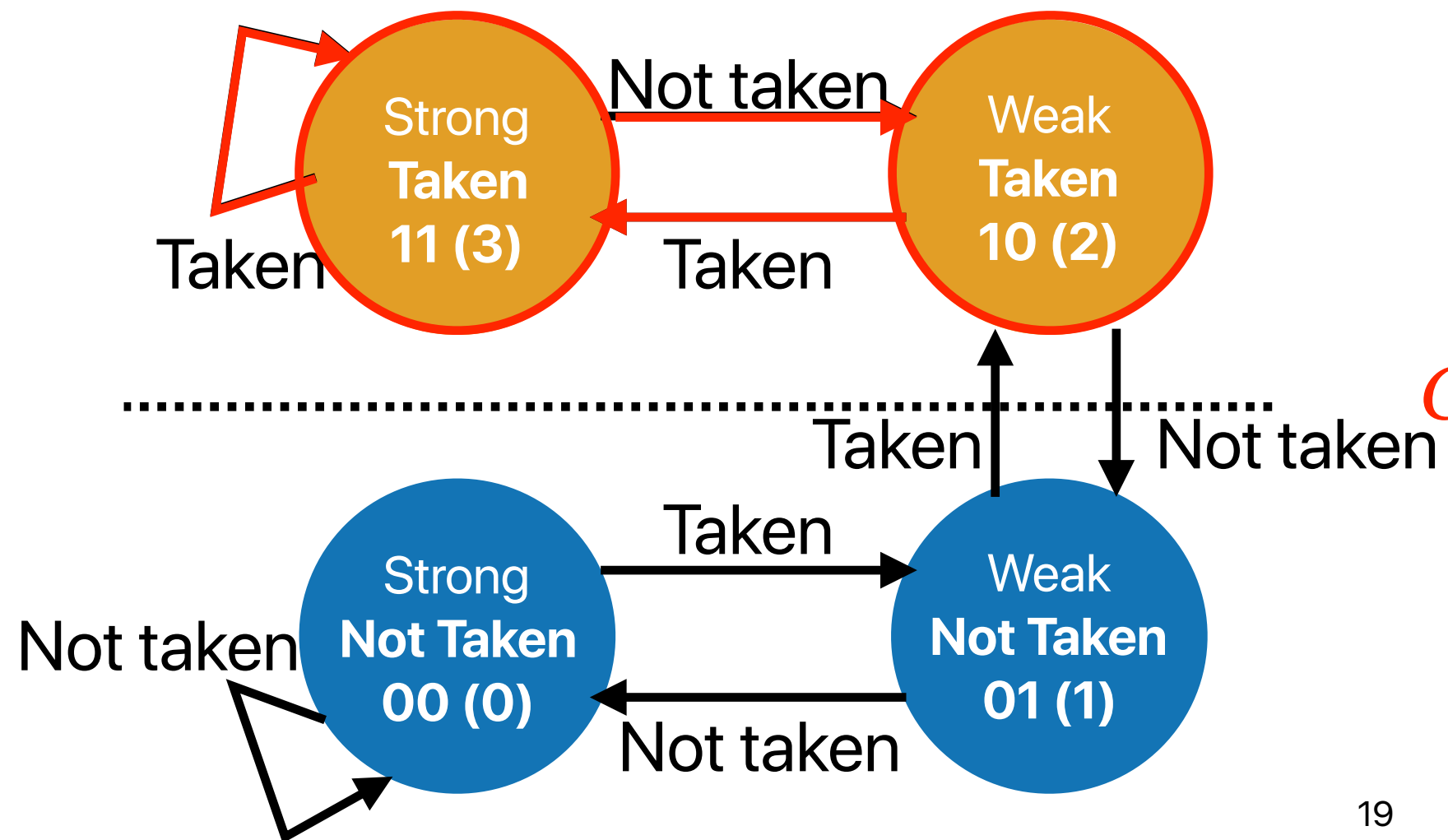
	branch PC	target PC	State
	0x400048	0x400032	10
Predict Taken	0x400080	0x400068	11
	0x401080	0x401100	00
	0x4000F8	0x400100	01



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!

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

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%**

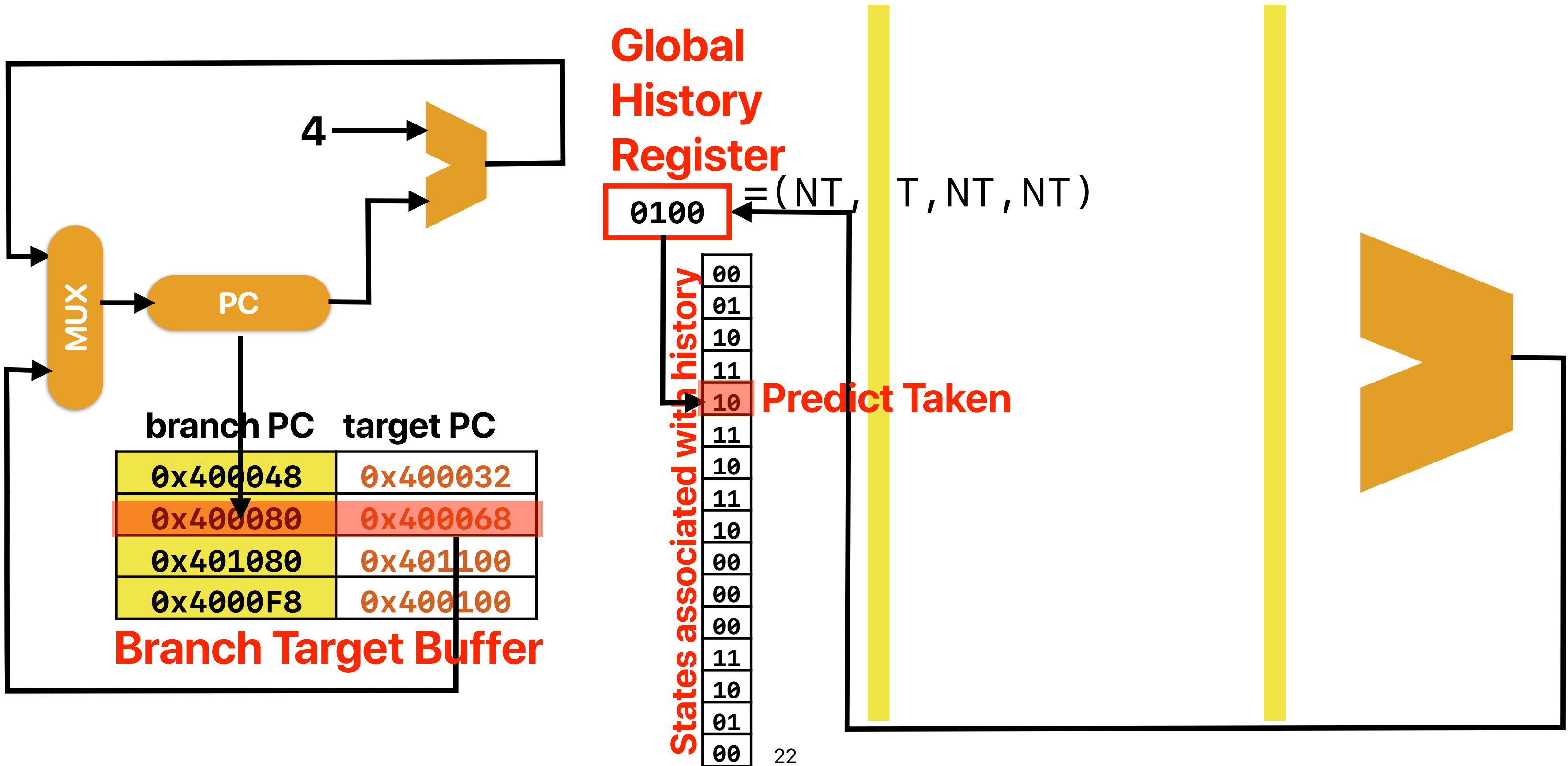
For branch Y, almost 100%,
For branch X, only 50%

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

Two-level global predictor

Reading: Scott McFarling. Combining Branch Predictors. Technical report WRL-TN-36, 1993.

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




Better predictor?

- Consider two predictors — (L) 2-bit local predictor with unlimited BTB entries and (G) 4-bit global history with 2-bit predictors. How many of the following code snippet would allow (G) to outperform (L)?

about the same

about the same

L could be better


 $-$

```
i = 0;
do {
    if( i % 10 != 0)
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100);
```

$=$

```
i = 0;
do {
    a[i] += i;
} while ( ++i < 100);
```

\equiv

 \equiv

```
i = 0;
do {
    j = 0;
    do {
        sum += A[i*2+j];
    }
    while( ++j < 2);
} while ( ++i < 100);
```

\geq

```
i = 0;
do {
    if( rand() %2 == 0)
        a[i] *= 2;
    a[i] += i;
} while ( ++i < 100)
```

A. 0

B. 1

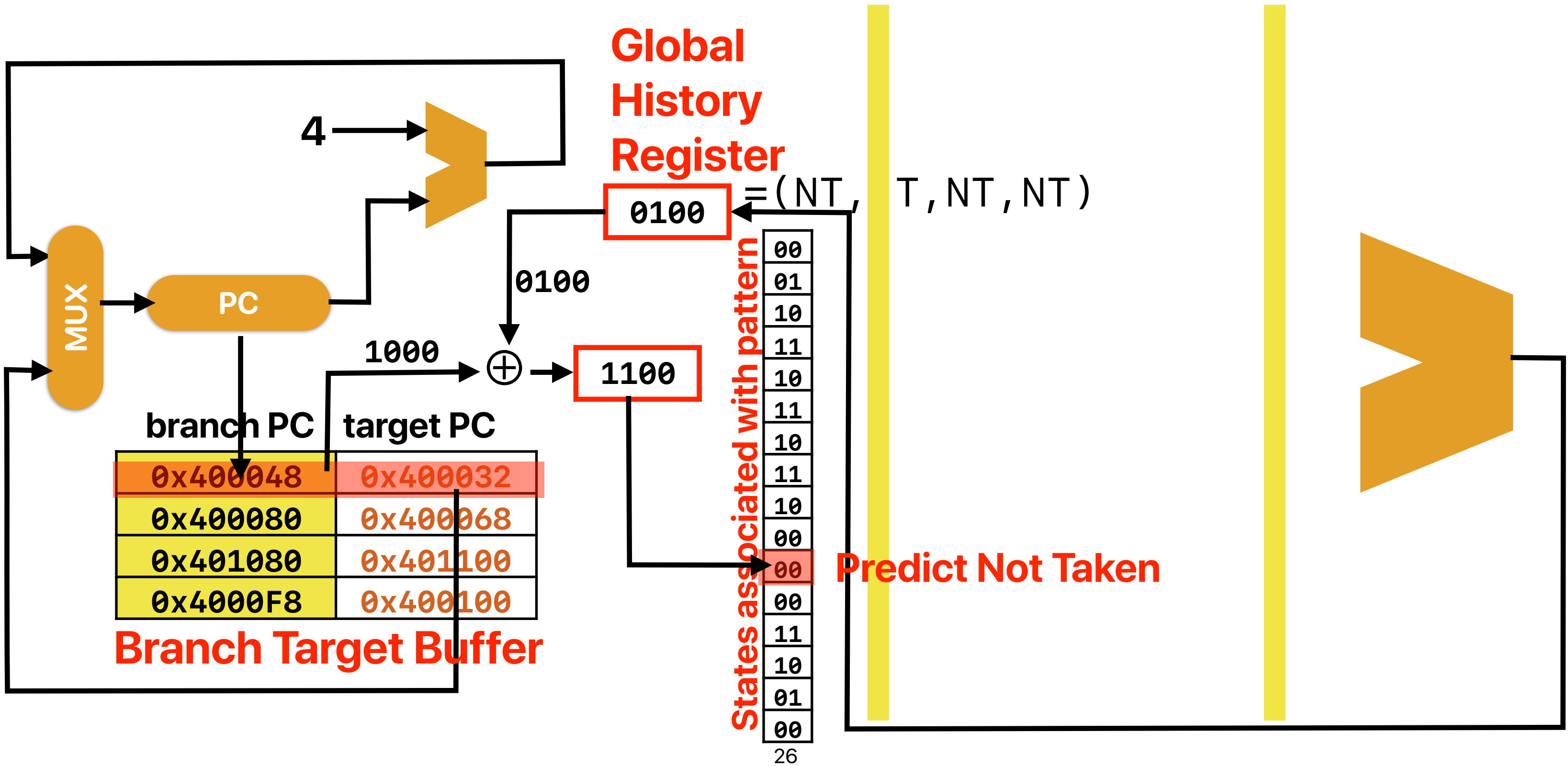
C. 2

D. 3

E. 4

Hybrid predictors

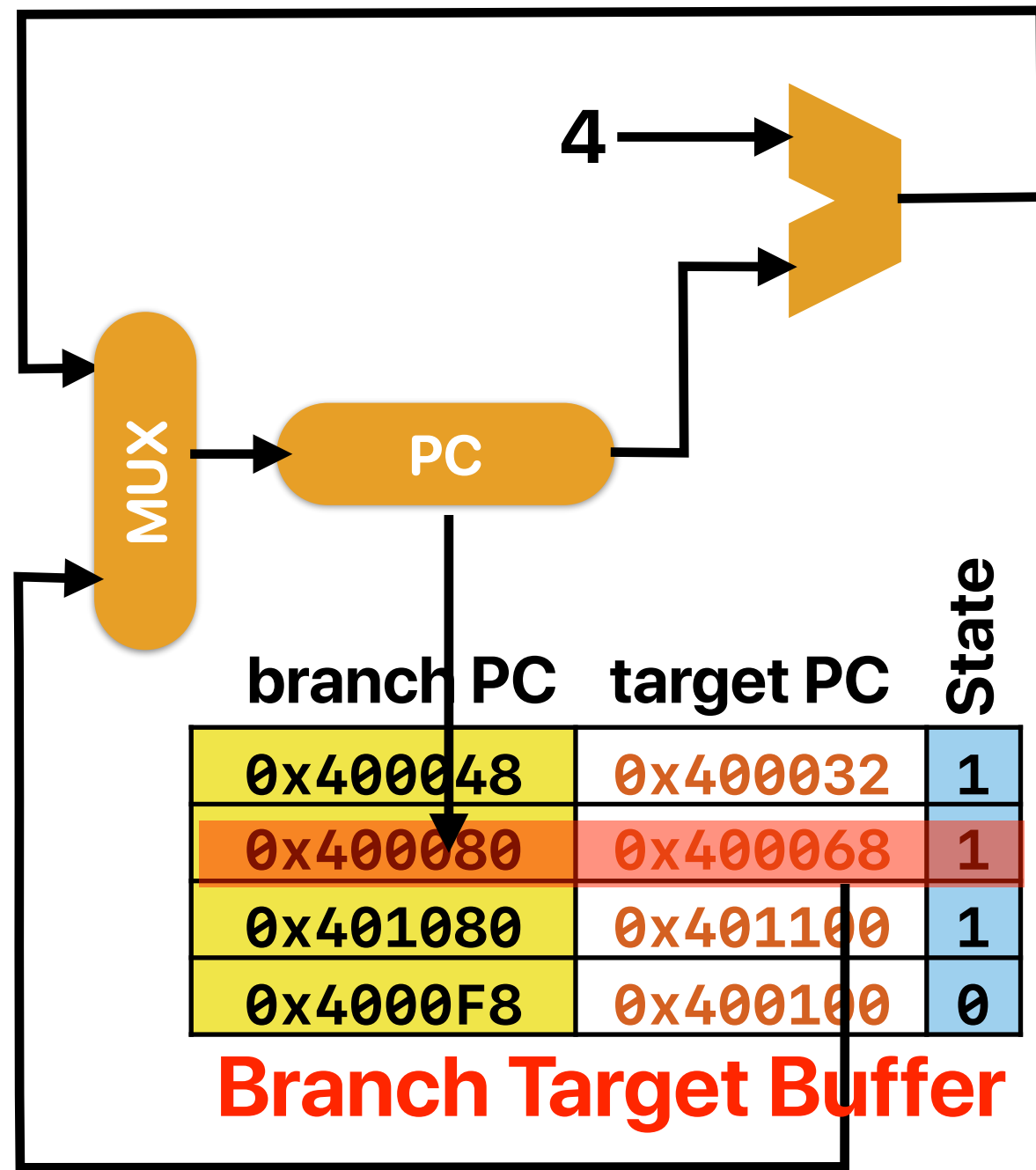
gshare predictor



gshare predictor

- Allowing the predictor to identify both branch address but also use global history for more accurate prediction

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

Branch predictor 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

- 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 ++;
    }
}
```


Demo revisited

- Why the performance is better when option is not "0"
 - ① The amount of dynamic instructions needs to execute is a lot smaller
 - ② The amount of branch instructions to execute is smaller
 - ✓③ The amount of branch mis-predictions is smaller
 - ④ The amount of data accesses is smaller

A. 0 `if(option)`

B. 1

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

C. 2 `for (unsigned i = 0; i < 100000; ++i) {`
`int threshold = std::rand();`

D. 3 `for (unsigned i = 0; i < arraySize; ++i)`
`if (data[i] >= threshold)` **branch X**

E. 4 `sum ++;`

`}`

`}`

	Without sorting	With sorting
The prediction accuracy of X before threshold	50%	100%
The prediction accuracy of X after threshold	50%	100%

Four implementations

- Which of the following implementations will perform the best on modern pipeline processors?

A

```
inline int popcount(uint64_t x){
    int c=0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

C

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    while(x) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

B

```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

D

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++)
    {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

Why is C better than B?

- How many of the following statements explains the reason why B outperforms C with compiler optimizations

- ① ☒ C has lower dynamic instruction count than B
— C only needs one load, one add, one shift, the same amount of iterations
- ② C has significantly lower branch mis-prediction rate than B
— the same number being predicted.
- ③ C has significantly fewer branch instructions than B — the same amount of branches
- ④ C can incur fewer data hazards
— Probably not. In fact, the load may have negative effect without architectural supports

A. 0

B. 1

C. 2

D. 3

E. 4

```
inline int popcount(uint64_t x) {  
    int c = 0;  
    int table[16] = {0, 1, 1, 2, 1,  
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};  
    while(x) {  
        c += table[(x & 0xF)];  
        x = x >> 4;  
    }  
    return c;  
}
```

```
inline int popcount(uint64_t x) {  
    int c = 0;  
    while(x) {  
        c += x & 1;  
        x = x >> 1;  
        c += x & 1;  
        x = x >> 1;  
        c += x & 1;  
        x = x >> 1;  
        c += x & 1;  
        x = x >> 1;  
    }  
    return c;  
}
```

Why is D better than C?

- How many of the following statements explains the main reason why B outperforms C with compiler optimizations

- ① ✓ D has lower dynamic instruction count than C
— Compiler can do loop unrolling — no branches
- ② ✓ D has significantly lower branch mis-prediction rate than C
— Could be
- ③ ✓ D has significantly fewer branch instructions than C
— maybe eliminated through loop unrolling...
- ④ D can incur fewer data hazards than C
— about the same

A. 0

B. 1

C. 2

D. 3

E. 4

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    while(x) {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

```
inline int popcount(uint64_t x) {
    int c = 0;
    int table[16] = {0, 1, 1, 2, 1,
2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
    for (uint64_t i = 0; i < 16; i++)
    {
        c += table[(x & 0xF)];
        x = x >> 4;
    }
    return c;
}
```

All branches are gone with loop unrolling

[illegible]

Without knowing "i<16" in the for-loop, this is not possible

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;
}
```

Solutions/work-around of pipeline hazards

- Structural
 - Stall
 - More read/write ports
 - Split hardware units (e.g., instruction/data caches)
- Control
 - Stalls
 - Branch predictions
 - Compiler optimizations with ISA supports (e.g., delayed branch)

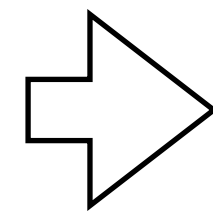
Data Hazards

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

Example: vector scaling

```
i = 0;  
do {  
    vector[i] += scale;  
} while ( ++i < size )
```

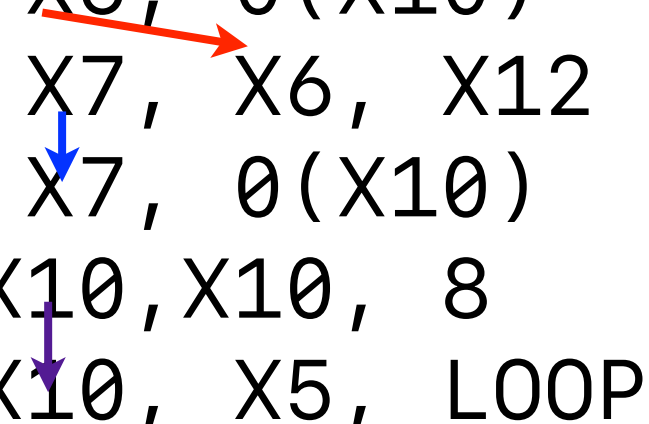


```
                                shl    X5, X11, 3  
                                add    X5, X5, X10  
LOOP: ld    X6, 0(X10)  
                                add    X7, X6, X12  
                                sd     X7, 0(X10)  
                                addi   X10, X10, 8  
                                bne    X10, X5, LOOP
```

How many dependencies do we have?

- How many pairs of data dependencies are there in the following RISC-V instructions?

```
ld      X6, 0(X10)
add     X7, X6, X12
sd      X7, 0(X10)
addi    X10, X10, 8
bne     X10, X5, LOOP
```



A. 1

B. 2

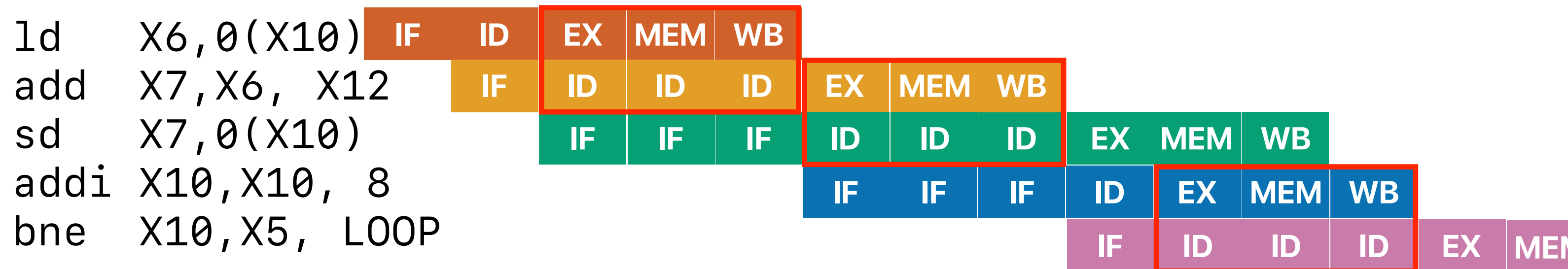
C. 3

D. 4

E. 5

Stalls on data hazards

- How many pairs of instructions in the following RISC-V instructions will results in data hazards/stalls in a basic 5-stage RISC-V pipeline?



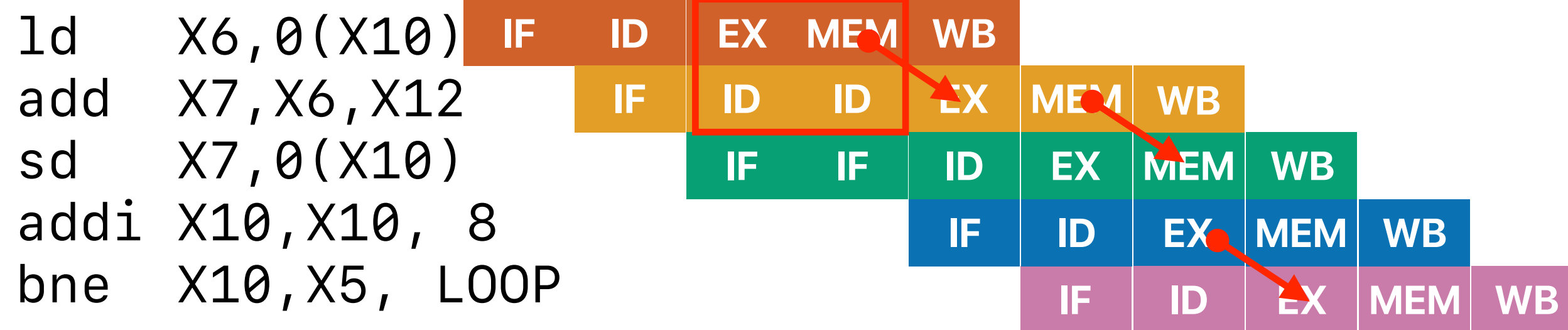
- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

Solution 2: Data forwarding

- Add logics/wires to forward the desired values to the demanding instructions
- In our five stage pipeline — if the instruction entering the EXE stage consumes a result from a previous instruction that is entering MEM stage or WB stage
 - A source of the instruction entering EXE stage is the destination of an instruction entering MEM/WB stage
 - The previous instruction must be an instruction that updates register file

Do we still have to stall?

- How many pairs of instructions in the following RISC-V instructions will results in data hazards/stalls in a basic 5-stage RISC-V pipeline with "full" data forwarding?



A. 0

B. 1

C. 2

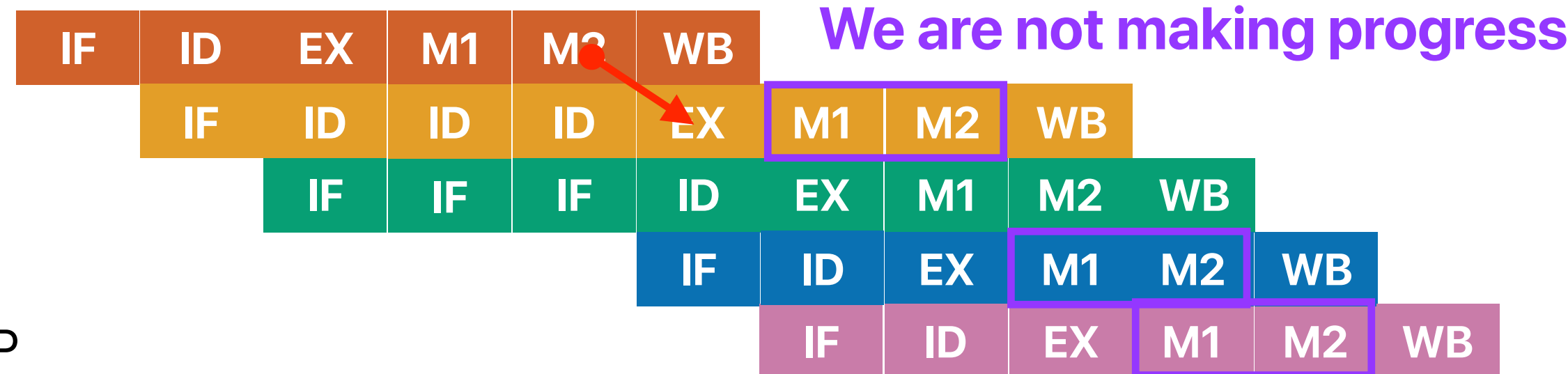
D. 3

E. 4

Problems with data forwarding

- What if our pipeline gets deeper? — Considering a newly designed pipeline where memory stage is split into 2 stages and the memory access finishes at the 2nd memory stage. By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

① ld X6, 0(X10)
 ② add X7, X6, X12
 ③ sd X7, 0(X10)
 ④ addi X10, X10, 8
 ⑤ bne X10, X5, LOOP



The effect of code optimization

- By reordering which pair of the following instruction stream can we eliminate all stalls without affecting the correctness of the code?

