CRASH COURSE ON COMPUTER ARCHITECTURE

Areg Melik-Adamyan, PhD

Engineering Manager, Intel Developer Products Division

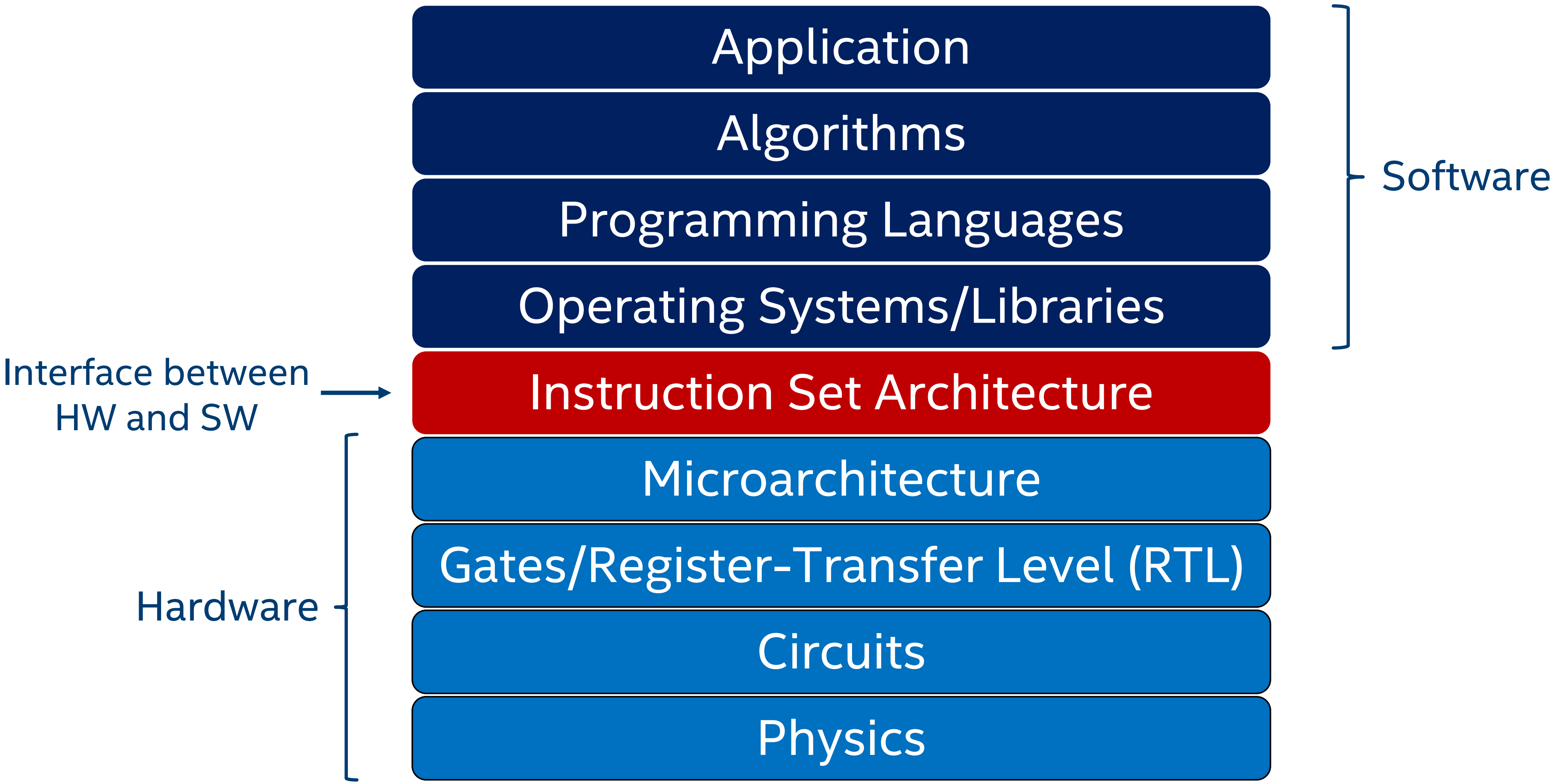
Topics covered in lecture

- Instruction level parallelism (ILP)
  - Pipelined execution
  - Superscalar
  - Out-of-order execution
  - Limit on ILP: data and control dependences between instructions
- Out-of-order execution implementation
  - Out of order execution, in-order completion (commit)
  - RAW dependences: forwarding
  - WAR and WAW dependences: register renaming
  - Control dependences: branch prediction and speculative execution

Texbooks and References

- Try to hit the tip of the iceberg
- Explain main concepts only
- Not enough to develop your own microprocessor…
- But allow better understand behavior and performance of your program
- Hennesy, Patterson, Computer Architecture: Quantative Approach, 6th Ed.
- Blaauw, Brooks, Computer Architecture: Concepts and Evolution

Layers of Abstraction



Application

Algorithms

Programming Languages

Operating Systems/Libraries

Software

Interface between HW and SW

Instruction Set Architecture

Microarchitecture

Gates/Register-Transfer Level (RTL)

