CS-200
Computer Architecture
—
Part 4f. Instruction Level Parallelism
Besides and Beyond Superscalars

Paolo Ienne
<paolo.ienne@epfl.ch>

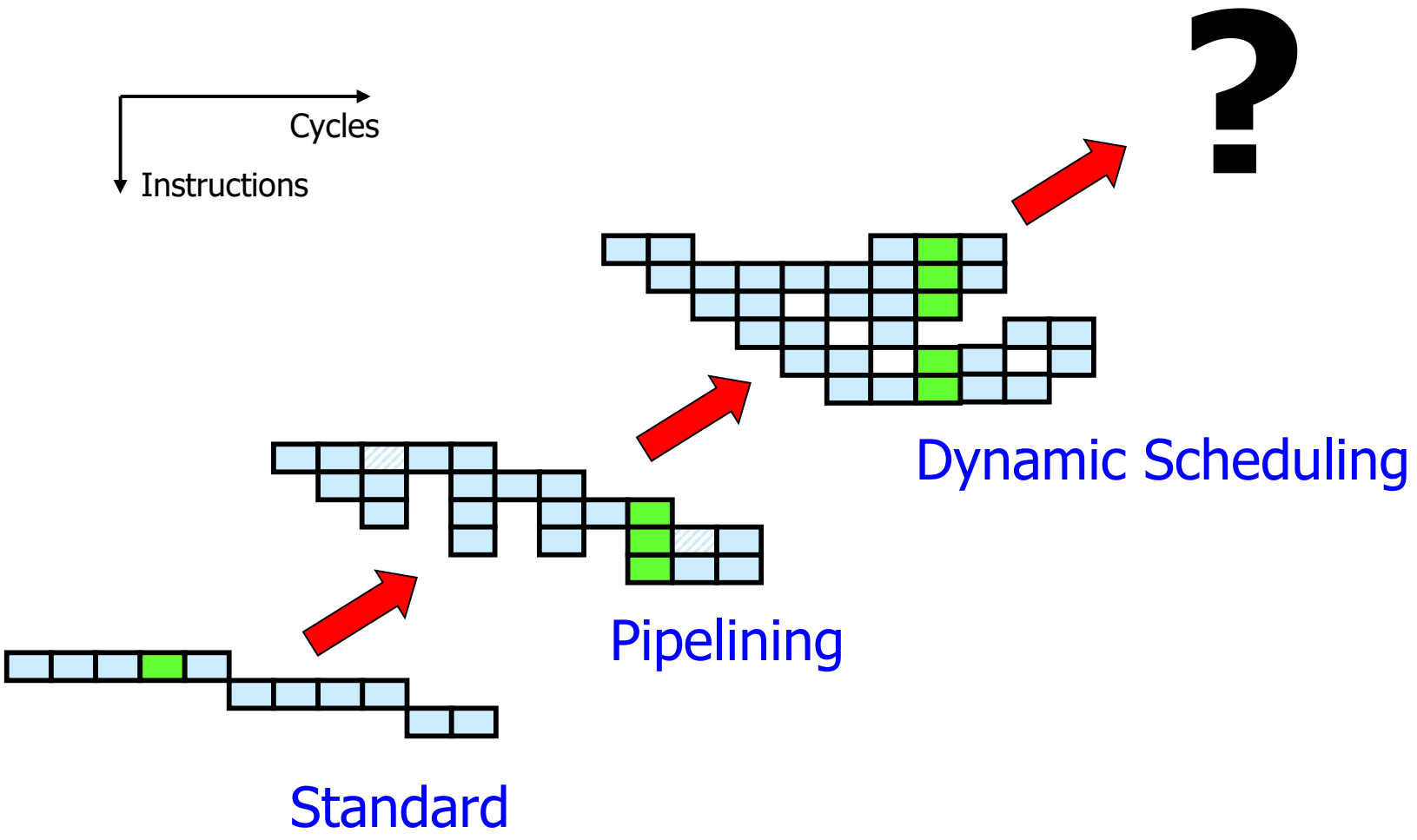
Content of This Lecture

1. Superscalar processors
2. Speculative execution
3. Simultaneous multithreading
4. Nonblocking caches
5. Very Long Instruction Word (VLIW) processors

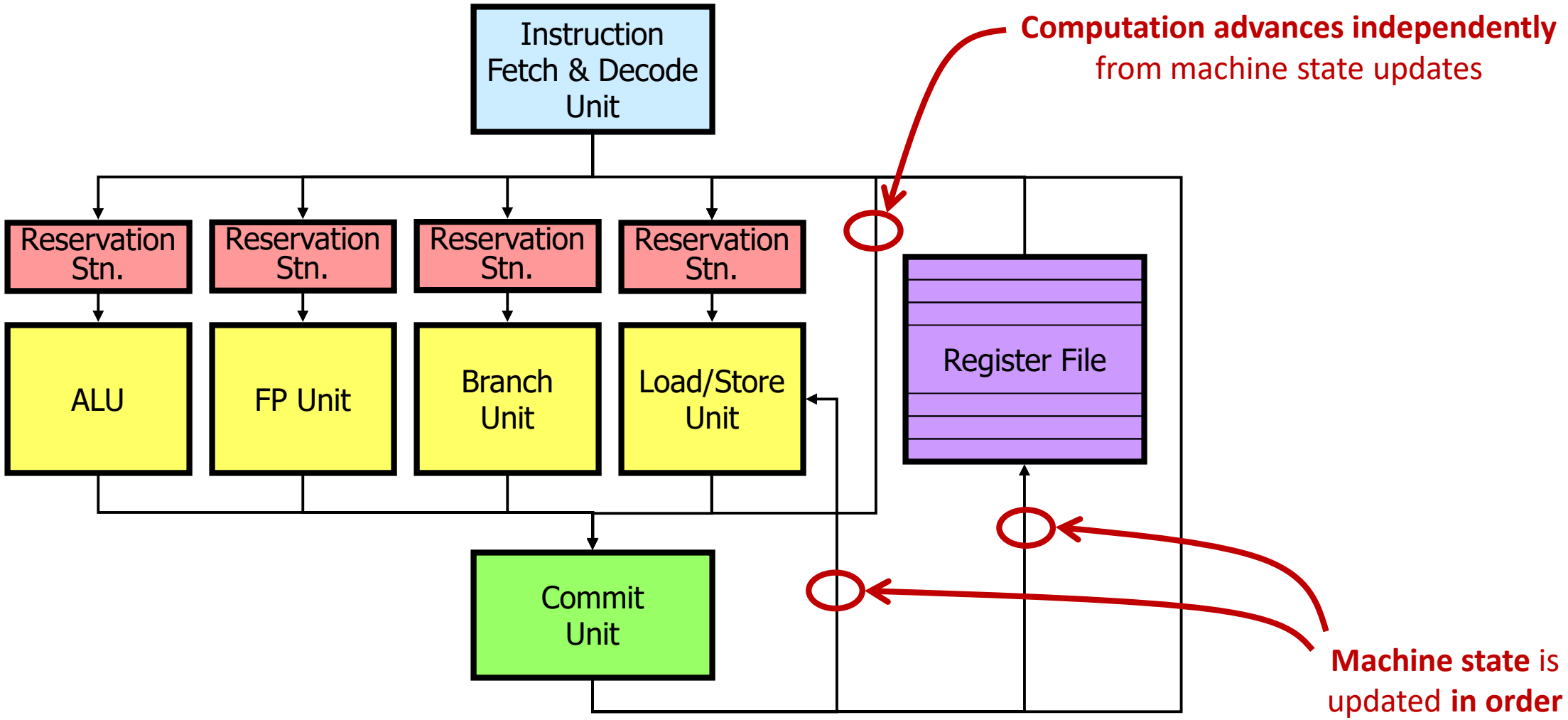
1

Superscalar Processors

ILP So Far…


Cycles
Instructions
?
Dynamic Scheduling
Pipelining
Standard

Dynamically Scheduled Processor

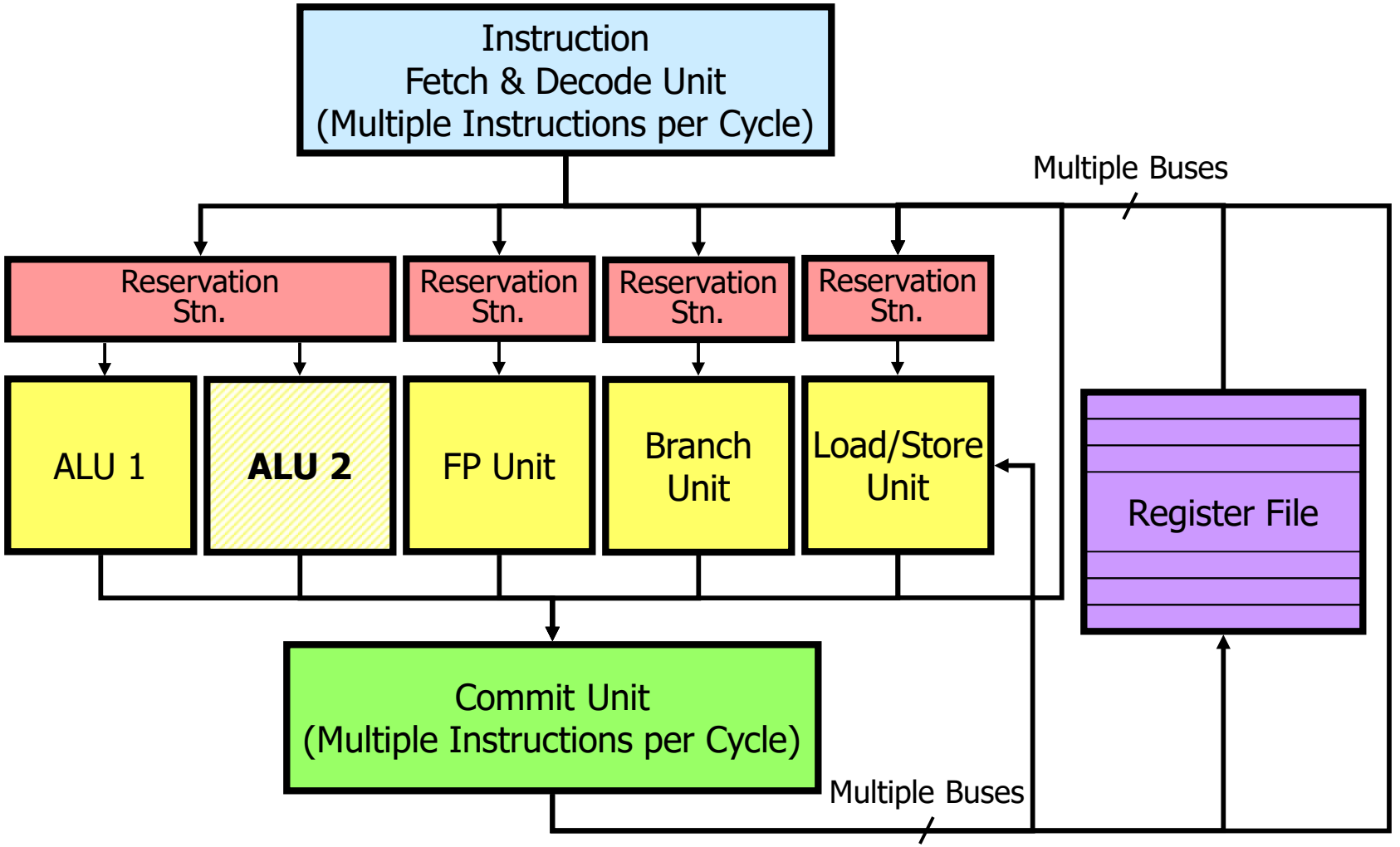


Superscalar Execution

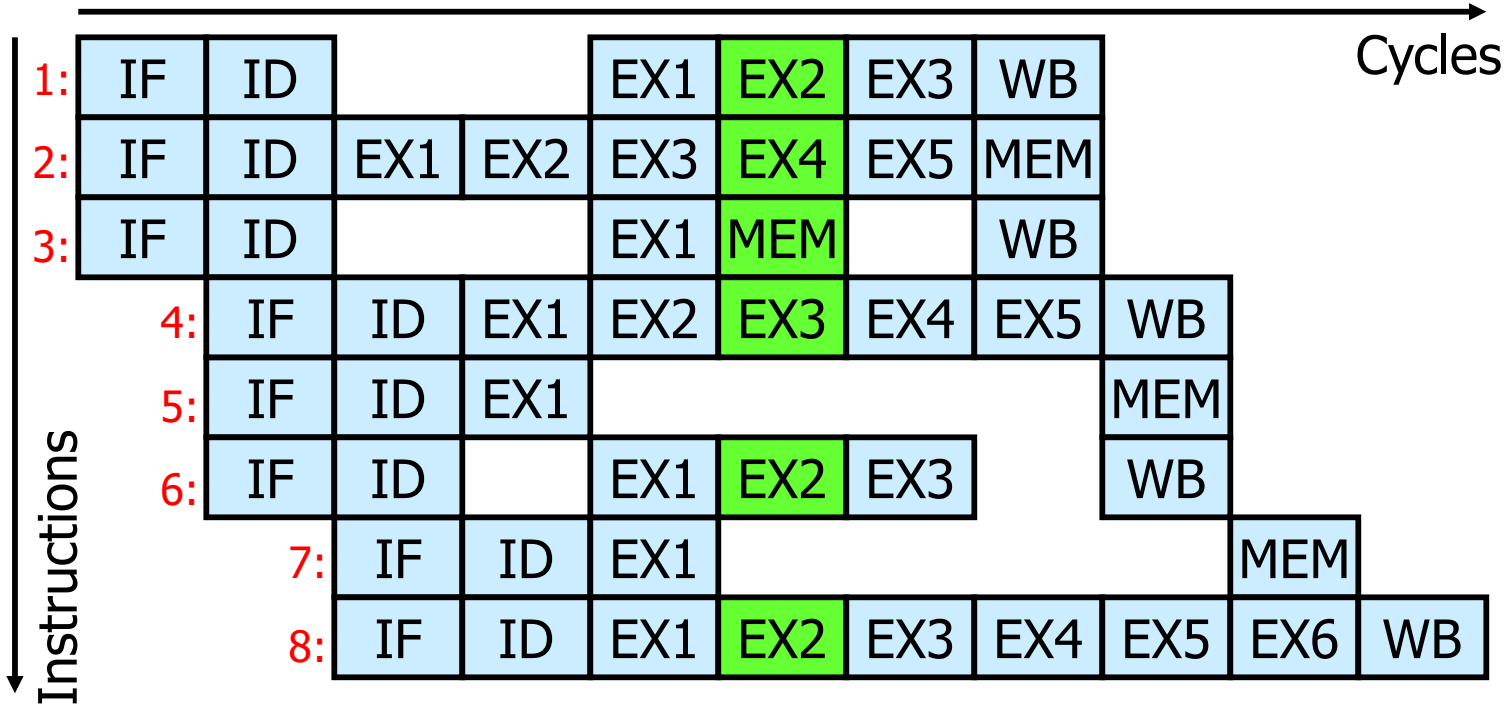
- Why not to **issue** (= start execution) **more than one instruction per cycle**?
  - In fact, we are already doing this…
- Key improvements and requirements:
  - **Fetch more instruction per cycle**: no big difficulty if the instruction cache can sustain the bandwidth
  - **Commit more instruction per cycle**: the ROB and the register file must have enough ports
  - Obey data and control dependencies: dynamic scheduling already takes care of this

