Modern processor design

Hung-Wei Tseng

Outline

- SuperScalar
- Dynamic scheduling/Out-of-order execution

SuperScalar

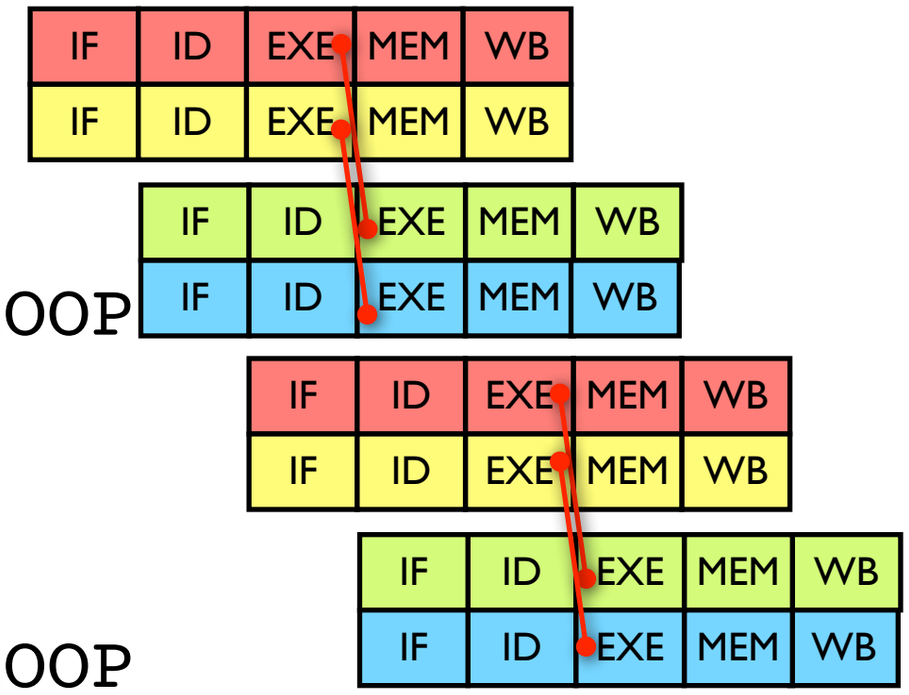
Pipeline

SuperScalar!

SuperScalar

- Improve ILP by widen the pipeline
  - The processor can handle more than one instructions in one stage
  - Instead of fetching one instruction, we fetch multiple instructions!
- CPI = 1/n for an n-issue SS processor in the best case.

```
add  $t1, $a0, $a1
addi $a1, $a1, -1
add  $t2, $a0, $t1
bne  $a1, $zero, LOOP
add  $t1, $a0, $a1
addi $a1, $a1, -1
add  $t2, $a0, $t1
bne  $a1, $zero, LOOP
```

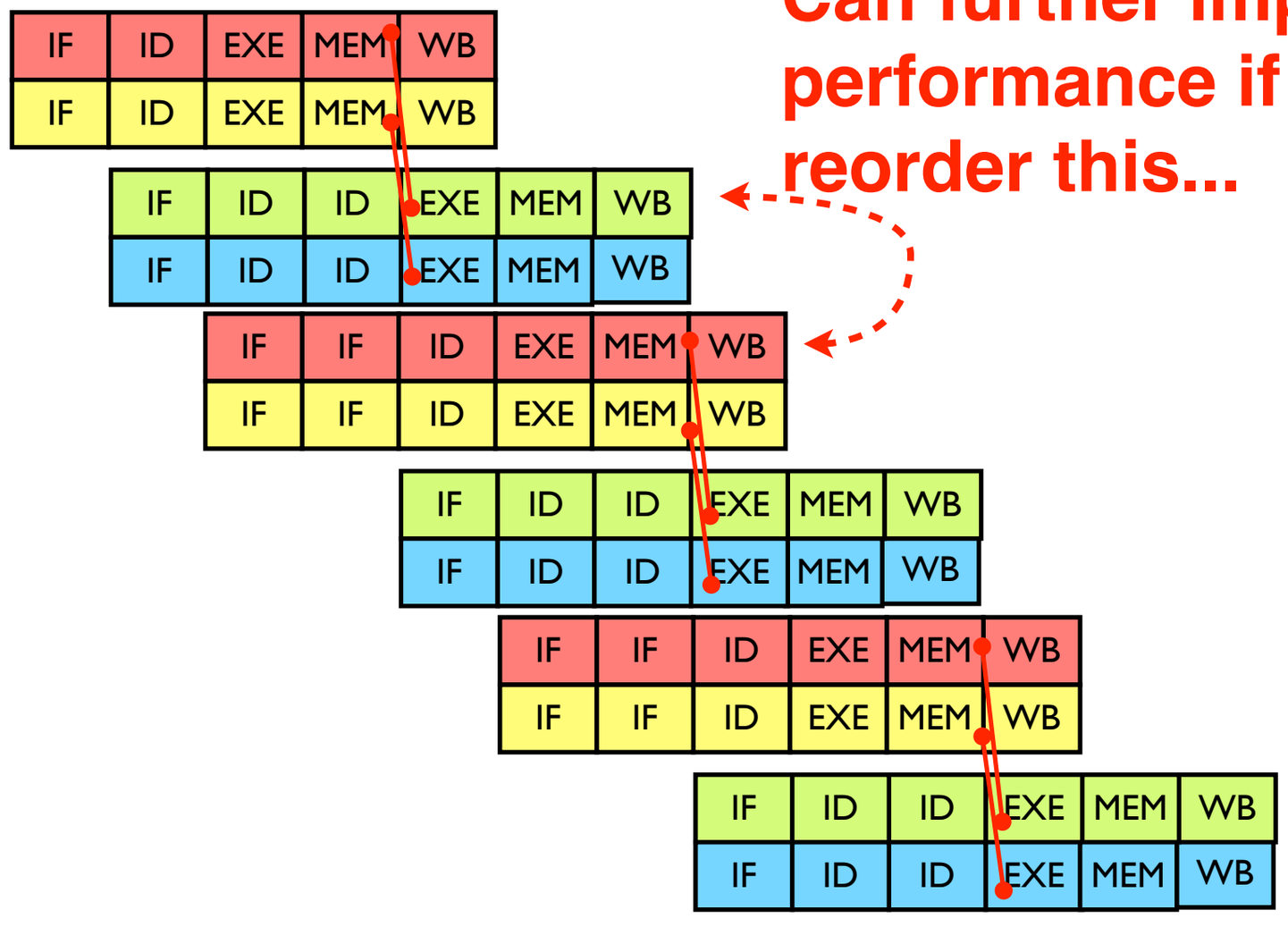


2 cycle per loop with perfect branch prediction: CPI = 0.5!
Pipeline takes 4 cycles per loop

Running compiler optimized code

- We can use compiler optimization to reorder the instruction sequence
- Compiler optimization requires no hardware change

```
lw   $t1, 0($a0)
addi $a0, $a0, 4
add  $v0, $v0, $t1
bne  $a0, $t0, LOOP
lw   $t1, 0($a0)
addi $a0, $a0, 4
add  $v0, $v0, $t1
bne  $a0, $t0, LOOP
lw   $t1, 0($a0)
addi $a0, $a0, 4
add  $v0, $v0, $t1
bne  $a0, $t0, LOOP
```



Can further improve performance if we can reorder this...