- ① `ld X6, 0(X10)`
- ② `add X7, X6, X12`
- ③ `sd X7, 0(X10)`
- ④ `addi X10, X10, 8`
- ⑤ `bne X10, X5, LOOP`

A. (1) & (2)

B. (2) & (3)

C. (3) & (4)

D. (4) & (5)

E. None of the pairs can be reordered

If we can predict the future ...

- Consider the following dynamic instructions:

- ① ld X6, 0(X10)
- ② add X7, X6, X12
- ③ sd X7, 0(X10)
- ④ addi X10, X10, 8
- ⑤ bne X10, X5, LOOP
- ⑥ ld X6, 0(X10)
- ⑦ add X7, X6, X12
- ⑧ sd X7, 0(X10)
- ⑨ addi X10, X10, 8
- ⑩ bne X10, X5, LOOP

Can we use "branch prediction" to predict the future and reorder instructions across the branch?

Which of the following pair can we reorder without affecting the correctness if the **branch prediction is perfect**?

- A. (2) and (4)
- B. (3) and (5)
- C. (5) and (6)
- D. (6) and (9)
- E. (9) and (10)

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

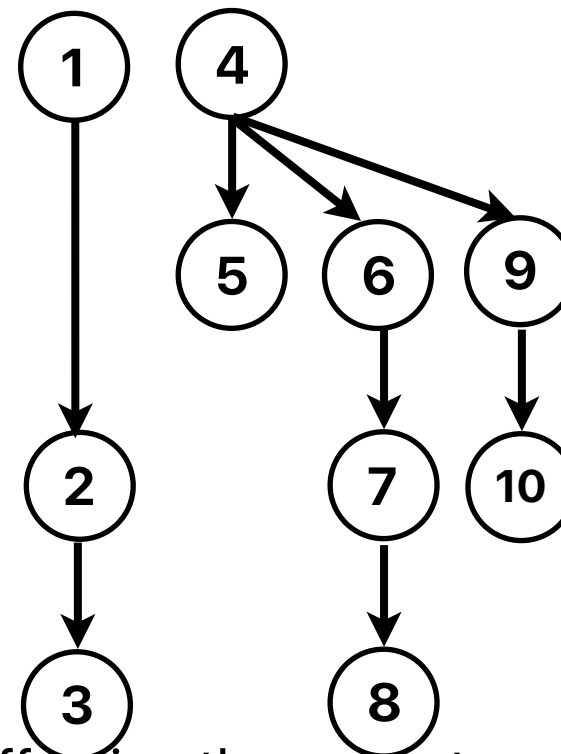
Tips of drawing a pipeline diagram

- Each instruction has to go through all 5 pipeline stages: IF, ID, EXE, MEM, WB in order — only valid if it's single-issue, RISC-V 5-stage pipeline
- An instruction can enter the next pipeline stage in the next cycle if
 - No other instruction is occupying the next stage
 - This instruction has completed its own work in the current stage
 - The next stage has all its inputs ready
- Fetch a new instruction only if
 - We know the next PC to fetch
 - We can predict the next PC
 - Flush an instruction if the branch resolution says it's mis-predicted.

If we can predict the future ...

- Consider the following dynamic instructions:

① ld X6, 0(X10)
② add X7, X6, X12
③ sd X7, 0(X10)
④ addi X10, X10, 8
⑤ bne X10, X5, LOOP
⑥ ld X6, 0(X10)
⑦ add X7, X6, X12
⑧ sd X7, 0(X10)
⑨ addi X10, X10, 8
⑩ bne X10, X5, LOOP



Which of the following pair can we reorder without affecting the correctness if the **branch prediction is perfect**?

- A. (2) and (4)
- B. (3) and (5)
- C. (5) and (6)
- D. (6) and (9)
- E. (9) and (10)

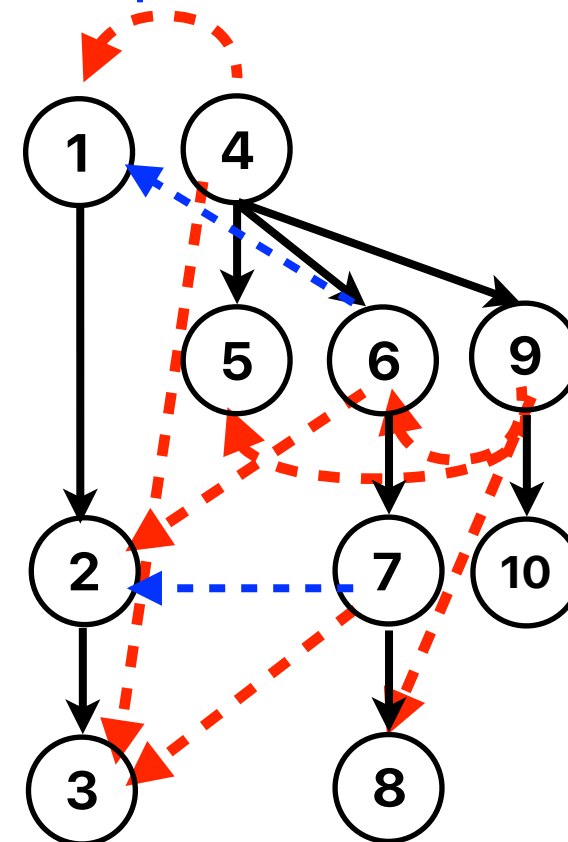
**We still can only reorder (5) and (6)
even though (2) & (4) are not
depending on each other!**

False dependencies

- We are still limited by **false dependencies** — because we have instructions without dependencies sharing registers!
- 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
 - 4 and 1 4 and 3, 6 and 2, 7 and 3, 9 and 5, 9 and 6, 9 and 8
 - **WAW (Write After Write):** a later instruction overwrites the output of an earlier one
 - 6 and 1, 7 and 2

```
① ld X6, 0(X10)
② add X7, X6, X12
③ sd X7, 0(X10)
④ addi X10, X10, 8
⑤ bne X10, X5, LOOP
⑥ ld X6, 0(X10)
⑦ add X7, X6, X12
⑧ sd X7, 0(X10)
⑨ addi X10, X10, 8
⑩ bne X10, X5, LOOP
```

53

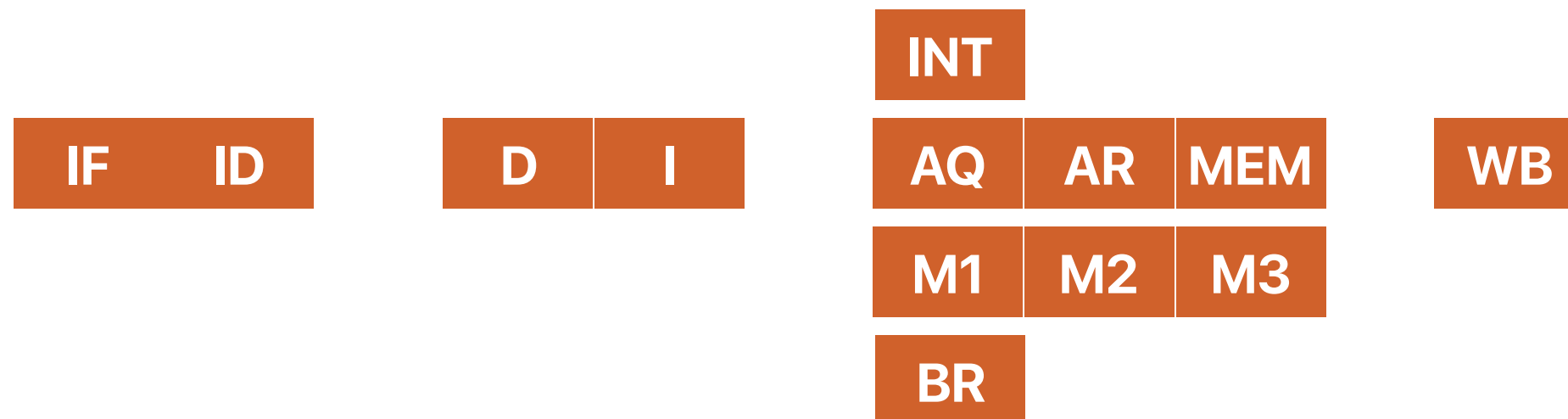


Eliminating false dependencies

- We're constrained in scheduling because instructions
 - WAR — a "logically" later instruction **overwrites** the input register
 - WAW — a "logically" later instruction **overwrites** the output register
- Since all the issues arises from "overwriting" — rename the output registers for instructions
- Two different mechanisms introduced
 - Tomasulo algorithm — rename/remap output to reservation stations
 - Register renaming — rename/remap output to physical registers

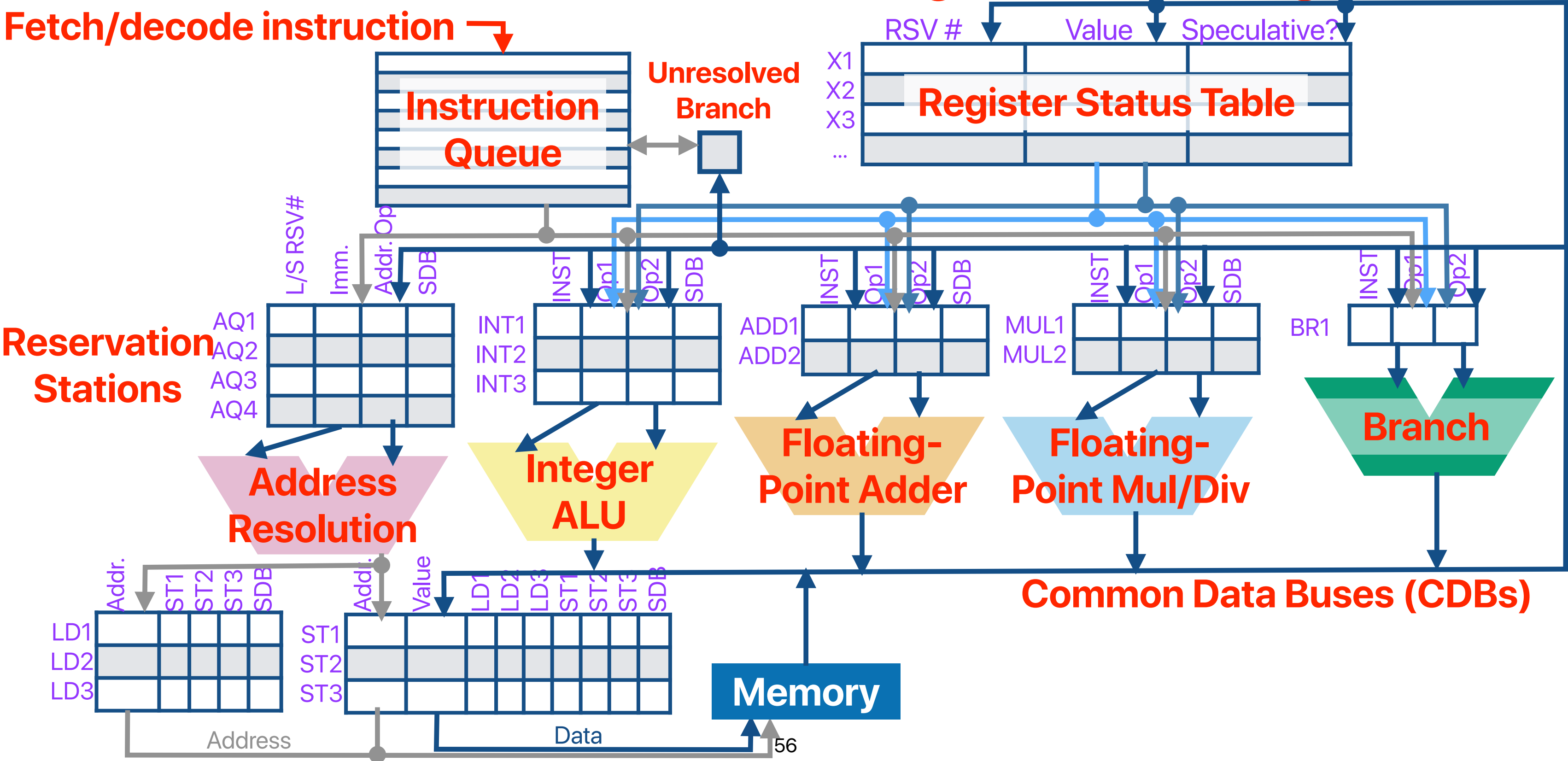
Pipeline in Tomasulo

- Dispatch (D) — allocate a “reservation station” for a decoded instruction
- Issue (I) — collect pending values/branch outcome from common data bus
- Execute (INT, AQ/AQ/MEM, M1/M2/M3, BR) — send the instruction to its corresponding pipeline if no structural hazards
- Write Back (WB) — broadcast the result through CDB



Overview of a processor supporting Tomasulo's algorithm

Fetch/decode instruction ↴



Takes 13 cycles to issue all instructions

①	ld	X6, 0(X10)	D	I	AQ	AR	MEM	WB																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																			
---	----	------------	---	---	----	----	-----	----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Register renaming

- 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.

Tomasulo in motion

①	ld	X6,0(X10)	D	I	AQ	AR	MEM	WB																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																			
---	----	-----------	---	---	----	----	-----	----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

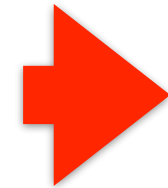
Recap: Why is B better than A?

A

```
inline int popcount(uint64_t x){
    int c=0;
    while(x) {
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```

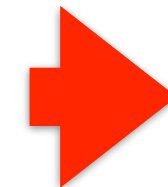
B

```
inline int popcount(uint64_t x) {
    int c = 0;
    while(x) {
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
        c += x & 1;
        x = x >> 1;
    }
    return c;
}
```



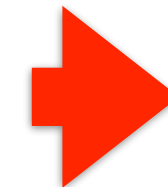
```
and    x2, x1, 1
add    x3, x3, x2
shr    x1, x1, 1
bne    x1, x0, LOOP
```

4*n instructions



```
and    x2, x1, 1
add    x3, x3, x2
shr    x1, x1, 1
and    x2, x1, 1
add    x3, x3, x2
shr    x1, x1, 1
and    x2, x1, 1
add    x3, x3, x2
shr    x1, x1, 1
bne    x1, x0, LOOP
```

13*(n/4) = 3.25*n instructions

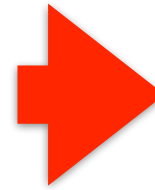


```
and    x2, x1, 1
shr    x4, x1, 1
shr    x5, x1, 2
shr    x6, x1, 3
shr    x1, x1, 4
and    x7, x4, 1
and    x8, x5, 1
and    x9, x6, 1
add    x3, x3, x2
add    x3, x3, x7
add    x3, x3, x8
add    x3, x3, x9
bne    x1, x0, LOOP
```

60 Only one branch for four iterations in A

Recap: Why is B better than A?