Data and control hazards are
the **ultimate limit** to **parallelism**

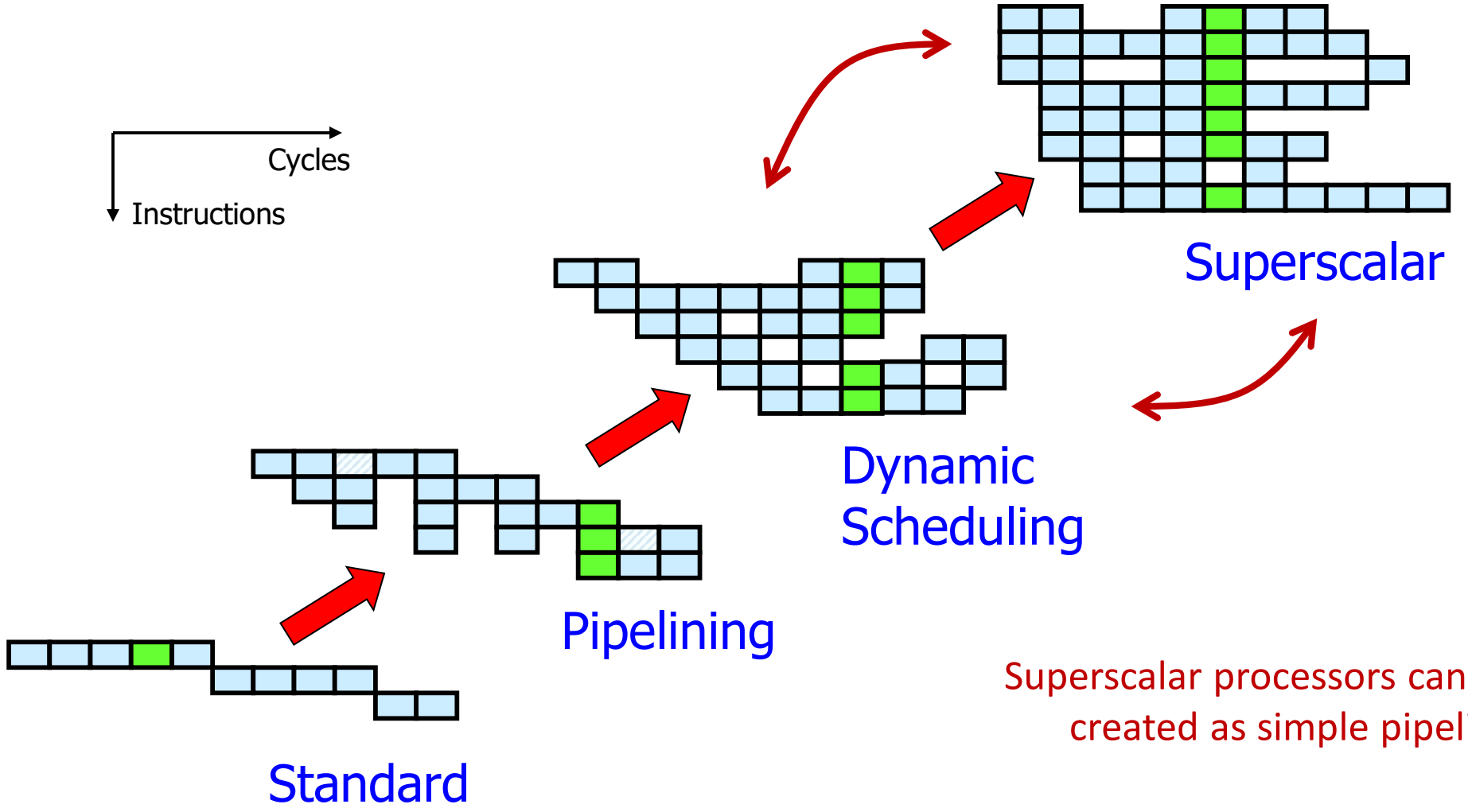
Superscalar Processor



Third Step: Superscalar Execution



Several Steps in Exploiting ILP



Superscalar processors can also be created as simple pipelines

2

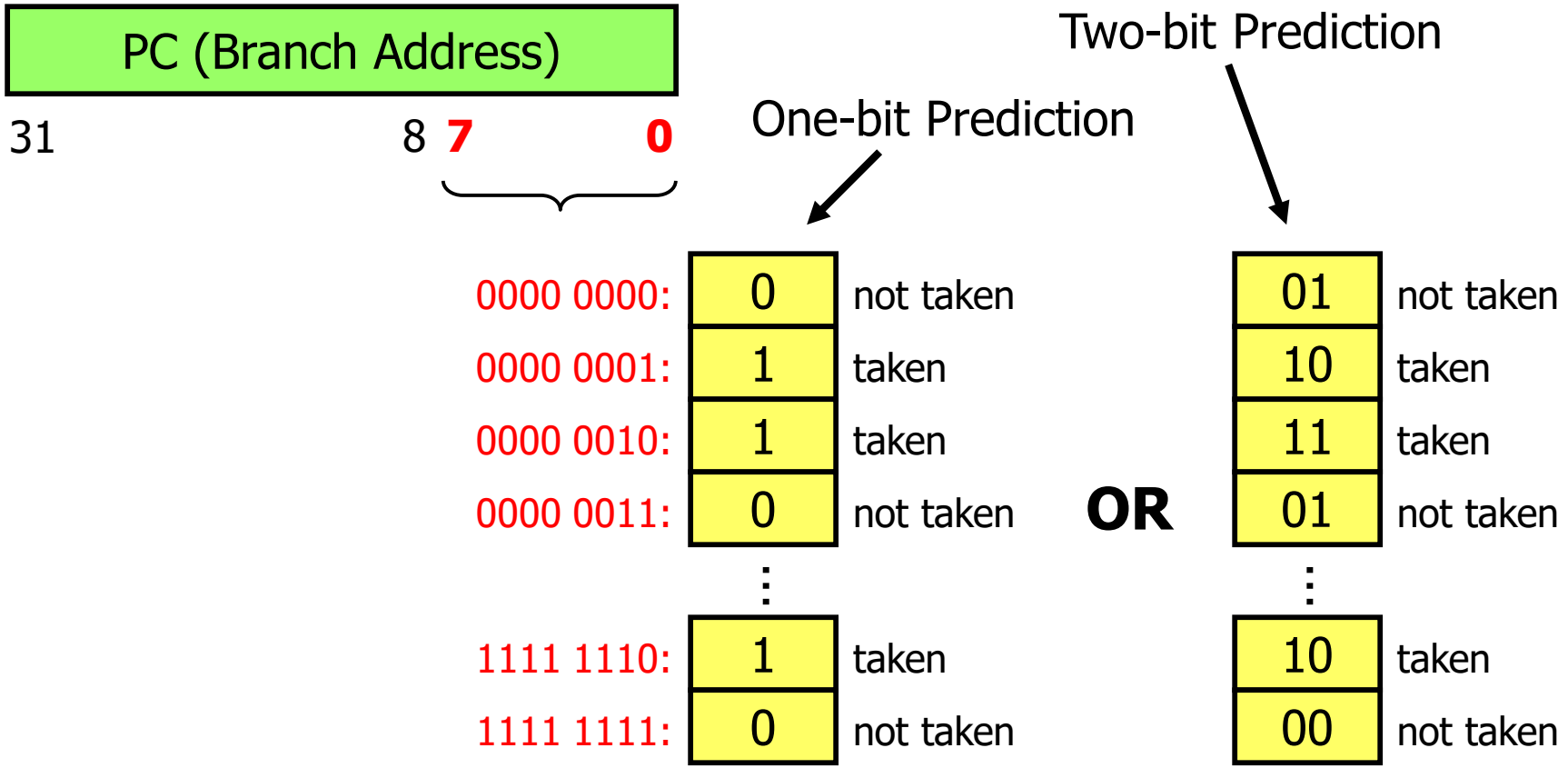
Speculative Execution

Dynamic Branch Prediction

- The **biggest problem** left to continue extracting Instruction Level Parallelism are
  - **True data dependencies**: instructions **cannot** be executed! Not much we can do about that…
  - **Branches**: where to look for other candidate instructions?

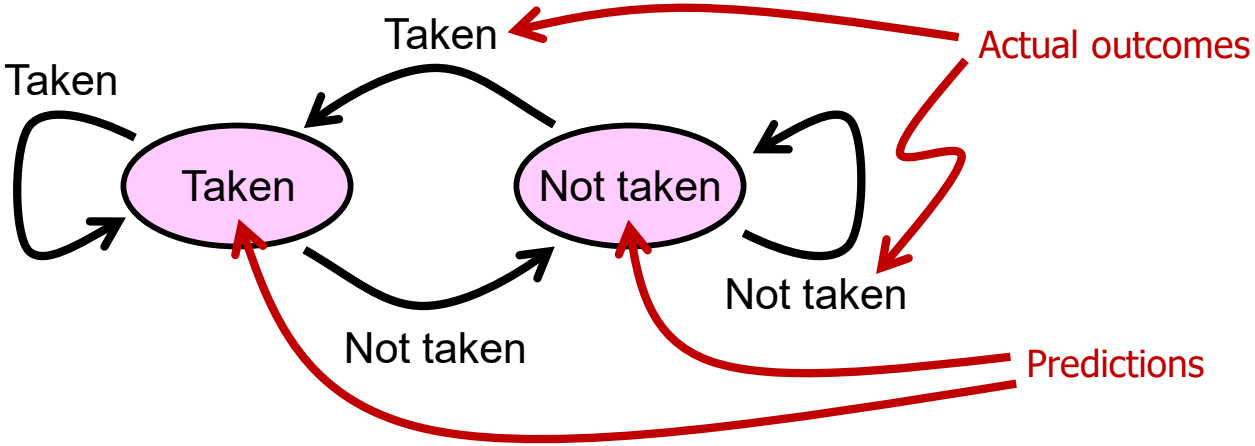
- **Static** prediction not very accurate and somehow hard to use
  - Never-taken, Always-taken-backward, Compiler-Specified
  - How does one know which one is right?
- **Dynamic** prediction: learn from history
  - Count how often a branch was taken in the past

Branch History Table

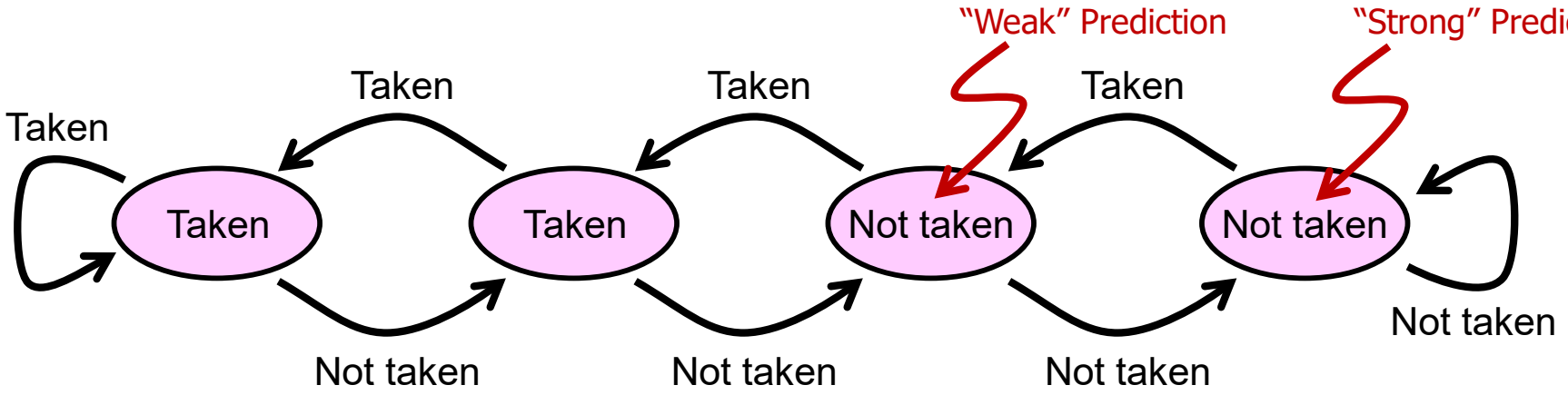


One- vs. Two-Bit Prediction Schemes

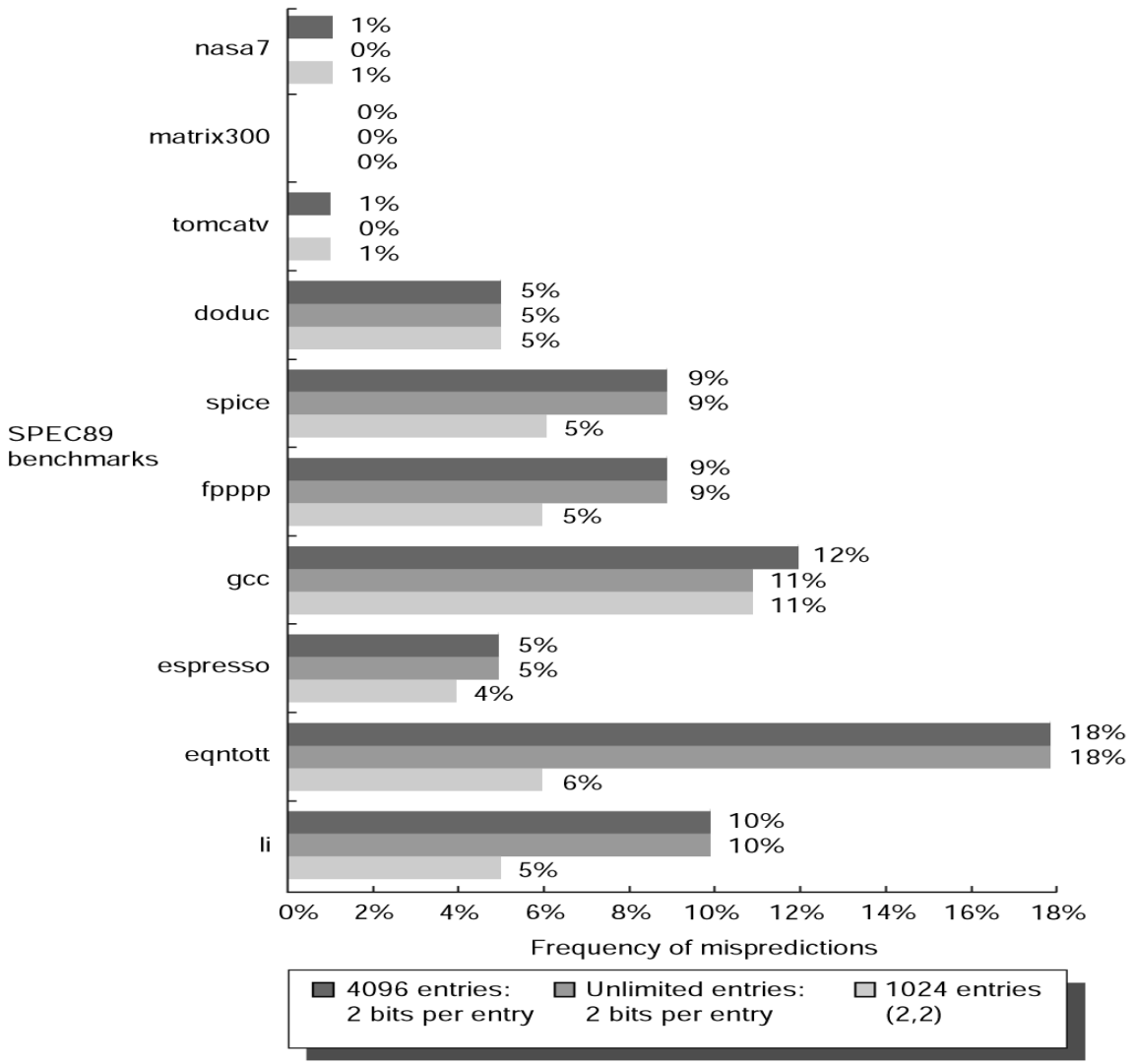
- Simplest one-bit predictor: “do the same as last time”