3 cycles if the processor predicts branch perfectly, CPI = 0.75

Limitations of compiler optimizations

- Compiler can only see/optimize **static instructions**, instructions in the compiled binary
- Compiler cannot optimize **dynamic instructions**, the real instruction sequence when executing the program
  - Compiler cannot re-order 3, 5 or 4,5
  - Compiler cannot predict cache misses
- Compiler optimization is constrained by **false dependencies** due to limited number of registers (even worse for x86)
  - Instructions `lw $t1, 0($a0)` and `addi $a0, $a0, 4` do not depend on each other
- Compiler optimizations do not work for all architectures
  - The code optimization in the previous example works for single pipeline, but not for superscalar

Static instructions

```
LOOP: lw   $t1, 0($a0)
      addi $a0, $a0, 4
      add  $v0, $v0, $t1
      bne  $a0, $t0, LOOP
      lw   $t0, 0($sp)
      lw   $t1, 4($sp)
```

Dynamic instructions

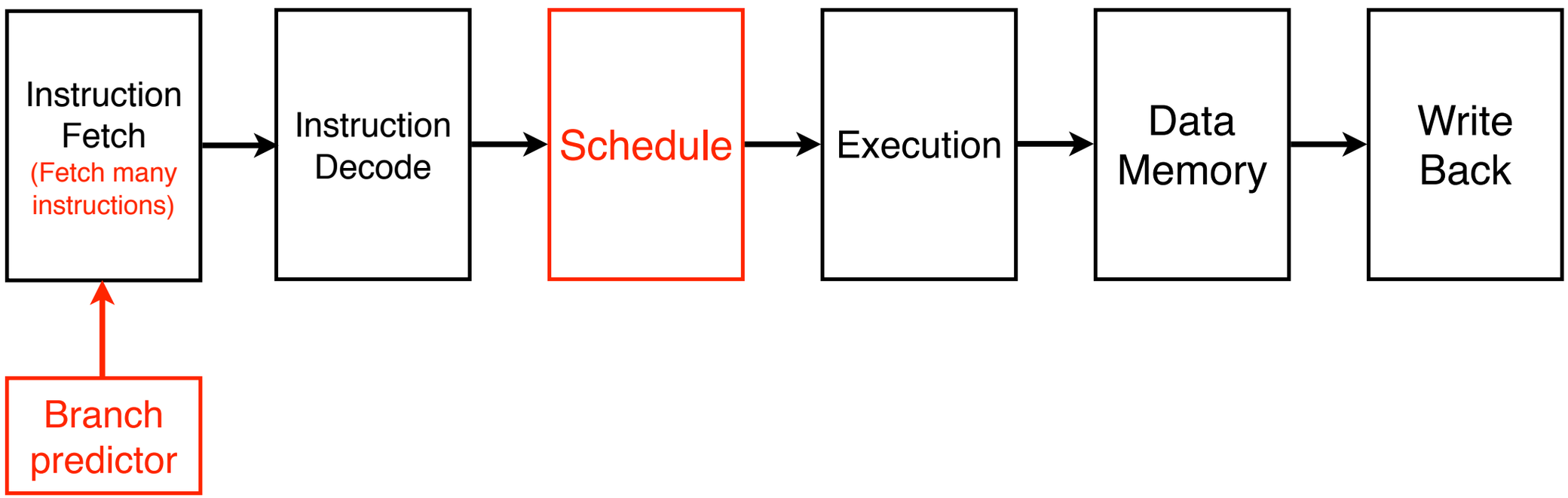
```
1: lw   $t1, 0($a0)
2: addi $a0, $a0, 4
3: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
5: lw   $t1, 0($a0)
6: addi $a0, $a0, 4
7: add  $v0, $v0, $t1
8: bne  $a0, $t0, LOOP
```

Simply superscalar + compiler optimization is not enough

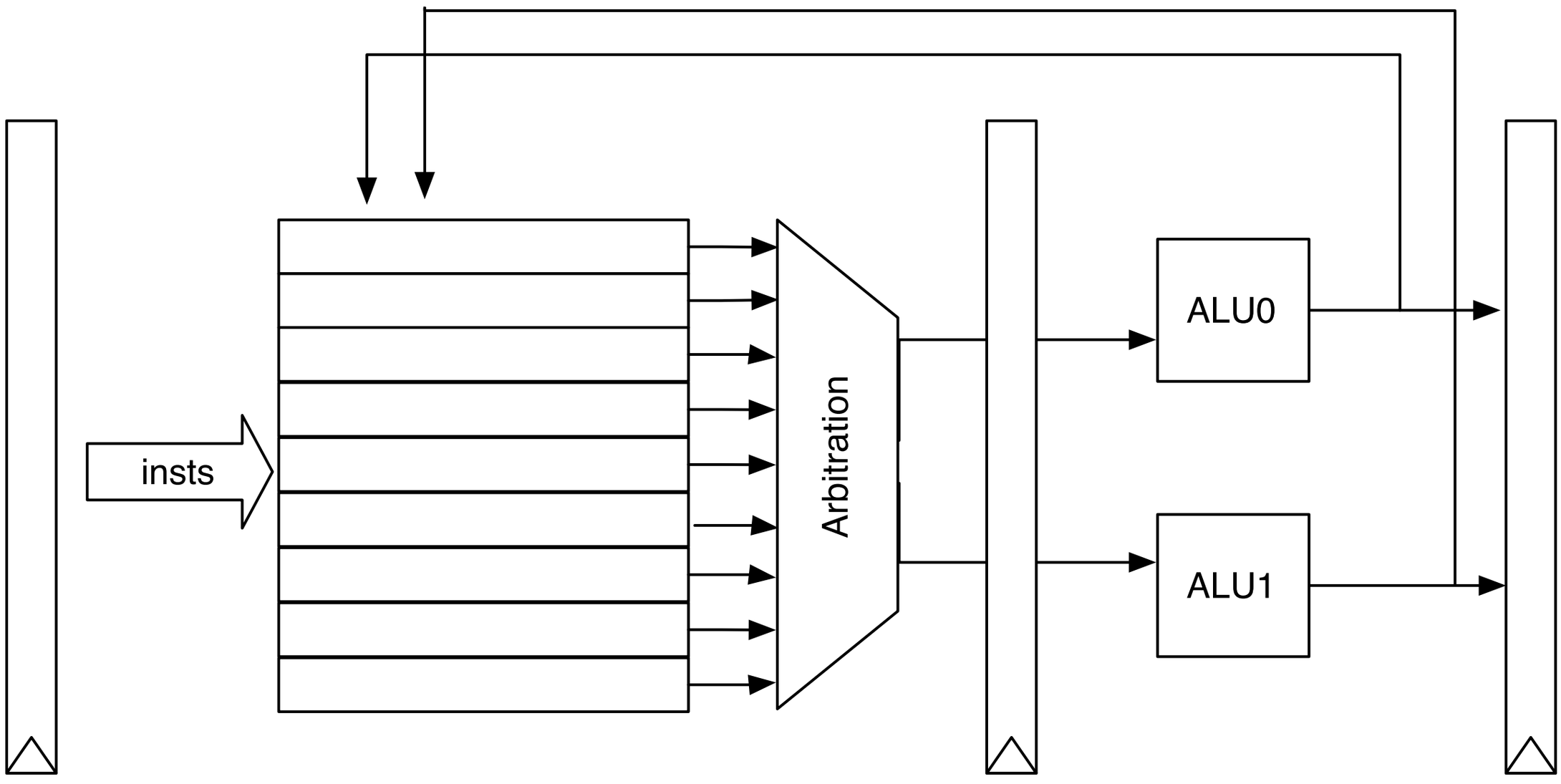
Dynamic out-of-order execution

Designing an out-of-order processor

- The goal is to “reorder/optimize instructions using **dynamic instructions**”
  - Needs to fetch multiple instructions at the same time so that we have more instructions to schedule
  - Needs the help of **branch prediction** to fetches instructions across the branch
- The hardware can schedule the execution of these fetched instructions