```
and x2, x1, 1
add x3, x3, x2
shr x1, x1, 1
and x2, x1, 1
add x3, x3, x2
shr x1, x1, 1
and x2, x1, 1
add x3, x3, x2
shr x1, x1, 1
and x2, x1, 1
add x3, x3, x2
shr x1, x1, 1
bne x1, x0, LOOP
```



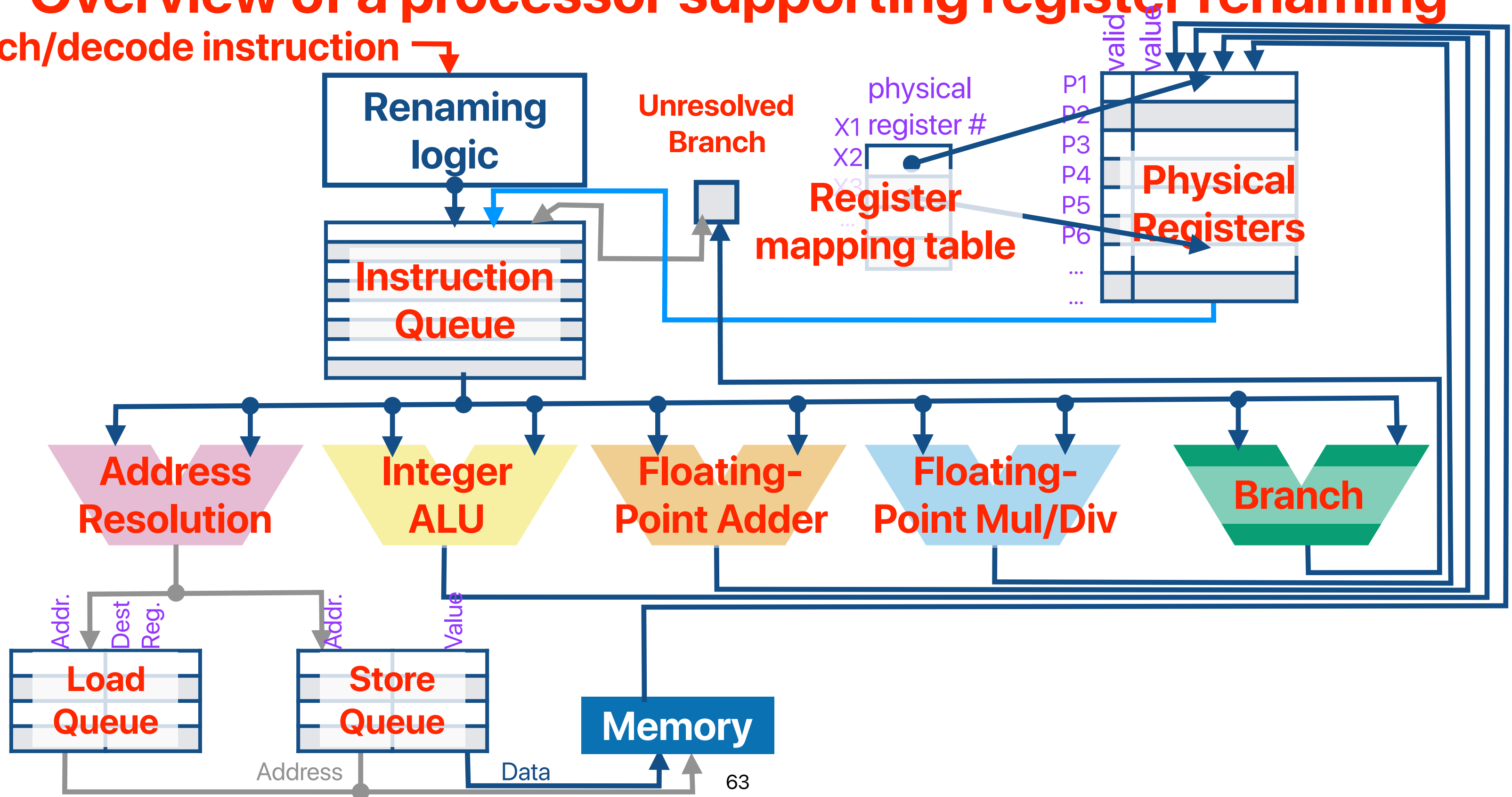
```
and x2, x1, 1
shr x4, x1, 1
shr x5, x1, 2
shr x6, x1, 3
shr x1, x1, 4
and x7, x4, 1
and x8, x5, 1
and x9, x6, 1
add x3, x3, x2
add x3, x3, x7
add x3, x3, x8
add x3, x3, x9
bne x1, x0, LOOP
```

Register renaming

- Decouple “reservation stations” from functional units
- Provide a set of “physical registers” and a mapping table mapping “architectural registers” to “physical registers”
- Allocate a physical register for a new output
- Stages
 - Dispatch (D) — allocate a “physical” for the output of a decoded instruction
 - Issue (I) — collect pending values/branch outcome from common data bus
 - Execute (INT, AQ/AQ/MEM, M1/M2/M3, BR) — send the instruction to its corresponding pipeline if no structural hazards
 - Write Back (WB) — broadcast the result through CDB

Overview of a processor supporting register renaming

Fetch/decode instruction →



Register renaming in motion

- ①

ld

X6, 0(X10)

R
- ②

add

X7, X6, X12
- ③

sd

X7, 0(X10)
- ④

addi

X10, X10, 8
- ⑤

bne

X10, X5, LOOP
- ⑥

ld

X6, 0(X10)
- ⑦

add

X7, X6, X12
- ⑧

sd

X7, 0(X10)
- ⑨

addi

X10, X10, 8
- ⑩

bne

X10, X5, LOOP

Renamed instruction		
1	ld	P1, 0(X10)
2		
3		
4		
5		
6		
7		
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	
X10	
X12	

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

Register renaming in motion

①

ld

X6, 0(X10)

R

I

②

add

X7, X6, X12

R

③

sd

X7, 0(X10)

④

addi

X10, X10, 8

⑤

bne

X10, X5, LOOP

⑥

ld

X6, 0(X10)

⑦

add

X7, X6, X12

⑧

sd

X7, 0(X10)

⑨

addi

X10, X10, 8

⑩

bne

X10, X5, LOOP

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3		
4		
5		
6		
7		
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	P2
X10	
X12	

	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	ld	X6, 0(X10)	R	I	AR
②	add	X7, X6, X12		R	I
③	sd	X7, 0(X10)			R
④	addi	X10, X10, 8			
⑤	bne	X10, X5, LOOP			
⑥	ld	X6, 0(X10)			
⑦	add	X7, X6, X12			
⑧	sd	X7, 0(X10)			
⑨	addi	X10, X10, 8			
⑩	bne	X10, X5, LOOP			

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4		
5		
6		
7		
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	P2
X10	
X12	

	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	ld	X6, 0(X10)	R	I	AR	LSQ
②	add	X7, X6, X12		R	I	I
③	sd	X7, 0(X10)			R	I
④	addi	X10, X10, 8				R
⑤	bne	X10, X5, LOOP				
⑥	ld	X6, 0(X10)				
⑦	add	X7, X6, X12				
⑧	sd	X7, 0(X10)				
⑨	addi	X10, X10, 8				
⑩	bne	X10, X5, LOOP				

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5		
6		
7		
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	P2
X10	P3
X12	

	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	ld	X6, 0(X10)	R	I	AR	LSQ	MEM
②	add	X7, X6, X12		R	I	I	I
③	sd	X7, 0(X10)			R	I	I
④	addi	X10, X10, 8				R	I
⑤	bne	X10, X5, LOOP					R
⑥	ld	X6, 0(X10)					
⑦	add	X7, X6, X12					
⑧	sd	X7, 0(X10)					
⑨	addi	X10, X10, 8					
⑩	bne	X10, X5, LOOP					

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6		
7		
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	P2
X10	P3
X12	

	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	ld	X6, 0(X10)	R	I	AR	LSQ	MEM	WB
②	add	X7, X6, X12		R	I	I	I	I
③	sd	X7, 0(X10)			R	I	I	I
④	addi	X10, X10, 8				R	I	INT
⑤	bne	X10, X5, LOOP					R	I
⑥	ld	X6, 0(X10)						R
⑦	add	X7, X6, X12						
⑧	sd	X7, 0(X10)						
⑨	addi	X10, X10, 8						
⑩	bne	X10, X5, LOOP						

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7		
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	P2
X10	P3
X12	

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

Register renaming in motion

①	ld	X6, 0(X10)	R	I	AR	LSQ	MEM	WB	
②	add	X7, X6, X12		R	I	I	I	I	INT
③	sd	X7, 0(X10)			R	I	I	I	I
④	addi	X10, X10, 8				R	I	INT	WB
⑤	bne	X10, X5, LOOP					R	I	I
⑥	ld	X6, 0(X10)						R	I
⑦	add	X7, X6, X12							R
⑧	sd	X7, 0(X10)							
⑨	addi	X10, X10, 8							
⑩	bne	X10, X5, LOOP							

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Register renaming in motion

①	ld	X6, 0(X10)	R	I	AR	LSQ	MEM	WB		
②	add	X7, X6, X12		R	I	I	I	I	INT	WB
③	sd	X7, 0(X10)			R	I	I	I	I	I
④	addi	X10, X10, 8				R	I	INT	WB	
⑤	bne	X10, X5, LOOP					R	I	I	BR
⑥	ld	X6, 0(X10)						R	I	AR
⑦	add	X7, X6, X12							R	I
⑧	sd	X7, 0(X10)								R
⑨	addi	X10, X10, 8								
⑩	bne	X10, X5, LOOP								

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9		
10		

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Register renaming in motion

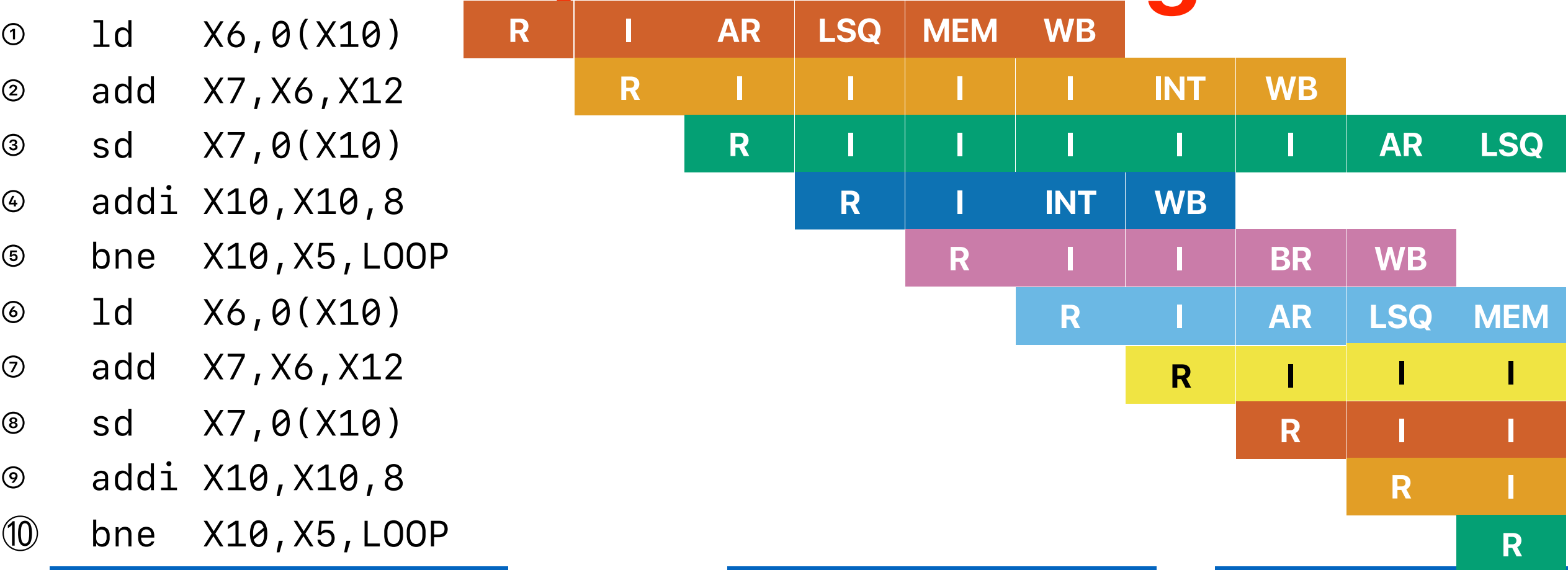
①	ld	X6, 0(X10)	R	I	AR	LSQ	MEM	WB										
②	add	X7, X6, X12		R	I	I	I	I	INT	WB								
③	sd	X7, 0(X10)			R	I	I	I	I	I	I	AR						
④	addi	X10, X10, 8				R	I	INT	WB									
⑤	bne	X10, X5, LOOP					R	I	I	BR	WB							
⑥	ld	X6, 0(X10)						R	I	AR	LSQ							
⑦	add	X7, X6, X12							R	I	I							
⑧	sd	X7, 0(X10)								R	I							
⑨	addi	X10, X10, 8										R						
⑩	bne	X10, X5, LOOP																

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10		

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Register renaming in motion

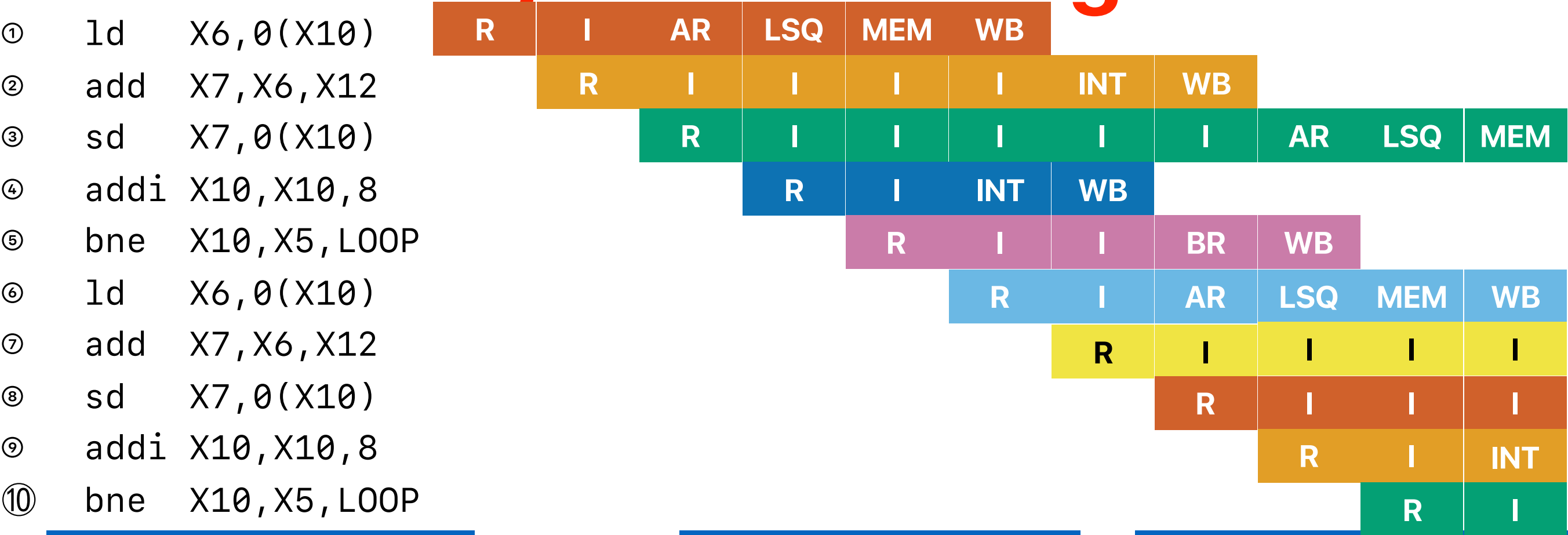


Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Register renaming in motion

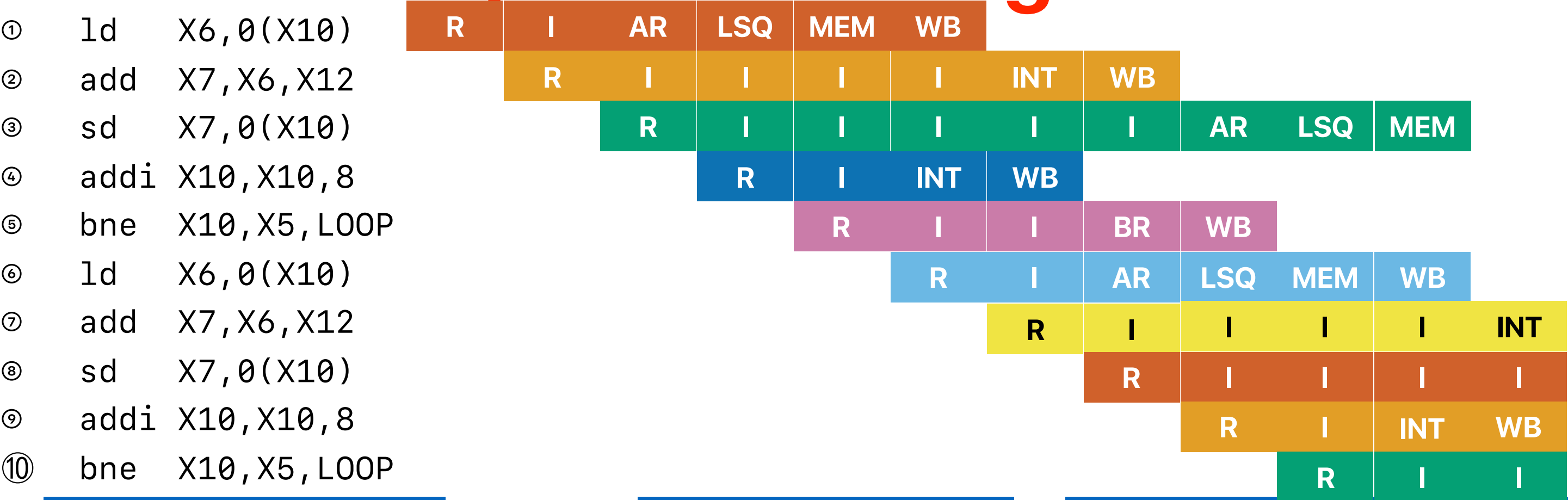


Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Register renaming in motion

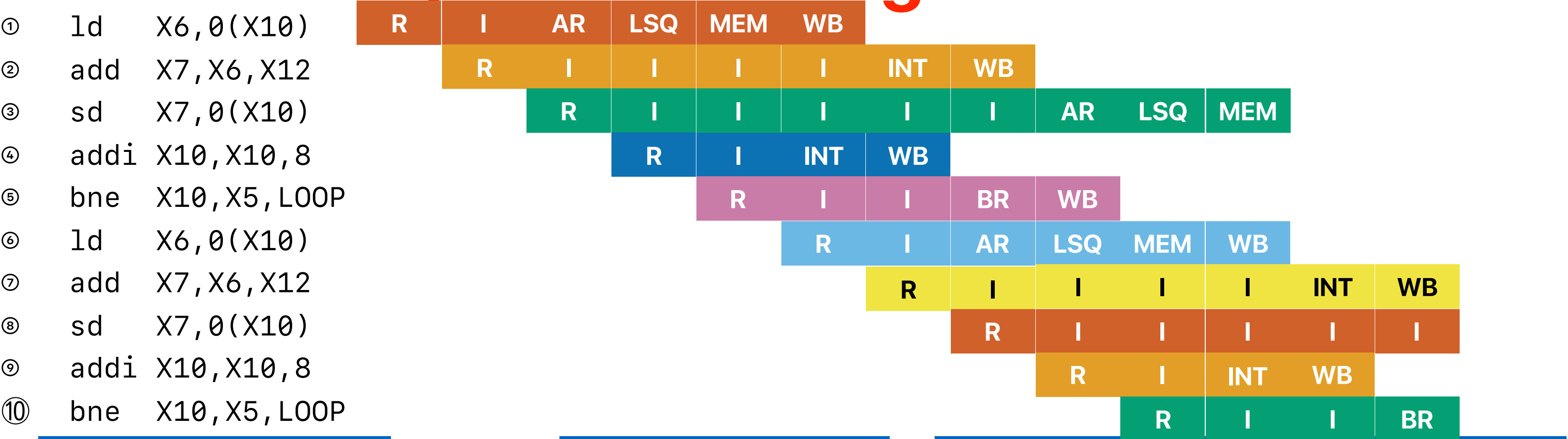


Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Register renaming in motion

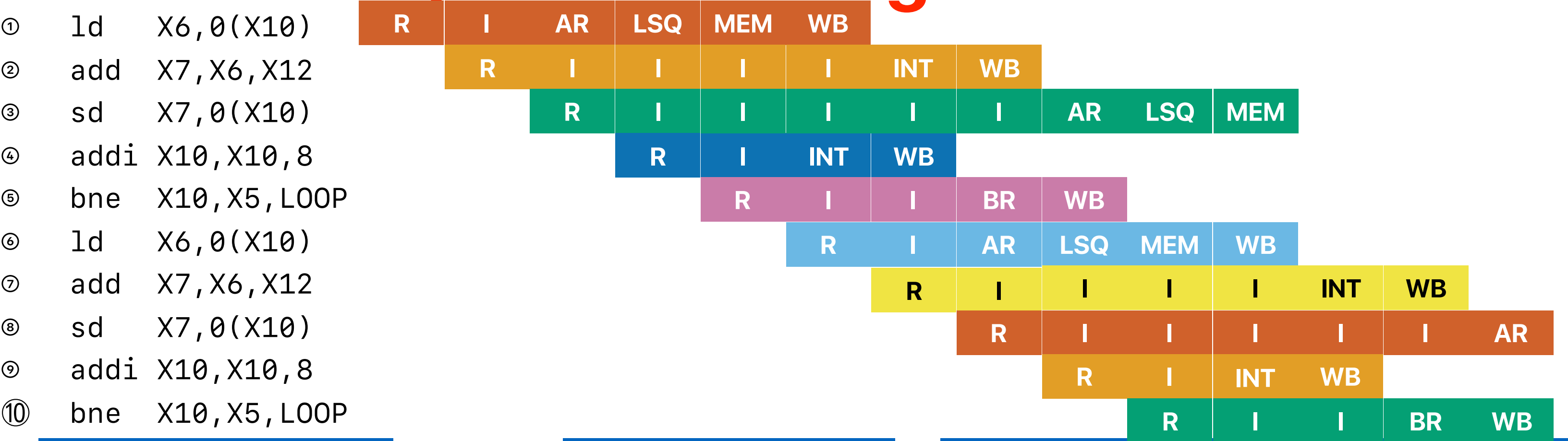


	Renamed instruction		
1	ld	P1,	0(X10)
2	add	P2,	P1, X12
3	sd	P2,	0(X10)
4	addi	P3,	X10, 8
5	bne	P3,	X5, LOOP
6	ld	P4,	0(P3)
7	add	P5,	P1, X12
8	sd	P5,	0(P3)
9	addi	P6,	P3, 8
10	bne	P6,	0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Register renaming in motion

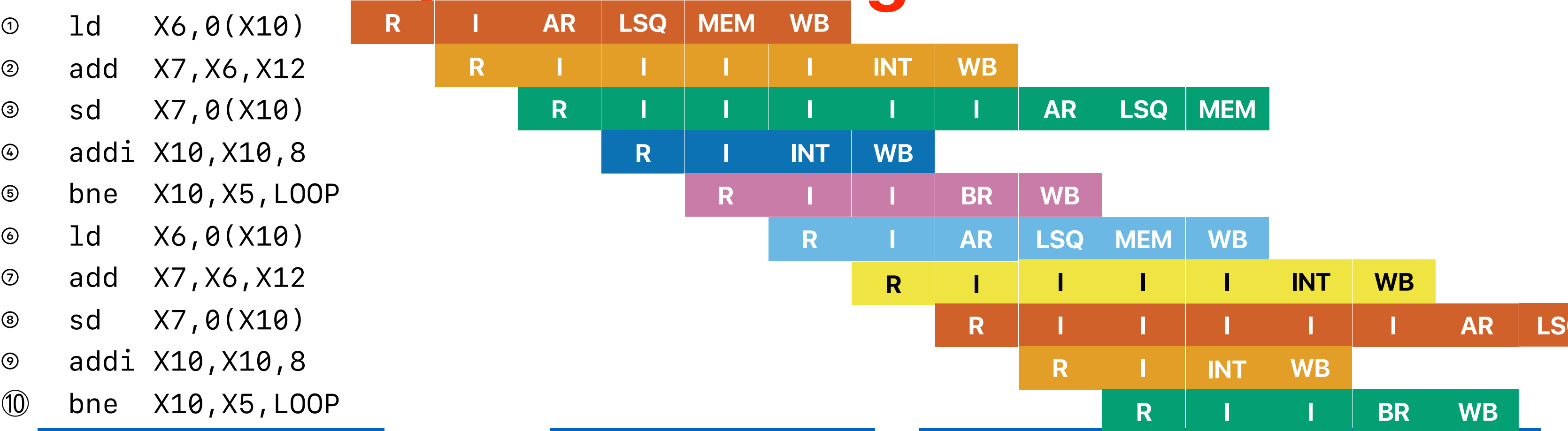


Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Register renaming in motion



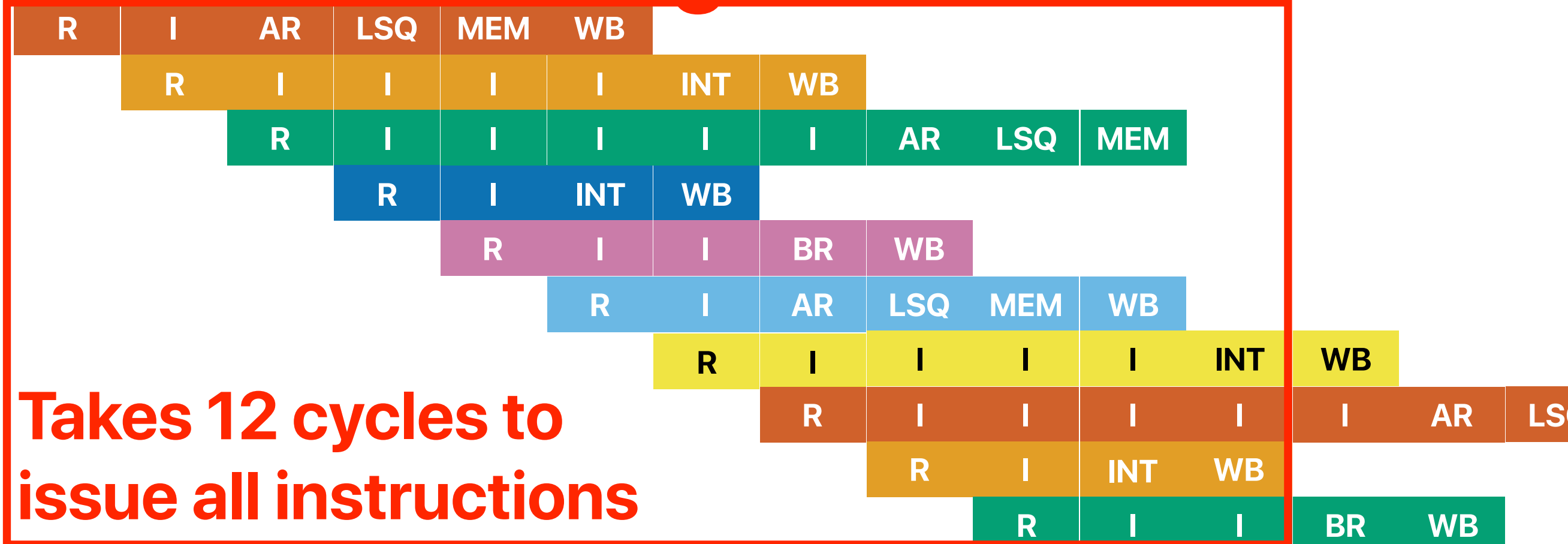
Renamed instruction	
1	ld P1, 0(X10)
2	add P2, P1, X12
3	sd P2, 0(X10)
4	addi P3, X10, 8
5	bne P3, X5, LOOP
6	ld P4, 0(P3)
7	add P5, P1, X12
8	sd P5, 0(P3)
9	addi P6, P3, 8
10	bne P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Register renaming in motion

- ① ld X6, 0(X10)
- ② add X7, X6, X12
- ③ sd X7, 0(X10)
- ④ addi X10, X10, 8
- ⑤ bne X10, X5, LOOP
- ⑥ ld X6, 0(X10)
- ⑦ add X7, X6, X12
- ⑧ sd X7, 0(X10)
- ⑨ addi X10, X10, 8
- ⑩ bne X10, X5, LOOP



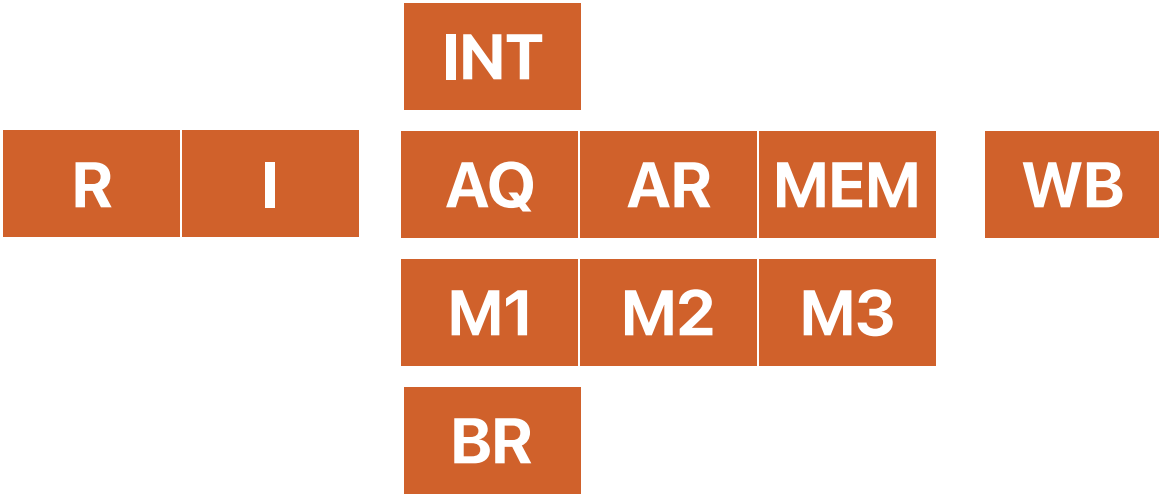
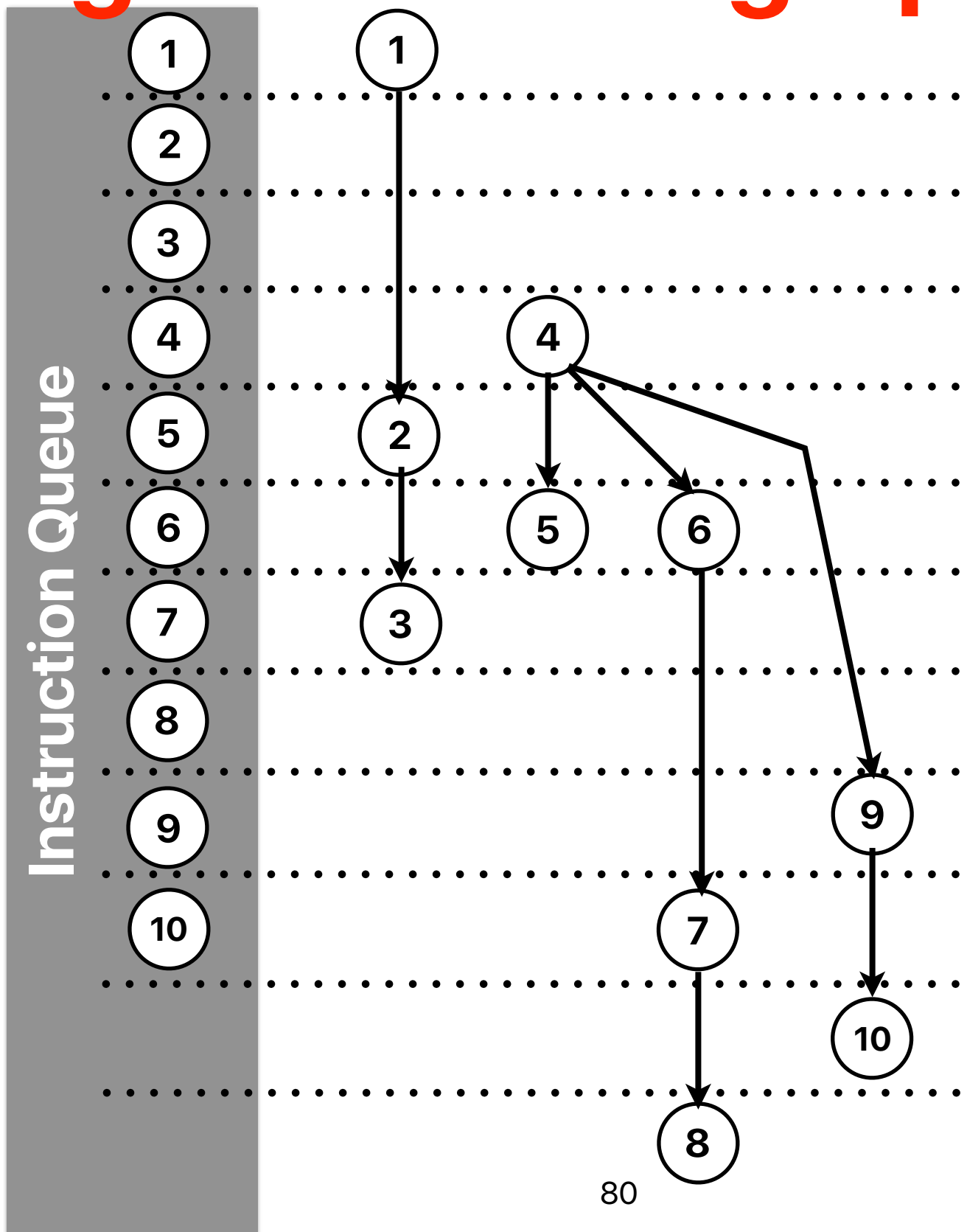
Renamed instruction	
1	ld P1, 0(X10)
2	add P2, P1, X12
3	sd P2, 0(X10)
4	addi P3, X10, 8
5	bne P3, X5, LOOP
6	ld P4, 0(P3)
7	add P5, P1, X12
8	sd P5, 0(P3)
9	addi P6, P3, 8
10	bne P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Through data flow graph analysis

① ld X6, 0(X10)
② add X7, X6, X12
③ sd X7, 0(X10)
④ addi X10, X10, 8
⑤ bne X10, X5, LOOP
⑥ ld X6, 0(X10)
⑦ add X7, X6, X12
⑧ sd X7, 0(X10)
⑨ addi X10, X10, 8
⑩ bne X10, X5, LOOP



INT — 2 cycles for depending instruction to start
MEM — 4 cycles for the depending instruction to start
MUL/DIV — 4 cycles for the depending instruction to start
BR — 2 cycles to resolve

Super Scalar

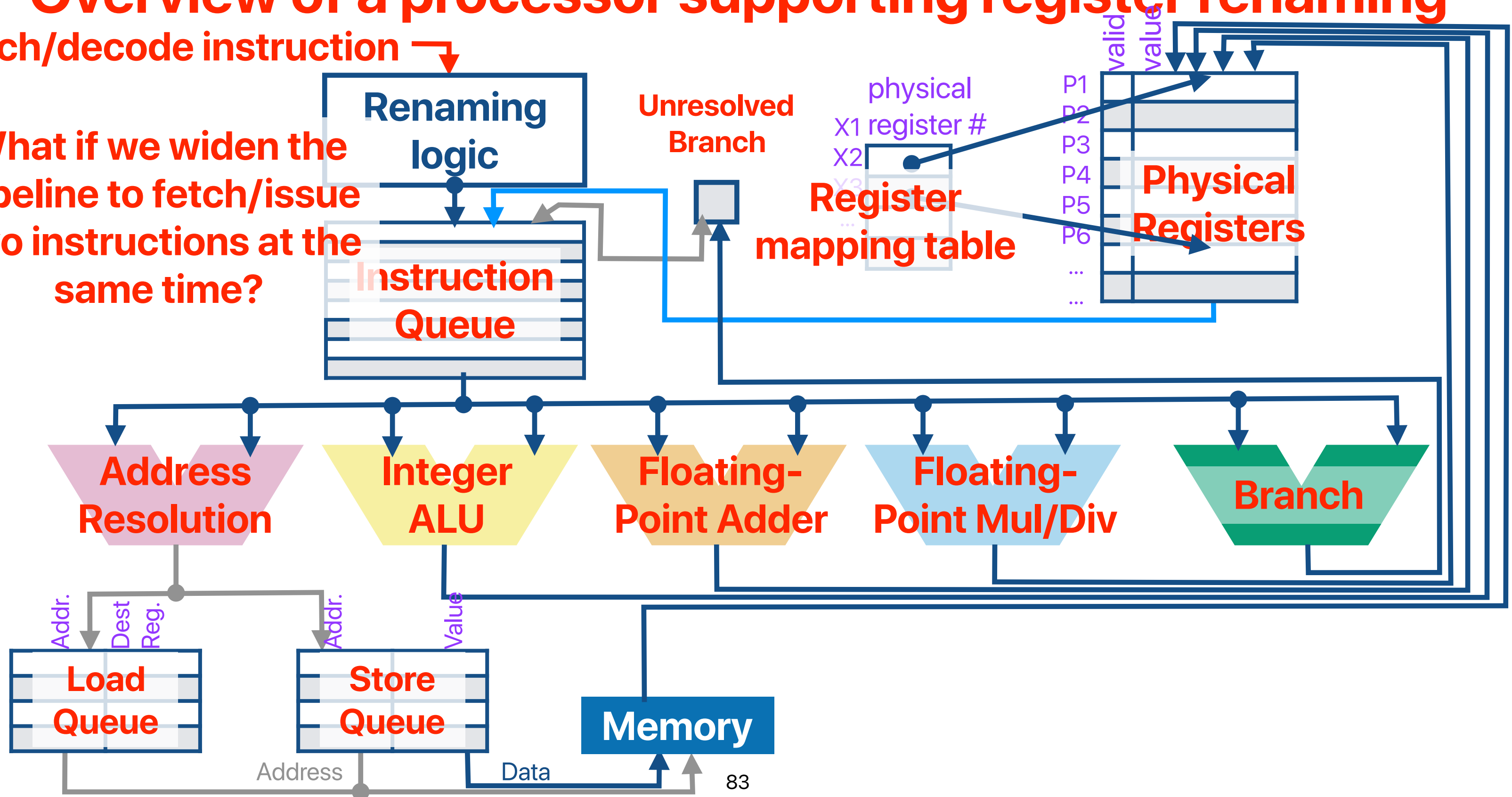
Superscalar

- Since we have more 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

Overview of a processor supporting register renaming

Fetch/decode instruction →

What if we widen the pipeline to fetch/issue two instructions at the same time?



2-issue RR processor in motion

- ① ld X6, 0(X10) R
- ② add X7, X6, X12 R
- ③ sd X7, 0(X10)
- ④ addi X10, X10, 8
- ⑤ bne X10, X5, LOOP
- ⑥ ld X6, 0(X10)
- ⑦ add X7, X6, X12
- ⑧ sd X7, 0(X10)
- ⑨ addi X10, X10, 8
- ⑩ bne X10, X5, LOOP

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3		
4		
5		
6		
7		
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	P2
X10	
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I
②	add	X7, X6, X12	R	I
③	sd	X7, 0(X10)		R
④	addi	X10, X10, 8		R
⑤	bne	X10, X5, LOOP		
⑥	ld	X6, 0(X10)		
⑦	add	X7, X6, X12		
⑧	sd	X7, 0(X10)		
⑨	addi	X10, X10, 8		
⑩	bne	X10, X5, LOOP		

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5		
6		
7		
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	P2
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR
②	add	X7, X6, X12	R	I	I
③	sd	X7, 0(X10)		R	I
④	addi	X10, X10, 8		R	I
⑤	bne	X10, X5, LOOP			R
⑥	ld	X6, 0(X10)			R
⑦	add	X7, X6, X12			
⑧	sd	X7, 0(X10)			
⑨	addi	X10, X10, 8			
⑩	bne	X10, X5, LOOP			

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7		
8		
9		
10		

Physical Register	
X5	
X6	P1
X7	P2
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ
②	add	X7, X6, X12	R	I	I	I
③	sd	X7, 0(X10)		R	I	I
④	addi	X10, X10, 8		R	I	INT
⑤	bne	X10, X5, LOOP			R	I
⑥	ld	X6, 0(X10)			R	I
⑦	add	X7, X6, X12				R
⑧	sd	X7, 0(X10)				R
⑨	addi	X10, X10, 8				
⑩	bne	X10, X5, LOOP				

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9		
10		

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM
②	add	X7, X6, X12	R	I	I	I	I
③	sd	X7, 0(X10)		R	I	I	I
④	addi	X10, X10, 8		R	I	INT	WB
⑤	bne	X10, X5, LOOP			R	I	I
⑥	ld	X6, 0(X10)			R	I	I
⑦	add	X7, X6, X12				R	I
⑧	sd	X7, 0(X10)				R	I
⑨	addi	X10, X10, 8					R
⑩	bne	X10, X5, LOOP					R

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB
②	add	X7, X6, X12	R	I	I	I	I	I
③	sd	X7, 0(X10)		R	I	I	I	I
④	addi	X10, X10, 8		R	I	INT	WB	
⑤	bne	X10, X5, LOOP			R	I	I	BR
⑥	ld	X6, 0(X10)			R	I	I	AR
⑦	add	X7, X6, X12				R	I	I
⑧	sd	X7, 0(X10)				R	I	I
⑨	addi	X10, X10, 8					R	I
⑩	bne	X10, X5, LOOP					R	I

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB	
②	add	X7, X6, X12	R	I	I	I	I	I	INT
③	sd	X7, 0(X10)		R	I	I	I	I	I
④	addi	X10, X10, 8		R	I	INT	WB		
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ
⑦	add	X7, X6, X12				R	I	I	I
⑧	sd	X7, 0(X10)				R	I	I	I
⑨	addi	X10, X10, 8					R	I	I
⑩	bne	X10, X5, LOOP					R	I	I

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB		
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB
③	sd	X7, 0(X10)		R	I	I	I	I	I	I
④	addi	X10, X10, 8		R	I	INT	WB			
⑤	bne	X10, X5, LOOP			R	I	I	I	BR	WB
⑥	ld	X6, 0(X10)			R	I	I	I	AR	AQ
⑦	add	X7, X6, X12				R	I	I	I	I
⑧	sd	X7, 0(X10)				R	I	I	I	I
⑨	addi	X10, X10, 8					R	I	I	INT
⑩	bne	X10, X5, LOOP					R	I	I	I

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB											
②	add	X7, X6, X12	R	I	I	I	I	I	I	INT	WB								
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	I	I						AR	
④	addi	X10, X10, 8		R	I	INT	WB												
⑤	bne	X10, X5, LOOP			R	I	I	I	BR	WB									
⑥	ld	X6, 0(X10)			R	I	I	I	AR	AQ	MEM	WB							
⑦	add	X7, X6, X12				R	I	I	I	I	I	I	I						
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I	I						
⑨	addi	X10, X10, 8					R	I	I	I	INT	WB							
⑩	bne	X10, X5, LOOP					R	I	I	I	I	I	I						

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB											
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB									
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR	AQ							
④	addi	X10, X10, 8		R	I	INT	WB												
⑤	bne	X10, X5, LOOP			R	I	I	I	BR	WB									
⑥	ld	X6, 0(X10)			R	I	I	I	AR	AQ	MEM	WB							
⑦	add	X7, X6, X12				R	I	I	I	I	I	I	INT						
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I	I						
⑨	addi	X10, X10, 8					R	I	I	INT	WB								
⑩	bne	X10, X5, LOOP					R	I	I	I	I	I	BR						

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB						
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB				
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR	AQ		
④	addi	X10, X10, 8		R	I	INT	WB							
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB					
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ	MEM	WB			
⑦	add	X7, X6, X12				R	I	I	I	I	I	INT		
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I		
⑨	addi	X10, X10, 8					R	I	I	INT	WB			
⑩	bne	X10, X5, LOOP					R	I	I	I	I	BR		

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB											
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB									
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR	AQ	MEM						
④	addi	X10, X10, 8		R	I	INT	WB												
⑤	bne	X10, X5, LOOP			R	I	I	I	BR	WB									
⑥	ld	X6, 0(X10)			R	I	I	I	AR	AQ	MEM	WB							
⑦	add	X7, X6, X12				R	I	I	I	I	I	I	INT	WB					
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I	I	I					
⑨	addi	X10, X10, 8					R	I	I	INT	WB								
⑩	bne	X10, X5, LOOP					R	I	I	I	I	I	BR	WB					

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB											
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB									
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR	AQ	MEM						
④	addi	X10, X10, 8		R	I	INT	WB												
⑤	bne	X10, X5, LOOP			R	I	I	I	BR	WB									
⑥	ld	X6, 0(X10)			R	I	I	I	AR	AQ	MEM	WB							
⑦	add	X7, X6, X12				R	I	I	I	I	I	I	INT	WB					
⑧	sd	X7, 0(X10)					R	I	I	I	I	I	I	I	AR				
⑨	addi	X10, X10, 8						R	I	INT	WB								
⑩	bne	X10, X5, LOOP							R	I	I	I	I	BR	WB				

Takes 10 cycles to issue all instructions

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB										
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB								
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR	AQ	MEM					
④	addi	X10, X10, 8		R	I	INT	WB											
⑤	bne	X10, X5, LOOP			R	I	I	I	BR	WB								
⑥	ld	X6, 0(X10)			R	I	I	I	AR	AQ	MEM	WB						
⑦	add	X7, X6, X12				R	I	I	I	I	I	I	INT	WB				
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I	I	I	AR	AQ		
⑨	addi	X10, X10, 8					R	I	I	INT	WB							
⑩	bne	X10, X5, LOOP					R	I	I	I	I	I	BR	WB				

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB											
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB									
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR	AQ	MEM						
④	addi	X10, X10, 8		R	I	INT	WB												
⑤	bne	X10, X5, LOOP			R	I	I	I	BR	WB									
⑥	ld	X6, 0(X10)			R	I	I	I	AR	AQ	MEM	WB							
⑦	add	X7, X6, X12				R	I	I	I	I	I	I	INT	WB					
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I	I	I	AR	AQ	MEM		
⑨	addi	X10, X10, 8					R	I	I	INT	WB								
⑩	bne	X10, X5, LOOP					R	I	I	I	I	I	BR	WB					

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

What about “linked list”

- For the following C code and its translation in RISC-V, how many cycles it takes the processor to issue all instructions? Assume the current PC is already at the first instruction and this linked list has only three nodes. This processor only fetches 1 instruction per cycle, with exactly the same register renaming hardware and pipeline as we showed previously.