- Two-bit predictor (saturating counter): adding some “inertia” or “take some time to change your mind”



Prediction Accuracy



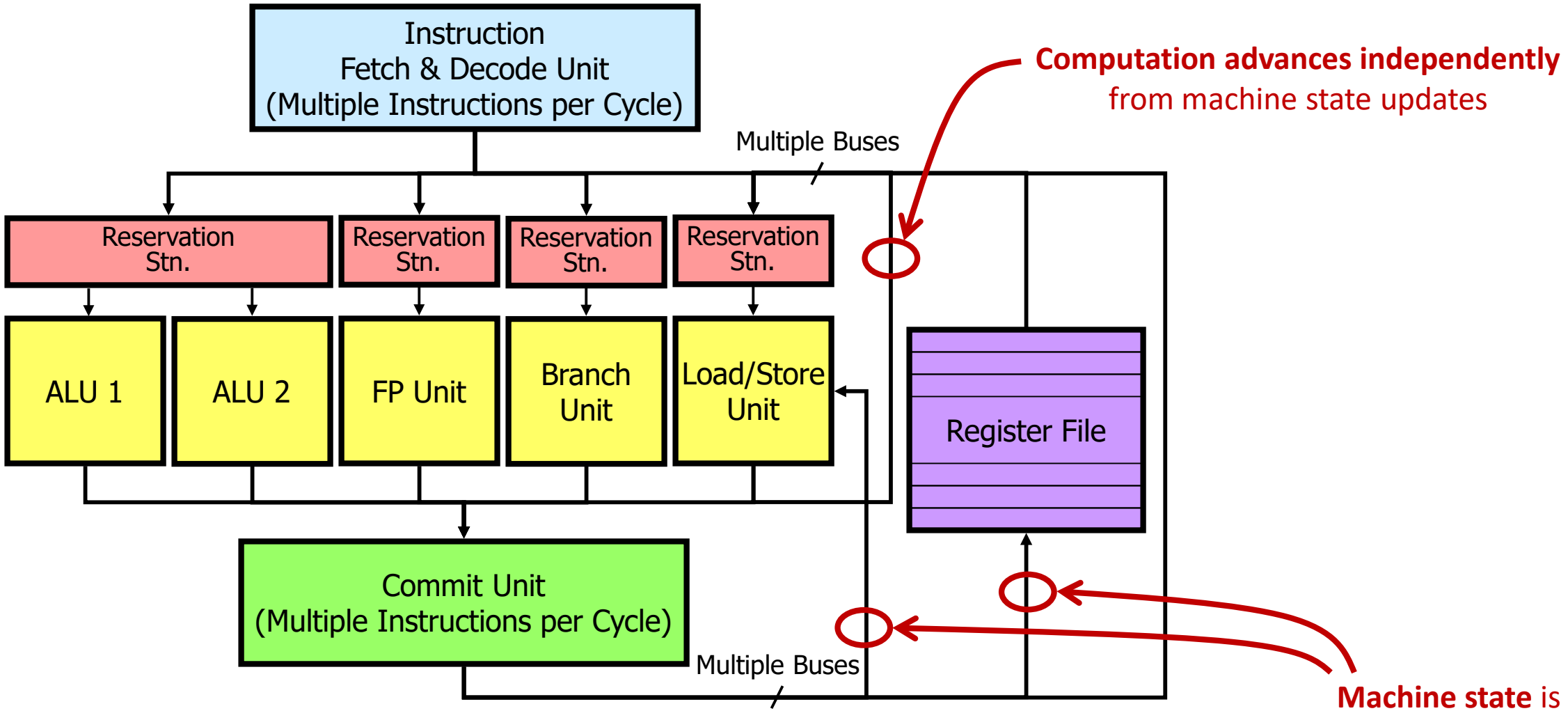
Source: AQA, © Morgan Kauffman 1996

Speculative Execution

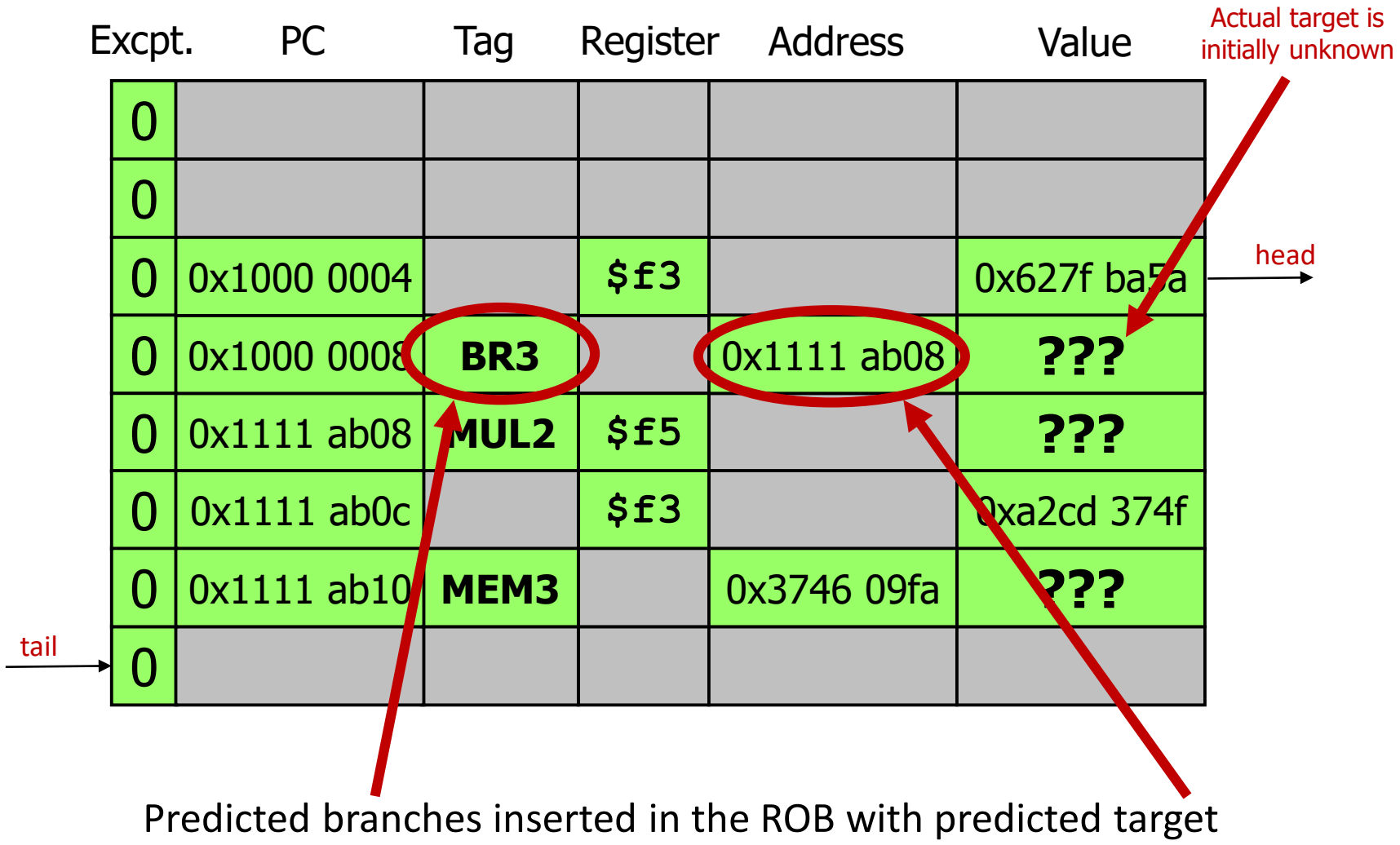
- We have been using **Dynamic Branch Prediction** only to tentatively **Fetch** and **Decode** instructions → no effect on registers and memory, so **easy to squash**
- More aggressively, one could **Execute** instructions (and use their results) before the branch target is known: **Speculative Execution**

- We need to **prevent changes to the architectural state** of the processor until the correctness of the prediction is known:
  - Was it right? Good!
  - Was it wrong? **Squash it!**

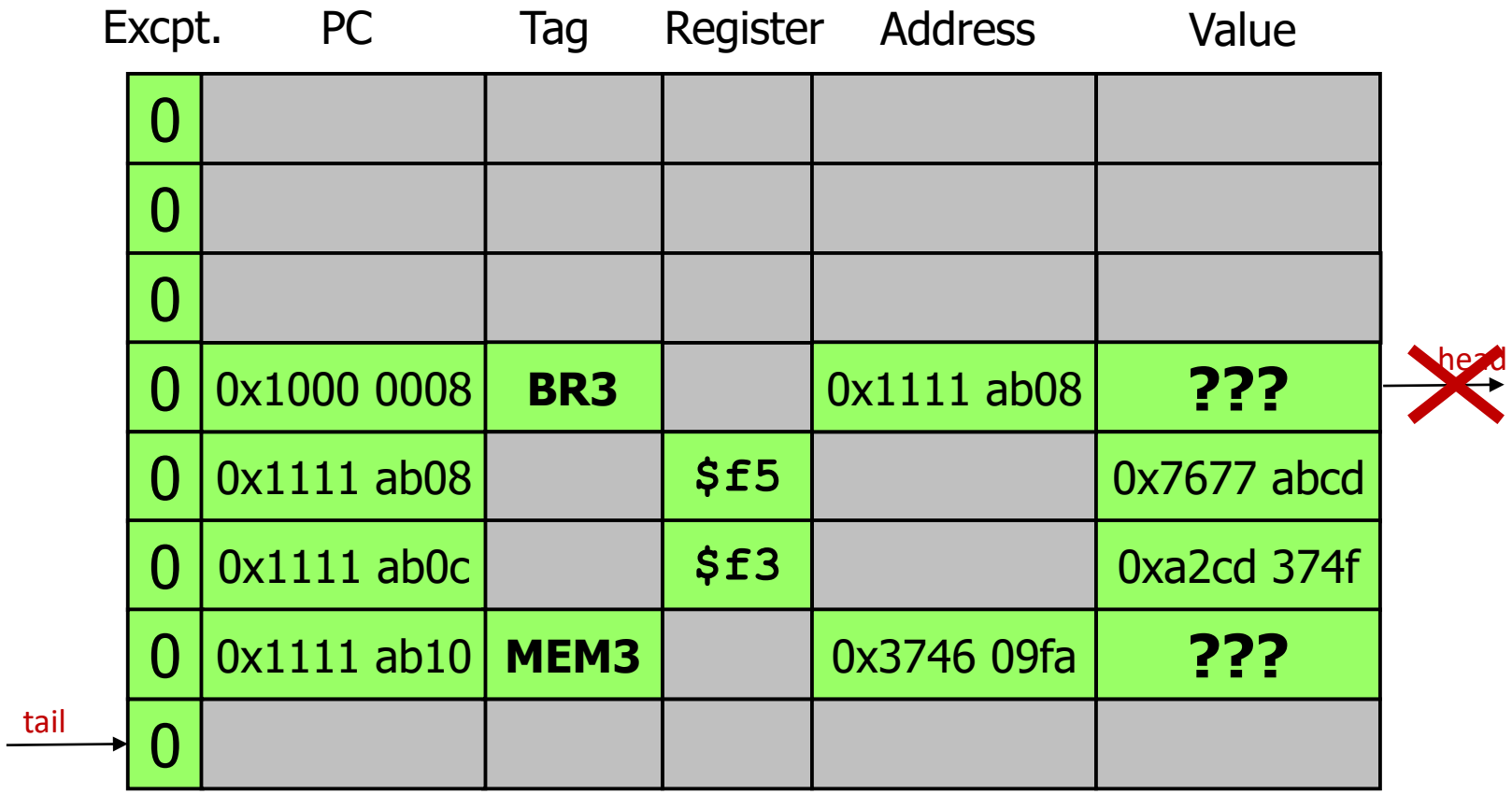
Dynamically Scheduled Superscalar Processor