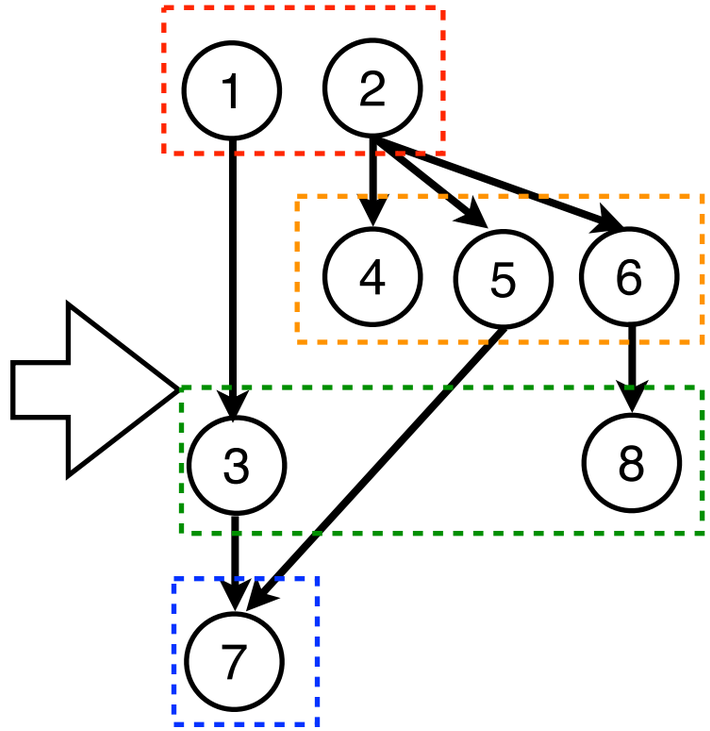
The instruction window



Scheduling instructions: based on data dependencies

- Draw the data dependency graph, put an arrow if an instruction depends on the other.
  - RAW (Read after write)

```
1: lw   $t1, 0($a0)
2: addi $a0, $a0, 4
3: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
5: lw   $t1, 0($a0)
6: addi $a0, $a0, 4
7: add  $v0, $v0, $t1
8: bne  $a0, $t0, LOOP
```

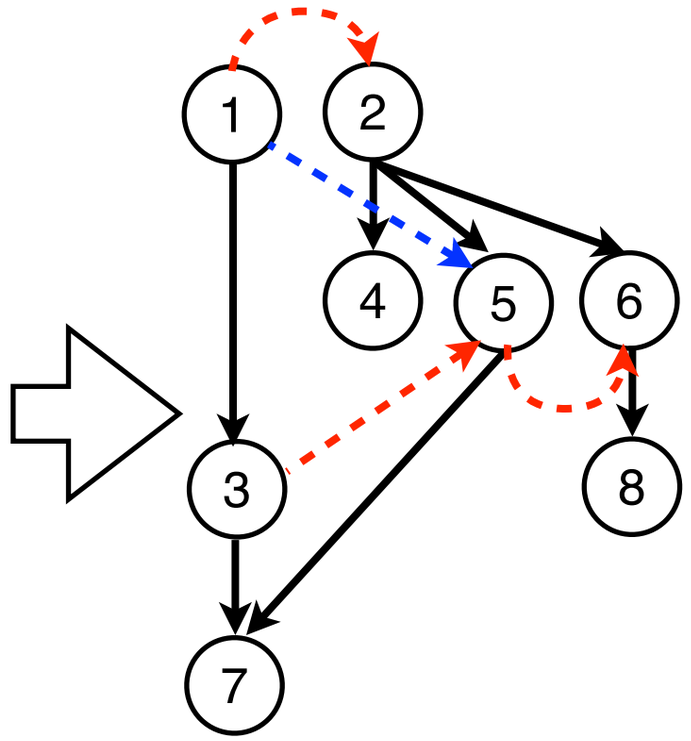


- **In theory**, instructions without dependencies can be executed in parallel or out-of-order
- Instructions with dependencies can never be reordered

False dependencies

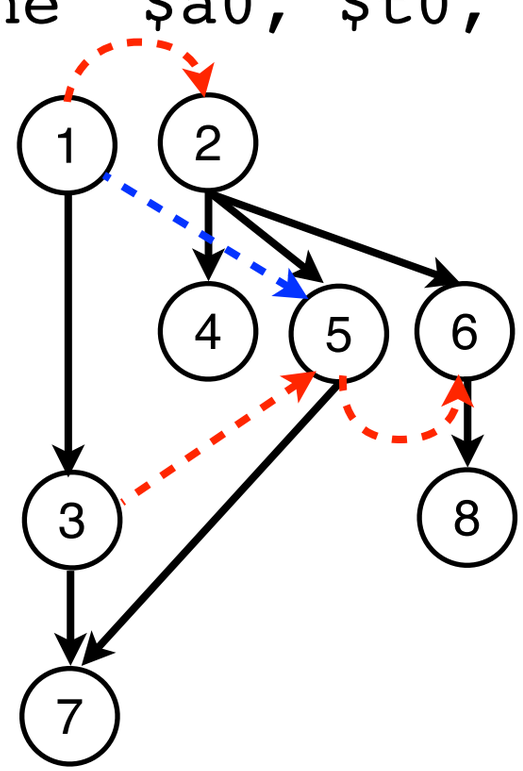
- We are still limited by **false dependencies**
- They are not "true" dependencies because they don't have an arrow in data dependency graph
  - WAR (Write After Read): a later instruction overwrites the source of an earlier one
    - 1 and 2, 3 and 5, 5 and 6
  - WAW (Write After Write): a later instruction overwrites the output of an earlier one
    - 1 and 5

```
1: lw   $t1, 0($a0)
2: addi $a0, $a0, 4
3: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
5: lw   $t1, 0($a0)
6: addi $a0, $a0, 4
7: add  $v0, $v0, $t1
8: bne  $a0, $t0, LOOP
```

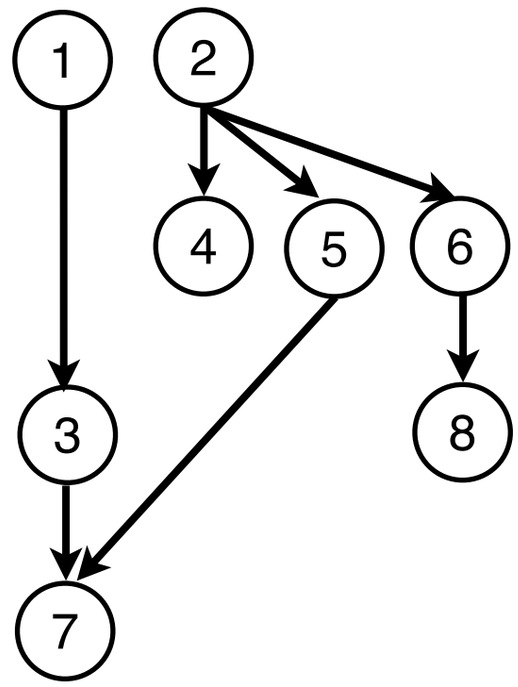


If we can transform the code ...

```
1: lw   $t1, 0($a0)
2: addi $a0, $a0, 4
3: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
5: lw   $t1, 0($a0)
6: addi $a0, $a0, 4
7: add  $v0, $v0, $t1
8: bne  $a0, $t0, LOOP
```




```
1: lw   $t1, 0($a0)
2: addi $a1, $a0, 4
3: add  $v1, $v0, $t1
4: bne  $a1, $t0, LOOP
5: lw   $t2, 0($a1)
6: addi $a2, $a1, 4
7: add  $v2, $v1, $t2
8: bne  $a2, $t0, LOOP
```



- We can get rid of the problem if each new output can use a different register!
- Compiler cannot do this because compiler cannot know if the second loop will executed or not!