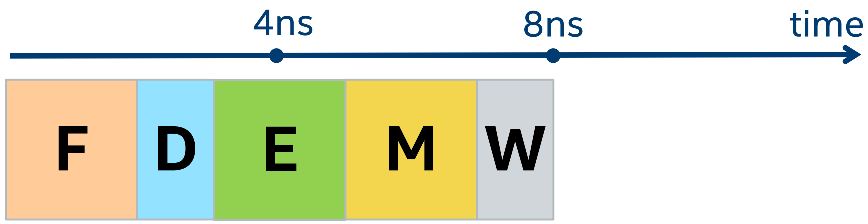
Circuits

Physics

Hardware

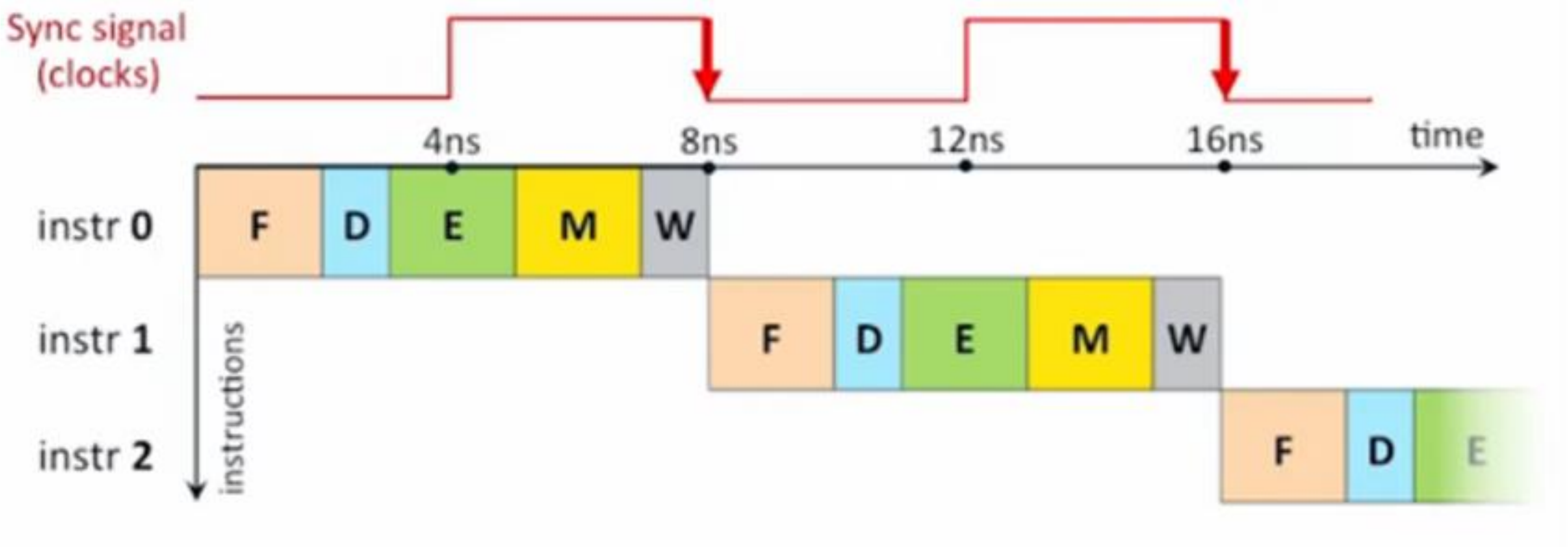
Optimization Notice

Basic CPU Actions



1. Fetch instruction by PC from memory
2. Decode it and read its operands from registers
3. Execute calculations
4. Read/write memory
5. Write the result into registers and update PC

Non-Pipelined Processing

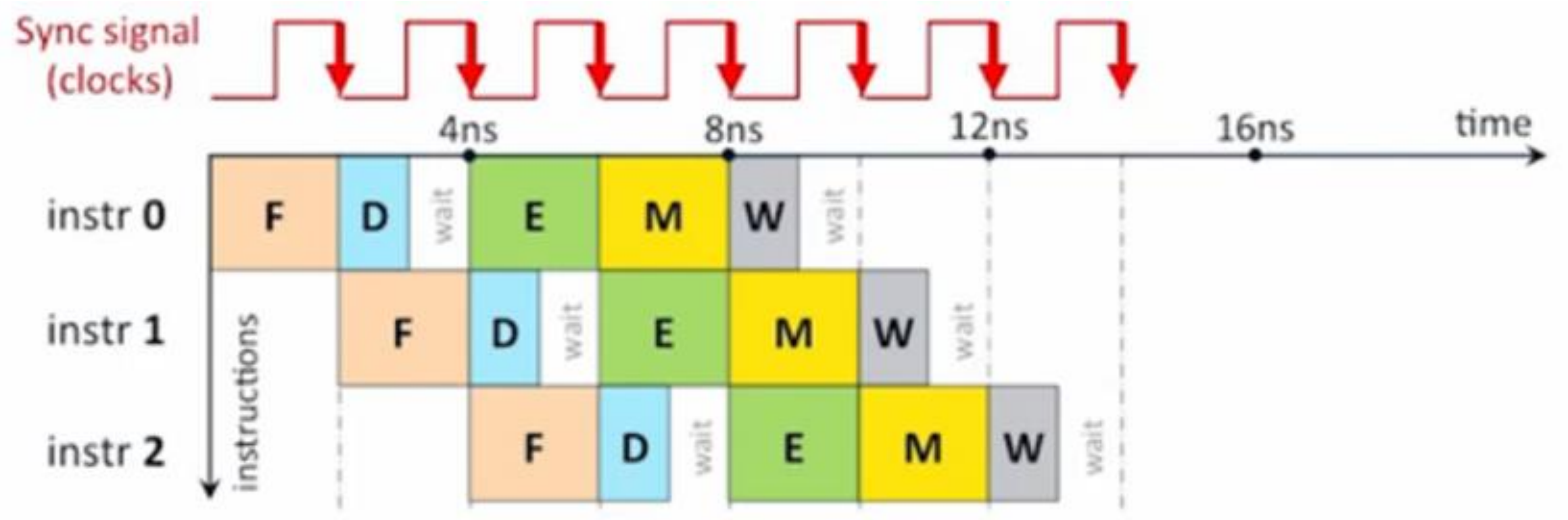


- Instructions are processed sequentially, one per cycle
- How to speed-up?
  - SW: decrease number of instructions
  - HW: decrease the time to process one instruction