Branches in the ROB

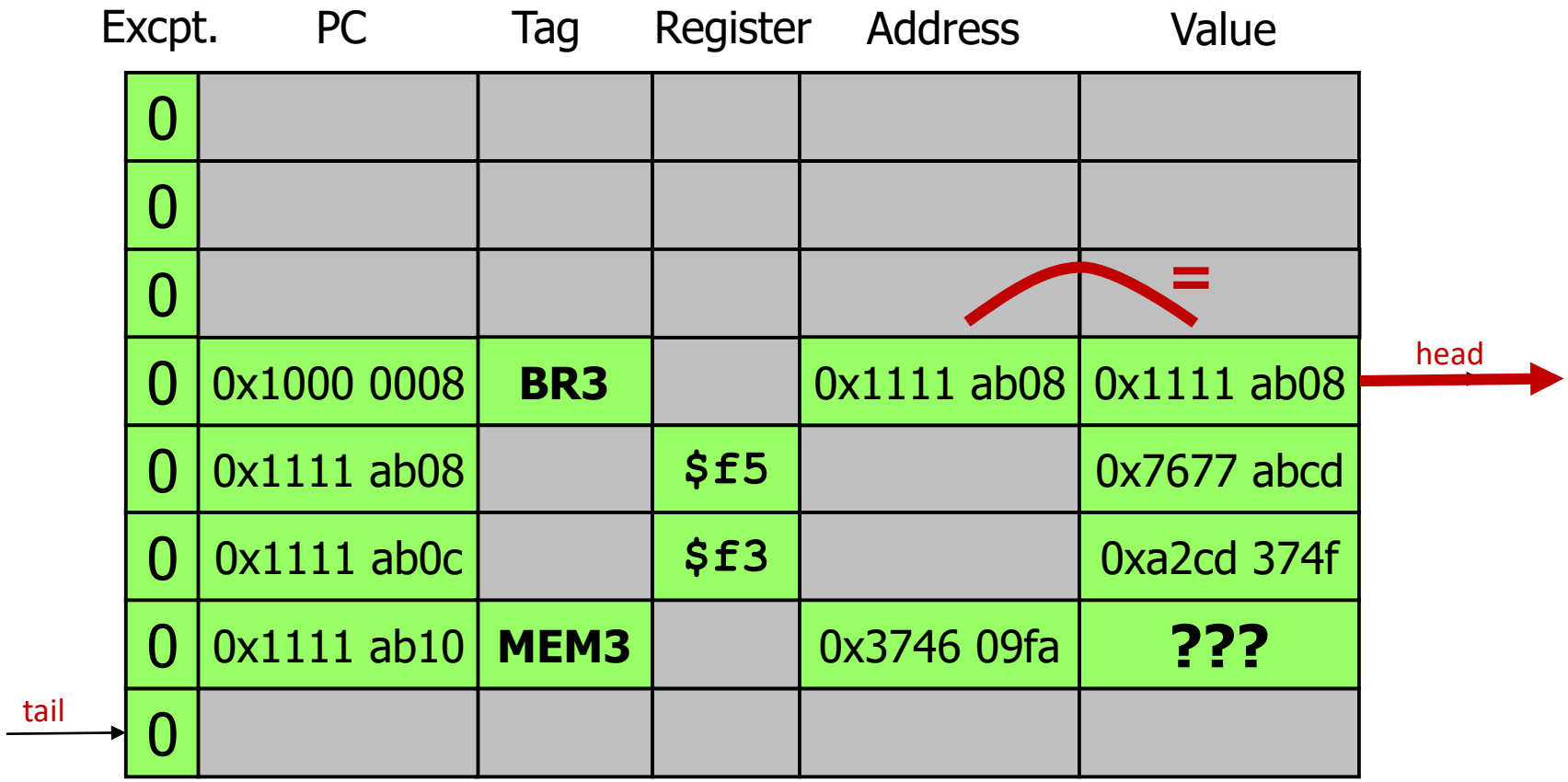


Branches without Outcome Block the ROB



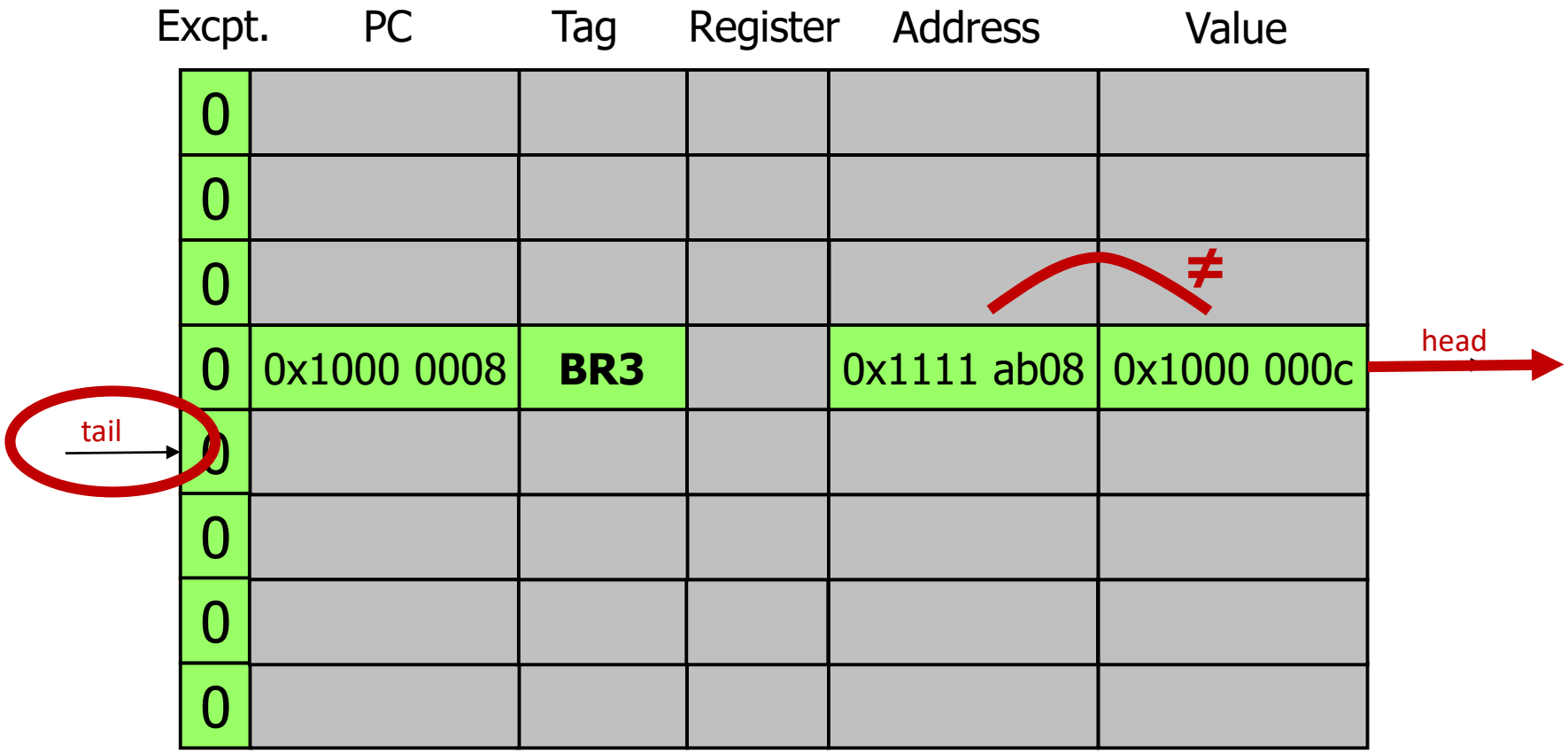
A predicted branch whose outcome is **unknown** cannot be committed

Correctly Predicted Branches Are Ignored



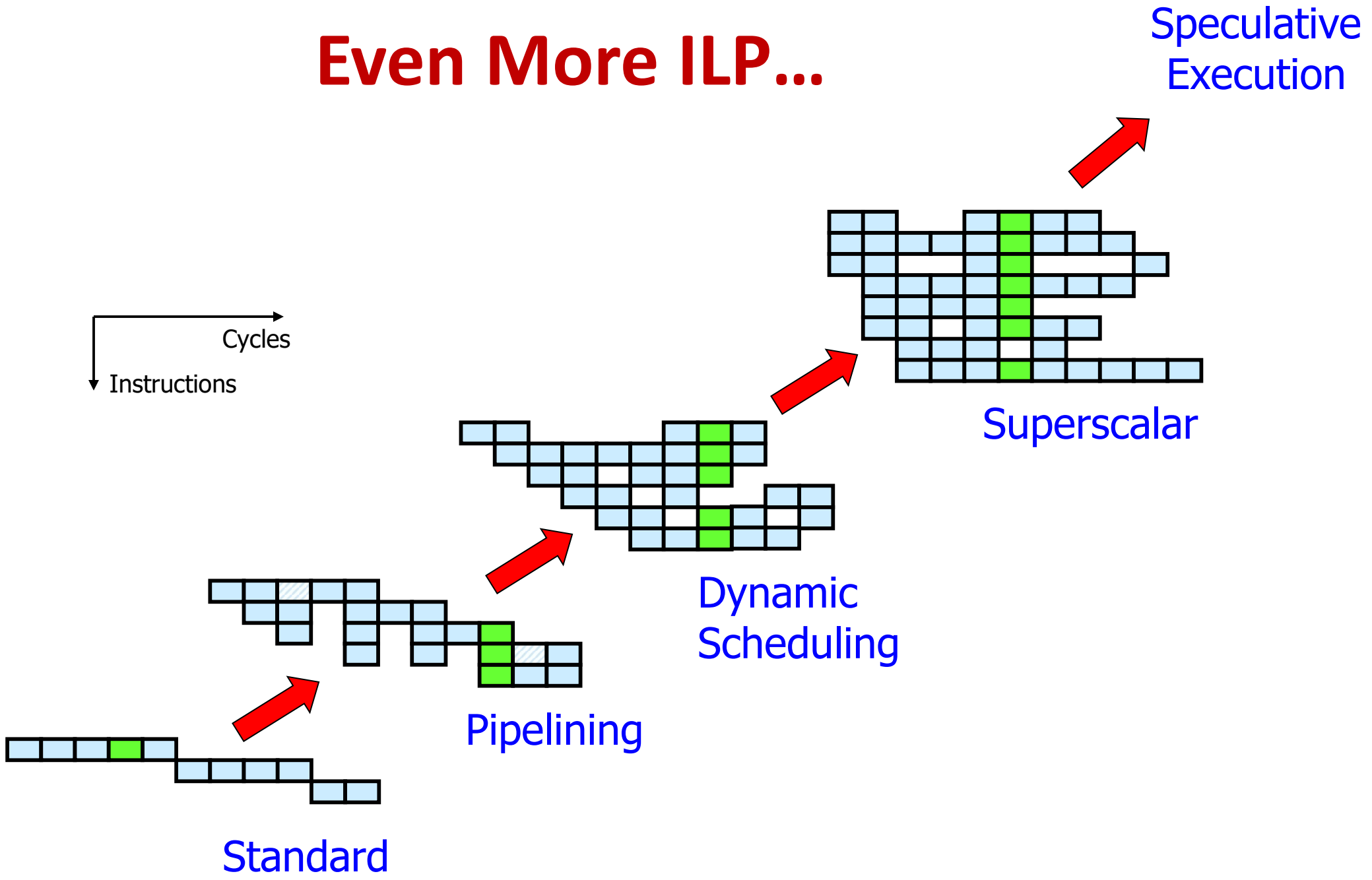
BR3 can commit (= do nothing and remove from ROB)