```
do {  
    number_of_nodes++;  
    current = current->next;  
} while ( current != NULL )
```

```
LOOP: ld    X10, 8(X10)  
      addi  X7, X7, 1  
      bne   X10, X0, LOOP
```

- A. 9
- B. 10
- C. 11
- D. 12
- E. 13

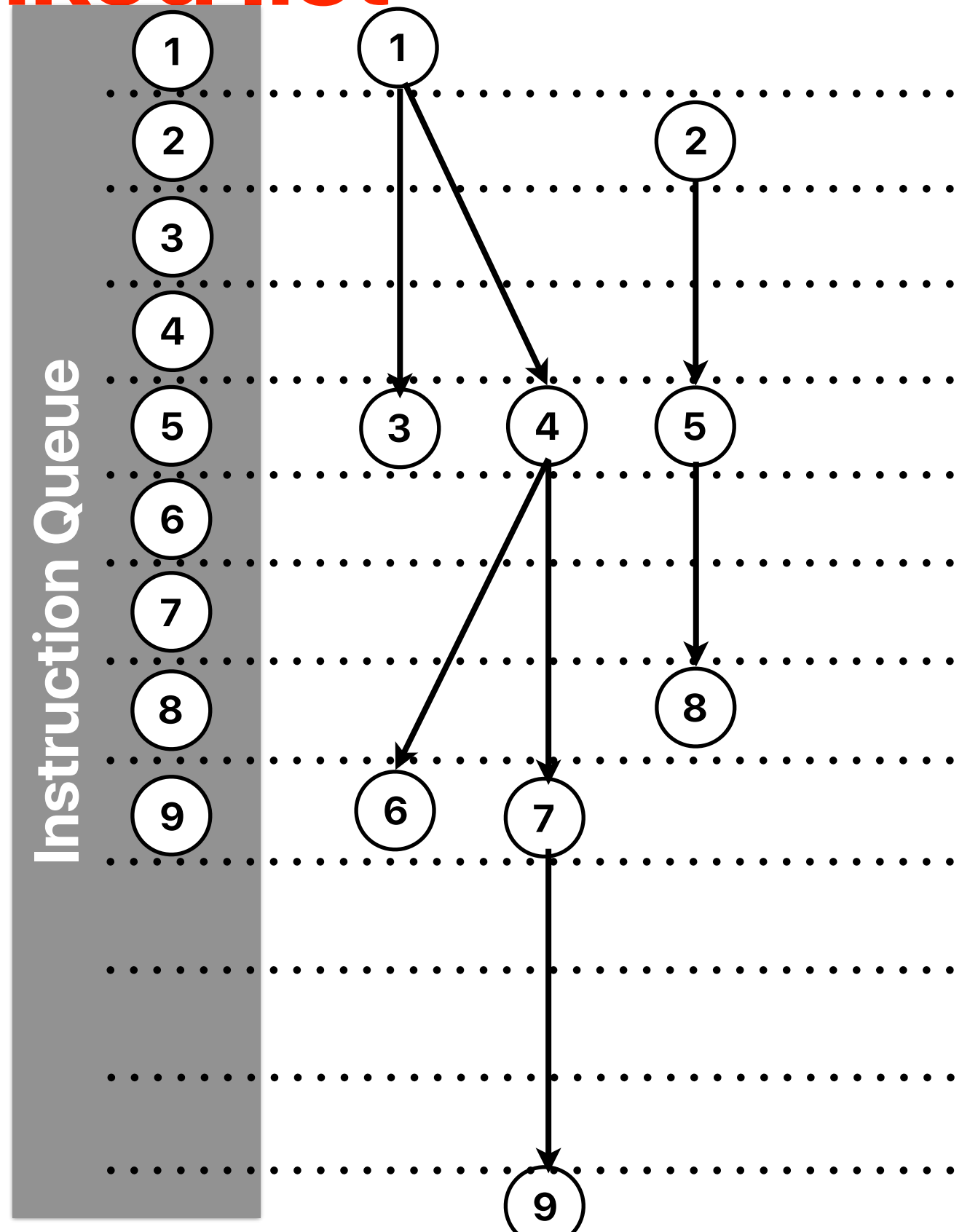
What about "linked list"

Static instructions

```
LOOP: ld    X10, 8(X10)
      addi  X7, X7, 1
      bne   X10, X0, LOOP
```

Dynamic instructions

```
① ld    X10, 8(X10)
② addi  X7, X7, 1
③ bne   X10, X0, LOOP
④ ld    X10, 8(X10)
⑤ addi  X7, X7, 1
⑥ bne   X10, X0, LOOP
⑦ ld    X10, 8(X10)
⑧ addi  X7, X7, 1
⑨ bne   X10, X0, LOOP
```



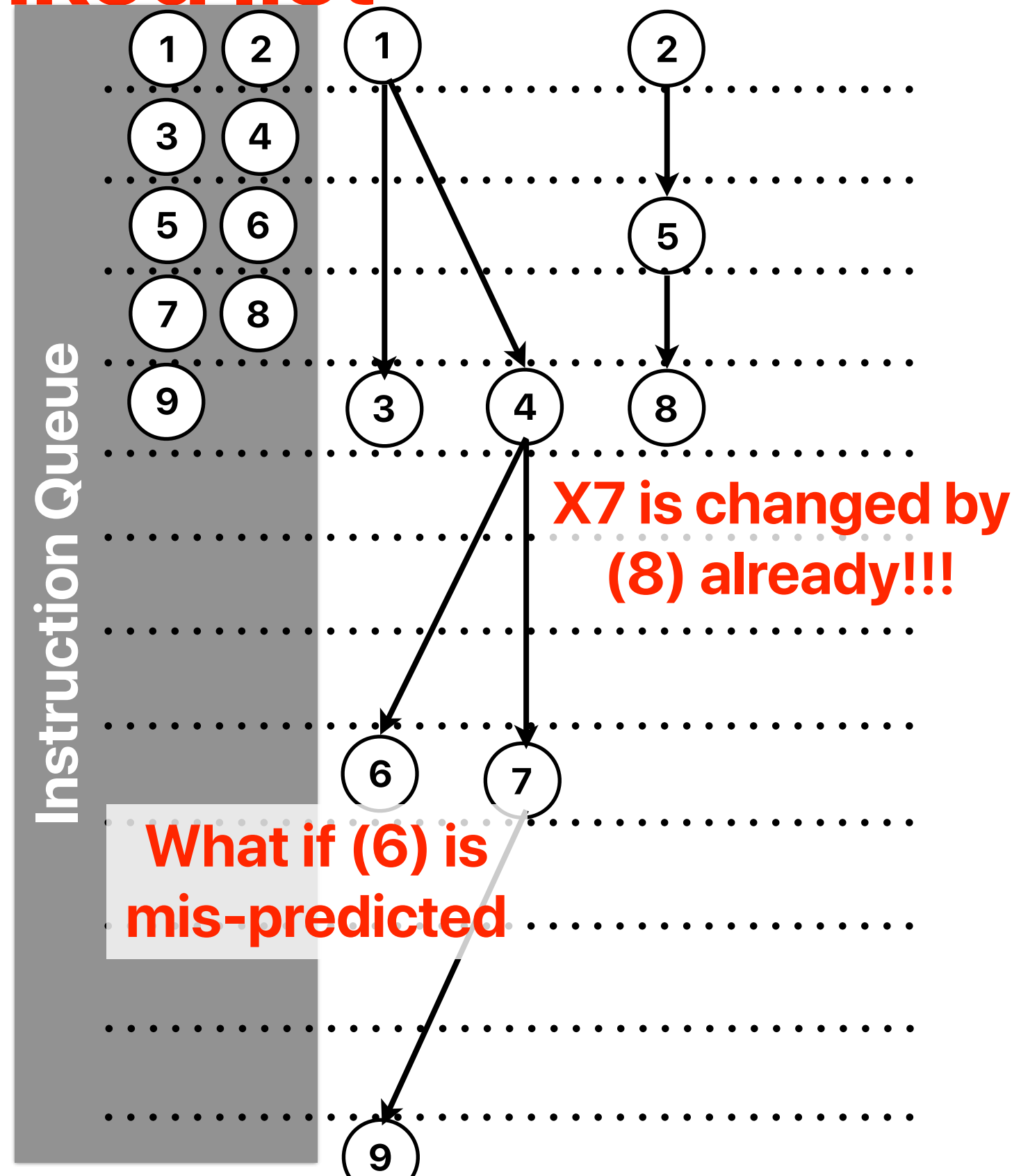
What about "linked list"

Static instructions

```
LOOP: ld    X10, 8(X10)
      addi  X7, X7, 1
      bne   X10, X0, LOOP
```

Dynamic instructions