    or overlap their processing. i.e. make pipeline

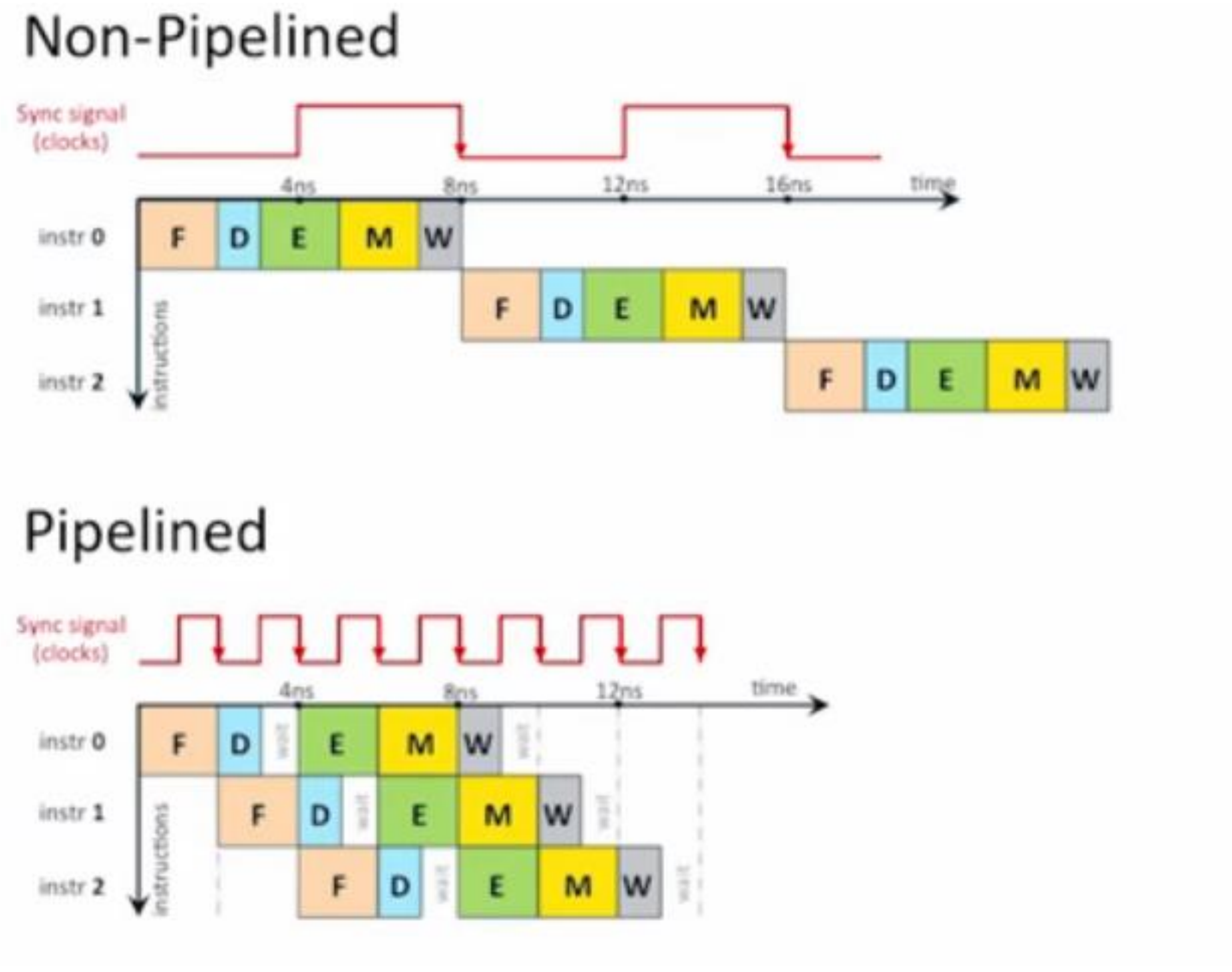
Optimization Notice

Pipeline

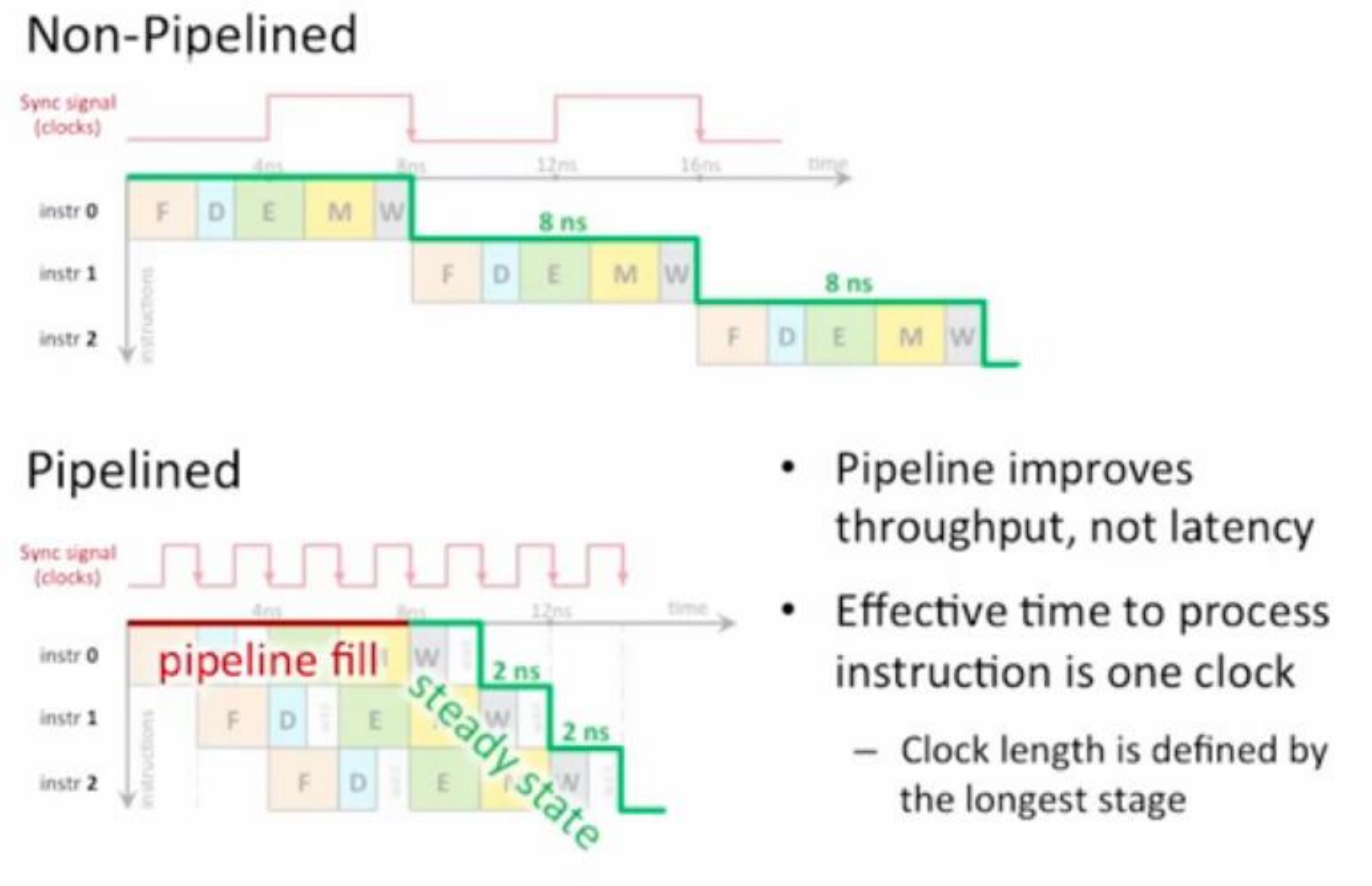


- Processing is split into several steps called “stages”
  - Each stage takes one cycle
  - The clock cycle is determined by the longest stage
- Instructions are overlapped
  - A new instruction occupies a stage as soon as the previous one leaves it

Pipeline vs Non-Pipeline



Pipeline vs Non-Pipeline



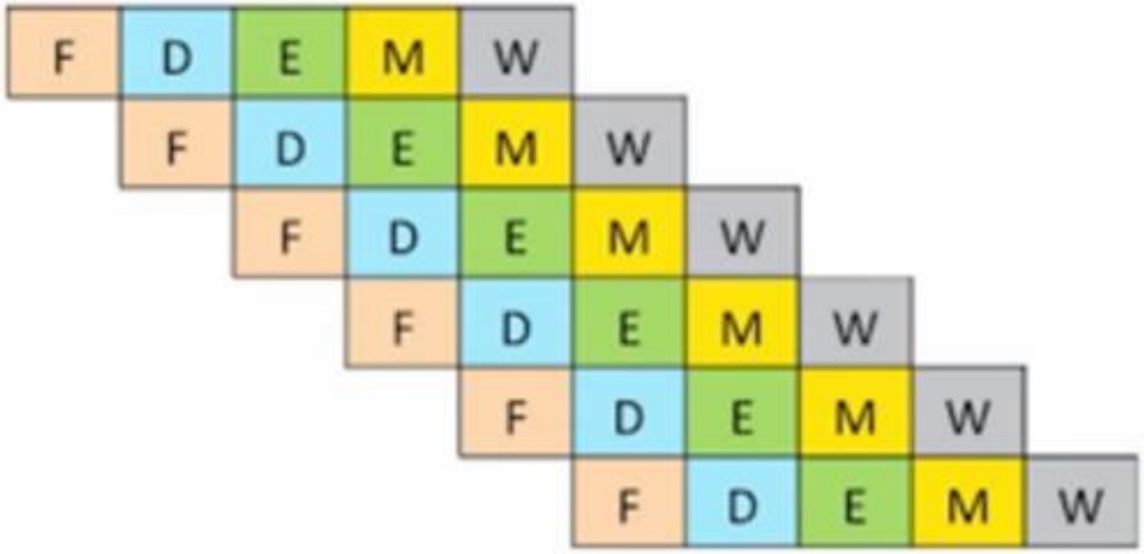
Non-Pipelined

Pipelined

- Pipeline improves throughput, not latency
- Effective time to process instruction is one clock
  - Clock length is defined by the longest stage