Register renaming

- We can remove false dependencies if we can store each new output in a different register
- Architectural registers: an **abstraction** of registers visible to compilers and programmers
  - Like MIPS $0 -- $31
- Physical registers: the internal registers used for execution
  - Larger number than architectural registers
  - Modern processors have 128 physical registers
  - Invisible to programmers and compilers
- Maintains a mapping table between “physical” and “architectural” registers

Register renaming

Original code

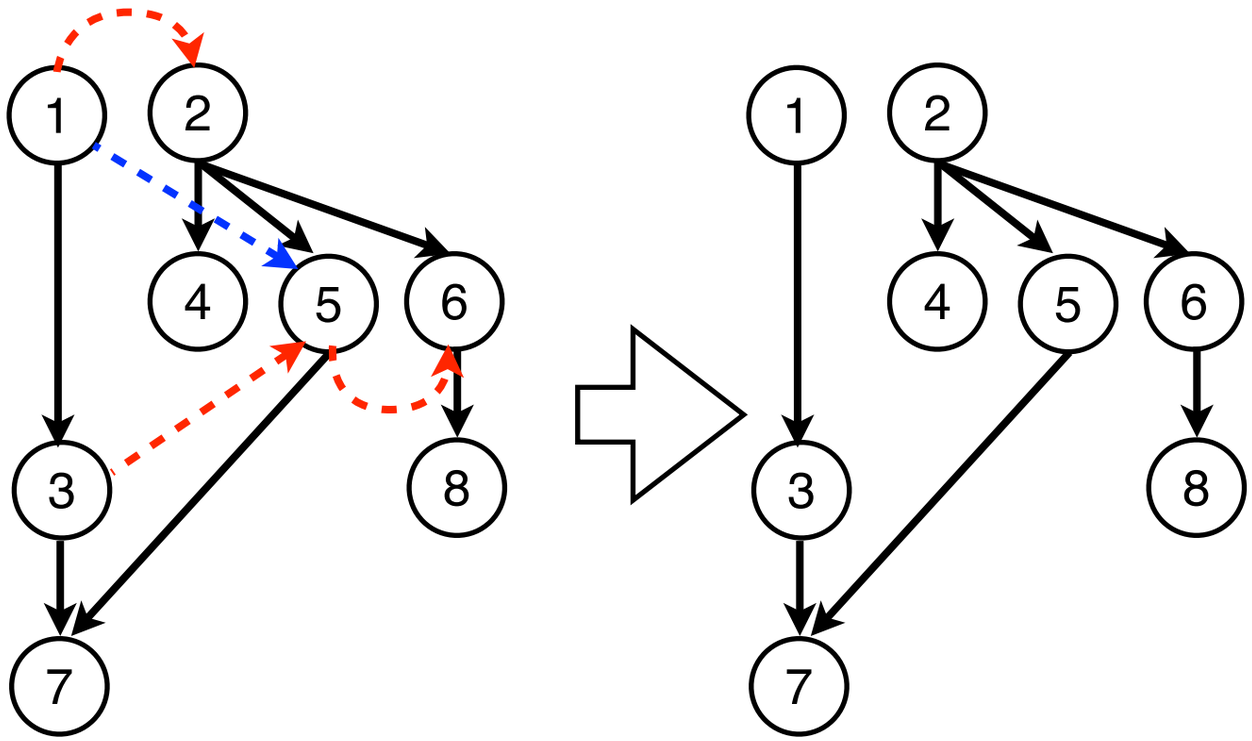
```
1: lw   $t1, 0($a0)
2: addi $a0, $a0, 4
3: add  $v0, $v0, $t1
4: bne  $a0, $t0, LOOP
5: lw   $t1, 0($a0)
6: addi $a0, $a0, 4
7: add  $v0, $v0, $t1
8: bne  $a0, $t0, LOOP
```

After renamed

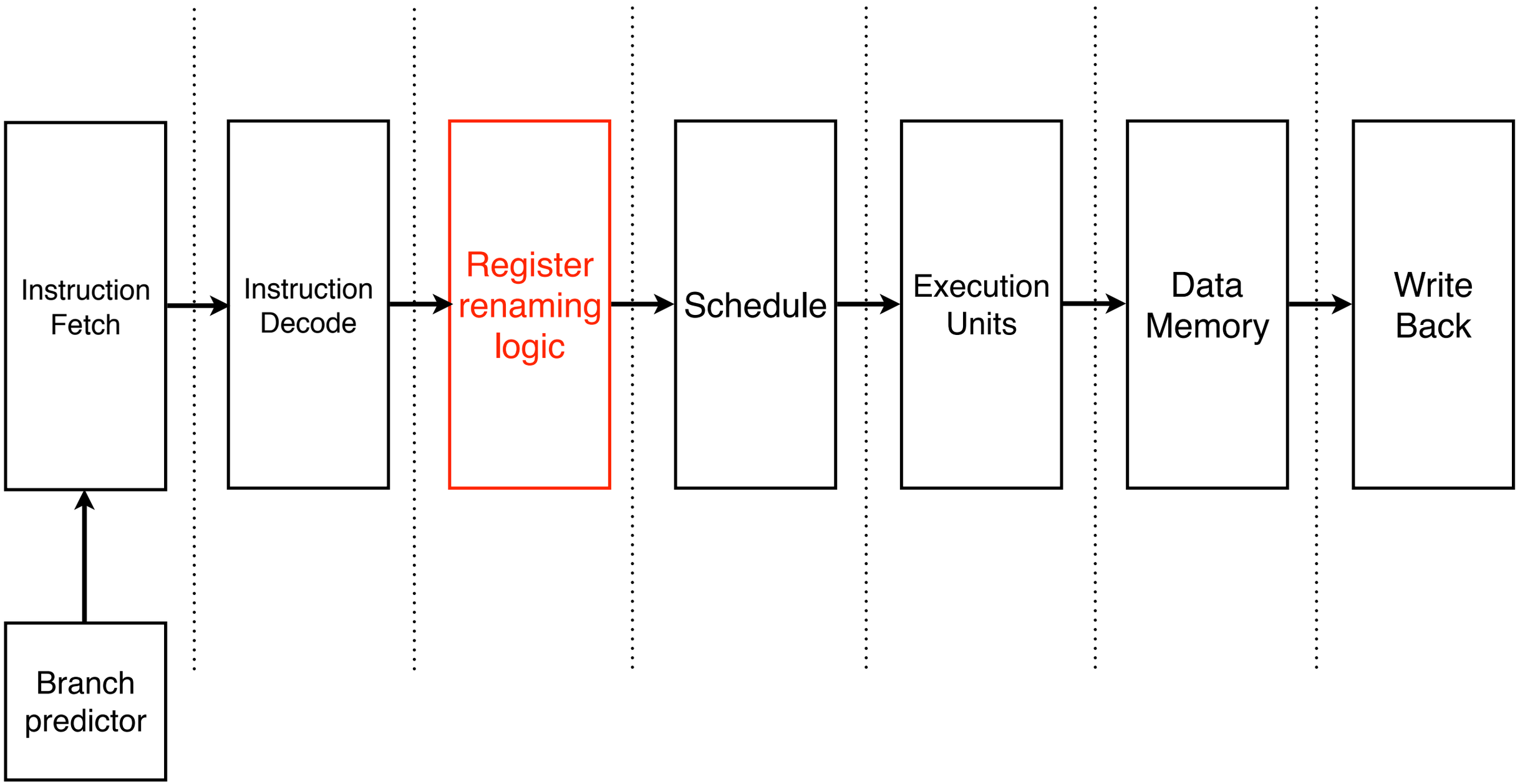
```
1: lw   $p5 , 0($p1)
2: addi $p6 , $p1, 4
3: add  $p7 , $p4, $p5
4: bne  $p6 , $p2, LOOP
5: lw   $p8 , 0($p6)
6: addi $p9 , $p6, 4
7: add  $p10, $p7, $p8
8: bne  $p9 , $p2, LOOP
```