```
① ld    X10, 8(X10)
② addi  X7, X7, 1
③ bne   X10, X0, LOOP
④ ld    X10, 8(X10)
⑤ addi  X7, X7, 1
⑥ bne   X10, X0, LOOP
⑦ ld    X10, 8(X10)
⑧ addi  X7, X7, 1
⑨ bne   X10, X0, LOOP
```



Without additional mechanisms...

- If we meet a branch
 - Any instruction after the branch cannot be issued
 - If the branch is mispredicted — flush the instruction queue entries of later instructions

Hurt performance!

In which pipeline stage can we have exceptions?

- How many of the following pipeline stages can we have exceptions?

- ① IF — **page fault, illegal address**
- ② ID — **unknown instruction**
- ③ EXE — **divide by zero, overflow, underflow**
- ④ MEM — **page fault, illegal address**
- ⑤ WB

A. 1

B. 2

C. 3

D. 4

E. 5

Without additional mechanisms...

- If we meet a branch
 - Any instruction after the branch cannot be issued
 - If the branch is mispredicted — flush the instruction queue entries of later instructions
- If we have to handle exceptions precisely
 - All instruction cannot be issued out-of-order!

Hurt performance!
OoO becomes useless!

Make OoO great again!
— Reorder Buffer (ROB)

Speculative Execution

- Any execution of an instruction before any 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

2-issue RR processor in motion

- ① ld X6, 0(X10) R
- ② add X7, X6, X12 R
- ③ sd X7, 0(X10)
- ④ addi X10, X10, 8
- ⑤ bne X10, X5, LOOP
- ⑥ ld X6, 0(X10)
- ⑦ add X7, X6, X12
- ⑧ sd X7, 0(X10)
- ⑨ addi X10, X10, 8
- ⑩ bne X10, X5, LOOP

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3		
4		
5		
6		
7		
8		
9		
10		

head
tail

Physical Register	
X5	
X6	P1
X7	P2
X10	
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I
②	add	X7, X6, X12	R	I
③	sd	X7, 0(X10)		R
④	addi	X10, X10, 8		R
⑤	bne	X10, X5, LOOP		
⑥	ld	X6, 0(X10)		
⑦	add	X7, X6, X12		
⑧	sd	X7, 0(X10)		
⑨	addi	X10, X10, 8		
⑩	bne	X10, X5, LOOP		

Renamed instruction			
1	ld	P1, 0(X10)	head
2	add	P2, P1, X12	
3	sd	P2, 0(X10)	tail
4	addi	P3, X10, 8	
5			
6			
7			
8			
9			
10			

Physical Register	
X5	
X6	P1
X7	P2
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR
②	add	X7, X6, X12	R	I	I
③	sd	X7, 0(X10)		R	I
④	addi	X10, X10, 8		R	I
⑤	bne	X10, X5, LOOP			R
⑥	ld	X6, 0(X10)			R
⑦	add	X7, X6, X12			
⑧	sd	X7, 0(X10)			
⑨	addi	X10, X10, 8			
⑩	bne	X10, X5, LOOP			

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7		
8		
9		
10		

← head

← tail

Physical Register	
X5	
X6	P1
X7	P2
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ
②	add	X7, X6, X12	R	I	I	I
③	sd	X7, 0(X10)		R	I	I
④	addi	X10, X10, 8		R	I	INT
⑤	bne	X10, X5, LOOP			R	I
⑥	ld	X6, 0(X10)			R	I
⑦	add	X7, X6, X12				R
⑧	sd	X7, 0(X10)				R
⑨	addi	X10, X10, 8				
⑩	bne	X10, X5, LOOP				

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9		
10		

← head

← tail

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM
②	add	X7, X6, X12	R	I	I	I	I
③	sd	X7, 0(X10)		R	I	I	I
④	addi	X10, X10, 8		R	I	INT	WB
⑤	bne	X10, X5, LOOP			R	I	I
⑥	ld	X6, 0(X10)			R	I	I
⑦	add	X7, X6, X12				R	I
⑧	sd	X7, 0(X10)				R	I
⑨	addi	X10, X10, 8					R
⑩	bne	X10, X5, LOOP					R

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

← head

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

← tail

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB
②	add	X7, X6, X12	R	I	I	I	I	I
③	sd	X7, 0(X10)		R	I	I	I	I
④	addi	X10, X10, 8		R	I	INT	WB	C
⑤	bne	X10, X5, LOOP			R	I	I	BR
⑥	ld	X6, 0(X10)			R	I	I	AR
⑦	add	X7, X6, X12				R	I	I
⑧	sd	X7, 0(X10)				R	I	I
⑨	addi	X10, X10, 8					R	I
⑩	bne	X10, X5, LOOP					R	I

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

← head

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

← tail

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB	C
②	add	X7, X6, X12	R	I	I	I	I	I	INT
③	sd	X7, 0(X10)		R	I	I	I	I	I
④	addi	X10, X10, 8		R	I	INT	WB	C	C
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ
⑦	add	X7, X6, X12				R	I	I	I
⑧	sd	X7, 0(X10)				R	I	I	I
⑨	addi	X10, X10, 8					R	I	I
⑩	bne	X10, X5, LOOP					R	I	I

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

← head

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

← tail

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB	C	
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB
③	sd	X7, 0(X10)		R	I	I	I	I	I	I
④	addi	X10, X10, 8		R	I	INT	WB	C	C	C
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB	C
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ	MEM
⑦	add	X7, X6, X12				R	I	I	I	I
⑧	sd	X7, 0(X10)				R	I	I	I	I
⑨	addi	X10, X10, 8					R	I	I	INT
⑩	bne	X10, X5, LOOP					R	I	I	I

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

← head

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

← tail

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB	C		
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB	C
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR
④	addi	X10, X10, 8		R	I	INT	WB	C	C	C	C
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB	C	C
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ	MEM	WB
⑦	add	X7, X6, X12				R	I	I	I	I	I
⑧	sd	X7, 0(X10)				R	I	I	I	I	I
⑨	addi	X10, X10, 8					R	I	I	INT	WB
⑩	bne	X10, X5, LOOP					R	I	I	I	I

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

← head

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

← tail

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB	C		
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB	C
③	sd	X7, 0(X10)		R	I	I	I	I	I	AR	AQ
④	addi	X10, X10, 8		R	I	INT	WB	C	C	C	C
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB	C	C
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ	MEM	WB
⑦	add	X7, X6, X12				R	I	I	I	I	INT
⑧	sd	X7, 0(X10)				R	I	I	I	I	I
⑨	addi	X10, X10, 8					R	I	I	INT	WB
⑩	bne	X10, X5, LOOP					R	I	I	I	I

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

← head

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

← tail

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB	C				
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB	C		
③	sd	X7, 0(X10)		R	I	I	I	I	I	AR	AQ	MEM	
④	addi	X10, X10, 8		R	I	INT	WB	C	C	C	C	C	
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB	C	C	C	
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ	MEM	WB	C	C
⑦	add	X7, X6, X12				R	I	I	I	I	I	INT	WB
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I	I
⑨	addi	X10, X10, 8					R	I	I	INT	WB	C	C
⑩	bne	X10, X5, LOOP					R	I	I	I	I	BR	WB

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

← head

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

← tail

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB	C										
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB	C								
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR	AQ	MEM	C					
④	addi	X10, X10, 8		R	I	INT	WB	C	C	C	C	C	C	C	C	C			
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB	C	C	C	C	C	C	C			
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ	MEM	WB	C	C	C	C	C			
⑦	add	X7, X6, X12				R	I	I	I	I	I	INT	WB	C					
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I	I	AR					
⑨	addi	X10, X10, 8					R	I	I	INT	WB	C	C	C	C				
⑩	bne	X10, X5, LOOP					R	I	I	I	I	BR	WB	C					

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

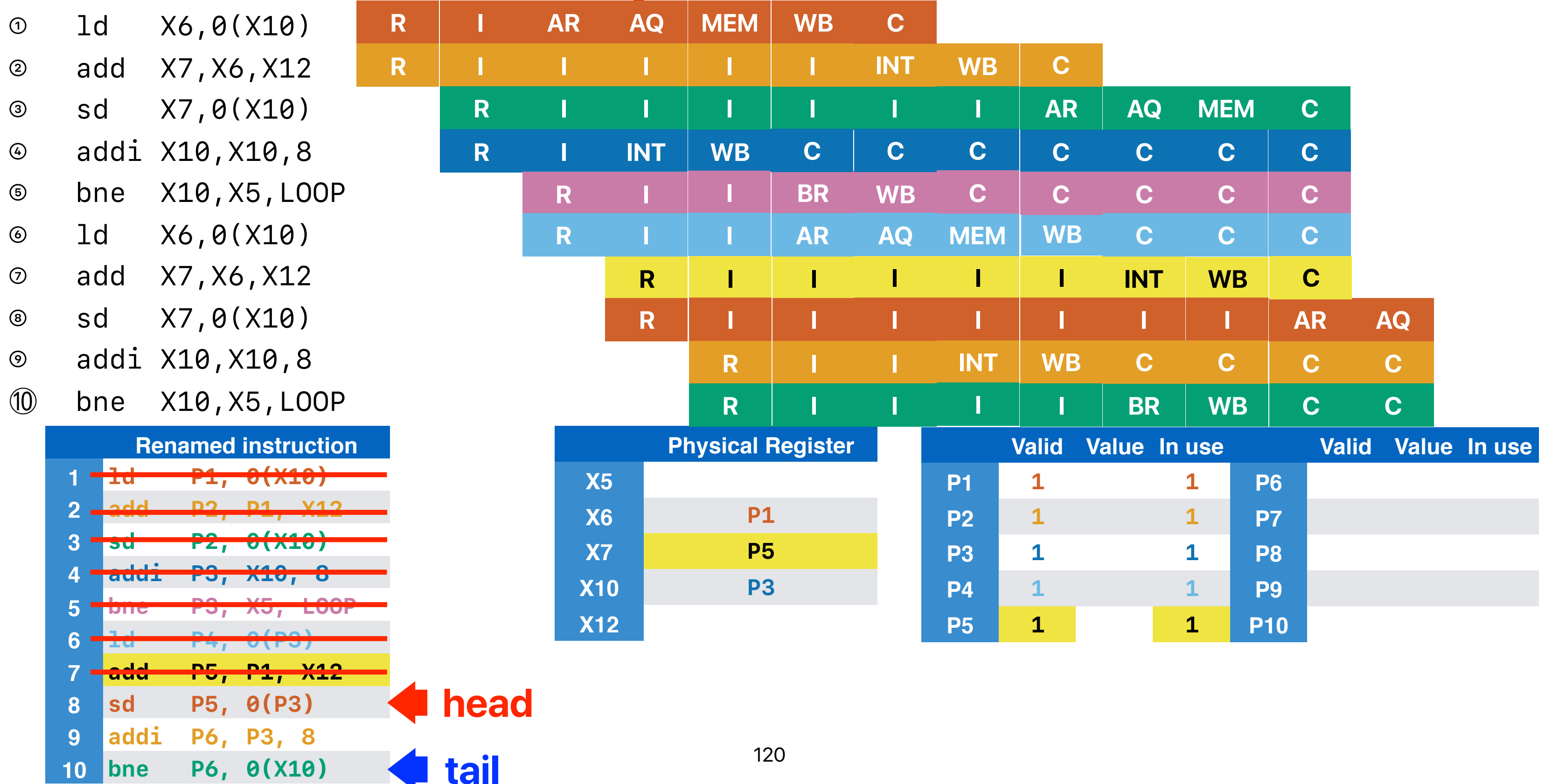
← head

← tail

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion



2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB	C										
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB	C								
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR	AQ	MEM	C					
④	addi	X10, X10, 8		R	I	INT	WB	C	C	C	C	C	C	C	C	C			
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB	C	C	C	C	C	C				
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ	MEM	WB	C	C	C					
⑦	add	X7, X6, X12				R	I	I	I	I	I	INT	WB	C					
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I	AR	AQ	MEM				
⑨	addi	X10, X10, 8					R	I	I	INT	WB	C	C	C	C	C			
⑩	bne	X10, X5, LOOP						R	I	I	I	I	BR	WB	C	C	C		

Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

← head

← tail

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

2-issue RR processor in motion

①	ld	X6, 0(X10)	R	I	AR	AQ	MEM	WB	C										
②	add	X7, X6, X12	R	I	I	I	I	I	INT	WB	C								
③	sd	X7, 0(X10)		R	I	I	I	I	I	I	AR	AQ	MEM	C					
④	addi	X10, X10, 8		R	I	INT	WB	C	C	C	C	C	C	C	C	C			
⑤	bne	X10, X5, LOOP			R	I	I	BR	WB	C	C	C	C	C	C				
⑥	ld	X6, 0(X10)			R	I	I	AR	AQ	MEM	WB	C	C	C	C				
⑦	add	X7, X6, X12				R	I	I	I	I	I	INT	WB	C					
⑧	sd	X7, 0(X10)				R	I	I	I	I	I	I	I	AR	AQ	MEM	C		
⑨	addi	X10, X10, 8					R	I	I	INT	WB	C	C	C	C	C	C		
⑩	bne	X10, X5, LOOP					R	I	I	I	I	BR	WB	C	C	C	C	C	

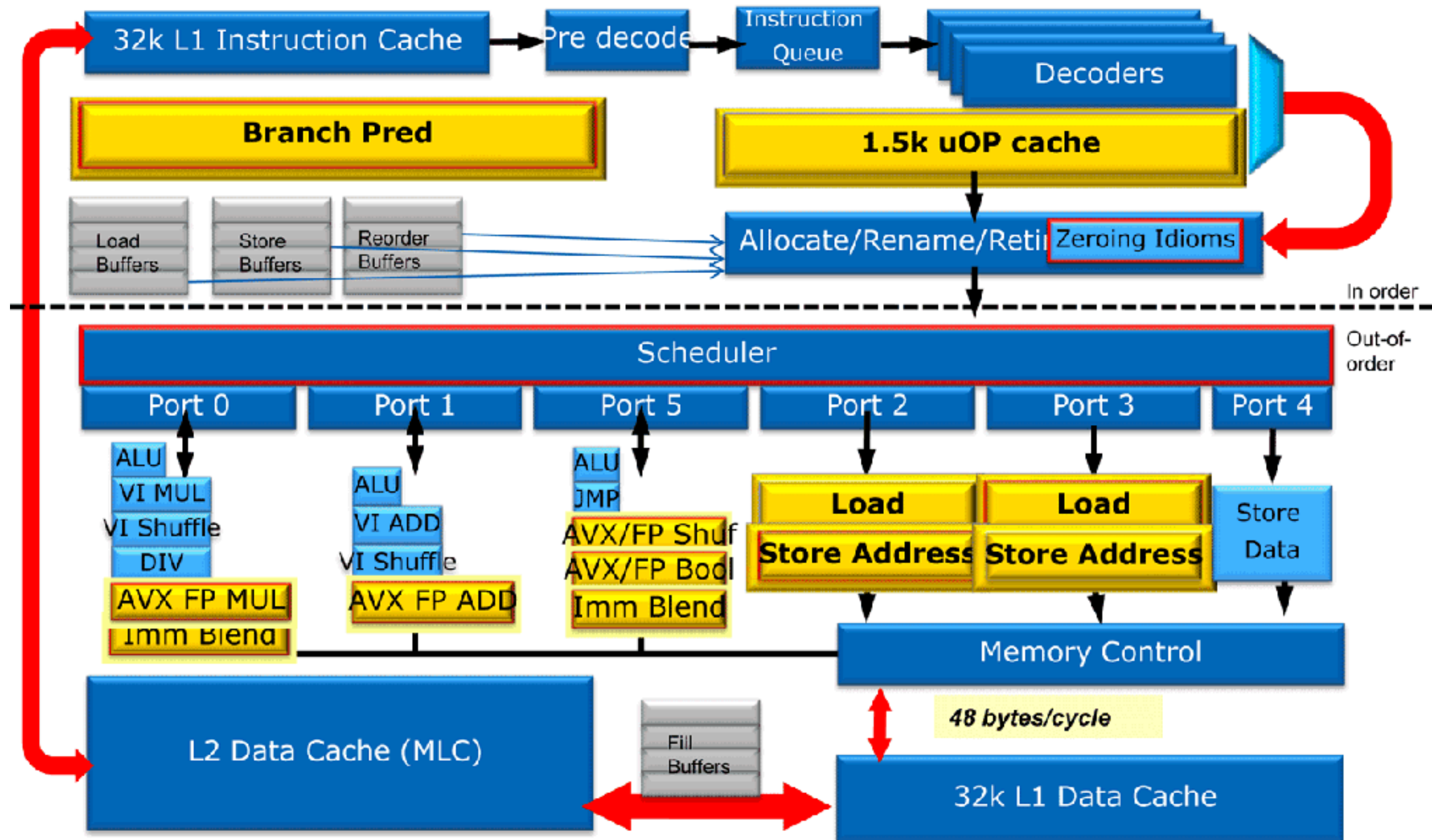
Renamed instruction		
1	ld	P1, 0(X10)
2	add	P2, P1, X12
3	sd	P2, 0(X10)
4	addi	P3, X10, 8
5	bne	P3, X5, LOOP
6	ld	P4, 0(P3)
7	add	P5, P1, X12
8	sd	P5, 0(P3)
9	addi	P6, P3, 8
10	bne	P6, 0(X10)

tail

Physical Register	
X5	
X6	P1
X7	P5
X10	P3
X12	

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

Intel Sandy Bridge



How good is SS/OoO/ROB with this code?

- Consider the following dynamic instructions

① ld X1, 0(X10)
② addi X10, X10, 8
③ add X20, X20, X1
④ bne X10, X2, LOOP

Assume a superscalar processor with issue width as 2 & unit delay as 1 cycle. The processor can fetch up to 4 instructions per cycle, 3 cycles to execute each instruction and the loop will execute for 10,000 times, what's the average CPI?

A. 0.5

B. 0.75

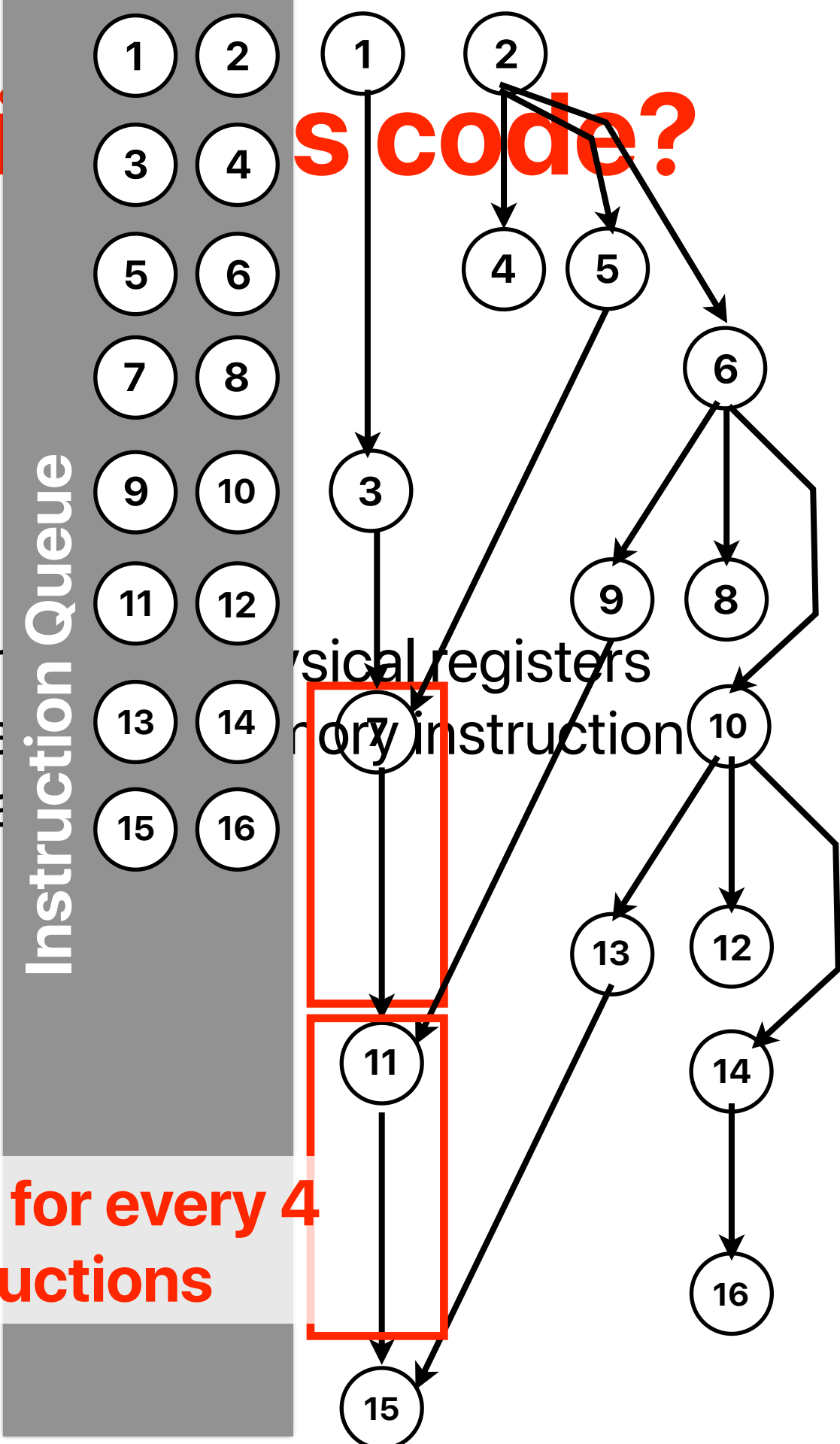
C. 1

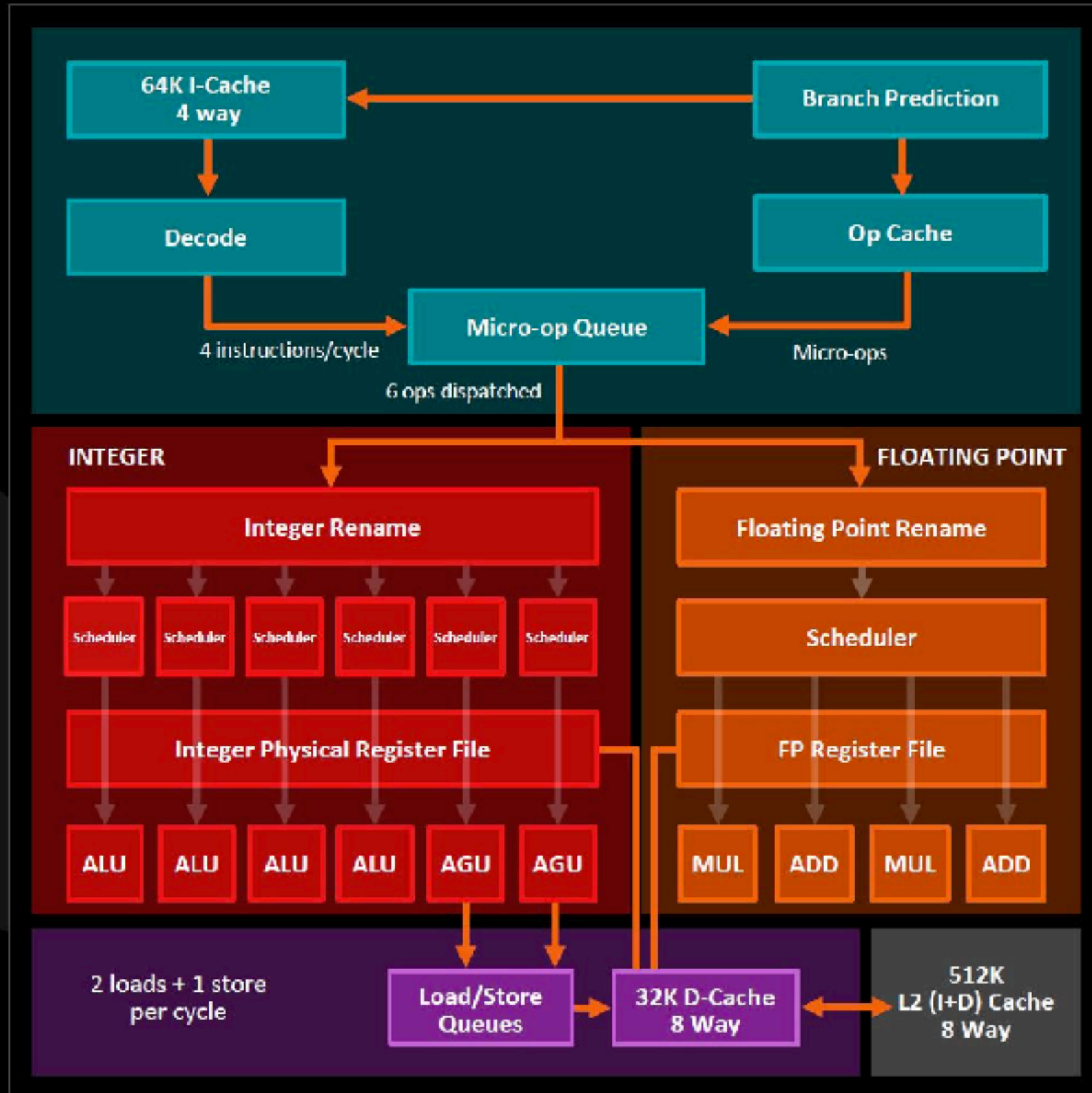
D. 1.25

E. 1.5

① ld X1, 0(X10)
② addi X10, X10, 8
③ add X20, X20, X1
④ bne X10, X2, LOOP
⑤ ld X1, 0(X10)
⑥ addi X10, X10, 8
⑦ add X20, X20, X1
⑧ bne X10, X2, LOOP
⑨ ld X1, 0(X10)
⑩ addi X10, X10, 8
⑪ add X20, X20, X1
⑫ bne X10, X2, LOOP

124





ZEN MICROARCHITECTURE

- ▲ Fetch Four x86 instructions
- ▲ Op Cache instructions
- ▲ 4 Integer units
 - Large rename space – 168 Registers
 - 192 instructions in flight/8 wide retire
- ▲ 2 Load/Store units
 - 72 Out-of-Order Loads supported
- ▲ 2 Floating Point units x 128 FMACs
 - built as 4 pipes, 2 Fadd, 2 Fmul
- ▲ I-Cache 64K, 4-way
- ▲ D-Cache 32K, 8-way
- ▲ L2 Cache 512K, 8-way
- ▲ Large shared L3 cache
- ▲ 2 threads per core

Solutions/work-around of pipeline hazards

- Structural
 - Stall
 - More read/write ports
 - Split hardware units (e.g., instruction/data caches)
- Control
 - Stalls
 - Branch predictions
 - Compiler optimizations with ISA supports (e.g., delayed branch)
- Data
 - Stalls
 - Data forwarding
 - Compiler optimizations
 - Dynamic scheduling

Why compiler optimization is insufficient

- Compiler cannot predict “dynamic” events
 - Compiler doesn’t know if the branch is going or likely to be taken or not
 - Loop unrolling doesn’t always work
 - Compiler doesn’t know if the memory access is going to be a miss or not
- Compiler can only optimize on features exposed by hardware
 - Compiler can only see “architectural registers”, but cannot utilize “physical registers”
 - Has very limited power in “renaming”
 - Creates “false dependencies”
 - Compiler optimization cannot be adaptive to micro architectural changes — what if the pipeline changes?

Recap: What about "linked list"

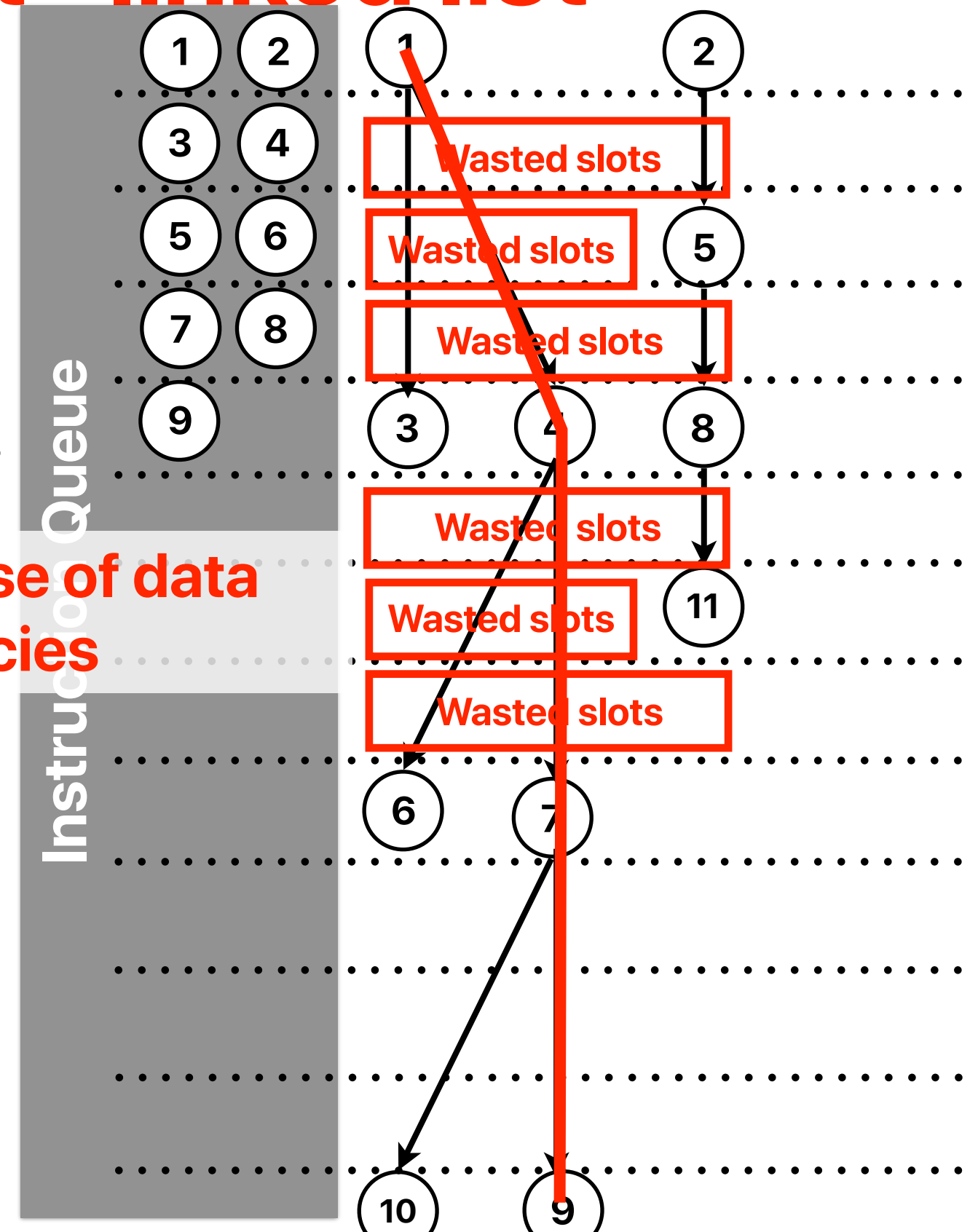
Static instructions

```
LOOP: ld    X10, 8(X10)
      addi  X7, X7, 1
      bne   X10, X0, LOOP
```

Dynamic instructions