Pipeline Limitations

- Max speed of the pipeline is one instruction per clock
- It is rare due to dependencies among instructions (**data** or **control**) and **in-order** processing



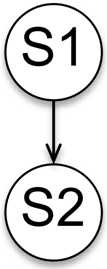
Recap from basic architecture course: Data Dependences

- A statement/instruction S2 is said to be data dependent on statement/instruction S1 if
  - S1 executes before S2 in the original program
  - S1 and S2 access the same data item
  - At least one of the accesses is a write.

Data Dependence

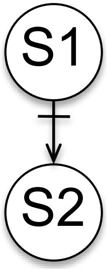
Flow dependence (RAW, True dependence)

S1: X = A+B
S2: C= X+A



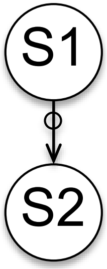
Anti dependence (WAR)

S1: A = X + B
S2: X= C + D



Output dependence (WAW)

S1: X = A+B
S2: X= C + D

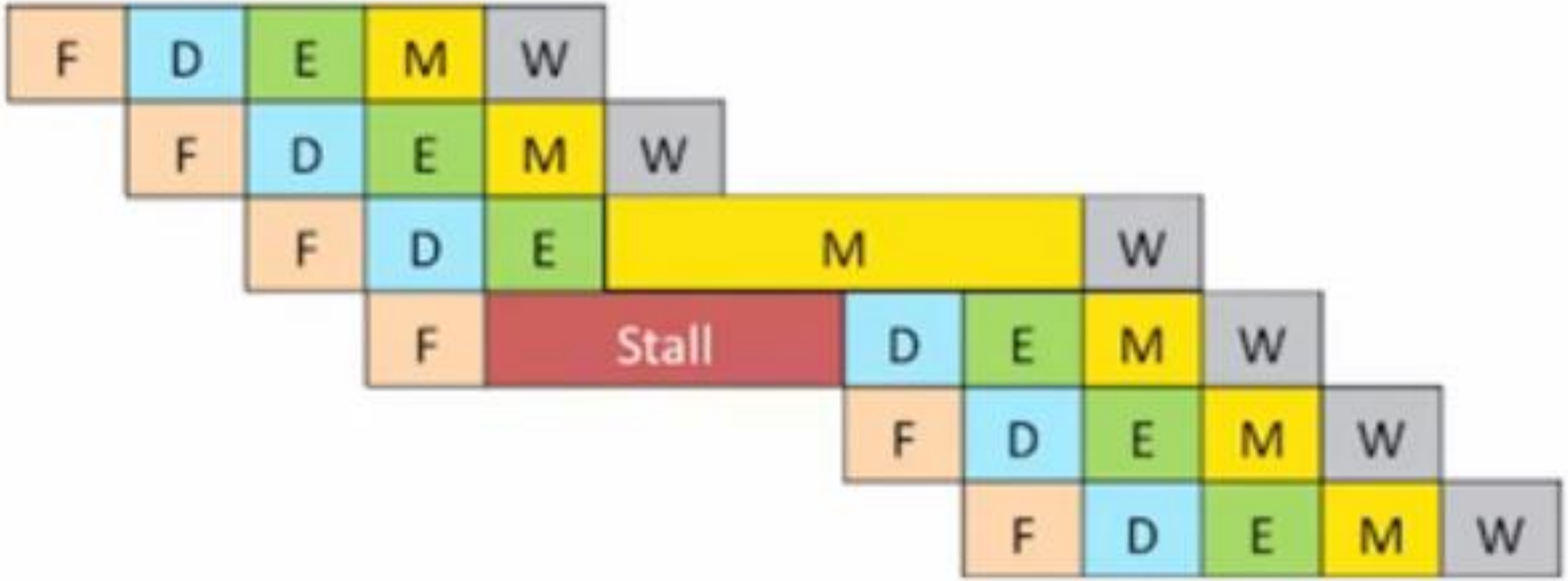


Data Dependence

- Dependences indicate an execution order that must be honored.
- Executing statements/instructions in the order of the dependences guarantee correct results.
- Statements/instructions not dependent on each other can be reordered, executed in parallel, or coalesced into a vector operation.

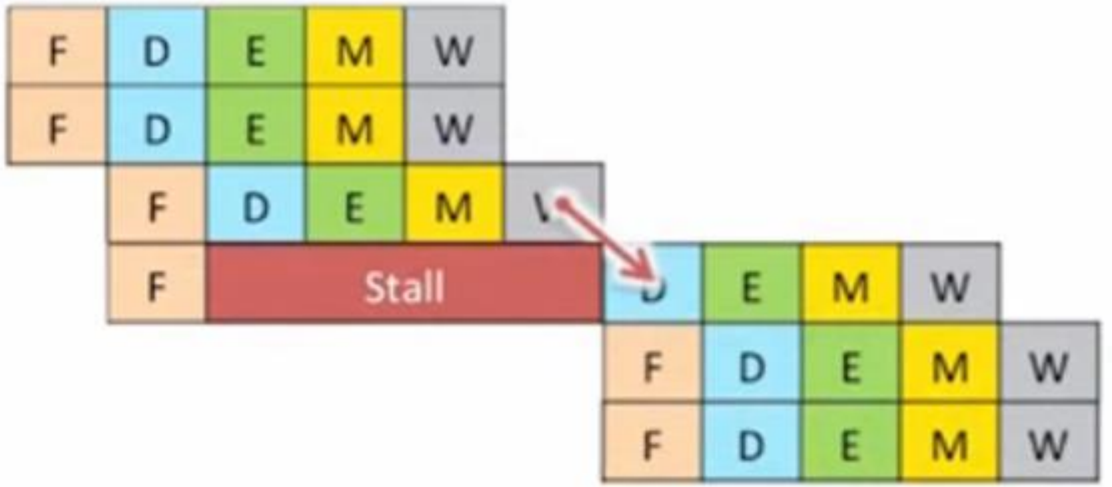
Pipeline Limitations

- Various types of hazards:
  - read after write (RAW), *true/flow dependence*
  - write after read (WAR), *anti-dependence*
  - write after write (WAW), *output dependence*