Register map

cycle	$a0	$t0	$t1	$v0
0	p1	p2	p3	p4
1	p1	p2	p5	p4
2	p6	p2	p5	p4
3	p6	p2	p5	p7
4	p6	p2	p5	p7
5	p6	p2	p8	p7
6	p9	p2	p8	p7
7	p9	p2	p8	p10
8	p9	p2	p8	p10



Simplified OOO pipeline



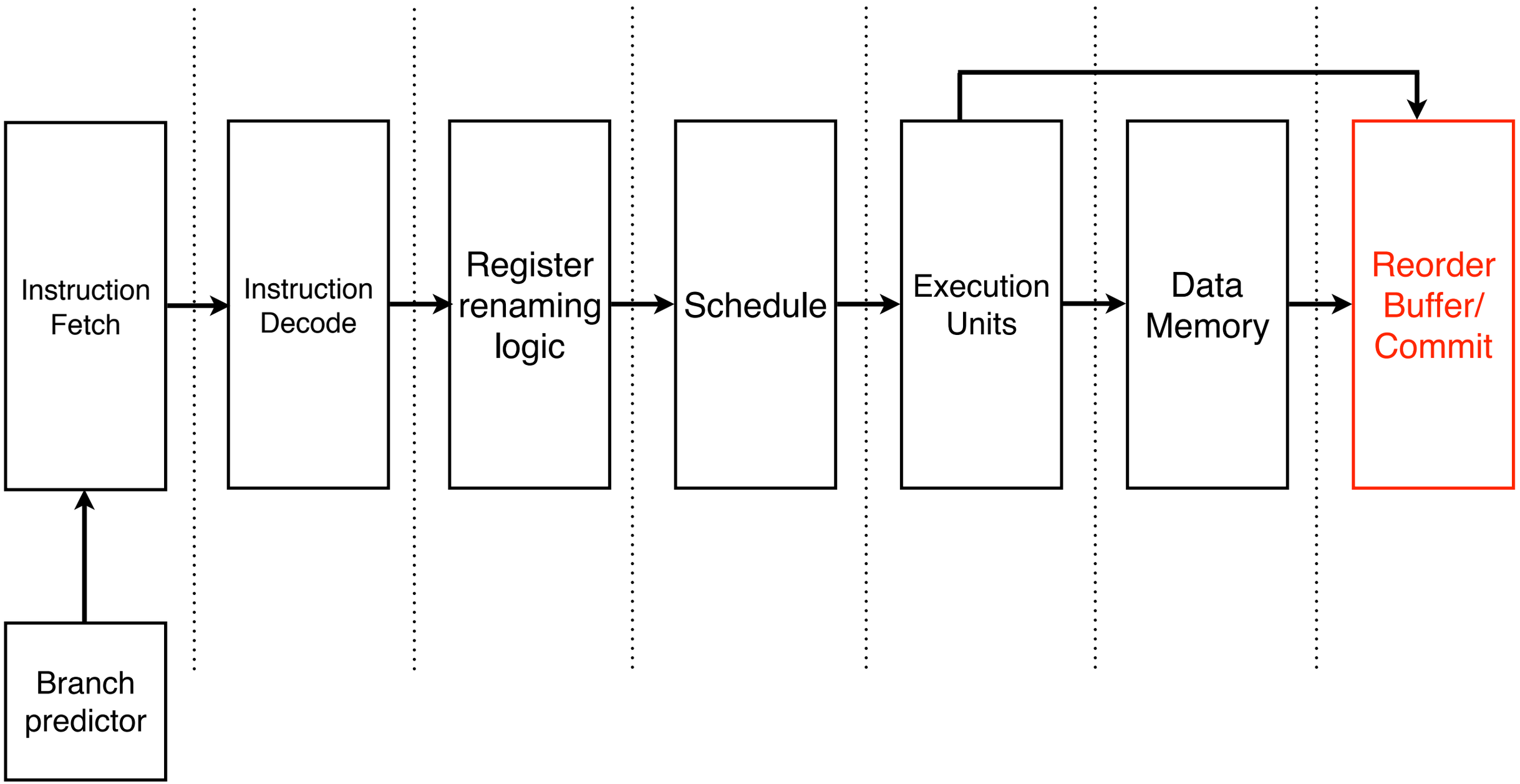
Scheduling across branches

- Hardware can schedule instruction across branch instructions with the help of branch prediction
  - Fetch instructions according to the branch prediction
  - However, branch predictor can never be perfect
- Execute instructions across branches
  - Speculative execution: execute an instruction before the processor know if we need to execute or not
  - Execute an instruction all operands are ready (the values of depending physical registers are generated)
  - Store results in “reorder buffer” before the processor knows if the instruction is going to be executed or not.

Reorder buffer

- An instruction will be given an reorder buffer entry number
- A instruction can “retire”/ “commit” only if all its previous instructions finishes.
- If branch mis-predicted, “flush” all instructions with later reorder buffer indexes and clear the occupied physical registers
- We can implement the reorder buffer by extending instruction window or the register map.