```
① ld    X10, 8(X10)
② addi  X7, X7, 1
③ bne   X10, X0, LOOP
④ ld    X10, 8(X10)
⑤ addi  X7, X7, 1
⑥ bne   X10, X0, LOOP
⑦ ld    X10, 8(X10)
⑧ addi  X7, X7, 1
⑨ bne   X10, X0, LOOP
```

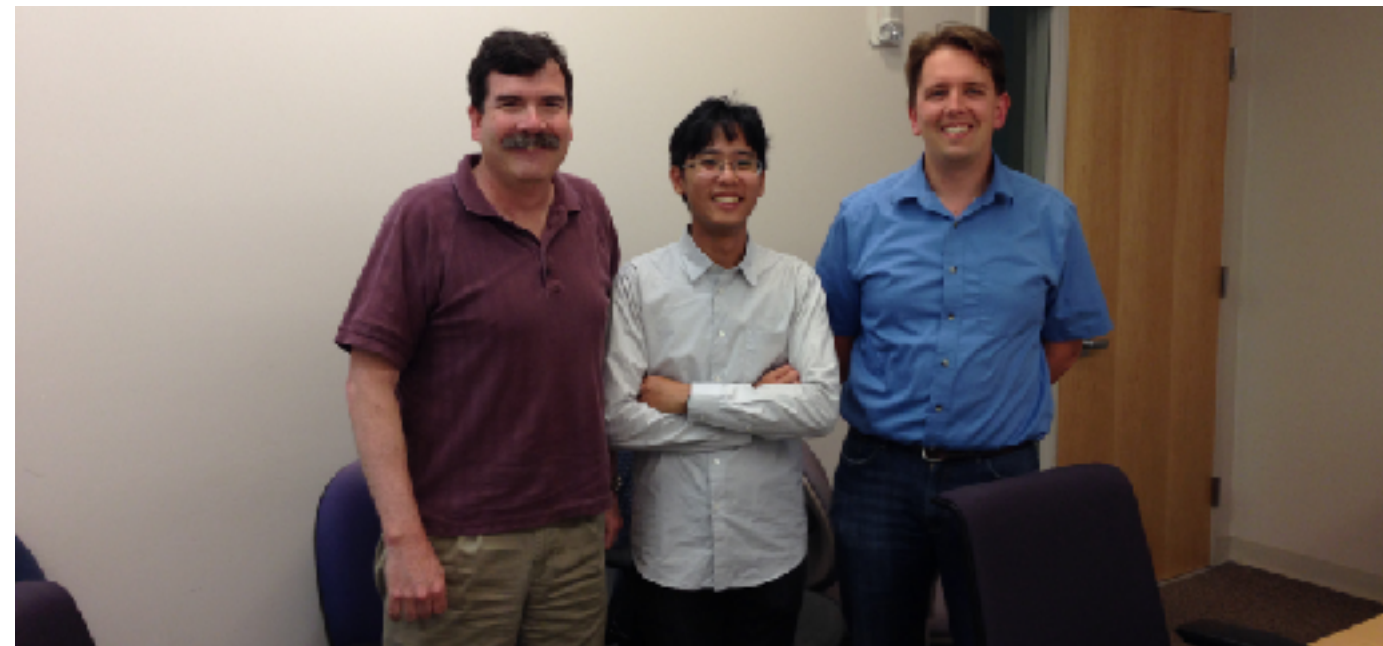
ILP is low because of data dependencies



Simultaneous multithreading: maximizing on-chip parallelism

Dean M. Tullsen, Susan J. Eggers, Henry M. Levy

Department of Computer Science and Engineering, University of Washington



Simultaneous multithreading

- The processor can schedule instructions from different threads/processes/programs
- Fetch instructions from different threads/processes to fill the not utilized part of pipeline
 - Exploit “thread level parallelism” (TLP) to solve the problem of insufficient ILP in a single thread
 - You need to create an illusion of multiple processors for OSs

Architectural support for simultaneous multithreading

- To create an illusion of a multi-core processor and allow the core to run instructions from multiple threads concurrently, how many of the following units in the processor must be duplicated/extended?

- ① Program counter — **you need to have one for each context**
- ② Register mapping tables — **you need to have one for each context**
- ③ Physical registers — **you can share**
- ④ ALUs — **you can share**
- ⑤ Data cache — **you can share**
- ⑥ Reorder buffer/Instruction Queue

A. 2 — **you need to indicate which context the instruction is from**

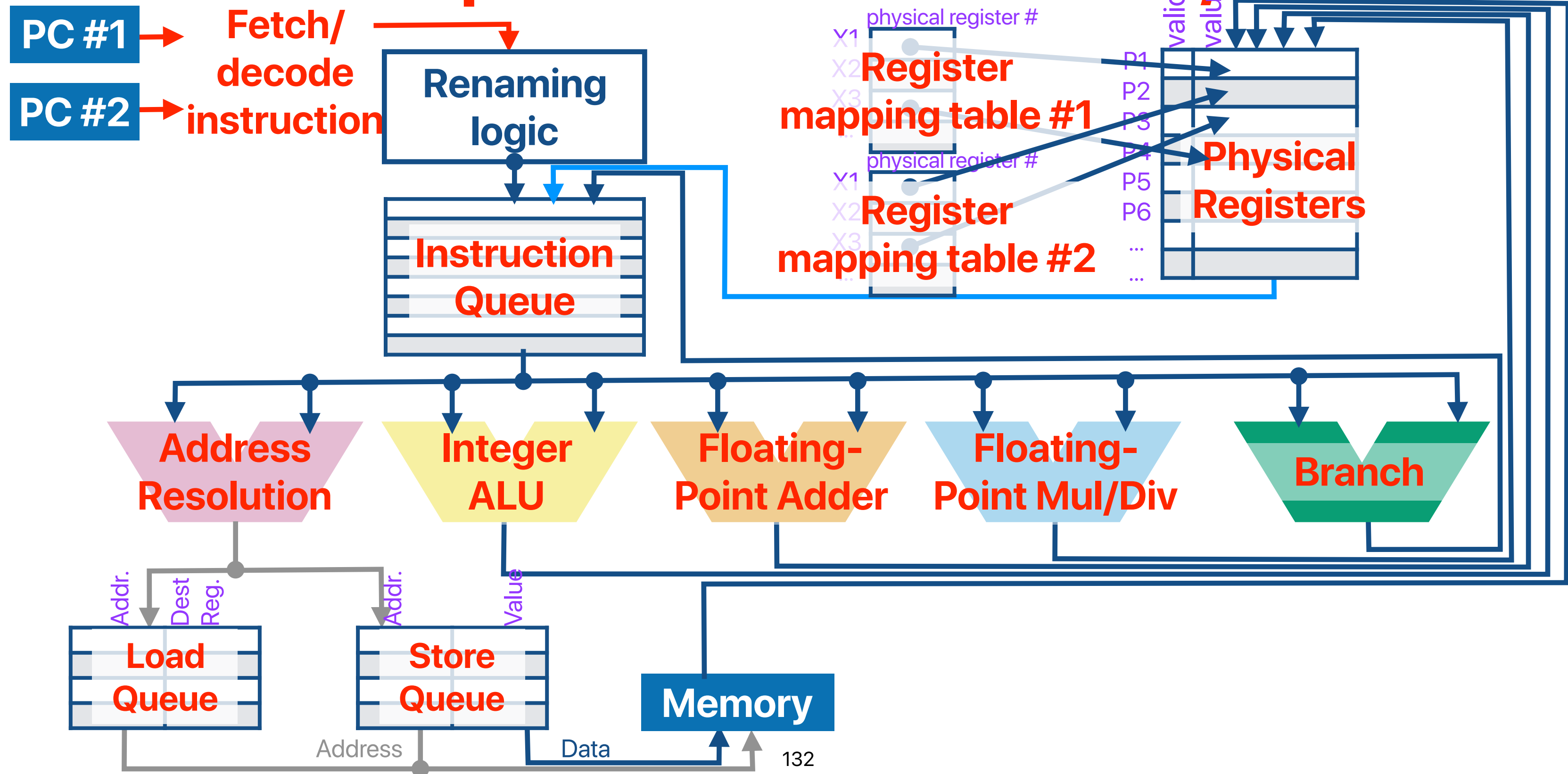
B. 3

C. 4

D. 5

E. 6

SMT SuperScalar Processor w/ ROB



SMT

- How many of the following about SMT are correct?
 - ① SMT makes processors with deep pipelines more tolerable to mis-predicted branches **We can execute from other threads/contexts instead of the current one**
hurt, b/c you are sharing resource with other threads.
 - ② SMT can ~~improve~~ the throughput of a single-threaded application
 - ③ SMT processors can better utilize hardware during cache misses comparing with superscalar processors with the same issue width **We can execute from other threads/contexts instead of the current one**
 - ④ SMT processors can have higher cache miss rates comparing with superscalar processors with the same cache sizes when executing the same set of applications.

A. 0 **b/c we're sharing the cache**

B. 1

C. 2

D. 3

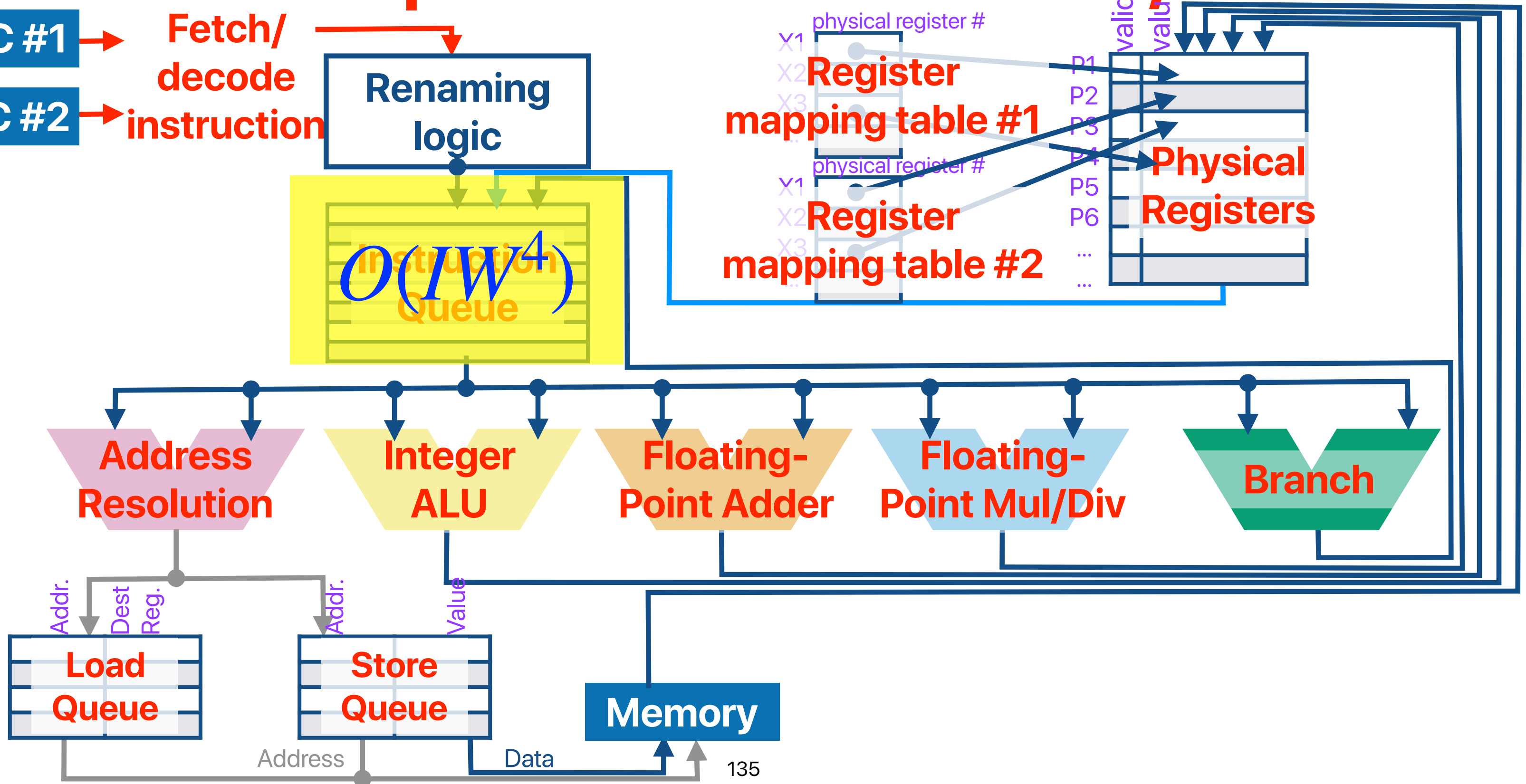
E. 4

SMT

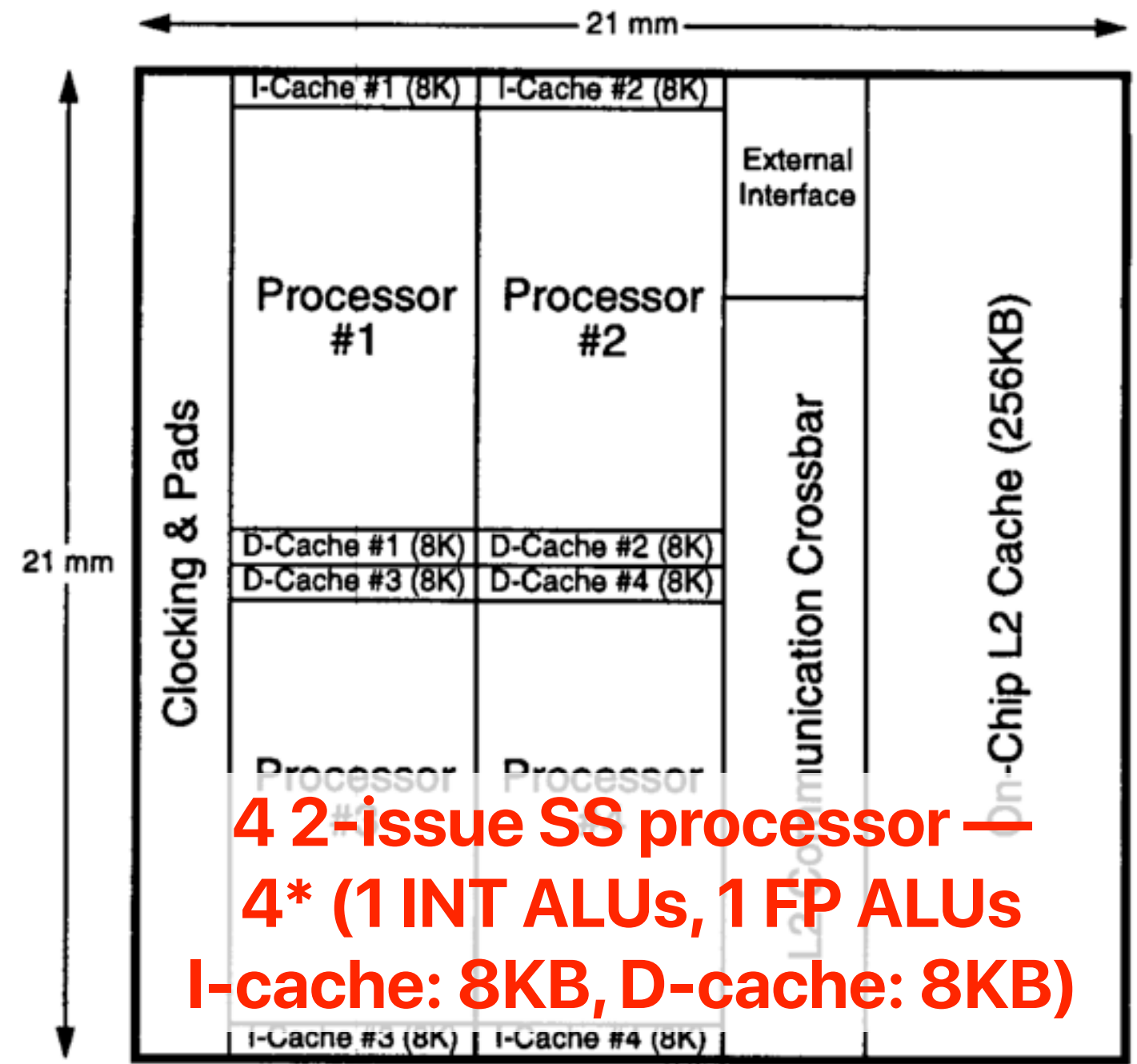
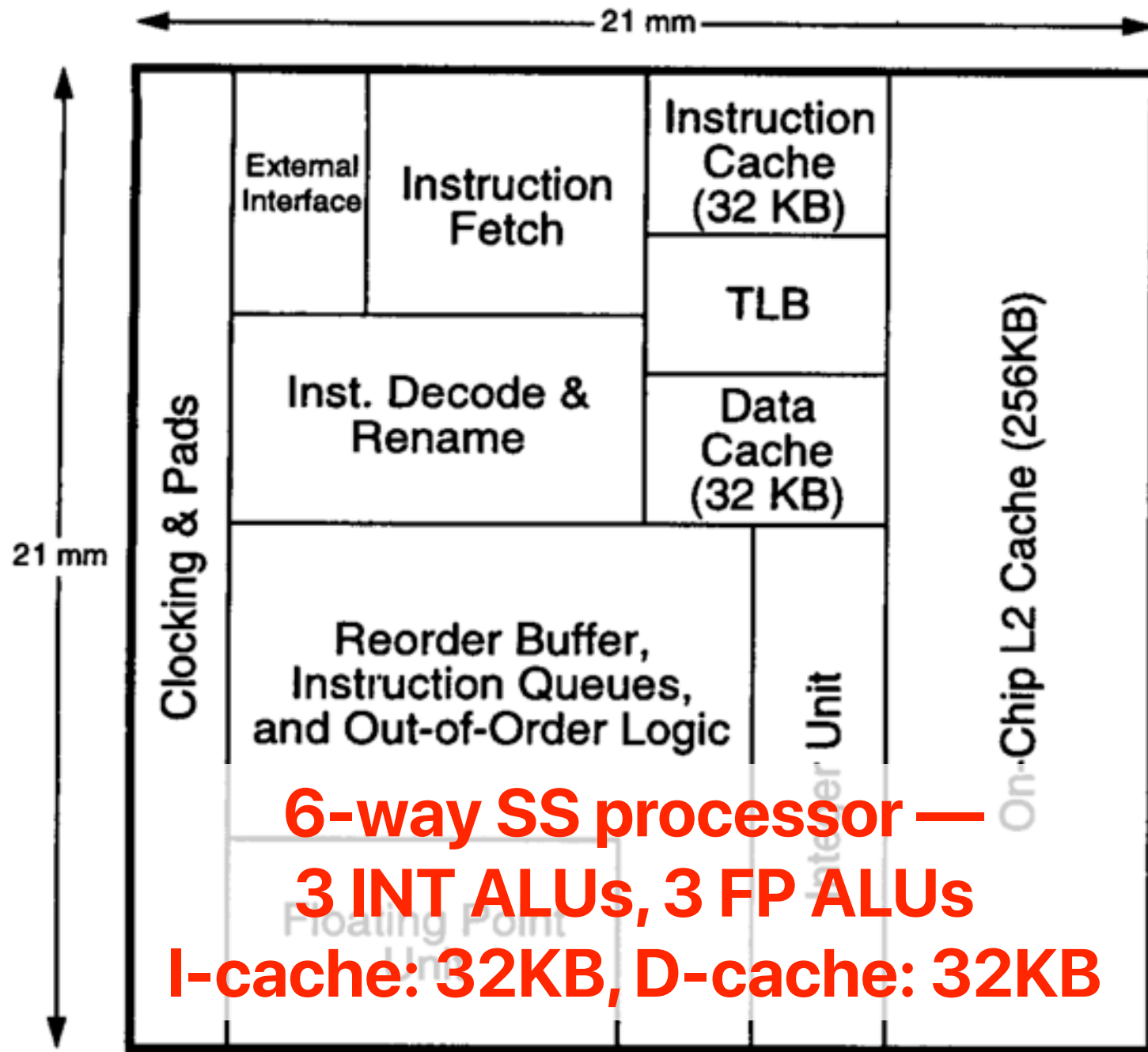
- Improve the throughput of execution
 - May increase the latency of a single thread
- Less branch penalty per thread
- Increase hardware utilization
- Simple hardware design: Only need to duplicate PC/Register Files
- Real Case:
 - Intel HyperThreading (supports up to two threads per core)
 - Intel Pentium 4, Intel Atom, Intel Core i7
 - AMD RyZen

The diagram illustrates a processor architecture with the following components and data flow:

- PC #1** and **PC #2** provide instructions to the **Fetch/decode instruction** stage.
- The **Fetch/decode instruction** stage feeds into the **Renaming logic**.
- The **Renaming logic** feeds into the **Instruction Queue** (labeled $O(IW^4)$).
- The **Instruction Queue** feeds into five functional units: **Address Resolution**, **Integer ALU**, **Floating-Point Adder**, **Floating-Point Mul/Div**, and **Branch**.
- The **Address Resolution** unit feeds into the **Load Queue** and **Store Queue**.
- The **Load Queue** and **Store Queue** feed into the **Memory** block.
- The **Memory** block feeds into the **Branch** unit.
- The **Branch** unit feeds back into the **PC #1** and **PC #2**.
- The **Register mapping table #1** and **Register mapping table #2** are used to map physical registers to logical registers (X1, X2, X3, ...).
- The **Physical Registers** are shown as a stack of registers (P1, P2, P3, P4, P5, P6, ...).

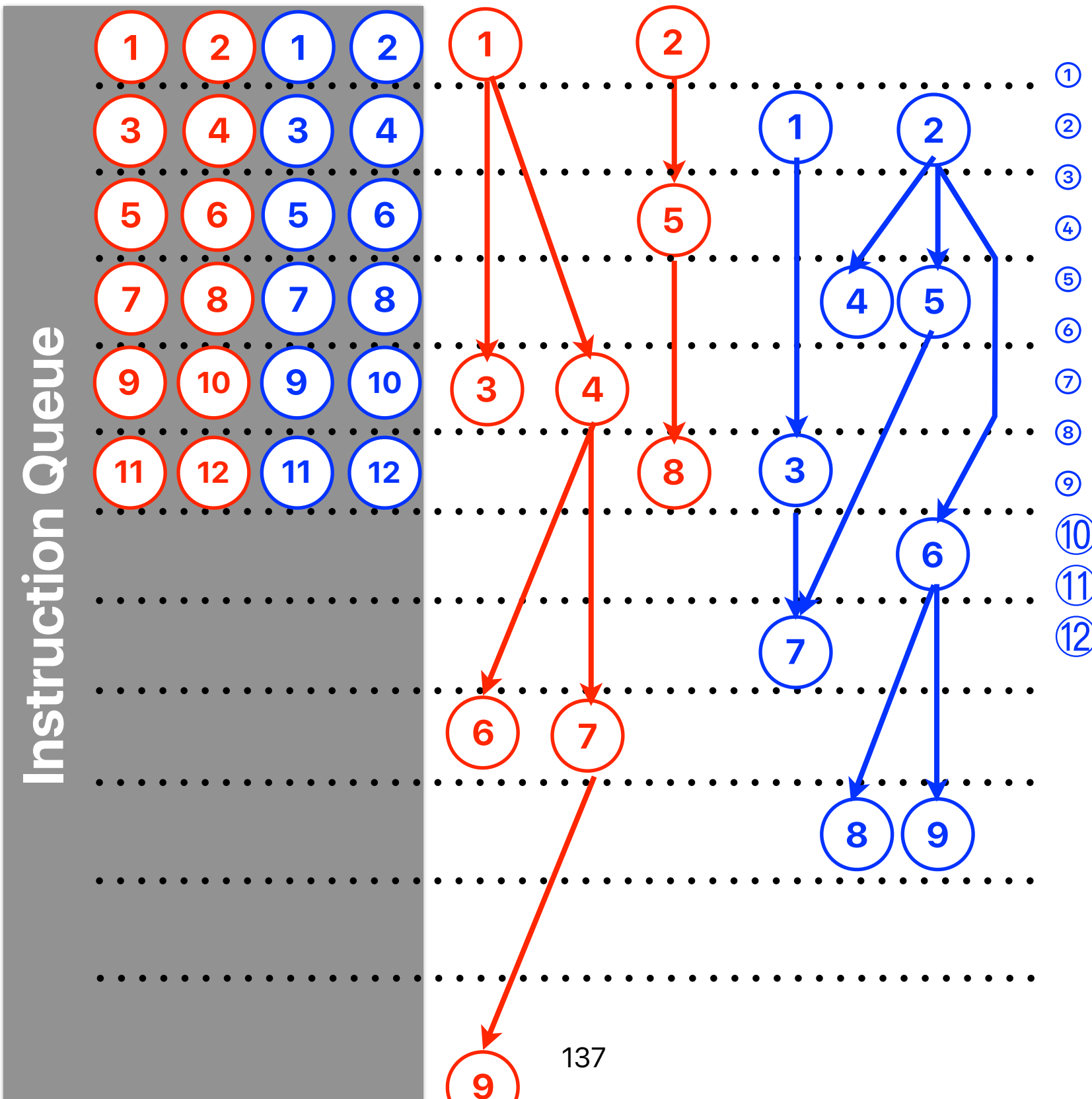


Wide-issue SS processor v.s. multiple narrower-issue SS processors



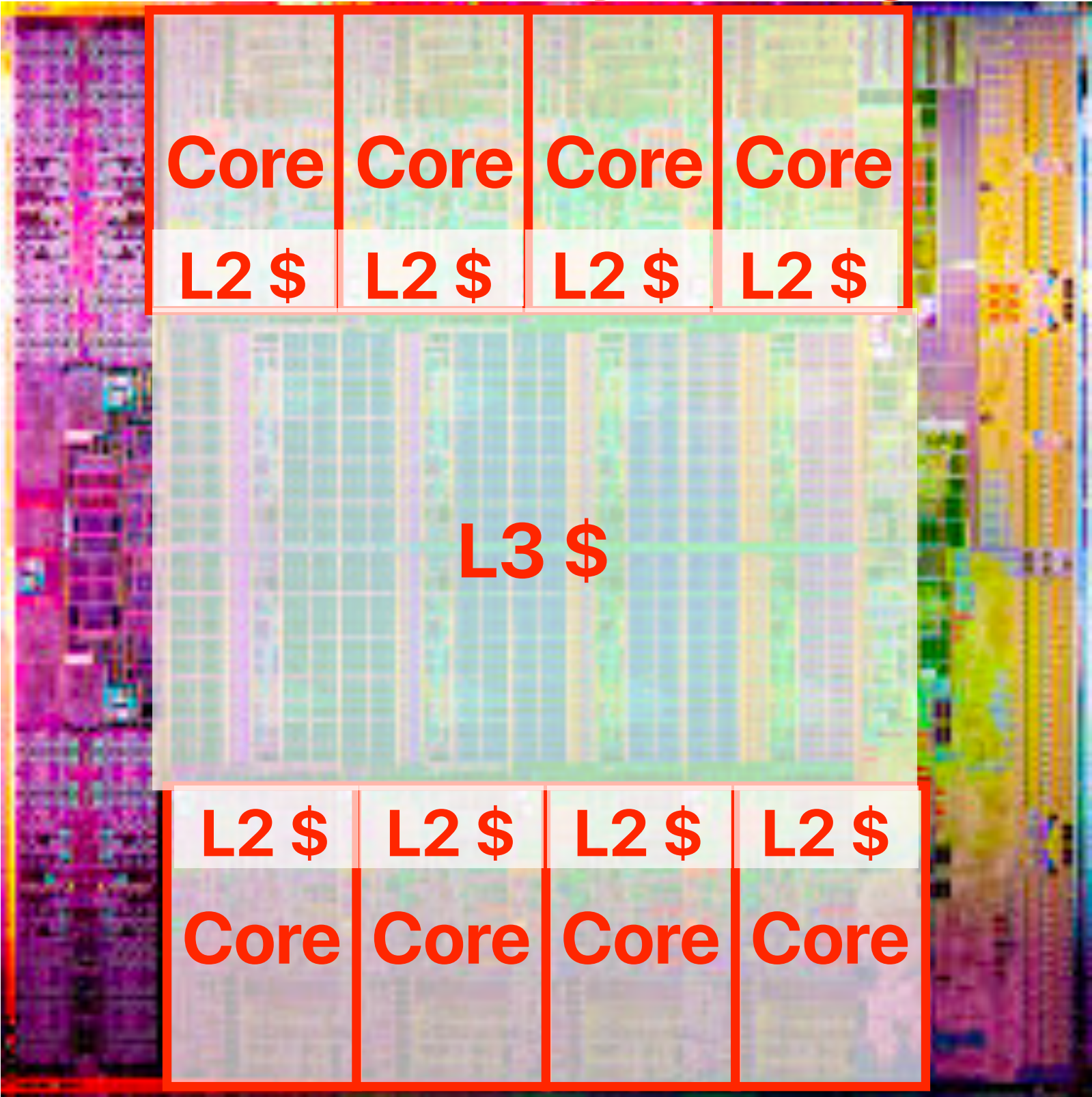
Simultaneous multithreading

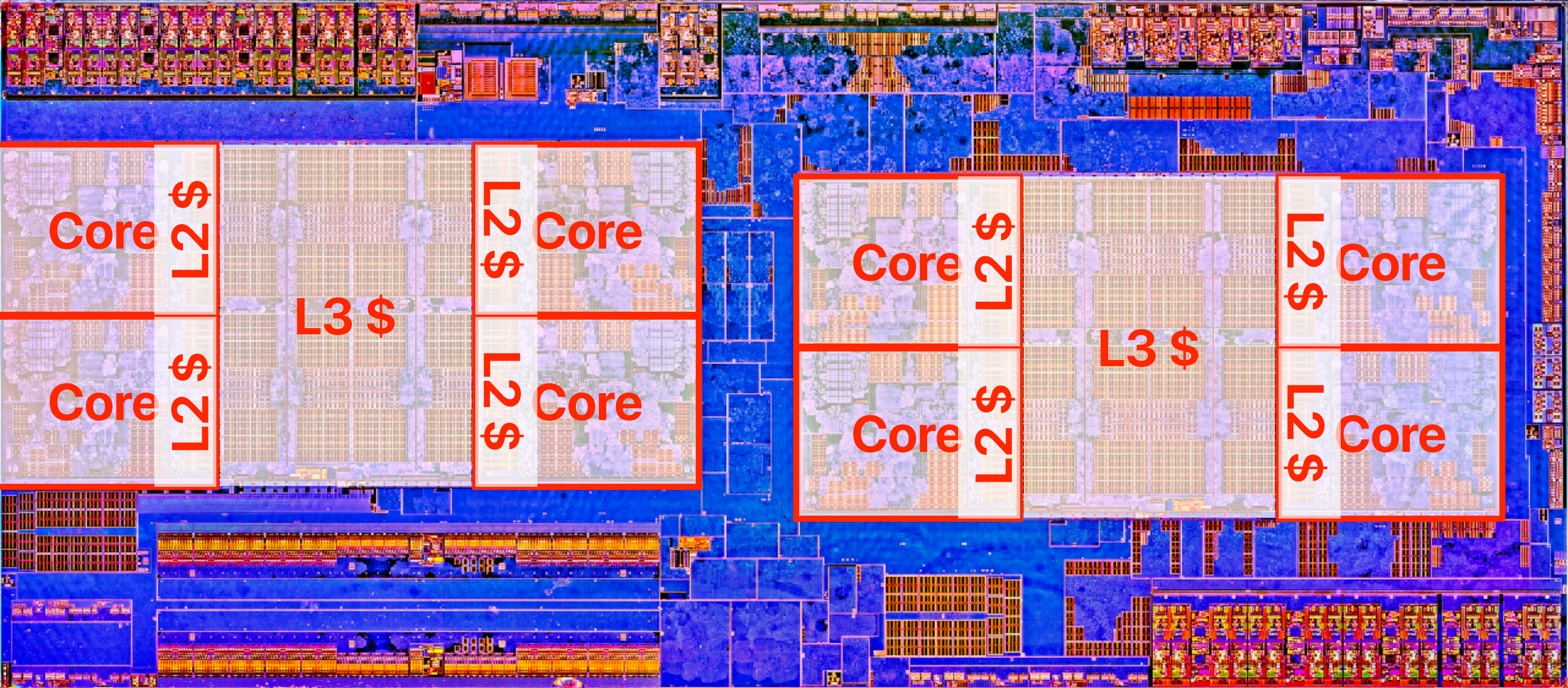
① ld X10, 8(X10)
② addi X7, X7, 1
③ bne X10, X0, LOOP
④ ld X10, 8(X10)
⑤ addi X7, X7, 1
⑥ bne X10, X0, LOOP
⑦ ld X10, 8(X10)
⑧ addi X7, X7, 1
⑨ bne X10, X0, LOOP



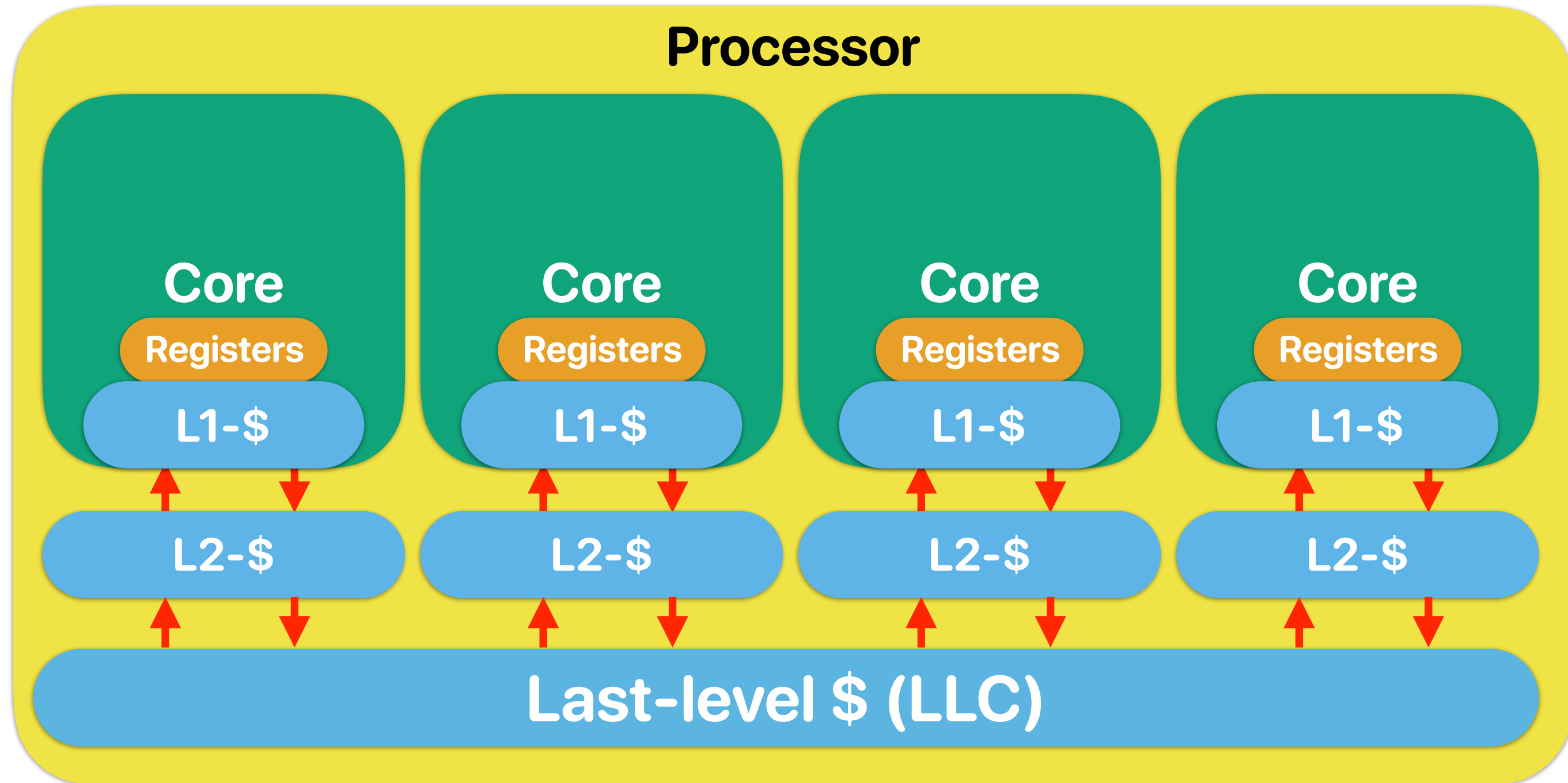
① ld X1, 0(X10)
② addi X10, X10, 8
③ add X20, X20, X1
④ bne X10, X2, LOOP
⑤ ld X1, 0(X10)
⑥ addi X10, X10, 8
⑦ add X20, X20, X1
⑧ bne X10, X2, LOOP
⑨ ld X1, 0(X10)
⑩ addi X10, X10, 8
⑪ add X20, X20, X1
⑫ bne X10, X2, LOOP

Intel Sandy Bridge

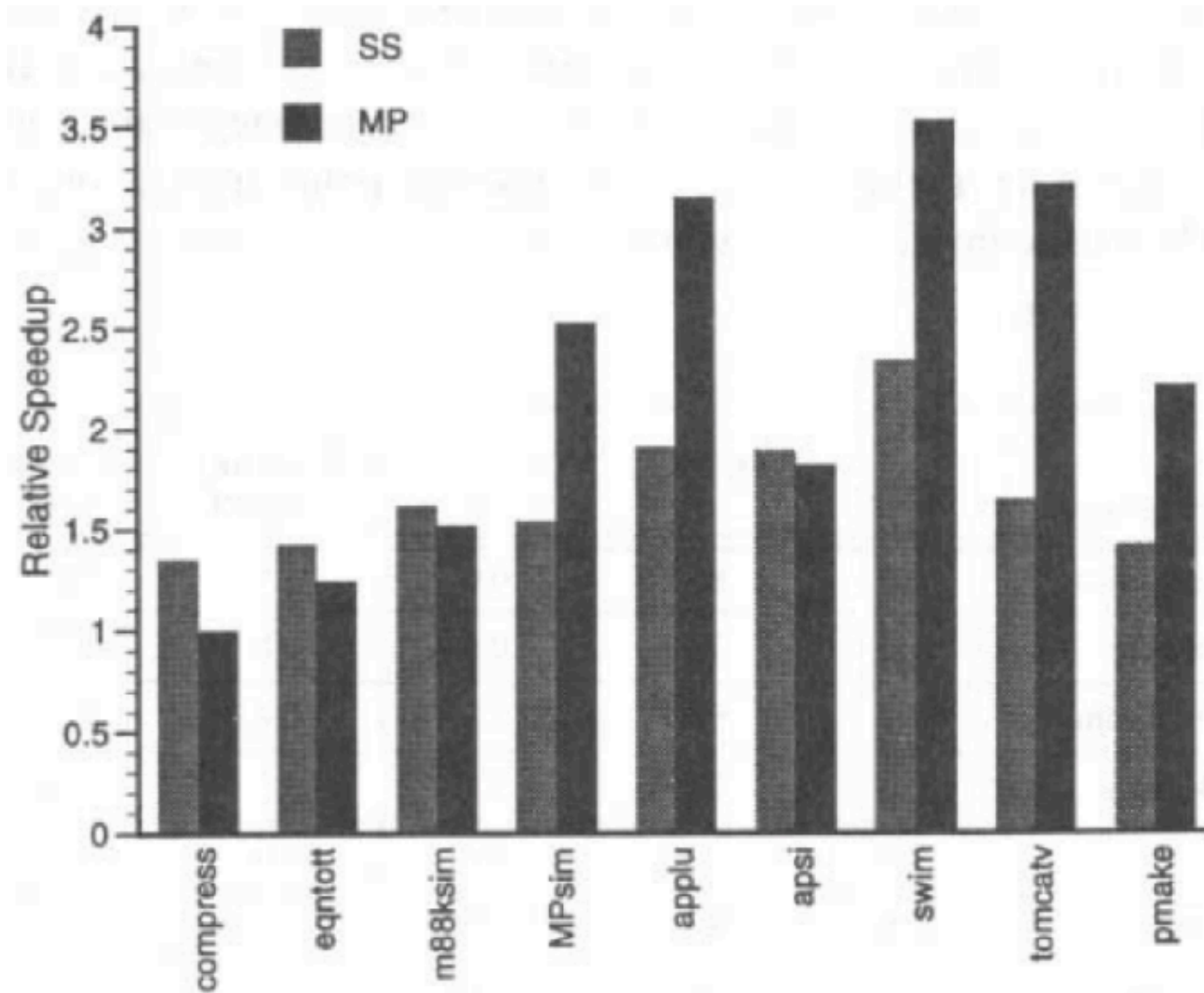




Concept of CMP



Performance of CMP



SMT v.s. CMP

- Both CMP & SMT exploit thread-level or task-level parallelism. Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

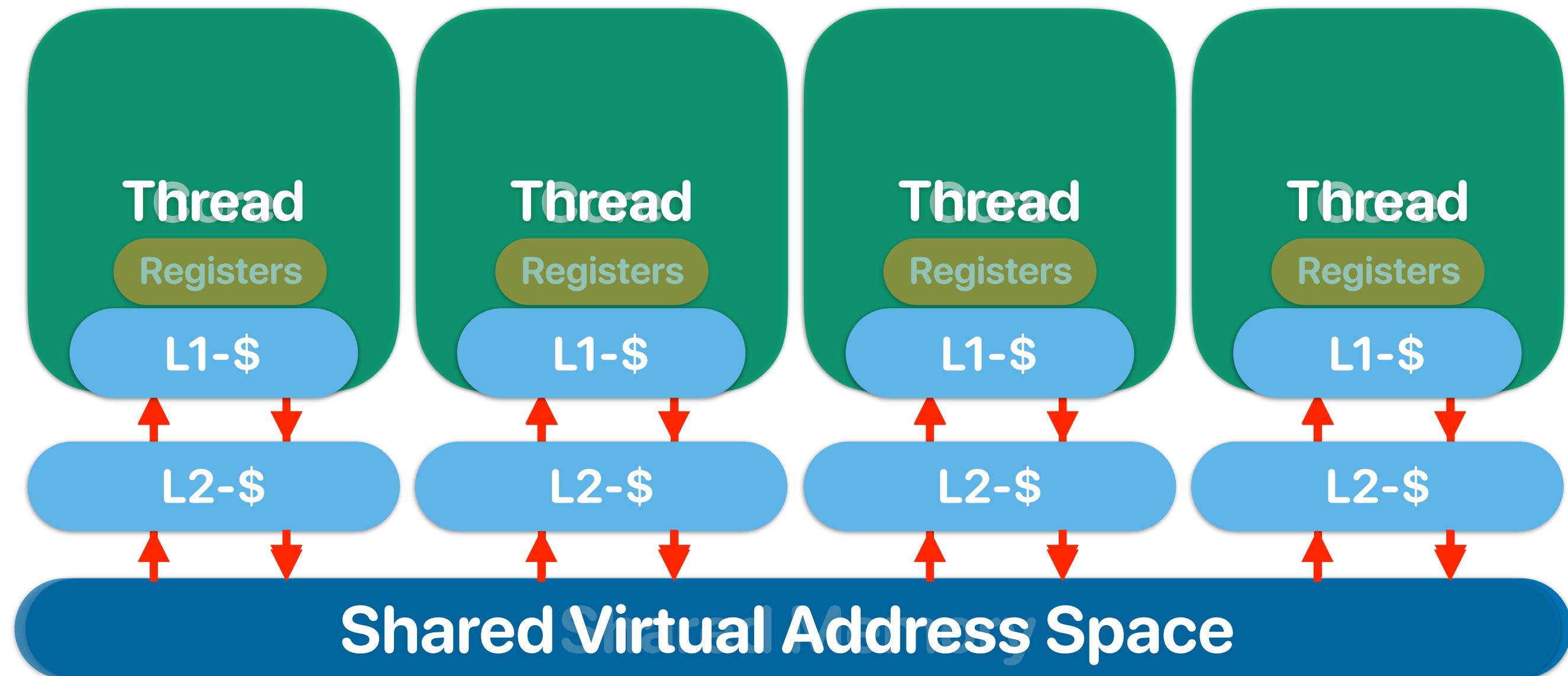
P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

Which one do you think is better?

- A. P1
- B. P2

Architectural Support for Parallel Programming

What software thinks about "multiprogramming" hardware

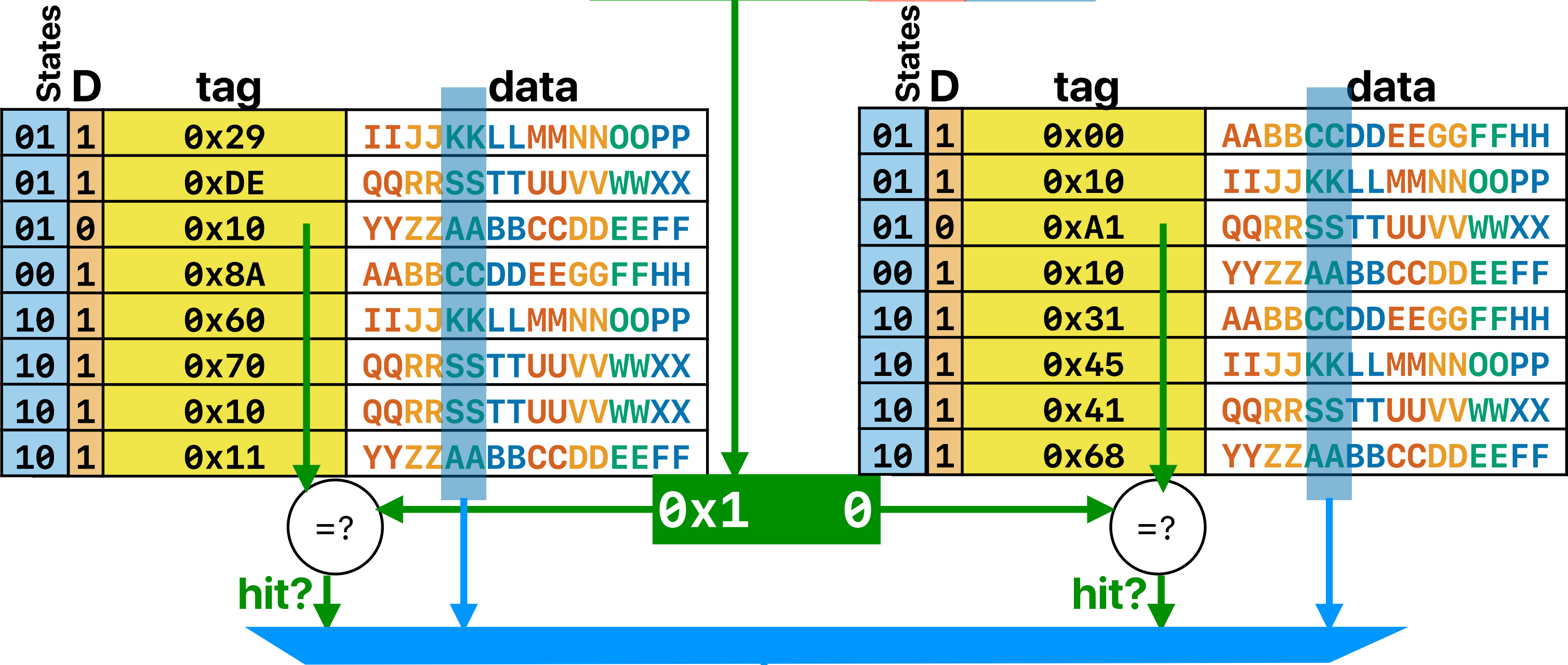


Coherency & Consistency

- Coherency — Guarantees all processors see the same value for a variable/memory address in the system when the processors need the value at the same time
 - What value should be seen
- Consistency — All threads see the change of data in the same order
 - When the memory operation should be done

Coherent way-associative cache

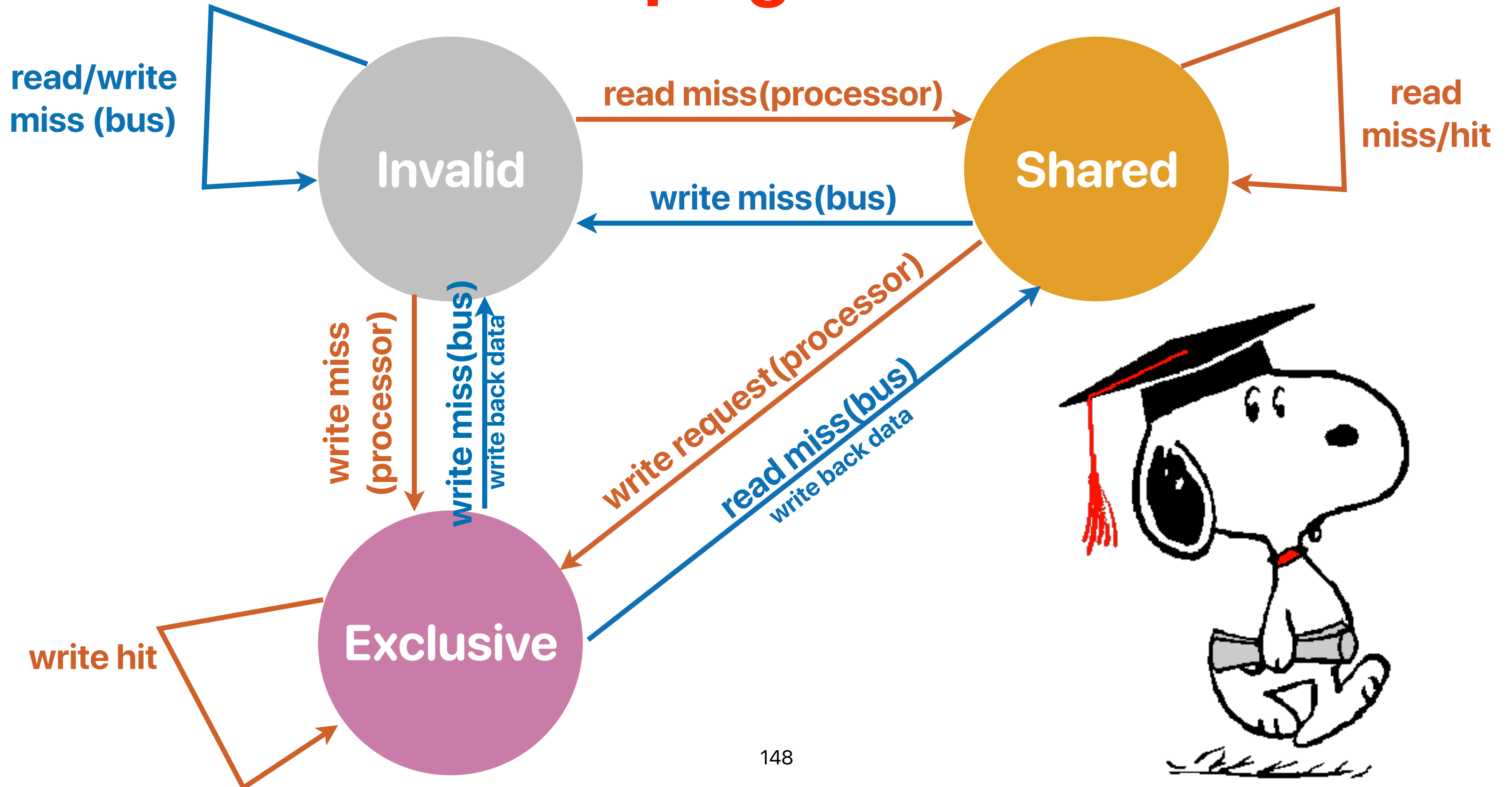
memory address: 0x0 8 2 4
tag set block
index offset
memory address: 0b00001000000100100



Simple cache coherency protocol

- Snooping protocol
 - Each processor broadcasts / listens to cache misses
- State associate with each block (cacheline)
 - Invalid
 - The data in the current block is invalid
 - Shared
 - The processor can read the data
 - The data may also exist on other processors
 - Exclusive
 - The processor has full permission on the data
 - The processor is the only one that has up-to-date data

Snooping Protocol



Cache coherency

- Assuming that we are running the following code on a CMP with a cache coherency protocol, how many of the following outputs are possible? (a is initialized to 0 as assume we will output more than 10 numbers)

thread 1	thread 2
<pre>while(1) printf("%d ", a);</pre>	<pre>while(1) a++;</pre>

- ① 0 1 2 3 4 5 6 7 8 9
② 1 2 5 9 3 6 8 10 12 13
③ 1 1 1 1 1 1 1 64 100
④ 1 1 1 1 1 1 1 1 1 100
A. 0
B. 1
C. 2
D. 3
E. 4

4Cs of cache misses

- 3Cs:
 - Compulsory, Conflict, Capacity
- Coherency miss:
 - A "block" invalidated because of the sharing among processors.

L v.s. R

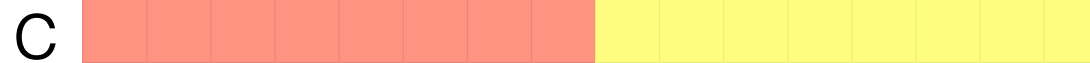
Version L

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid;i<ARRAY_SIZE;i+=NUM_OF_THREADS)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```



Version R

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS);i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS);i++)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```



Performance comparison

- Comparing implementations of thread_vadd — L and R, please identify which one will be performing better and why

Version L

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid; i<ARRAY_SIZE; i+=NUM_OF_THREADS)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

Version R