Superscalar: Wide Pipeline

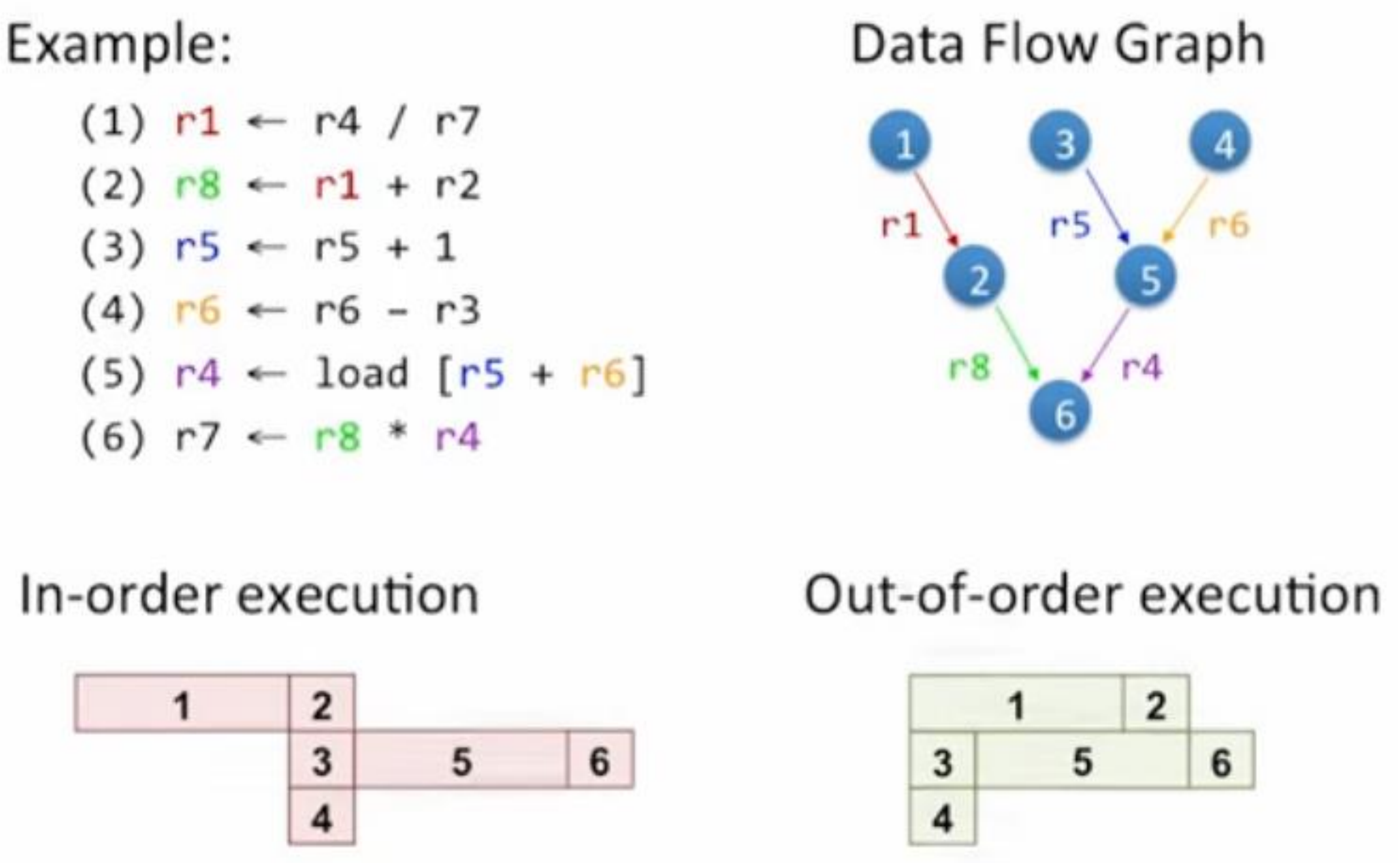
- Pipeline exploits instruction level parallelism (ILP)
- Can we improve? Execute, instructions in parallel
  - Need to double HW structures
  - Max speedup is 2 instructions per cycle (IPC=2)
  - The real speedup is less due to dependencies and in-order execution



Is Superscalar Good Enough?

- Theoretically can execute multiple instructions in parallel
    - Wide pipeline => more performance
- But…
    - Only independent subsequent instructions can be executed in parallel
    - Whereas subsequent instructions are often dependent
    - So the utilization of the second pipe is often low
- Solution: **out-of-order execution**
    - Execute instructions based on the "data flow" graph, rather than program order
    - Still need to keep the visibility of in-order execution

Data Flow Analysis



Example:

```
(1) r1 ← r4 / r7
(2) r8 ← r1 + r2
(3) r5 ← r5 + 1
(4) r6 ← r6 - r3
(5) r4 ← load [r5 + r6]
(6) r7 ← r8 * r4
```