Simplified OOO pipeline



Dynamic execution with register naming

- Register renaming with unlimited physical registers, dynamical scheduling with 2-issue pipeline
- Assume that we fetch/decode/renaming/retire 4 instructions into/from instruction window each cycle
- Assume load needs 2 cycles to execute (one cycle address calculation and one cycle memory access)

After renamed

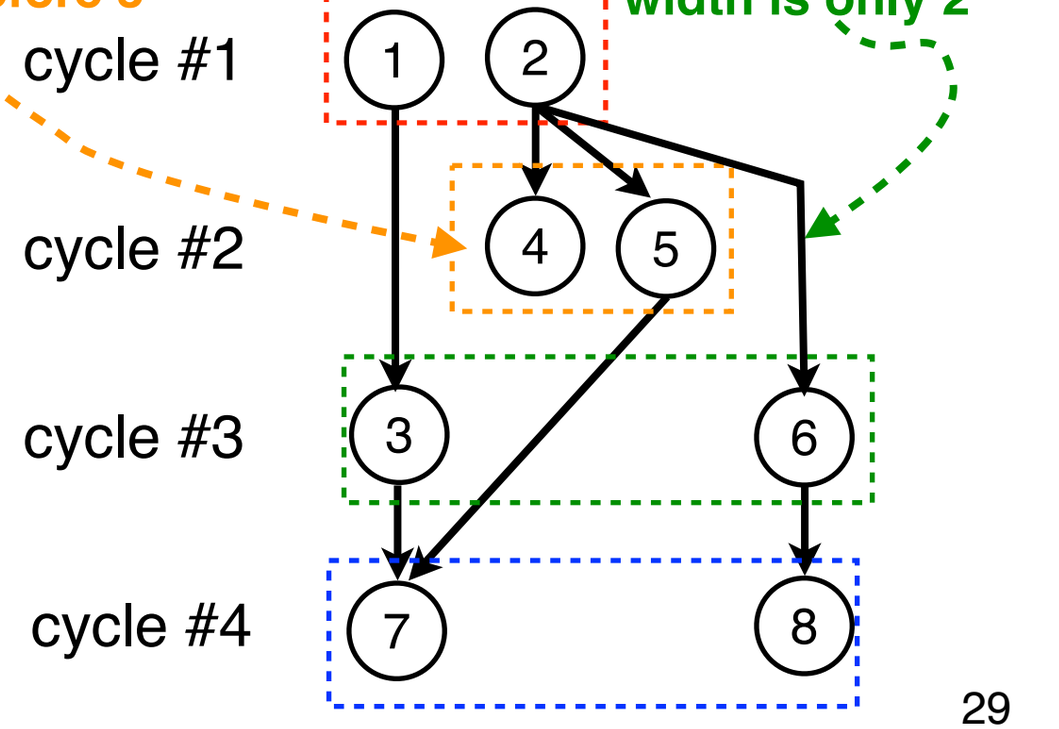
available in cycle #1

available in cycle #2

```
1: lw   $p5 , 0($p1)
2: addi $p6 , $p1, 4
3: add  $p7 , $p4, $p5
4: bne  $p6 , $p2, LOOP
5: lw   $p8 , 0($p6)
6: addi $p9 , $p6, 4
7: add  $p10, $p7, $p8
8: bne  $p9 , $p2, LOOP
```

4 and 5 are issues before 3

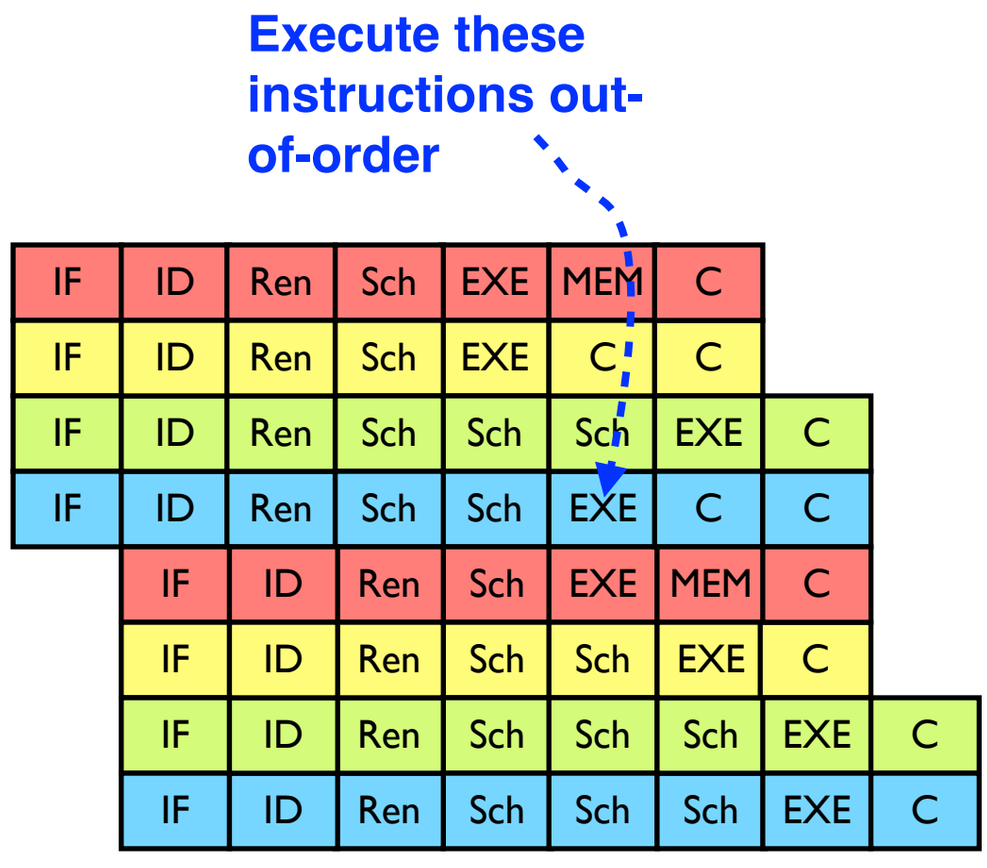
Cannot issue because the issue width is only 2



Dynamic execution with register naming

- Register renaming with unlimited physical registers, dynamical scheduling with 2-issue pipeline
- Assume that we fetch/decode/renaming/retire 4 instructions into/from instruction window each cycle

```
1: lw   $p5 , 0($p1)
2: addi $p6 , $p1, 4
3: add  $p7 , $p4, $p5
4: bne  $p6 , $p2, LOOP
5: lw   $p8 , 0($p6)
6: addi $p9 , $p6, 4
7: add  $p10, $p7, $p8
8: bne  $p9 , $p2, LOOP
```