```
void *threaded_vadd(void *thread_id)
{
    int tid = *(int *)thread_id;
    int i;
    for(i=tid*(ARRAY_SIZE/NUM_OF_THREADS); i<(tid+1)*(ARRAY_SIZE/NUM_OF_THREADS); i++)
    {
        c[i] = a[i] + b[i];
    }
    return NULL;
}
```

- A. L is better, because the cache miss rate is lower
- B. R is better, because the cache miss rate is lower**
- C. L is better, because the instruction count is lower
- D. R is better, because the instruction count is lower
- E. Both are about the same

Main thread

```
for(i = 0 ; i < NUM_OF_THREADS ; i++)
{
    tids[i] = i;
    pthread_create(&thread[i], NULL, threaded_vadd, &tids[i])
}
for(i = 0 ; i < NUM_OF_THREADS ; i++)
    pthread_join(thread[i], NULL);
```


Again — how many values are possible?

- Consider the given program. You can safely assume the caches are coherent. How many of the following outputs will you see?

① (0, 0)

② (0, 1)

③ (1, 0)

④ (1, 1)

A. 0

B. 1

C. 2

D. 3

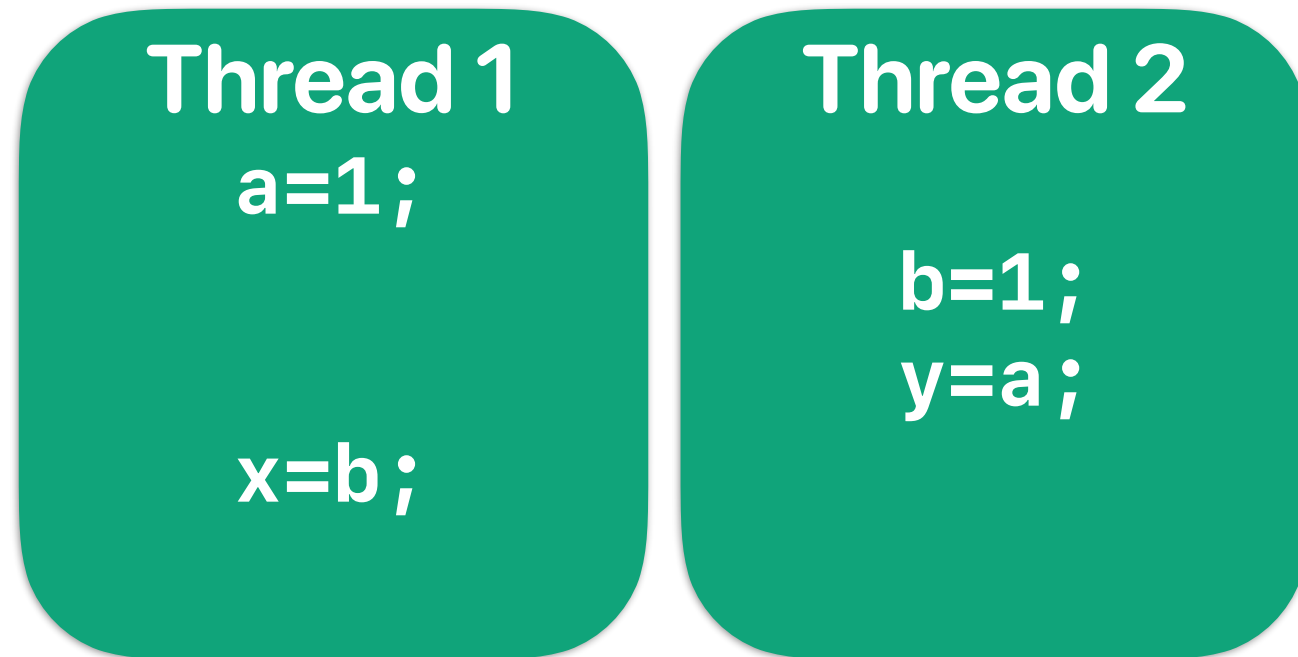
E. 4

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

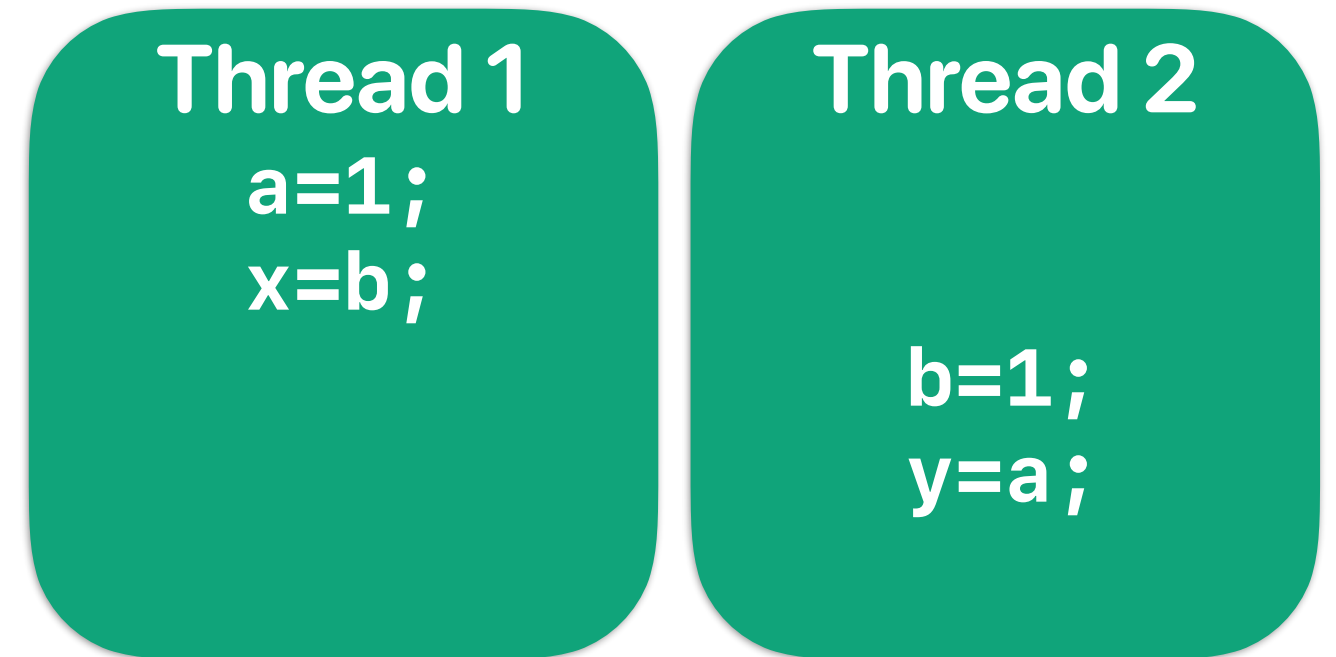
volatile int a,b;
volatile int x,y;
volatile int f;
void* modifya(void *z) {
    a=1;
    x=b;
    return NULL;
}
void* modifyb(void *z) {
    b=1;
    y=a;
    return NULL;
}
```

```
int main() {
    int i;
    pthread_t thread[2];
    pthread_create(&thread[0], NULL, modifya, NULL);
    pthread_create(&thread[1], NULL, modifyb, NULL);
    pthread_join(thread[0], NULL);
    pthread_join(thread[1], NULL);
    fprintf(stderr, "(%d, %d)\n", x, y);
    return 0;
}
```