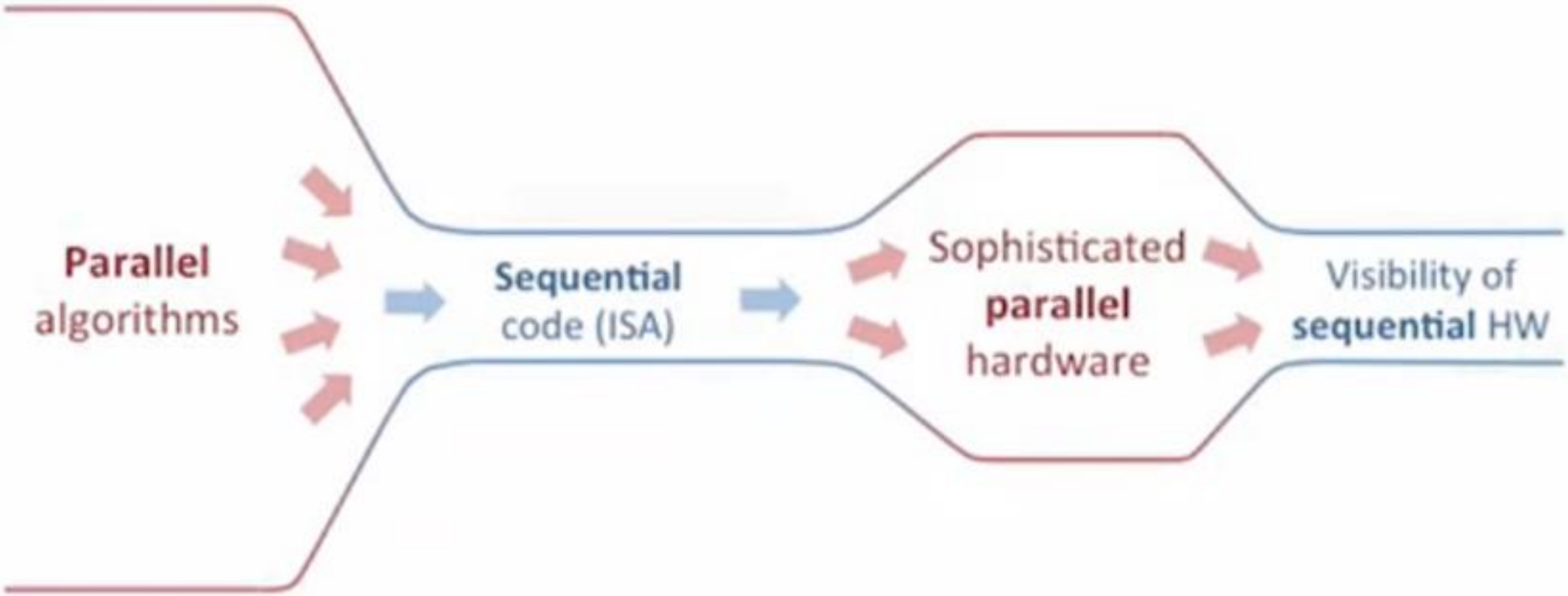
Data Flow Graph

In-order execution

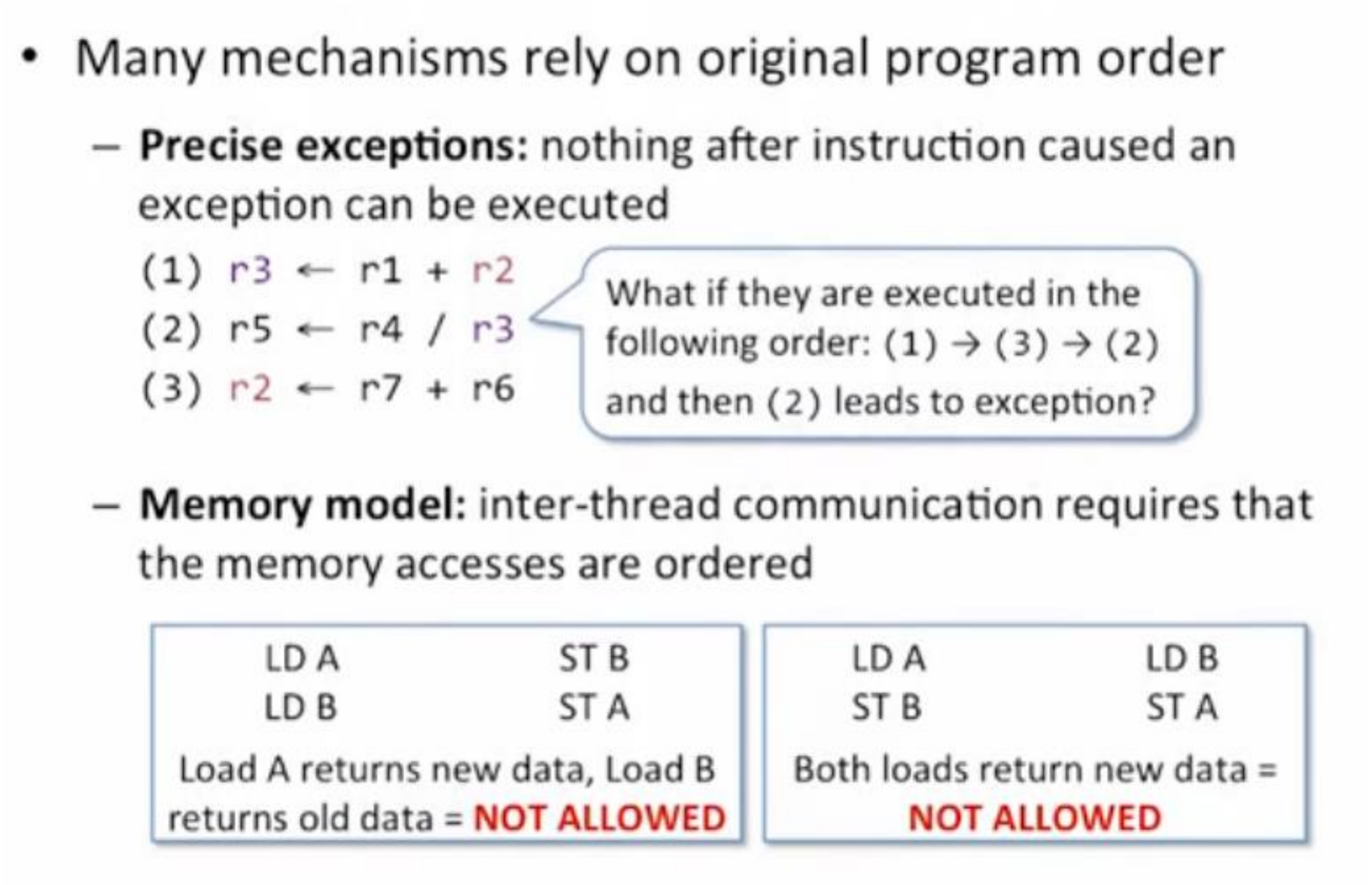
Out-of-order execution

Instruction “Grinder”

- Then technology allowed building wide HW, but the code representation remained sequential
- Decision: extract parallelism back by means of hardware
- Compatibility burden: needs to look like sequential hardware



Why Order is Important?



- Many mechanisms rely on original program order
  - **Precise exceptions:** nothing after instruction caused an exception can be executed

```
(1) r3 ← r1 + r2
(2) r5 ← r4 / r3
(3) r2 ← r7 + r6
```

What if they are executed in the following order: (1) → (3) → (2) and then (2) leads to exception?

  - **Memory model:** inter-thread communication requires that the memory accesses are ordered