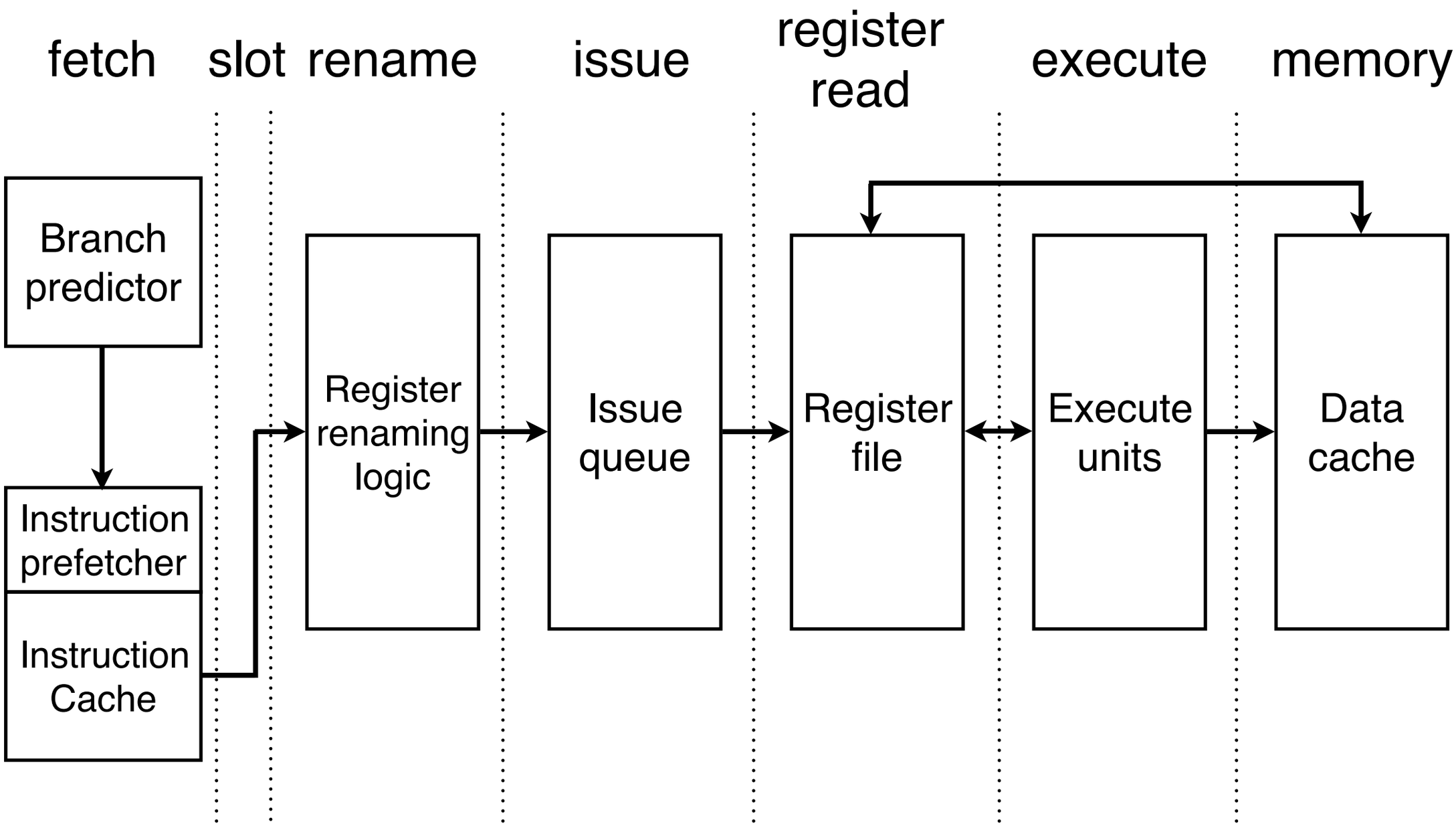


Execute/issue 2 instructions per cycle, CPI = 0.5

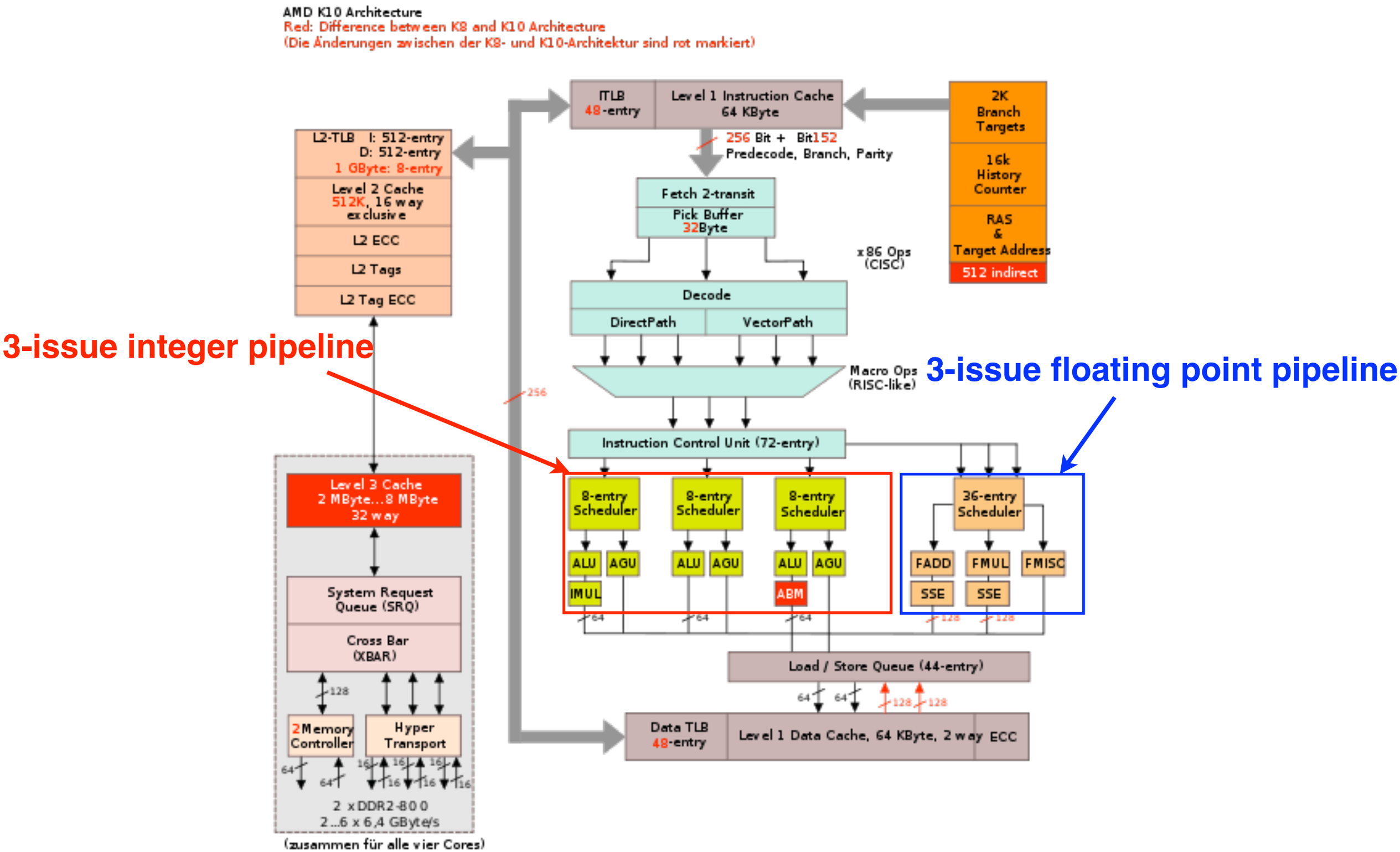
Problems with OOO+Superscalar

- The modern OOO processors have 3-6 issue widths
- Keeping instruction window filled is hard
  - Branches are every 4-5 instructions.
  - If the instruction window is 32 instructions the processor has to predict 6-8 consecutive branches correctly to keep IW full.
- The ILP within an application is low
  - Usually 1-2 per thread
  - ILP is even lower is data depends on memory operations (if cache misses) or long latency operations
- Demo