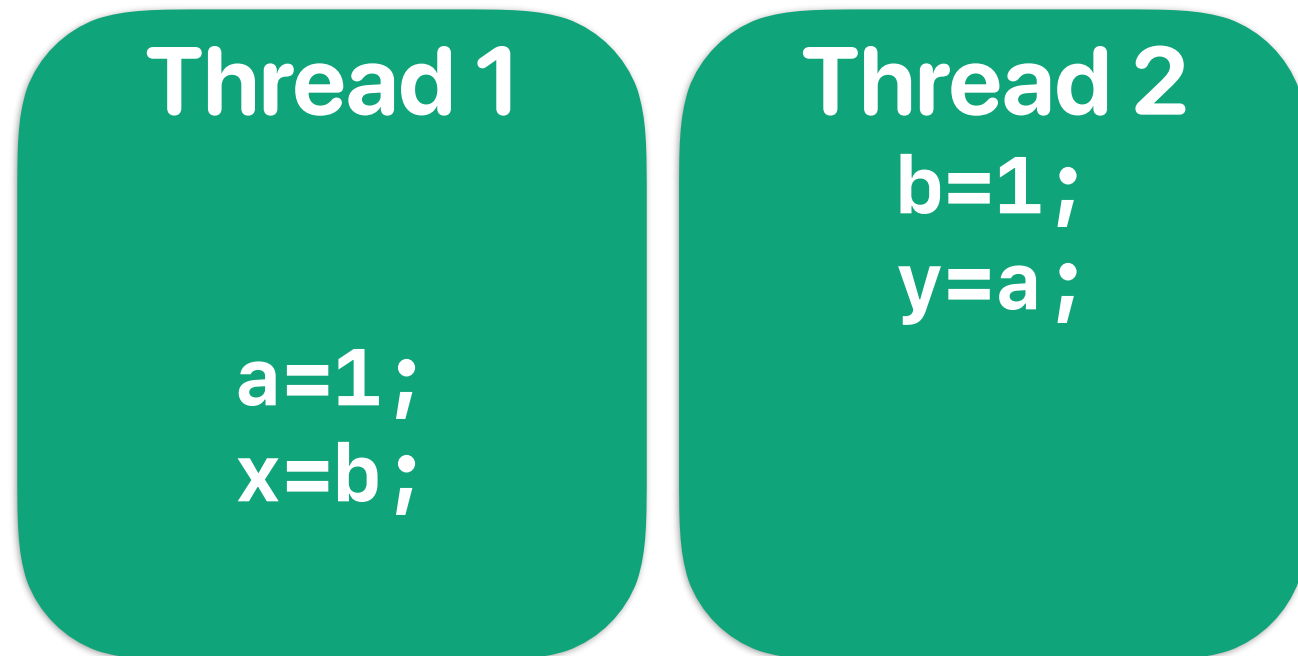
Possible scenarios



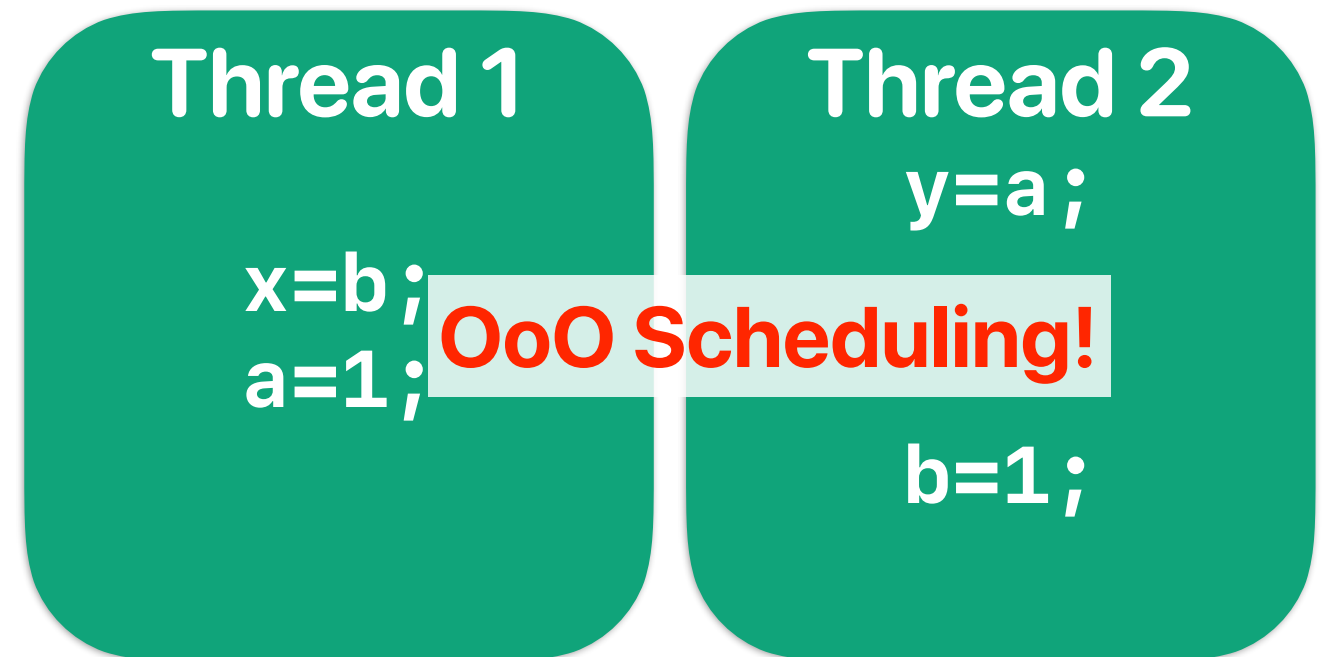
(1,1)



(0,1)



(1,0)



(0,0)

fence instructions

- x86 provides an "mfence" instruction to prevent reordering across the fence instruction
- x86 only supports this kind of "relaxed consistency" model. You still have to be careful enough to make sure that your code behaves as you expected

thread 1	thread 2
<pre>a=1; mfence a=1 must occur/update before mfence x=b;</pre>	<pre>b=1; mfence b=1 must occur/update before mfence y=a;</pre>

Power/Energy/Dark Silicon

Dynamic/Active Power

- The power consumption due to the switching of transistor states

- Dynamic power per transistor

$$P_{dynamic} \sim \alpha \times C \times V^2 \times f \times N$$

- α : average switches per cycle
- C : capacitance
- V : voltage
- f : frequency, usually linear with V
- N : the number of transistors

Static/Leakage Power

- The power consumption due to leakage — transistors do not turn all the way off during no operation
- Becomes the **dominant** factor in the most advanced process technologies.

$$P_{leakage} \sim N \times V \times e^{-V_t}$$

- N : number of transistors
- V : voltage
- V_t : threshold voltage where transistor conducts (begins to switch)

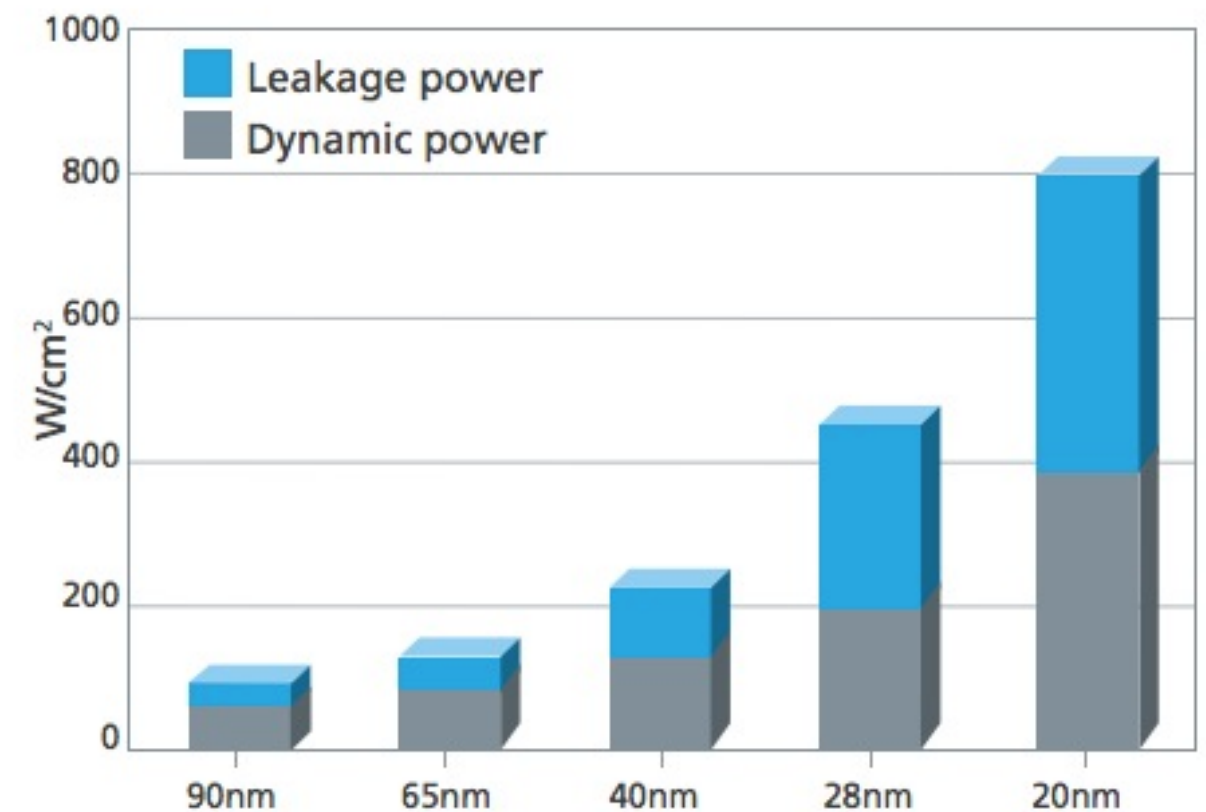


Figure 1: Leakage power becomes a growing problem as demands for more performance and functionality drive chipmakers to nanometer-scale process nodes (Source: IBS).

Dennardian Broken

- Given a scaling factor S

Parameter	Relation	Classical Scaling	Leakage Limited
Power Budget		1	1
Chip Size		1	1
Vdd (Supply Voltage)		$1/S$	1
Vt (Threshold Voltage)	$1/S$	$1/S$	1
tex (oxide thickness)		$1/S$	$1/S$
W, L (transistor)		$1/S$	$1/S$
Cgate (gate capacitance)	WL/tox	$1/S$	$1/S$
Isat (saturation current)	$WVdd/tox$	$1/S$	1
F (device frequency)	$Isat/(CgateVdd)$	S	S
D (Device/Area)	$1/(WL)$	S^2	S^2
p (device power)	$IsatVdd$	$1/S^2$	1
P (chip power)	Dp	1	S^2
U (utilization)	$1/P$	1	$1/S^2$

Power consumption to light on all transistors

Dennardian Scaling

Dennardian Broken

Chip

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

=49W

Chip

0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

=50W


Chip

1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

=100W!

More cores per chip, slower per core

Products Solutions Support



	<div>Intel® Xeon® Processor E7-8890 v4</div> <div>X</div>	<div>Intel® Xeon® Processor E7-8893 v4</div> <div>X</div>	<div>Intel® Xeon® Processor E7-8880 v4</div> <div>X</div>
Status	Launched	Launched	Launched
Launch Date ⓘ	Q2'16	Q2'16	Q2'16
Lithography ⓘ	14 nm	14 nm	14 nm
Performance			
# of Cores ⓘ	24	4	22
# of Threads ⓘ	48	8	44
Processor Base Frequency ⓘ	2.20 GHz	3.20 GHz	2.20 GHz
Max Turbo Frequency ⓘ	3.40 GHz	3.50 GHz	3.30 GHz
Cache ⓘ	60 MB	60 MB	55 MB
Bus Speed ⓘ	9.6 GT/s	9.6 GT/s	9.6 GT/s
# of QPI Links ⓘ	3	3	3
TDP ⓘ	165 W	140 W	150 W

What happens if power doesn't scale with process technologies?

- If we are able to cram more transistors within the same chip area (Moore's law continues), but the power consumption per transistor remains the same. Right now, if we power the chip with the same power consumption but put more transistors in the same area because the technology allows us to. How many of the following statements are true?

- ① The power consumption per chip will increase
- ② The power density of the chip will increase
- ③ Given the same power budget, we may not be able to power on all chip area if we maintain the same clock rate
- ④ Given the same power budget, we may have to lower the clock rate of circuits to power on all chip area

A. 0

B. 1

C. 2

D. 3

E. 4

Final Logistics

- Monday, December 9, 8:00 a.m. – 11:00 a.m.
 - No makeup, no re-schedule — if you miss it, you have to take responsibilities
 - Keep yourself health enough to attend the final is part of your job
- You may bring a calculator
 - Mobile phones/smartphones are NOT allowed
- Show your work (except for multiple choices)
 - You will get partial credits if you have some work done
 - You will get 0 if you only give us the answer
- Please print — if we cannot read your answers, we will not give you grades
- No cheatsheet is allowed
- No cheating
- No review this time — I will be out-of-town right after finishing the grading
- You can discuss **sample final** with me/TA or your friend in **private**, but I won't respond any on piazza

Sample Final

Format of finals

- Multiple choices (10-15 questions, TBD)
 - They're like your clicker/midterm multiple choices questions
 - Cumulative, don't forget your midterm and midterm review
- Homework style calculation/operation based questions * 3-4
- Brief discussion/Open-ended * 4-5
 - Explain your answer using less than 100 words. Some of them must be as short as 30 words
 - May not have a standard answer. You need to understand the concepts to provide a good answer

Multiple choices

How many dependencies do we have?

- How many pairs of data dependences are there in the following RISC-V instructions?

```
ld      X6, 0(X10)
add     X7, X6, X12
sd      X7, 0(X10)
addi    X10, X10, 8
bne     X10, X5, LOOP
```

- A. 1
- B. 2
- C. 3
- D. 4
- E. 5

False dependencies

- Consider the following dynamic instructions

- ① `ld X12, 0(X20)`
- ② `add X12, X10, X12`
- ③ `sub X18, X12, X10`
- ④ `ld X12, 8(X20)`
- ⑤ `add X14, X18, X12`
- ⑥ `add X18, X14, X14`
- ⑦ `sd X14, 16(X20)`
- ⑧ `addi X20, X20, 8`

which of the following pair is not a "false dependency"

- A. (1) and (4)
- B. (1) and (8)
- C. (5) and (7)
- D. (4) and (8)
- E. (7) and (8)

What about "linked list"

- For the following C code and its translation in RISC-V, how many cycles it takes the processor to issue all instructions? Assume the current PC is already at the first instruction and this linked list has only three nodes. This processor can fetch 2 instructions per cycle, with exactly the same register renaming hardware and pipeline as we showed previously.

```
do {  
    number_of_nodes++;  
    current = current->next;  
} while ( current != NULL )
```

```
LOOP: ld    X10, 8(X10)  
      addi  X7, X7, 1  
      bne   X10, X0, LOOP
```

- A. 9
- B. 10
- C. 11
- D. 12
- E. 13

CMP advantages

- How many of the following are advantages of CMP over traditional superscalar processor
 - ① CMP can provide better energy-efficiency within the same area
 - ② CMP can deliver better instruction throughput within the same die area (chip size)
 - ③ CMP can achieve better ILP for each running thread
 - ④ CMP can improve the performance of a single-threaded application without modifying code
- A. 0
- B. 1
- C. 2
- D. 3
- E. 4

How good is SS/OoO/ROB with this code?

- Consider the following dynamic instructions

- ① `ld X1, 0(X10)`
- ② `addi X10, X10, 8`
- ③ `add X20, X20, X1`
- ④ `bne X10, X2, LOOP`

Assume a superscalar processor with issue width as 2 & unlimited physical registers that can fetch up to 4 instructions per cycle, 3 cycles to execute a memory instruction and the loop will execute for 10,000 times, what's the average CPI?

- A. 0.5
- B. 0.75
- C. 1
- D. 1.25
- E. 1.5

Amdahl's Law on Multicore Architectures

- Regarding Amdahl's Law on multicore architectures, how many of the following statements is/are correct?
 - ① If we have unlimited parallelism, the performance of each parallel piece does not matter as long as the performance slowdown in each piece is bounded
 - ② With unlimited amount of parallel hardware units, single-core performance does not matter anymore
 - ③ With unlimited amount of parallel hardware units, the maximum speedup will be bounded by the fraction of parallel parts
 - ④ With unlimited amount of parallel hardware units, the effect of scheduling and data exchange overhead is minor
- A. 0
B. 1
C. 2
D. 3
E. 4

Summary of Optimizations

- Regarding the following cache optimizations, how many of them would help improve miss rate?
 - ① Non-blocking/pipelined/multibanked cache
 - ② Critical word first and early restart
 - ③ Prefetching
 - ④ Write buffer
- A. 0
B. 1
C. 2
D. 3
E. 4

Virtual indexed, physical tagged cache limits the cache size

- If you want to build a virtual indexed, physical tagged cache with 32KB capacity, which of the following configuration is possible? Assume the system use 4K pages.
 - A. 32B blocks, 2-way
 - B. 32B blocks, 4-way
 - C. 64B blocks, 4-way
 - D. 64B blocks, 8-way

Power & Energy

- Regarding power and energy, how many of the following statements are correct?
 - ① Lowering the power consumption helps extending the battery life
 - ② Lowering the power consumption helps reducing the heat generation
 - ③ Lowering the energy consumption helps reducing the electricity bill
 - ④ A CPU with 10% utilization can still consume 33% of the peak power

A. 0
B. 1
C. 2
D. 3
E. 4

MS' "Configurable Clouds"

- Regarding MS' configurable clouds that are powered by FPGAs, please identify how many of the following are correct
 - ① Each FPGA is dedicated to one machine
 - ② Each FPGA is connected through a network that is separated from the data center network
 - ③ FPGA can deliver shorter average latency for AES-CBC-128-SHA1 encryption and decryption than Intel's high-end processors
 - ④ FPGA-accelerated search queries are always faster than a pure software-based datacenter

A. 0
B. 1
C. 2
D. 3
E. 4

Free-answer questions

Performance evaluation with cache

- Consider the following cache configuration on RISC-V processor:

	I-L1	D-L1	L2	DRAM
size	32K	32K	256K	Big enough
block size	64 Bytes	64 Bytes	64 Bytes	4KB pages
associativity	2-way	2-way	8-way	
access time	1 cycle (no penalty if it's a hit)	1 cycle (no penalty if it's a hit)	10 cycles	100 cycles
local miss rate	2%	10%, 20% dirty	15% (i.e., 15% of L1 misses, also miss in the L2), 30% dirty	
Write policy	N/A	Write-back, write allocate		
Replacement	LRU replacement policy			

The application has 20% branches, 10% loads/stores, 70% integer instructions.

- Assume that TLB miss rate is 2% and it requires 100 cycles to handle a TLB miss. Also assume that the **branch predictor** has a hit rate of **87.5%**, what's the CPI of branch, L/S, and integer instructions? What is the average CPI?
- What if a load/store instruction becomes a load/store + an integer instruction to avoid address calculation?

Branch prediction

- For each labeled branch in the code below (except the backward branch on the outer loop) which of the following branch predictors will perform best.
 - Global history (3 bits) to select a 2-bit predictor
 - PC to select a 2-bit predictor
 - static (you can choose the direction)

```
for (j = 0; j < 100; j++) {  
    for (i = 0; i < 100; i++) {  
  
        // rand3() returns 1,2, or 3 with equal probability  
        if(rand3() <= 1) { // A  
        }  
  
        if(i % 3 == 0) { // B  
        }  
  
        if (i % 2 == 0) { // C  
        }  
  
        if(i % 6 == 0) { // D  
        }  
  
        if (i < 4) { // E  
        }  
  
    } // F  
}
```

Pipeline diagram

- Draw the pipeline diagram for the following instructions

① Loop: LD F1, 0(X3)
② FADD F2, F1, F4
③ FMUL F1, F2, F6
④ FADD F1, F1, F5
⑤ FADD F7, F7, F1
⑥ ADD X2, X2, -1
⑦ BNEZ X2, Loop
⑧ ADDI X6, X6, 4
⑨ LD F3, 0(X6)

- Assume we have a single-issue, in-order 7-stage pipeline: IF-ID-EX1/MEM1-EX2/MEM2-EX3/MEM3-EX4/MEM4-WB, predict taken, branch resolved in EX2
 - If the loop is taken twice
 - How many cycles would it take if the loop is taken 100 times? What's the average CPI?

Register renaming

- Draw the pipeline diagram for the following instructions
 - ① Loop: LD F1, 0(X3)
 - ② FADD F2, F1, F4
 - ③ FMUL F1, F2, F6
 - ④ FADD F1, F1, F5
 - ⑤ FADD F7, F7, F1
 - ⑥ ADD X2, X2, -1
 - ⑦ BNEZ X2, Loop
 - ⑧ ADDI X6, X6, 4
 - ⑨ LD F3, 0(X6)
- Assume we have a dual-fetch, dual-issue, out-of-order pipeline where
 - INT ALU takes 1 cycle
 - FP ALU takes 3 cycles
 - MEM pipeline: AR-AQ-MEM — 3 cycles in total
 - BR takes 1 cycle to resolve
- If the loop is taken twice, how many cycles it takes to issue all instructions?
- If the loop is taken 100 times, what's the average CPI?

Performance gain

- Your system has a memory latency of 160ns. You have setup timing on a carefully constructed loop to learn more about your system. The loop is below
 - ① Loop: LD F1, 0(X3)
 - ② FADD F2, F1, F4
 - ③ FMUL F1, F2, F6
 - ④ FADD F1, F1, F5
 - ⑤ FADD F7, F7, F1
 - ⑥ ADD X2, X2, -1
 - ⑦ BNEZ X2, Loop
 - ⑧ ADDI X6, X6, 4
 - ⑨ LD F3, 0(X6)
- On an out-of-order machine, your runtime is slightly greater than 20ns per loop iteration, what is the primary reason for this improvement (choose the best answer)?
 - A. Instruction Level Parallelism
 - B. Memory Level Parallelism
 - C. Thread Level Parallelism
 - D. Advanced Branch Prediction

Best cache configuration

- Consider the following code. Integers and pointers are both 4 bytes.

```
struct List {  
    List * next;  
    int data;  
}  
  
void foo(List *head) {  
    List * cur = head;  
    while(cur->next) {  
        cur = cur->next;  
    }  
}
```

- For a given total cache size, what cache line size will provide the best performance for this code?
(hint: Your answer should not depend on the number of lines or the associativity of the cache.)

Reverse caching

- Below, we have given you four different sequences of addresses generated by a program running on a processor with a data cache. Cache hit ratio for each sequence is also shown below. Assuming that the cache is initially empty at the beginning of each sequence, find out the following parameters of the processor's data cache (ensure that you sufficiently explain your answer)
 - Associativity (1, 2, or 4 ways)
 - Block size (1, 2, 4, 8, 16, or 32 bytes)
 - Total cache size (256B, or 512B)
 - Replacement policy (LRU or FIFO)
1. Address Sequence 1: [0, 2, 4, 8, 16, 32] Hit Ratio: 0.33
 2. Address Sequence 2: [0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0] Hit Ratio: 0.33
 3. Address Sequence 3: [0, 64, 128, 256, 512, 256, 128, 64, 0] Hit Ratio: 0.33
 4. Address Sequence 4: [0, 512, 1024, 0, 1536, 0, 2048, 512] Hit Ratio: 0.25

Open-ended questions

Code and cache miss rate

- Assume my cache has 16KB capacity, 16 byte block size and is 2-way set associative. Integers are 4 bytes. Give the C code for a loop that has a very poor hit rate in this cache but whose hit rate raises to almost 100% if we double the capacity to 32KB.

Branch predictions

- Increasing the size of a branch predictor typically reduces the chances of "aliasing" -- two branches sharing the same predictor. Usually, sharing results in negative interference (decreased prediction accuracy), but sometimes it can result in positive interference. Assuming a PC-indexed table of 2-bit predictors
 - Give an example of two branches (eg, show the T, N patterns for each, and how they are interleaved) that would result in positive interference (increased overall prediction accuracy).
 - Give an example of two branches that would result in negative interference.
 - Explain why most of the time you would expect to see negative interference with real code.

SMT v.s. CMP

- Both CMP & SMT exploit thread-level or task-level parallelism. Assuming both application X and application Y have similar instruction combination, say 60% ALU, 20% load/store, and 20% branches. Consider two processors:

P1: CMP with a 2-issue pipeline on each core. Each core has a private L1 32KB D-cache

P2: SMT with a 4-issue pipeline. 64KB L1 D-cache

Which one do you think is better?

- A. P1
- B. P2

Other open-ended questions

- Given the instruction front-end is decoupled from the backend of the pipeline ALUs, do you think ISA still affect performance?
- What features in modern processor architecture enable the potential of “Meltdown and Spectre” attacks? Should we live without those features?
- What compiler optimizations would not be effective given OoO execution hardware?
- If you’re asked to build an Xeon Phi type processor where each core also has many-way SMT, are you going to give the processor more cache or better branch predictor?

Other open-ended questions

- Can you name and briefly describe a few “trends” in the dark silicon era?
- If you’re asked to design a machine learning hardware, what will you do?
- Can we focus on improving the throughput of computing instead of latency? Can you give an example on what type of applications will not work well in this way