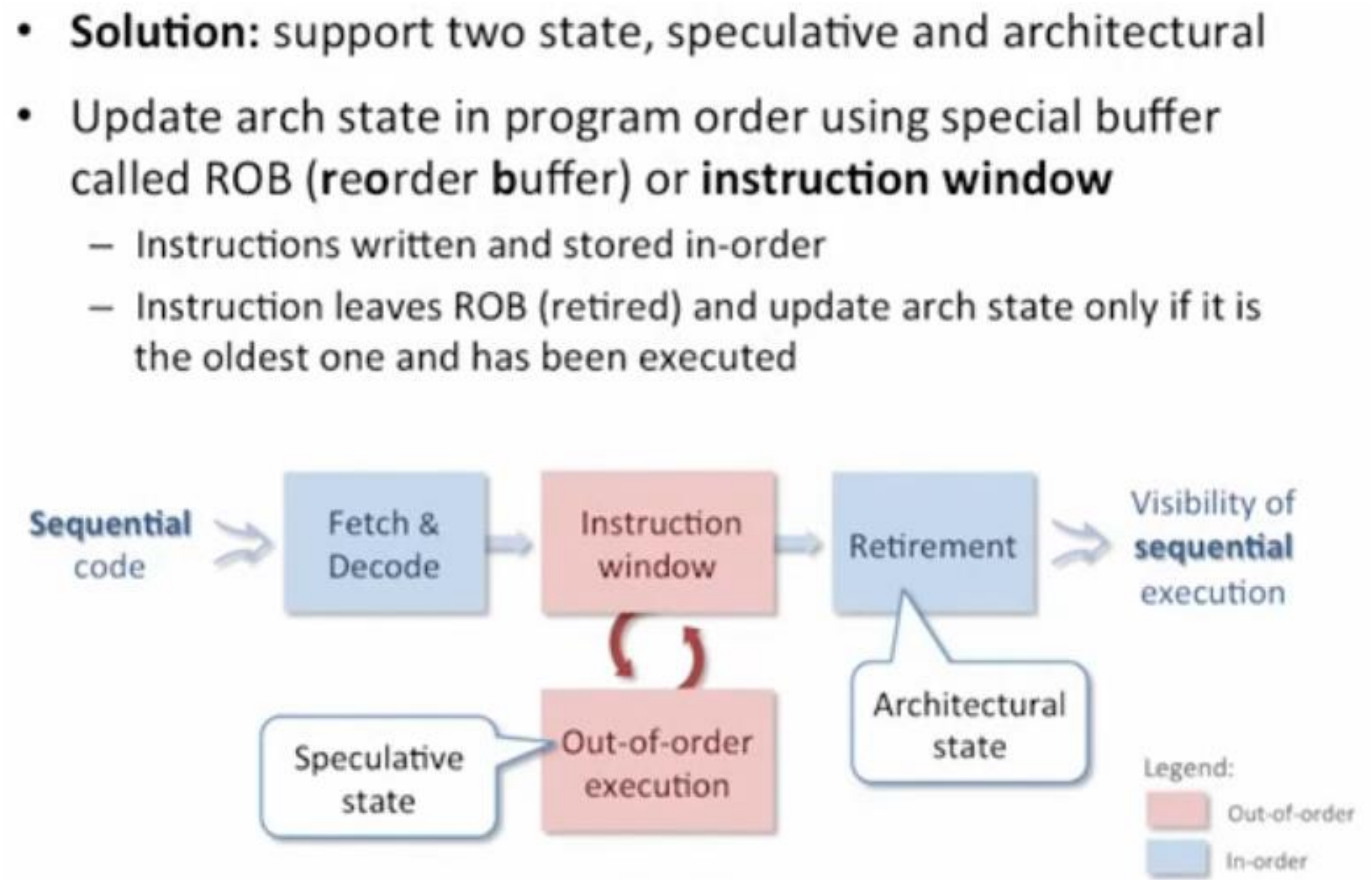
LD A	ST B
LD B	ST A

Load A returns new data, Load B returns old data = NOT ALLOWED

LD A	LD B
ST B	ST A

Both loads return new data = NOT ALLOWED

Maintaining Architectural State

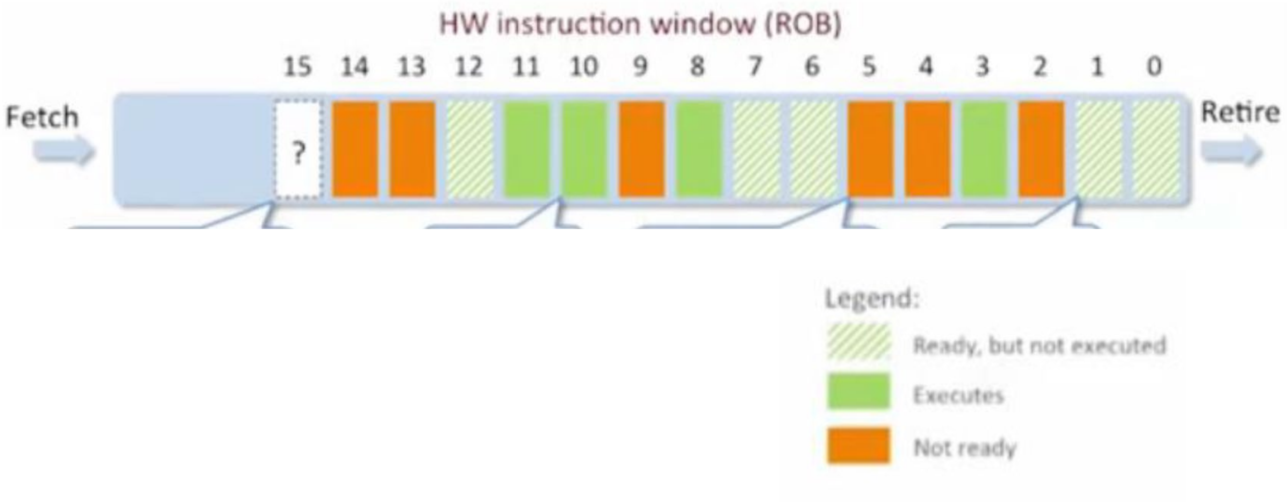


- **Solution:** support two state, speculative and architectural
- Update arch state in program order using special buffer called ROB (**re**order **b**uffer) or **instruction window**
  - Instructions written and stored in-order
  - Instruction leaves ROB (retired) and update arch state only if it is the oldest one and has been executed

Out-of-order execution: key ideas

- Out-of-order execution, in-order completion (commit/retire)
  - Precise exceptions, memory model of processor
- (I) Dataflow execution: instructions execute when operands are available
  - Reorder buffer (ROB): window of instructions
  - Instructions enter ROB in program order, retire from ROB in program order
  - Instructions in ROB execute when operands are available
  - However, results are stored in ROB until instruction retires
- (II) Register renaming
  - Goal: eliminate flow and anti-dependences on registers
  - Two register files
    - Architected register file: registers visible to programmer and ISA
    - Physical register file:
      - Larger set of registers that hold values for inflight instructions
      - Not visible to programmer or ISA, managed by hardware
  - OOO processors implement both dataflow execution and register renaming but useful to study separately
- (III) Branch prediction
  - How to get a big window of instructions to work on

OOO execution without renaming

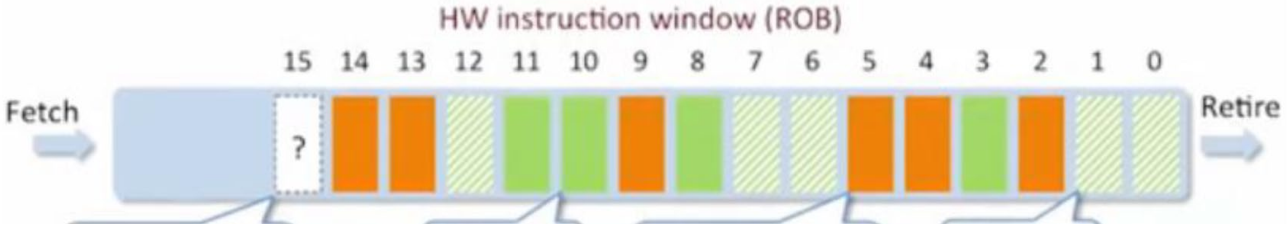


	v(alid)	value	consumers
R0			
R1			
…..			