Mispredicted Branches Trigger a Squash



BR3 triggers a **squash** and causes **fetch** to restart at **`0x1000000c`**

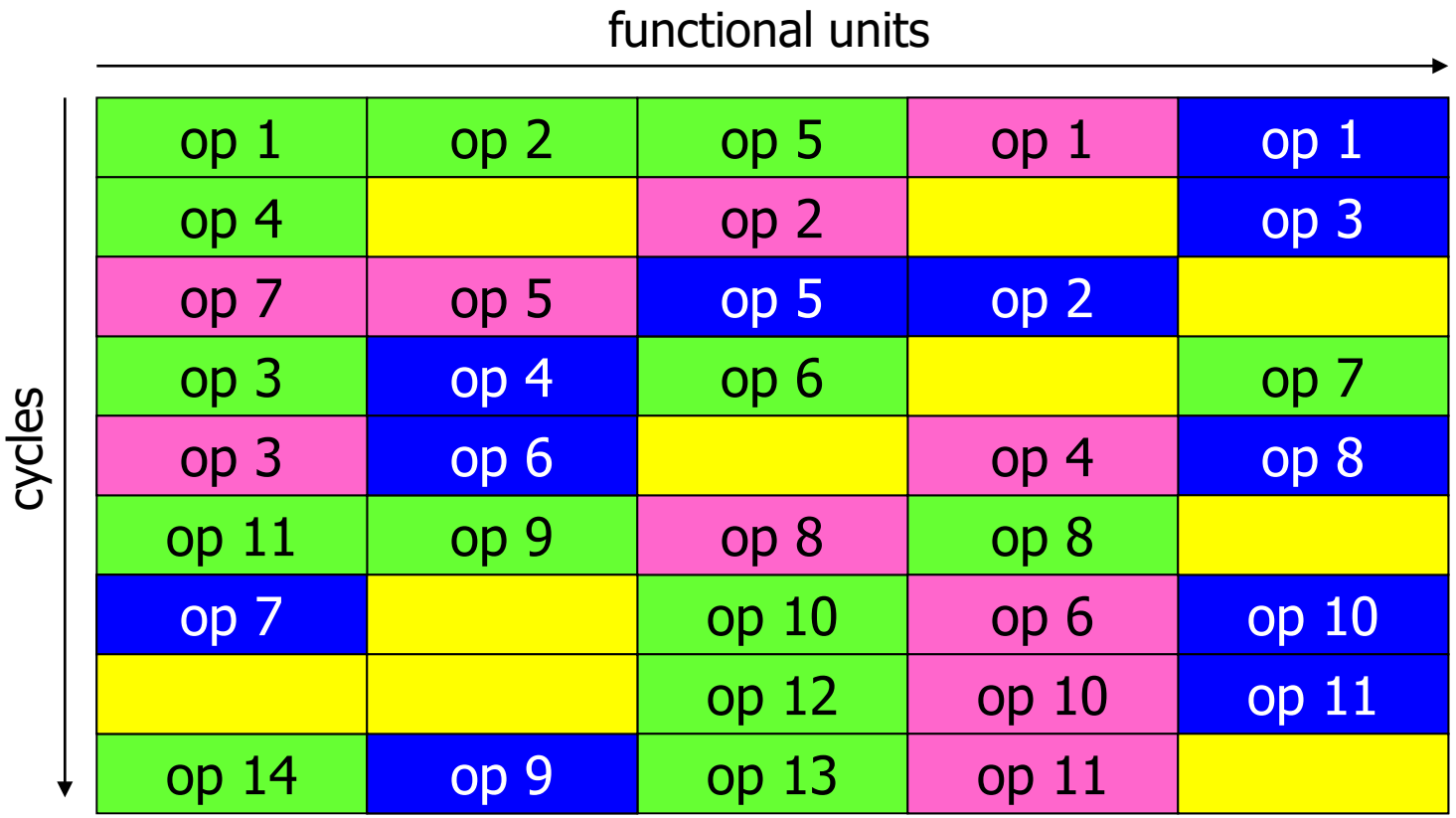
Even More ILP…



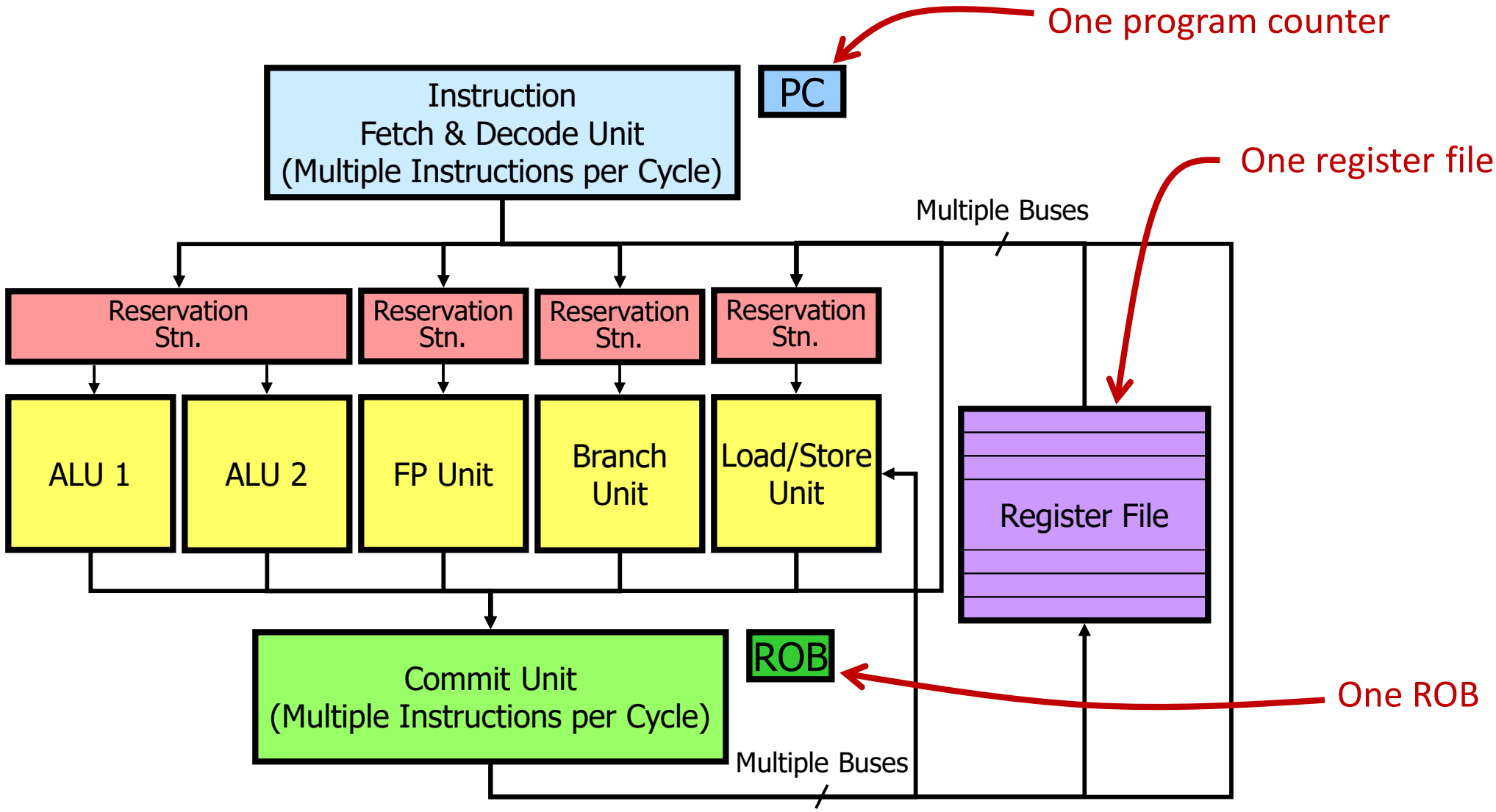
3

Simultaneous Multithreading

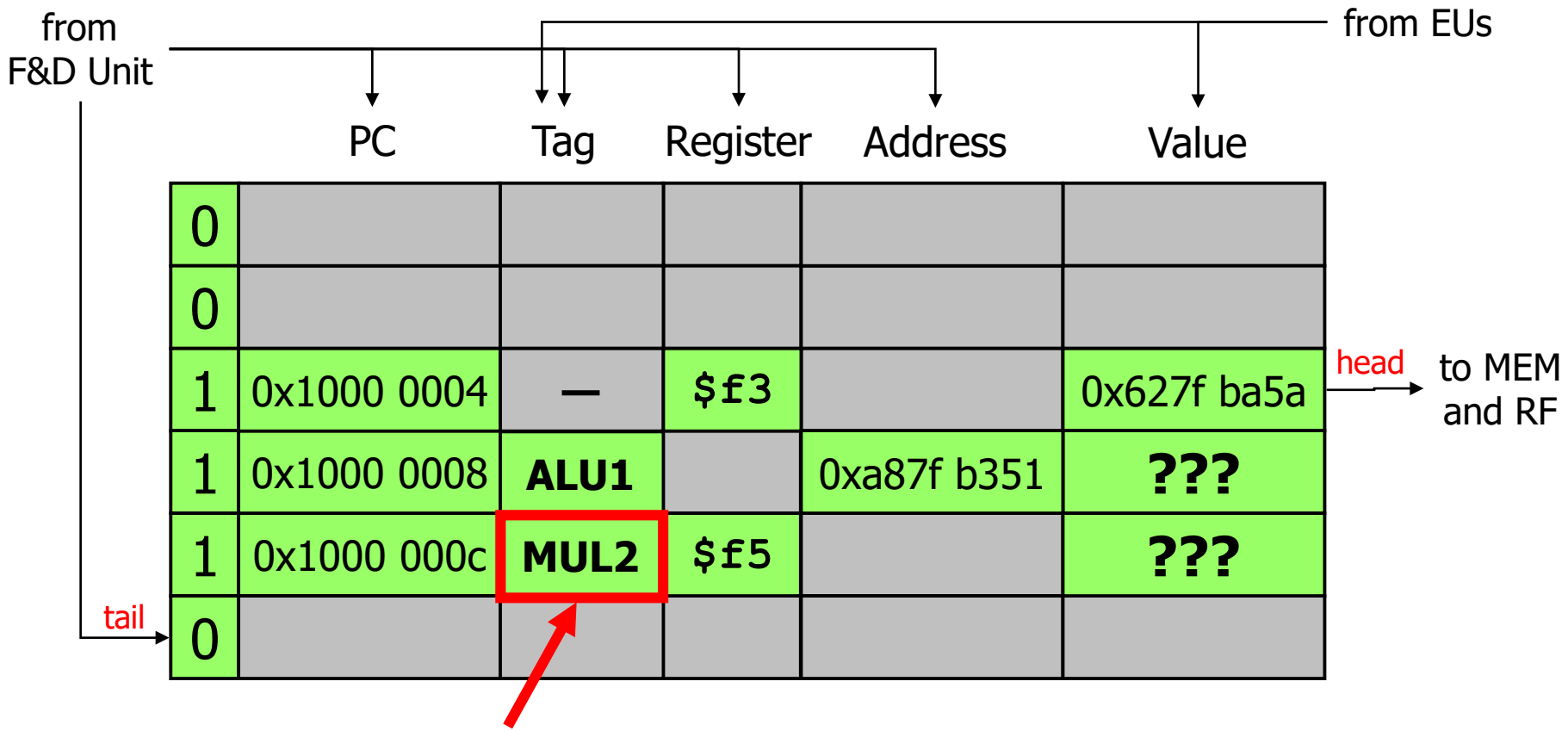
Simultaneous Multithreading (SMT): The Idea



Dynamically Scheduled Superscalar Processor



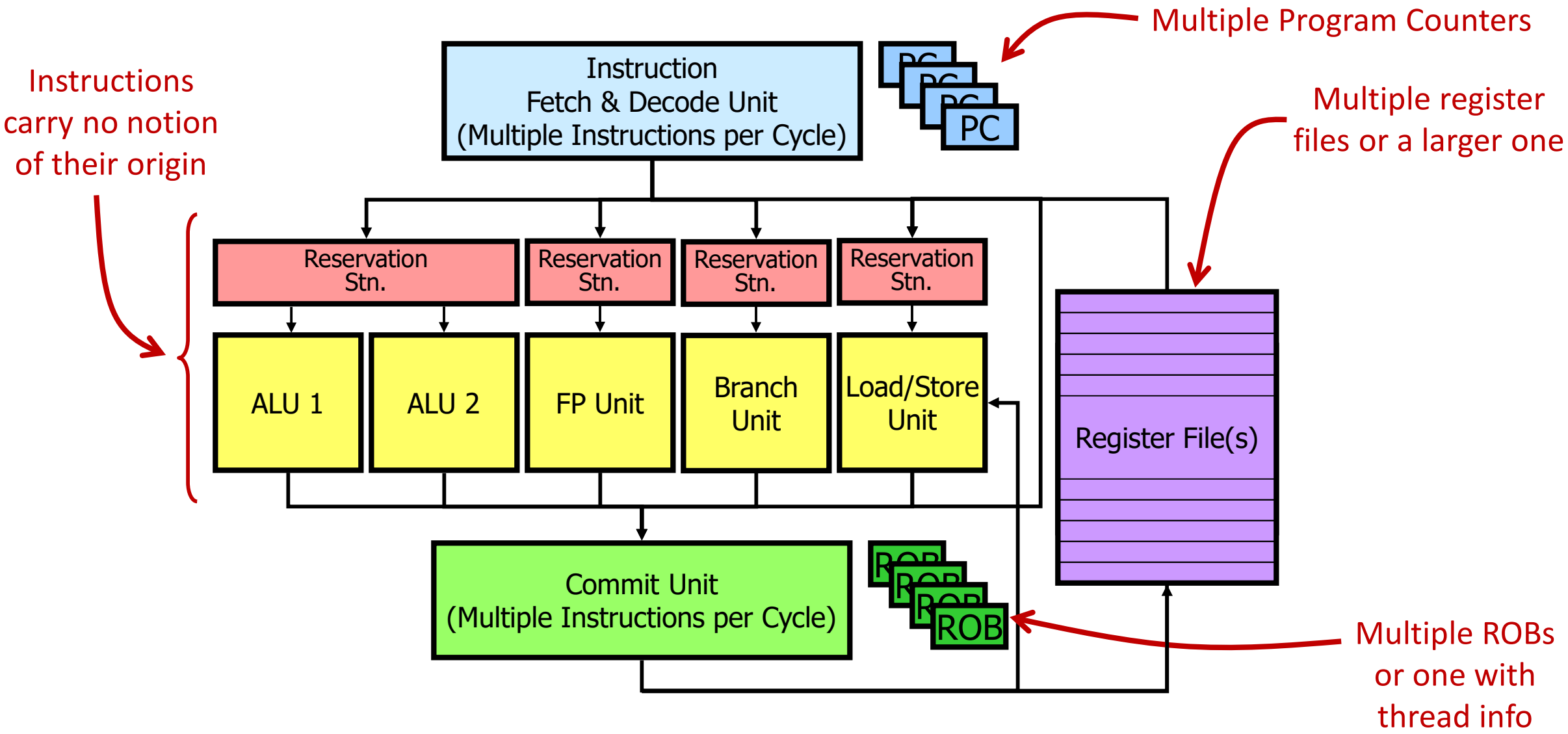
Reorder Buffer



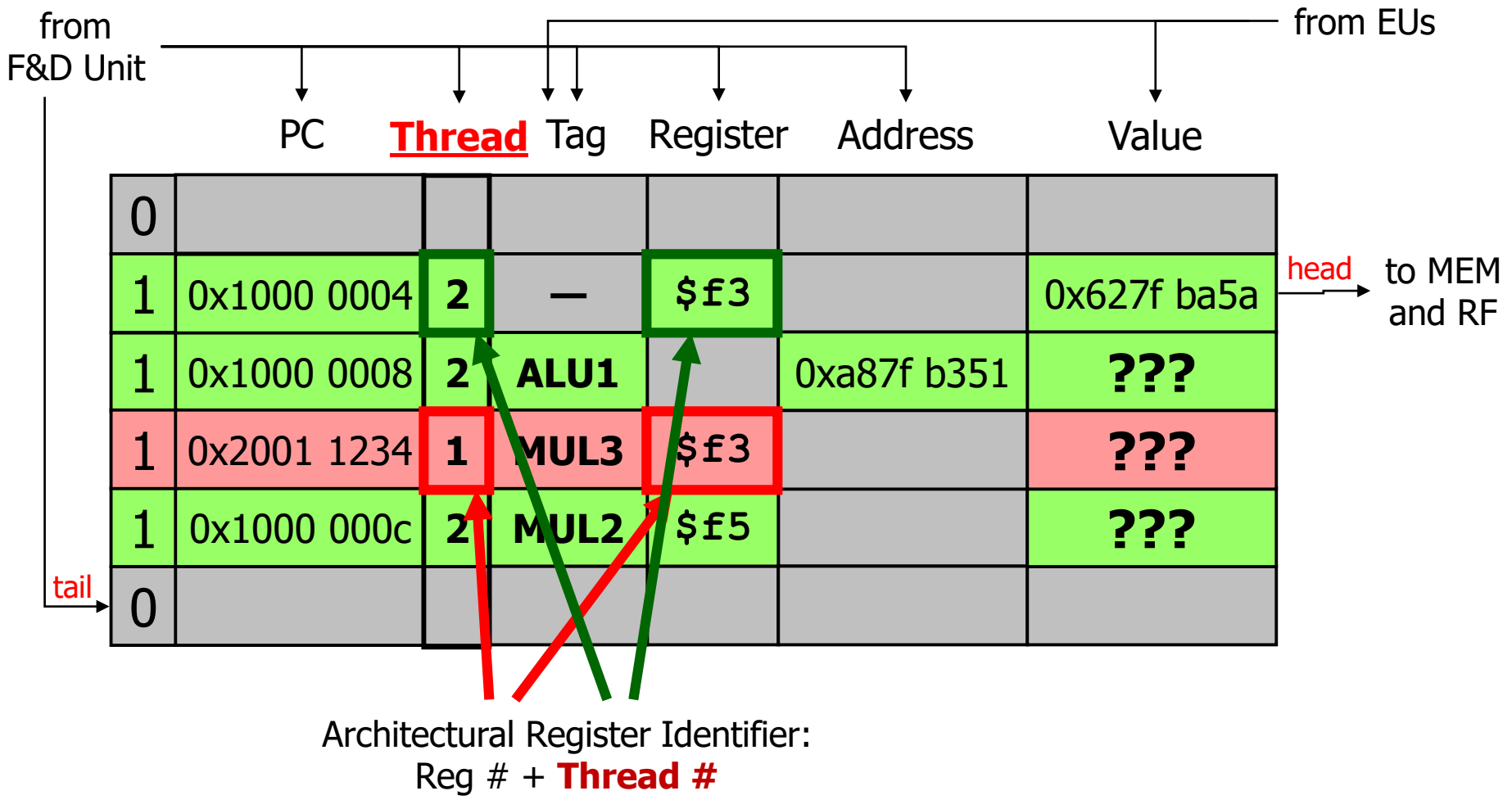
Register Renaming

(between fetch/decode and commit)