Example: Alpha 21264



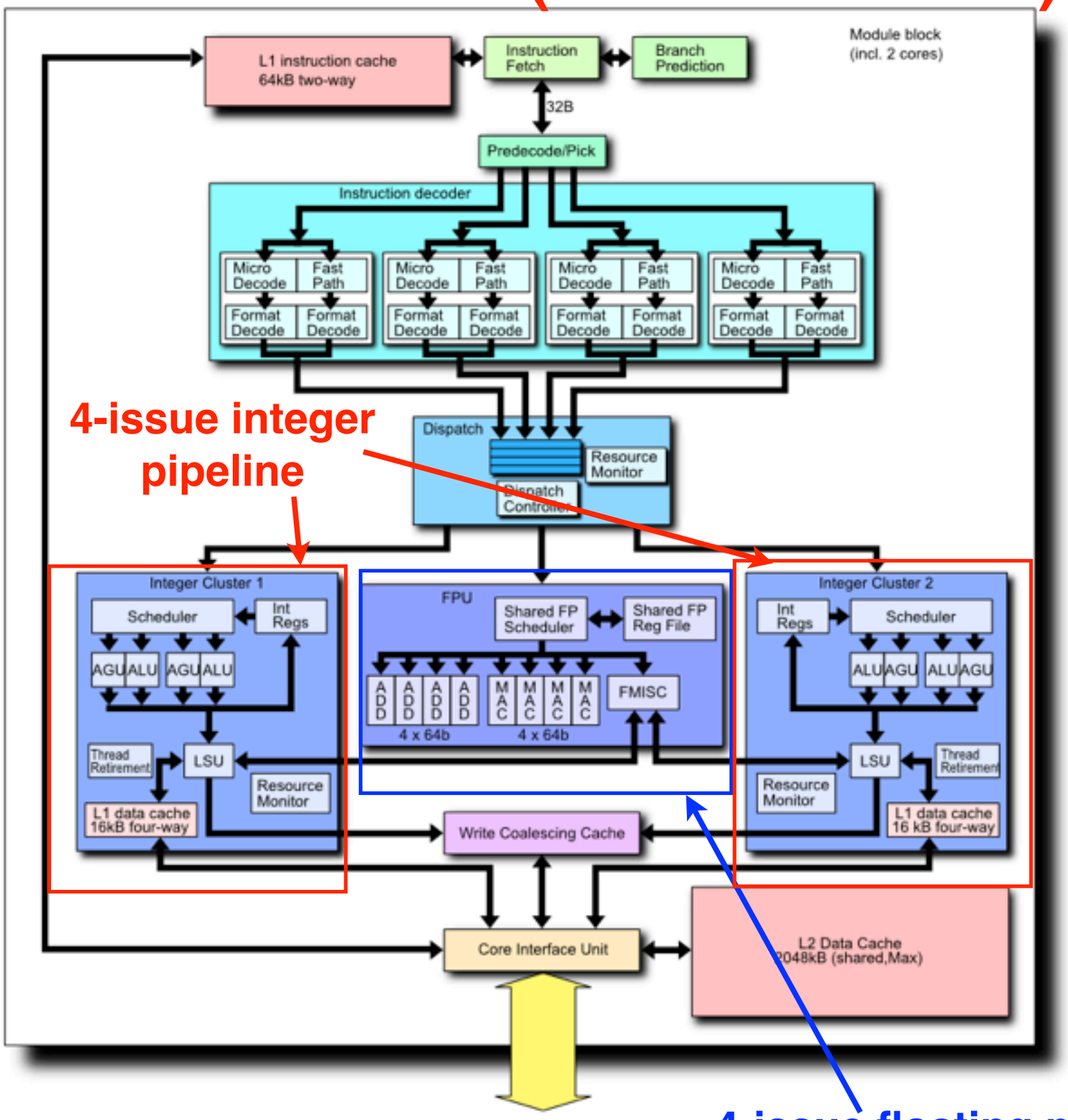
AMD K10 architecture



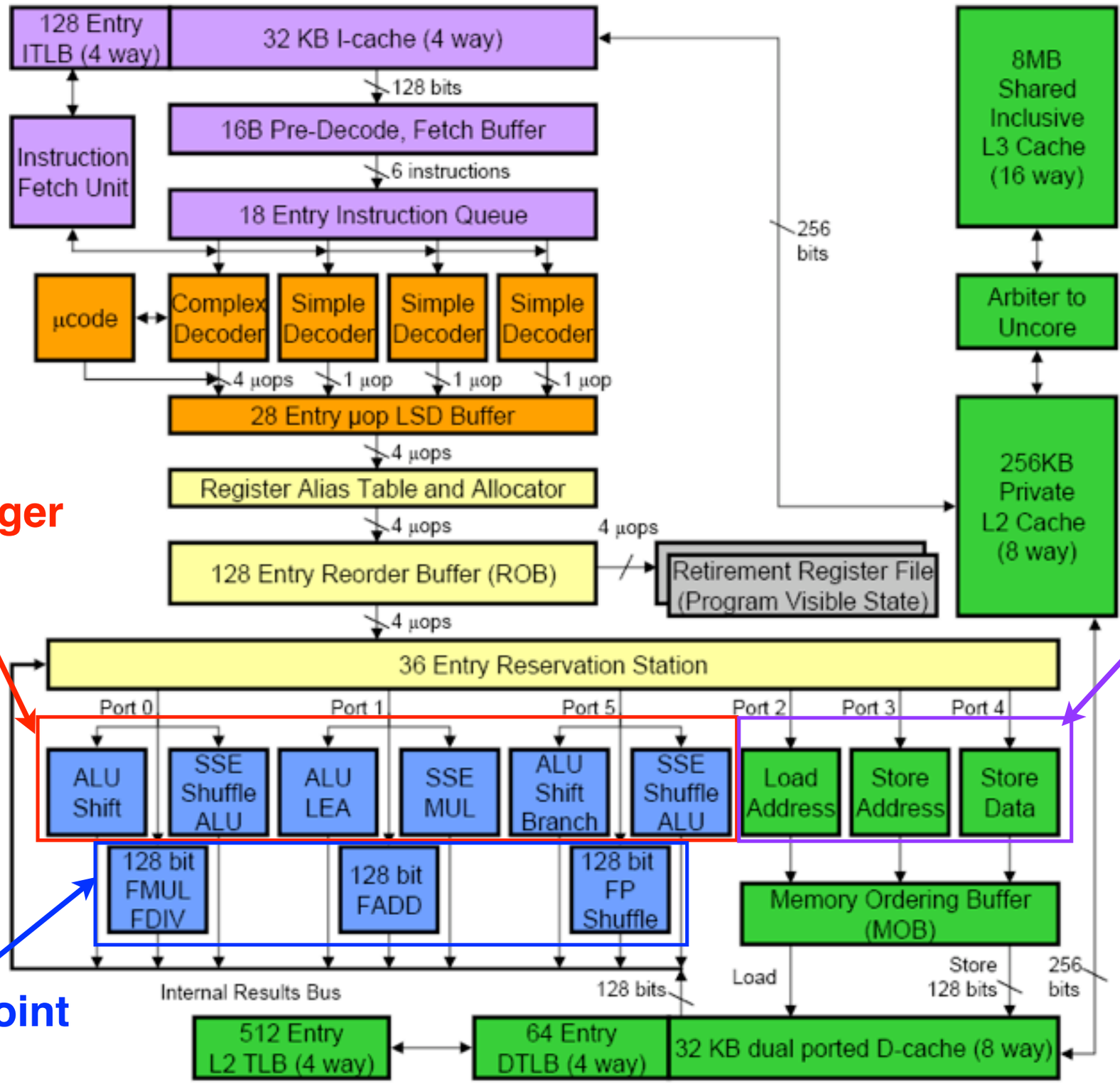
3-issue integer pipeline

3-issue floating point pipeline

AMD FX (Bulldozer)



intel Nehalem (1st gen core i7)



3-issue integer pipeline

3-issue memory pipeline

3-issue floating point pipeline

Intel SkyLake architecture