SMT Processor

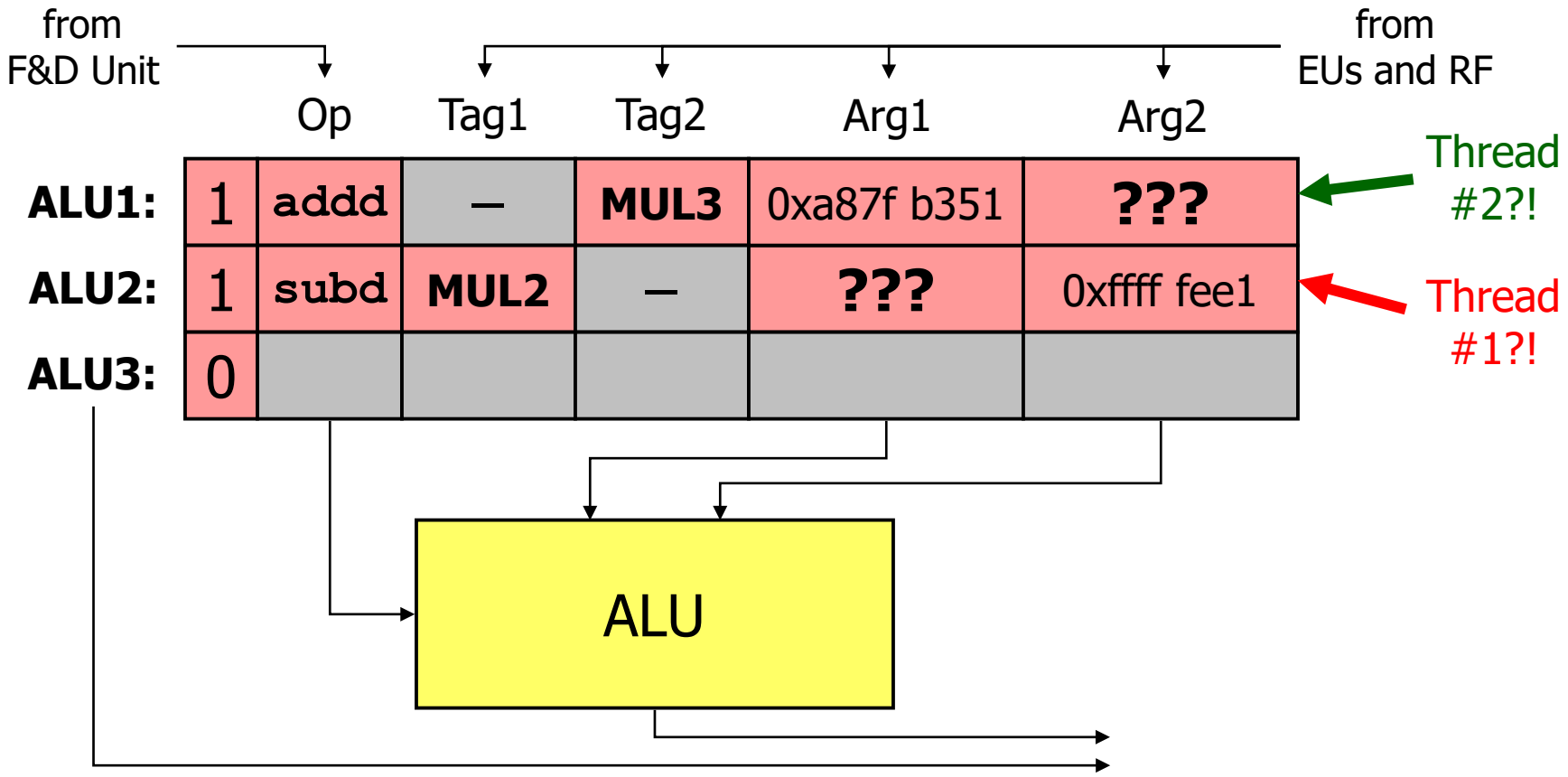


Reorder Buffer Remembers the Thread of Origin

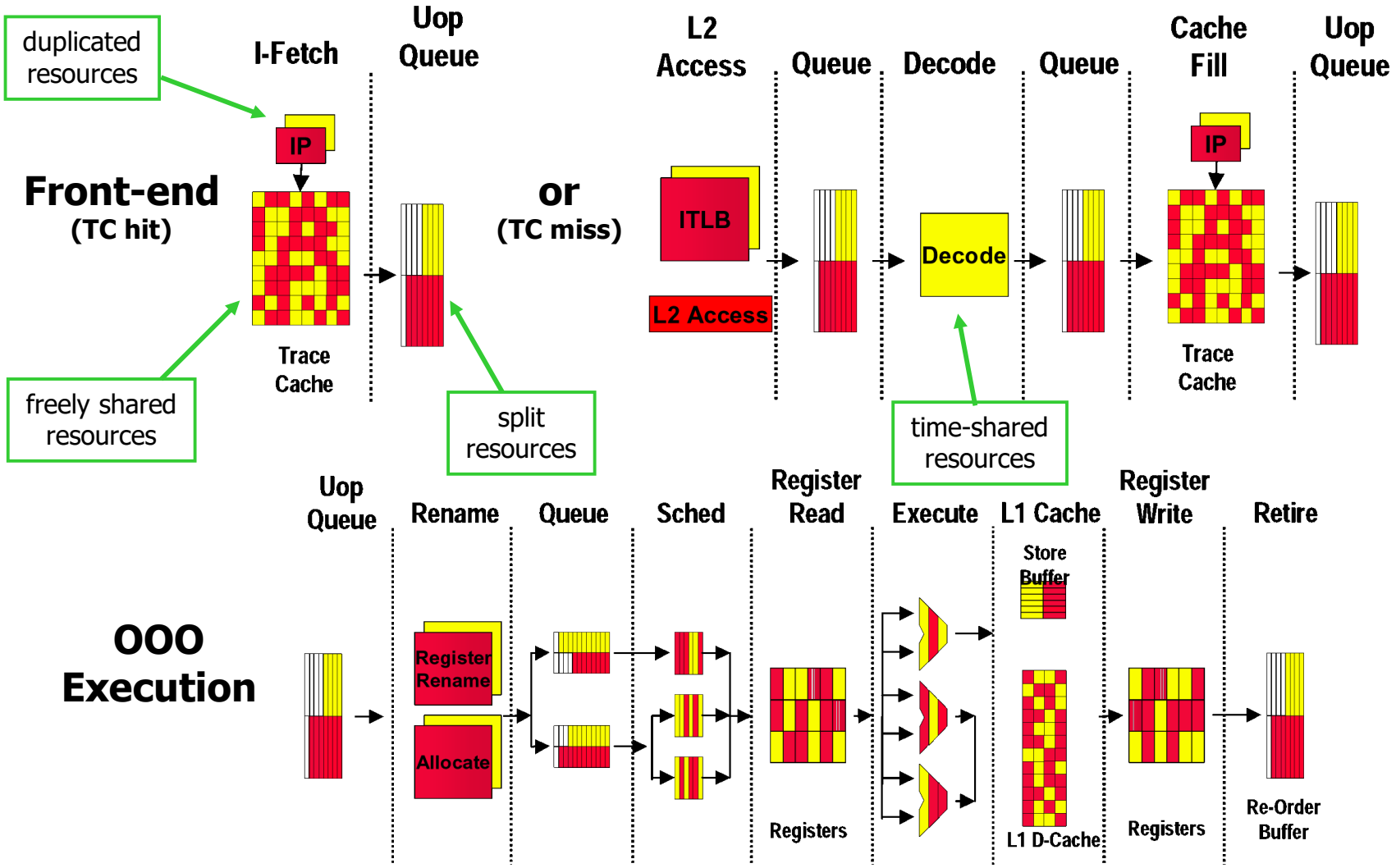


Reservation Stations

- Reservation stations **do not need to know** which thread an instruction belongs to



Intel SMT: Xeon Hyper-Threading Pipeline



Source: Marr et al., © Intel 2002

4

Nonblocking Caches

New Requirements for the Cache

- Consider the following code:

```
lw    $t2, 0($t0)        # t2 = mem[t0]
lw    $t3, 0($t1)        # t3 = mem[t1]
addi  $t3, $t3, 123
andi  $t3, $t3, 0xff
```

- If there is a cache miss for **`mem[t0]`**, one needs to **wait** for the (slow) main memory

- Of course, one wants the superscalar processor to **continue execution** as far as dependencies permit it

Nonblocking Caches

- The cache controller could serve a request, while waiting for the main memory, if the data are in the cache (**hit under miss**)
  - Hide the miss latency with useful work

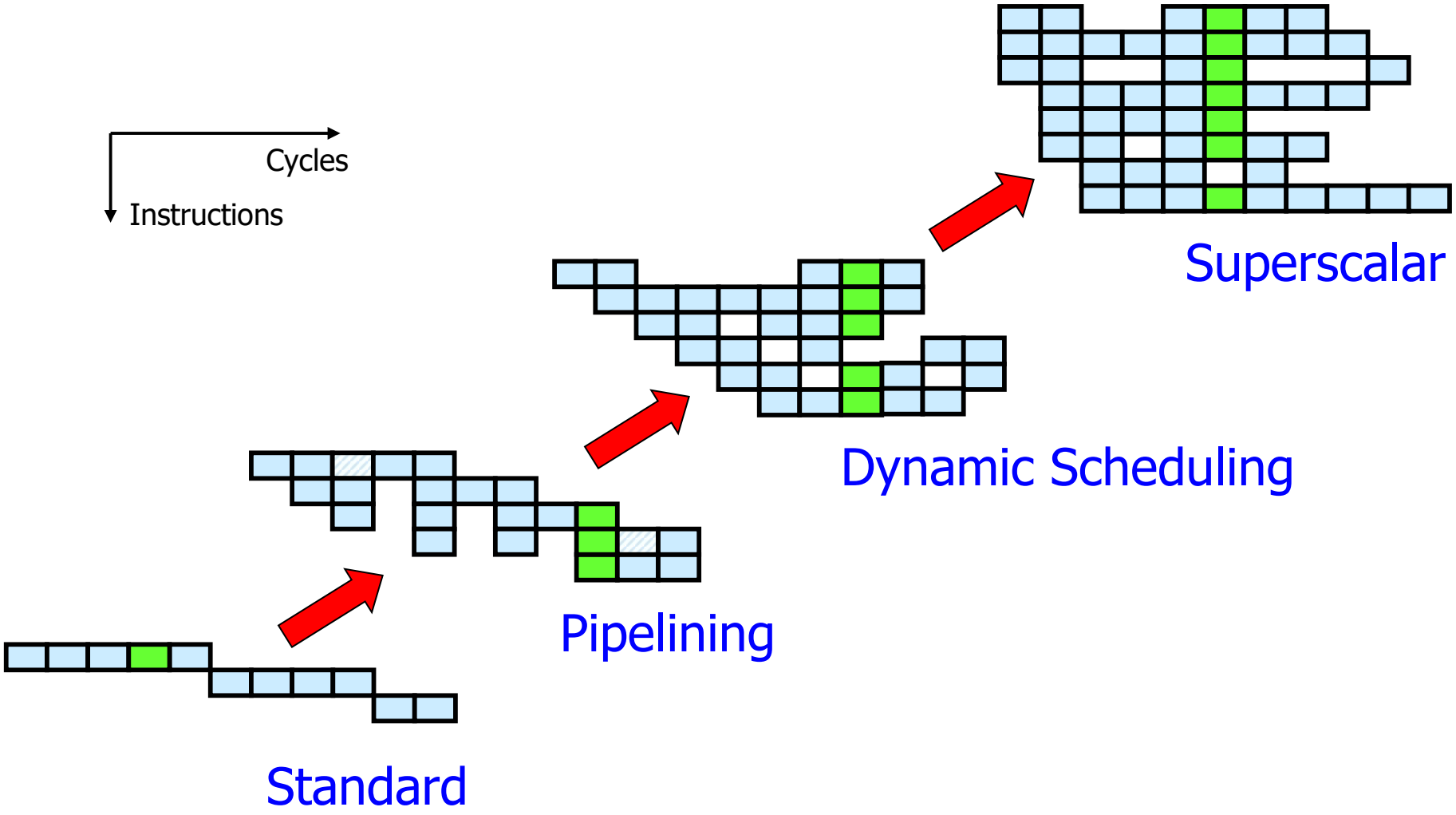
- The cache controller could serve a request, while waiting for the main memory, by issuing another request to memory (**miss under miss**)
  - Overlap the latency of the two misses

- **Nonblocking caches** are generally needed for dynamically scheduled superscalar processors

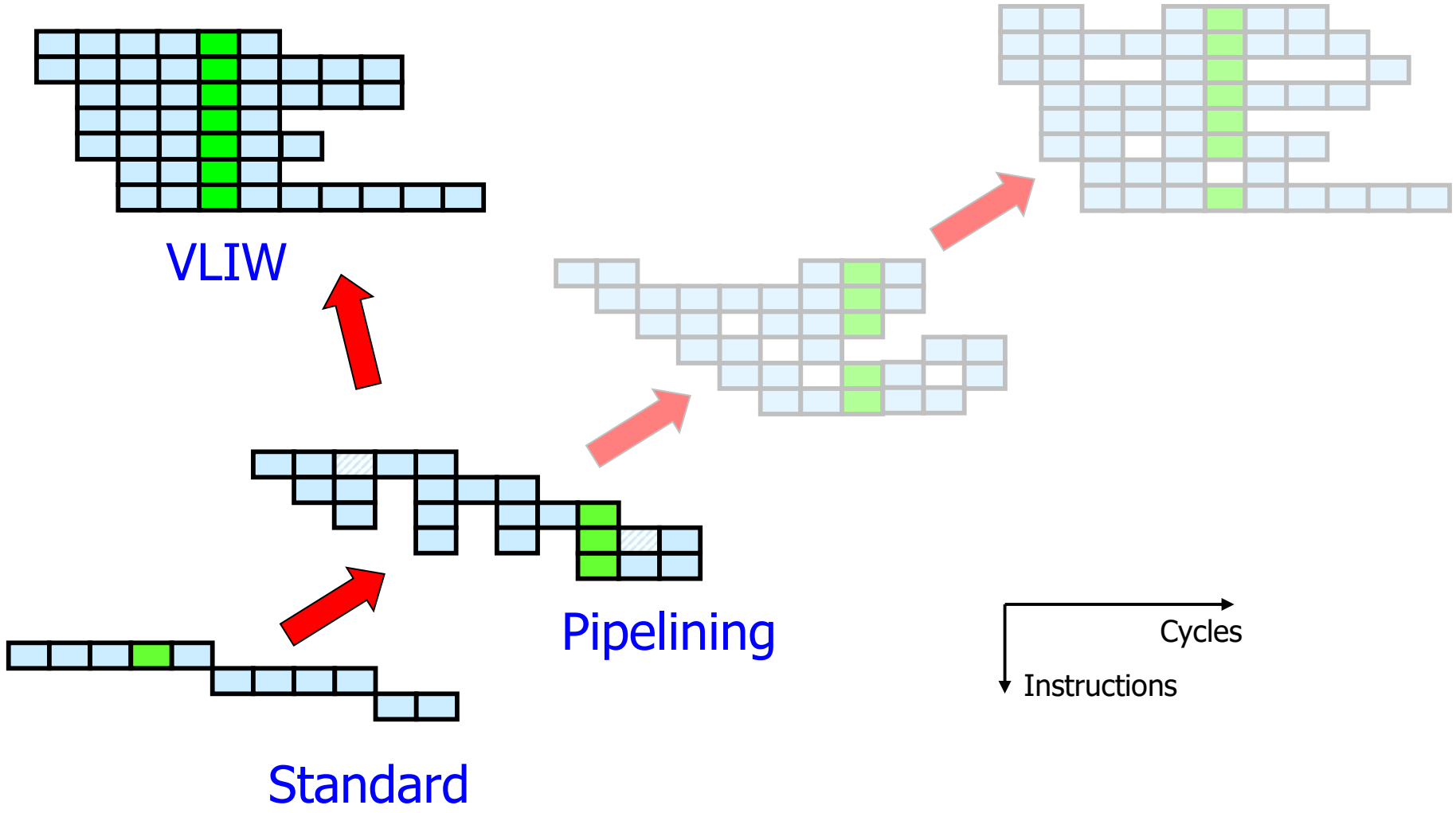
5

Very Long Instruction Word (VLIW) Processors

Last and Present Lesson Results: Several Steps in Exploiting ILP



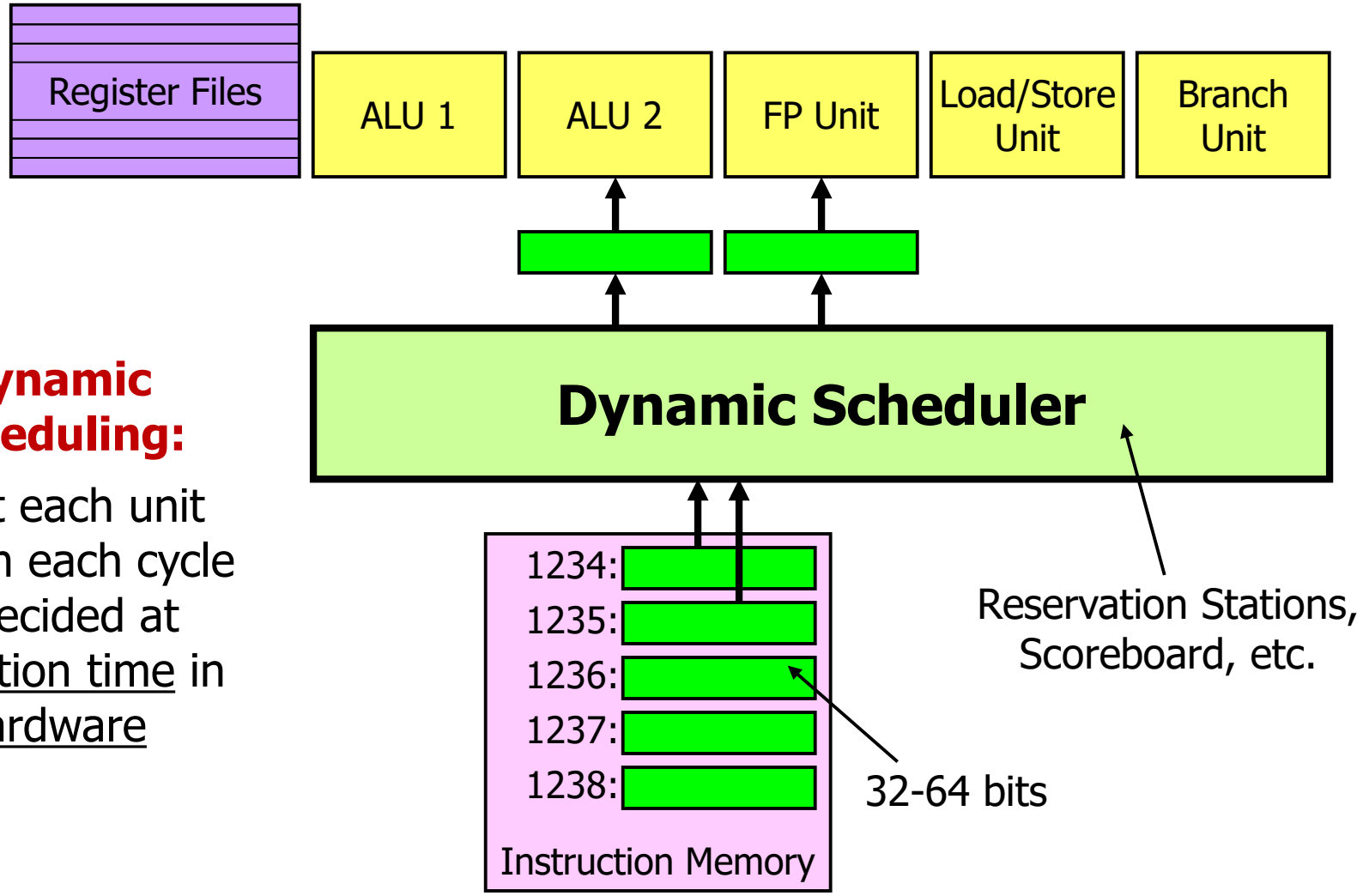
Very Long Instruction Word: An Alternate Way of Extracting ILP



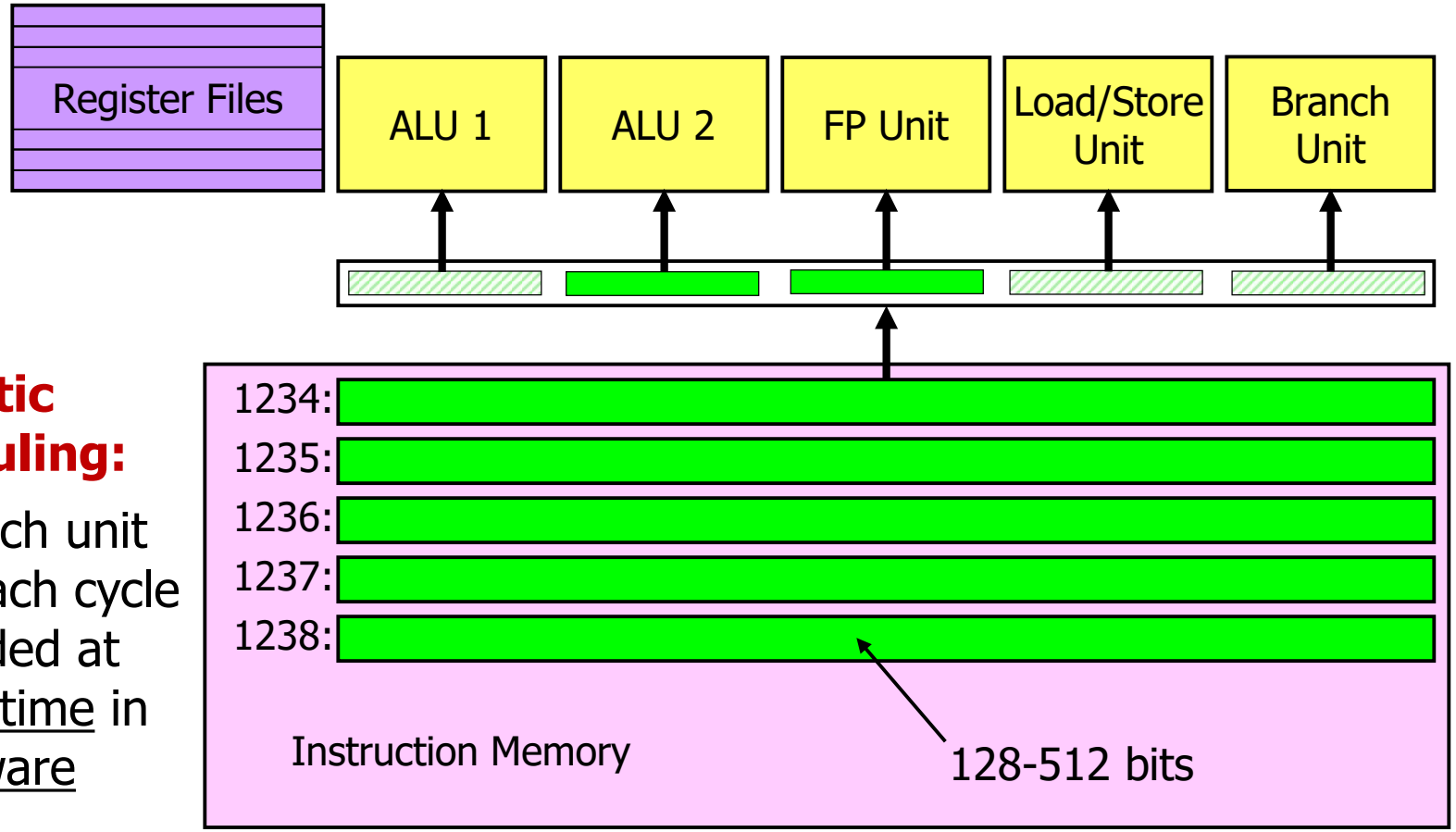
(Dynamically Scheduled) Superscalar Processor

Dynamic Scheduling:

What each unit does in each cycle is decided at execution time in hardware



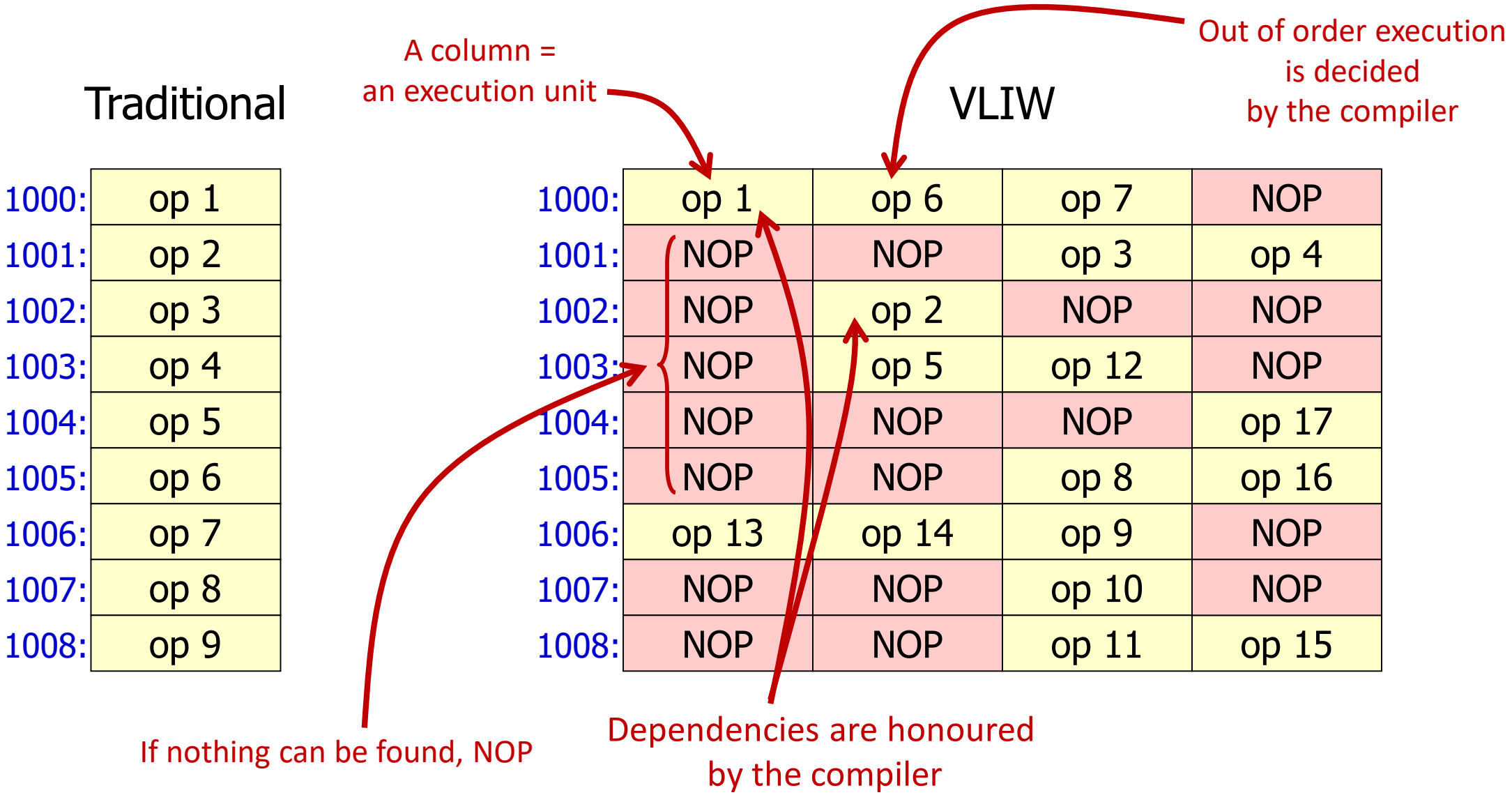
(Statically Scheduled)
Very Long Instruction Word Processor



Static Scheduling:

What each unit does in each cycle is decided at compile time in software

Traditional Code vs. VLIW Code



Challenges of VLIW

1. **Compiler Technology**
   - Most severe limitation until the end of the 90s (VLIW idea is around since the 70s!)

2. **Code Bloating**
   - All those NOPs occupy memory space and thus **cost**

3. **Binary Incompatibility**

What Kind of Information Is Missing at Compile Time?

- For example, consider:

```
sw   x3, 456(x1)
lw   x2, 123(x4)
```

- Is there a RAW dependence?
  - At run time:
    - Check if x1+456 = x4+123
    - Forwarding may even hide the memory latency…
  - At compile time:
    - ?!… (special techniques: *alias analysis*)

- Strong limitation for VLIW

Typical Code May Have Limited ILP

- Example:

```
Loop: ld     $f0, ($r1)        // read array elem.
      addd   $f4, $f0, $f2     // add constant
      sd     ($r1), $f4        // write array elem.

      subi   $r1, $r1, 8       // next element
      bnez   $r1, Loop
```

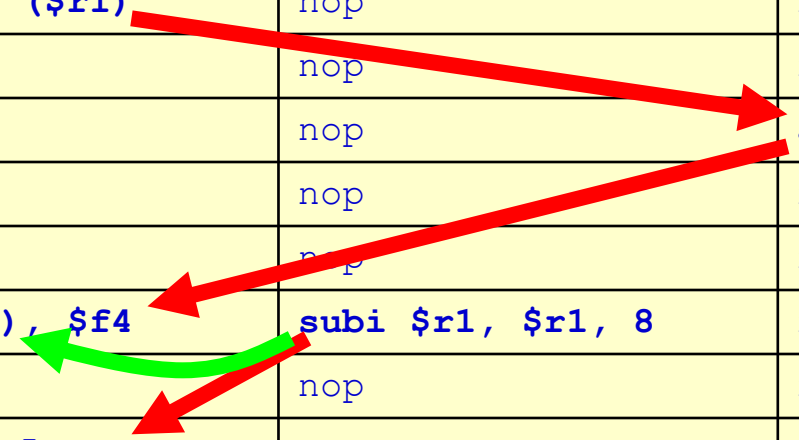
- Schedule on a VLIW processor
  - Slot 1: Load/Store Unit or Branch Unit
  - Slot 2: ALU
  - Slot 3: Floating-Point Unit
- Latencies:
  - Load/Store → 2 cycles
  - Integer → 2 cycles
  - Branch → 2 cycles
  - Floating Point → 3 cycles

Load/Store/Branch Unit	ALU	Floating-Point Unit	
			Cycle 1
			Cycle 2
			Cycle 3
			Cycle 4
			Cycle 5
			Cycle 6
			Cycle 7
			Cycle 8
			Cycle 9
			Cycle 10
			Cycle 11
			Cycle 12
			Cycle 13
			Cycle 14

Typical Code May Have Limited ILP

- Scheduled VLIW code:

Load/Store/Branch Unit	ALU	Floating-Point Unit	
`ld $f0, ($r1)`	nop	nop	Cycle 1
nop	nop	nop	Cycle 2
nop	nop	`addd $f4, $f0, $f2`	Cycle 3
nop	nop	nop	Cycle 4
nop	nop	nop	Cycle 5
`sd ($r1), $f4`	`subi $r1, $r1, 8`	nop	Cycle 6
nop	nop	nop	Cycle 7
`bnez $r1, Loop`	nop	nop	Cycle 8
nop	nop	nop	Cycle 9

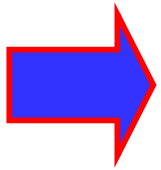


- Execution time for $r1 = 80:
  - 80 / 8 = 10 iterations; 9 cycles per iteration → **90 cycles**

Enlarge the Scope for ILP: Loop Unrolling

```
Loop: ld     $f0, ($r1)
      addd   $f4, $f0, $f2
      sd     ($r1), $f4

      subi   $r1, $r1, 8
      bnez   $r1, Loop
```



```
Loop: ld     $f0, ($r1)
      addd   $f4, $f0, $f2
      sd     ($r1), $f4

      ld     $f6, ($r1-8)
      addd   $f8, $f6, $f2
      sd     ($r1-8), $f8

      ld     $f10, ($r1-16)
      addd   $f12, $f10, $f2
      sd     ($r1-16), $f12

      ld     $f14, ($r1-24)
      addd   $f16, $f14, $f2
      sd     ($r1-24), $f16

      ld     $f18, ($r1-32)
      addd   $f20, $f18, $f2
      sd     ($r1-32), $f20

      subi   $r1, $r1, 40
      bnez   $r1, Loop
```

- Replicate body
- Update references
- Rename registers
- etc.

Loop Unrolling

Load/Store/Branch Unit	ALU	Floating-Point Unit	
`ld $f0, ($r1)`	nop	nop	Cycle 1
`ld $f6, ($r1-8)`	nop	nop	Cycle 2
`ld $f10, ($r1-16)`	nop	`addd $f4, $f0, $f2`	Cycle 3
`ld $f14, ($r1-24)`	nop	`addd $f8, $f6, $f2`	Cycle 4
`ld $f18, ($r1-32)`	nop	`addd $f12, $f10, $f2`	Cycle 5
`sd ($r1), $f4`	nop	`addd $f16, $f14, $f2`	Cycle 6
`sd ($r1-8), $f8`	nop	`addd $f20, $f18, $f2`	Cycle 7
`sd ($r1-16), $f12`	nop	nop	Cycle 8
`sd ($r1-24), $f16`	nop	nop	Cycle 9
`sd ($r1-32), $f20`	`subi $r1, $r1, 40`	nop	Cycle 10
nop	nop	nop	Cycle 11
`bnez $r1, Loop`	nop	nop	Cycle 12
nop	nop	nop	Cycle 13

- Now 80 / (5*8) = 2 iterations; 13 cycles per iteration → **26 cycles** (vs. 90 cycles, more than 3x faster!)

VLIW Compilation Techniques

- Many old and new techniques:
  - Aliasing analysis
  - Loop unrolling, peeling, fusion, and distribution
  - Software pipelining, modulo scheduling
  - Trace scheduling, superblock scheduling
  - With hardware support in the processor: Predication, hyperblock scheduling,…

- Usually advantage **not for free**:
  - Faster only on most frequent part of the code; penalty elsewhere
  - Difficulties to apply them in the general case
  - Larger code (worsens the performance of the I-cache)

Challenges of VLIW

1. **Compiler Technology**
   - Most severe limitation until the end of the 90s (VLIW idea is around since the 70s!)

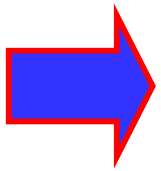
2. **Code Bloating**
   - All those NOPs occupy memory space and thus **cost**

3. **Binary Incompatibility**

VLIW Code Bloating

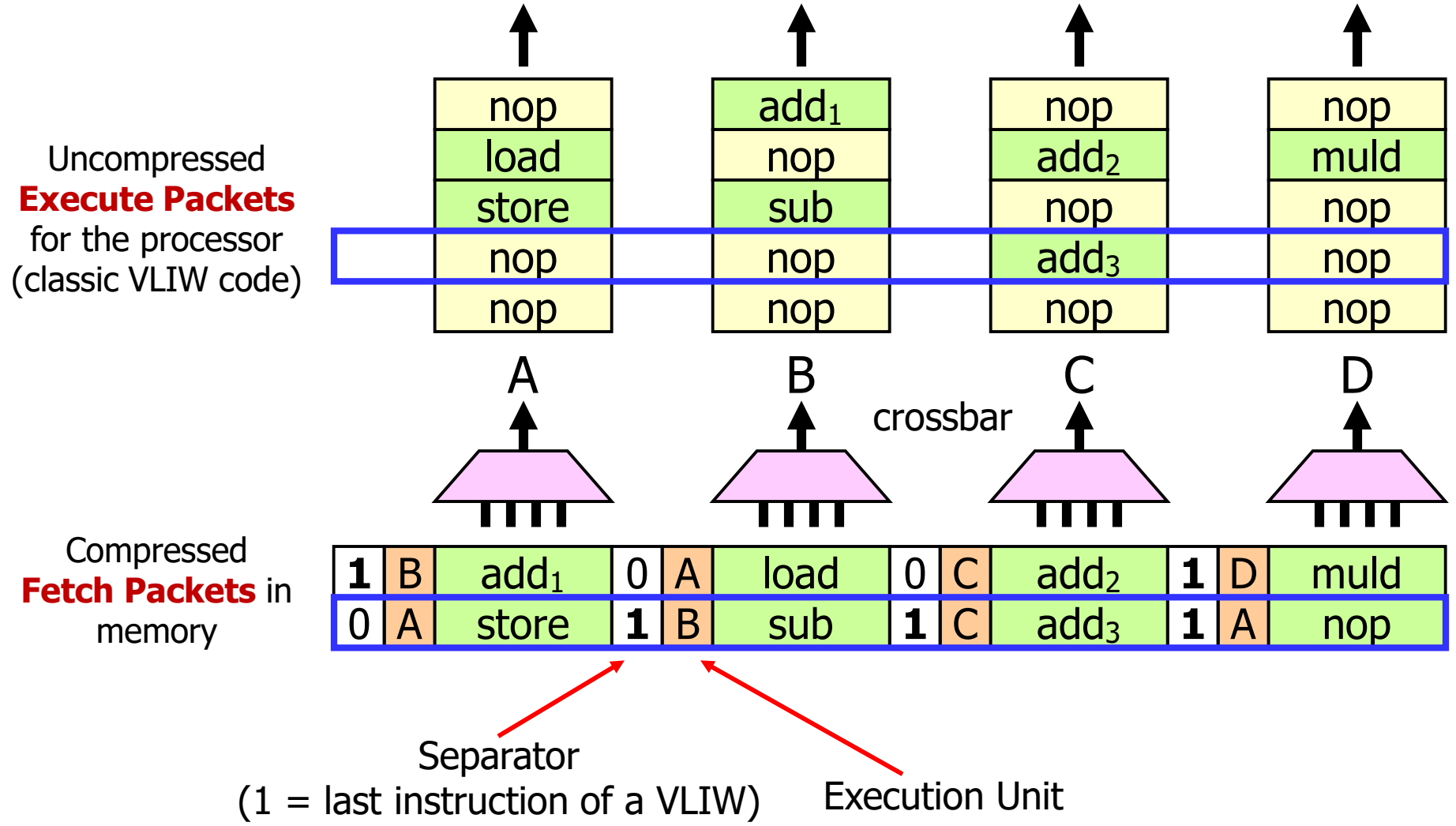
- VLIW code is often much larger than standard code: NOPs are explicit, aggressive unrolling, etc.
- Compare last example: **39** words vs. **5**! more than **50% are NOPs**!

```
ld   $f0, ($r1)
addd $f4, $f0, $f2
sd   ($r1), $f4
subi $r1, $r1, 8
bnez $r1, Loop
```



ld $f0, ($r1)	nop	nop
ld $f6, ($r1-8)	nop	nop
ld $f10, ($r1-16)	nop	addd $f4, $f0, $f2
ld $f14, ($r1-24)	nop	addd $f8, $f6, $f2
ld $f18, ($r1-32)	nop	addd $f12, $f10, $f2
sd ($r1), $f4	nop	addd $f16, $f14, $f2
sd ($r1-8), $f8	nop	addd $f20, $f18, $f2
sd ($r1-16), $f12	nop	nop
sd ($r1-24), $f16	nop	nop
sd ($r1-32), $f20	subi $r1, $r1, 40	nop
nop	nop	nop
bnez $r1, Loop	nop	nop
nop	nop	nop

Code Compression: Differentiate Fetch Packet and Execute Packet

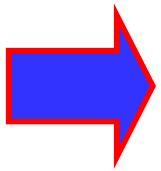


VLIW Code Bloating

- VLIW code is often much larger than standard code: NOPs are explicit, aggressive unrolling, etc.
- Compare last example: **39** words vs. **5**! more than **50% are NOPs**!

Not only a NOP problem, there are also **more** "real" instructions!

```
ld   $f0, ($r1)
addd $f4, $f0, $f2
sd   ($r1), $f4
subi $r1, $r1, 8
bnez $r1, Loop
```



ld $f0, ($r1)	nop	nop
ld $f6, ($r1-8)	nop	nop
ld $f10, ($r1-16)	nop	addd $f4, $f0, $f2
ld $f14, ($r1-24)	nop	addd $f8, $f6, $f2
ld $f18, ($r1-32)	nop	addd $f12, $f10, $f2
sd ($r1), $f4	nop	addd $f16, $f14, $f2
sd ($r1-8), $f8	nop	addd $f20, $f18, $f2
sd ($r1-16), $f12	nop	nop
sd ($r1-24), $f16	nop	nop
sd ($r1-32), $f20	subi $r1, $r1, 40	nop
nop	nop	nop
bnez $r1, Loop	nop	nop
nop	nop	nop

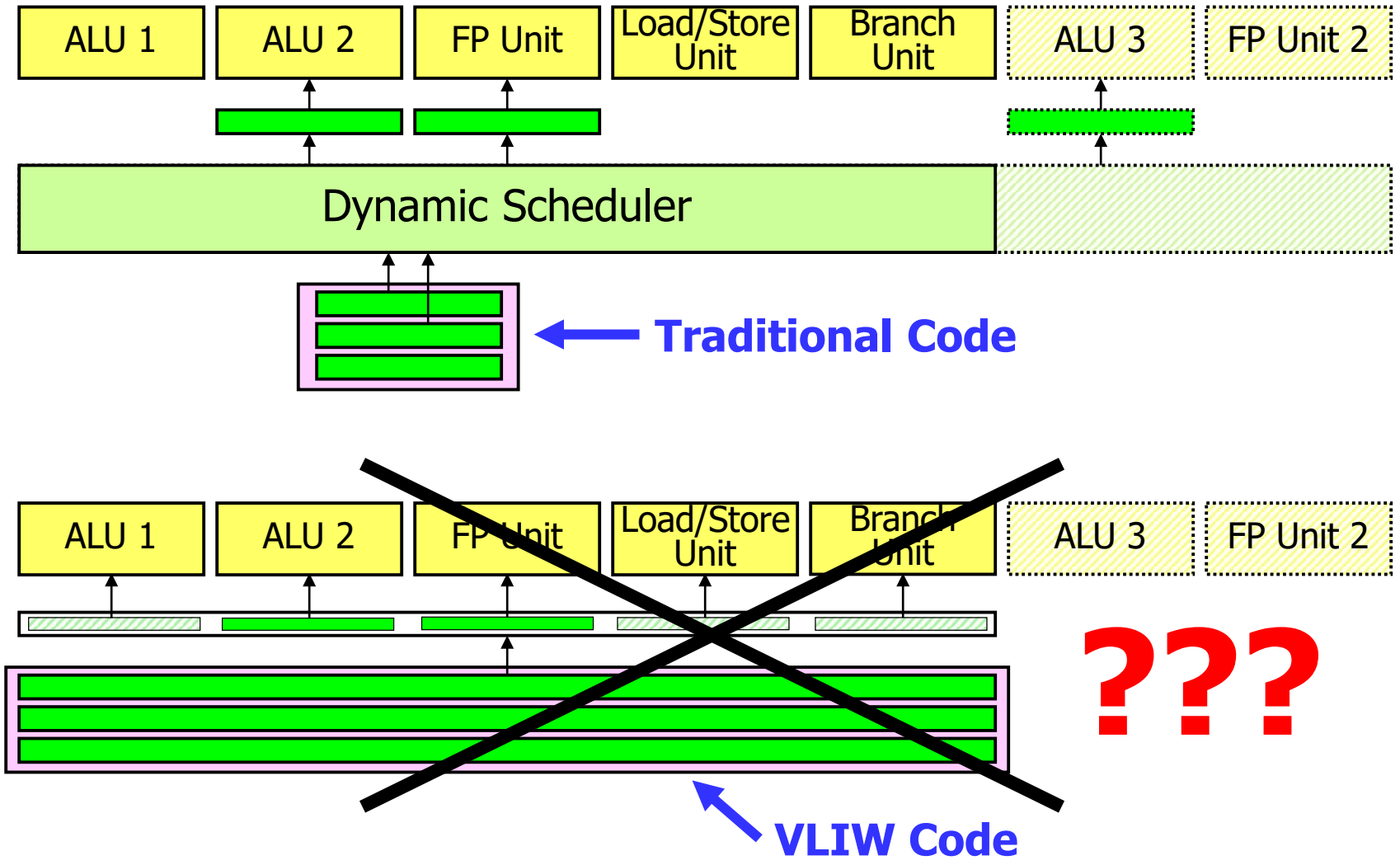
Challenges of VLIW

1. **Compiler Technology**
   - Most severe limitation until the end of the 90s (VLIW idea is around since the 70s!)

2. **Code Bloating**
   - All those NOPs occupy memory space and thus **cost**

3. **Binary Incompatibility**

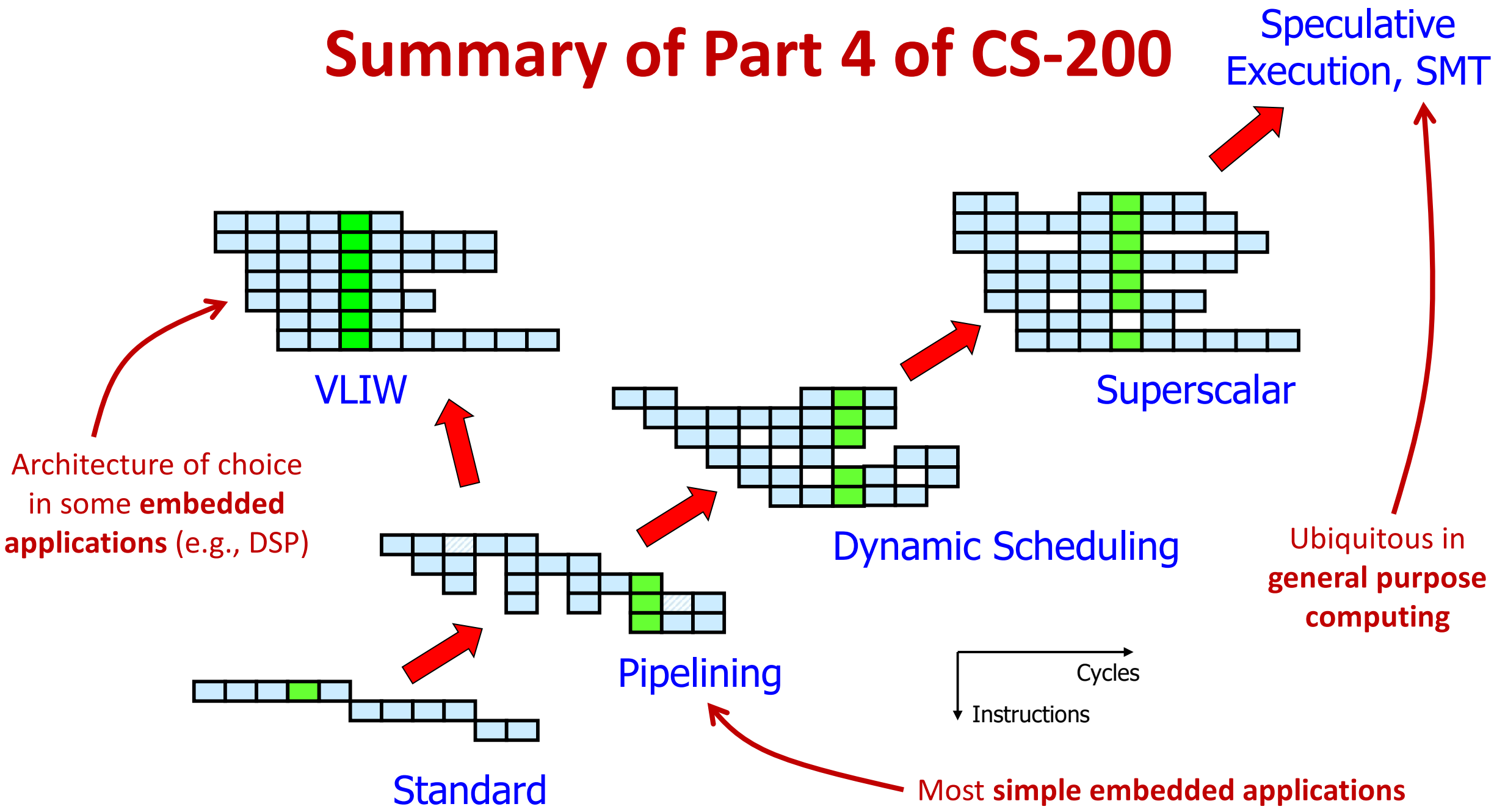
VLIW Binary Is Incompatible with More Aggressive Implementations



VLIW Binary Incompatibility

- More subtle sources of incompatibility
  - Changes in instruction latencies—e.g., load latencies increases *(logic-memory gap)*
- No fully satisfactory solution exists today
- Partial or research solutions:
  - Recompile (possible in some kind of systems—not for consumer market…)
  - Special VLIW coding/restrictions
  - Dynamic Binary Translation

Summary of Part 4 of CS-200



Summary of Part 4 of CS-200

- Dynamically-scheduled superscalar processors are the commercial state-of-the-art in general purpose computing (laptops, data centres): current high-end implementations of **x86 (Intel and AMD)** as well as **ARM** are all superscalar

- VLIW/EPIC processors represent an alternative, valuable in some situations: **Itanium 2** was a failed attempt by Intel to bring VLIWs in general-purpose computing; yet, practically all digital signal processors are VLIW (e.g., in all smartphones or for audio processing)

- Performance is the result of a **subtle balance** between exploiting possibilities in compilers and managing hardware implementation difficulties

References

- Patterson & Hennessy, COD – RISC-V Edition
  - **Section 4.11**
  - **Section 5.13** (Nonblocking Caches)
  - **Section 6.4** (SMT)