Registers

- Invariants (ROB):
  1. At most one instruction in flight writes to a given register
  2. Instruction executes when all operands available but result stored in result slot of ROB entry
  3. Destination register is updated only when instruction is retired
- Invariant (Registers):
  4. Rx.v → No instruction in flight writes to register rx
  5. ! Rx.v → An inflight instruction will write to rx and ROB entries for younger instructions that need that value are in Rx.consumers

Actions: (r3 ← r1 + 9 in ROB# n)



	v	value	consumers
R0			
R1			
.....			

- Enter:

```
if (! R3.v) //inflight instruction will modify r3
        wait until R3.v = true; (Invariant 1)
R3.v = false; (Invariant 1)
Enter instruction into ROB# n;
if (R1.v) then
        {ROB[n].slot1 = R1.value; (Invariant 4)
         Mark instruction as ready;}
else Add #n to R1.consumers; (Invariant 5)
```

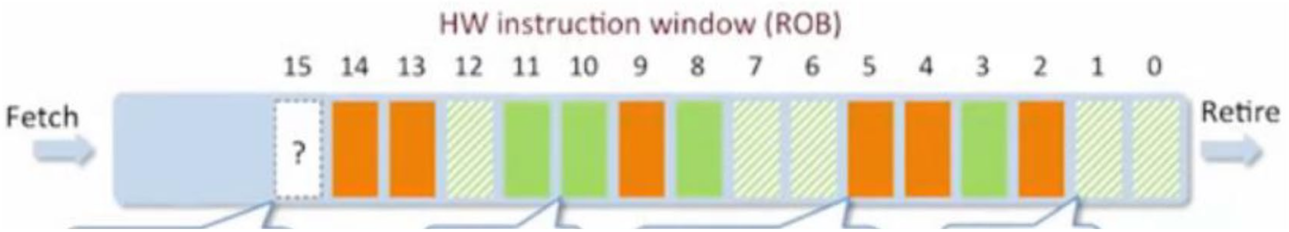
- Retire:

```
R3.value = ROB[n].result;
R3.v = true; (Invariant 3)
Notify all ROB entries in R3.consumers;  (Invariant 5)
```

- Execute:

```
Dispatch to free functional unit;
Write result back to ROB[n].result
```

Dependences and precise exceptions



	v	value	consumers
R0			
R1			
…..			

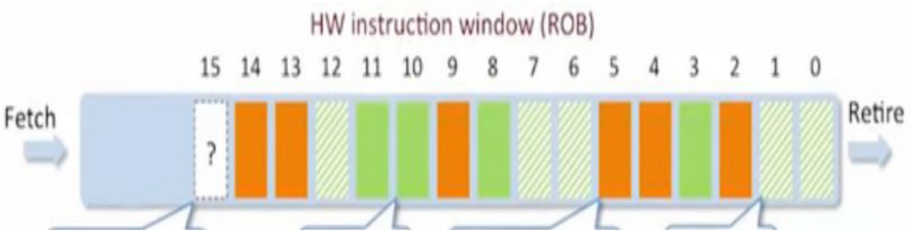
- Invariants (ROB):
  1. At most one instruction in flight writes to a given register
  2. Instruction executes when all operands available but result stored in result slot of ROB entry
  3. Destination register is updated only when instruction is retired
- Invariant (Registers):
  4. Rx.v = true ➔ No instruction in flight writes to register rx
  5. if (! Rx.v) ROB entries for all inflight instructions that read rx are in Rx.Consumers

- Flow dependences
  - Invariants (4) and (5,3)
- Anti- and output dependences
  - Invariant (1)
- Precise exceptions
  - Invariants (2) and (3)

Limitations of scheme:

- Consider instruction (r3 ← r1 + 9)
- Register r3 value not forwarded to consumers in ROB until instruction retires
- Why not forward value to consumers in ROB as soon as it is computed?
  - More parallelism
- One implementation of idea: register renaming

OOO execution with renaming



	v(alid)	value	PR#
R0			
R1			
.....			

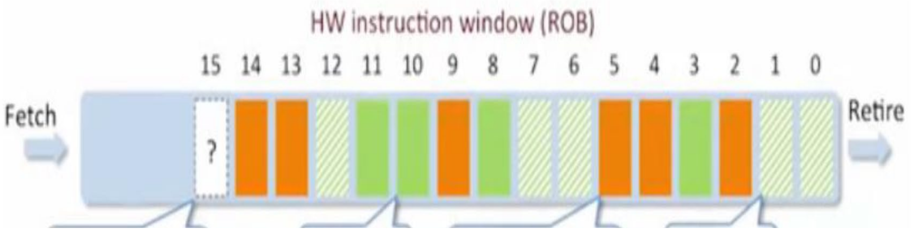
Architected Registers

	v(alid)	value	consumers
PR0			
PR1			
.....			

Physical Registers

- Two sets of registers
  - Architected registers: registers visible to the ISA and programmer
  - Physical registers:
    - Different and larger set of registers that hold values temporarily while instructions are in flight
    - Not visible to ISA or programmer, managed entirely by hardware
- Register renaming
  - Eliminate anti- and output-dependences within instruction window by using physical registers
- There can be several instructions in flight that write to same architected register, but they will write to different physical registers

OOO execution with renaming



	v(alid)	value	PR#
R0			
R1			
…..			

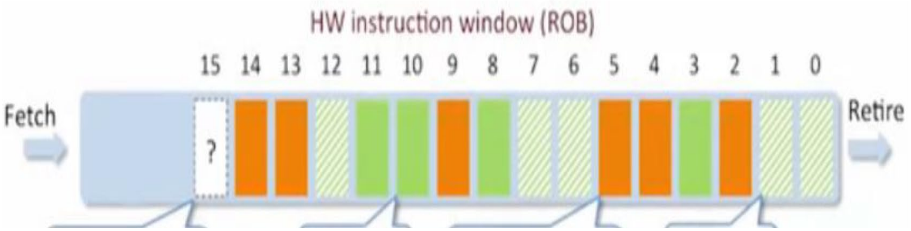
Architected Registers

	v(alid)	value	consumers
PR0			
PR1			
…..			

Physical Registers

- Invariants (ROB):
  1. At most one instruction in flight writes to a given *physical* register
  2. Instruction executes when all operands available but result stored in physical register
  3. Architected register is updated when instruction is retired
- Invariant (Registers):
  4. Rx.v → No instruction in flight writes to register rx
  5. (! Rx.v) → Youngest instruction that writes to rx will store result in Rx[PR#]
  6. (!Rx.v and Rx.PR#.v) → Youngest instruction that writes to rx has completed and value is in Rx.PR#.value
  7. (!Rx.v and ! Rx.PR#.v) → Youngest instruction that writes to rx has not completed and ROB entries for all inflight instructions that read its value are in Rx[PR#].consumers

OOO execution with renaming: (r3 ← r1 + 9 in ROB# n)



	v(alid)	value	PR#
R0			
R1			
…..			

Architected Registers

	v(alid)	value	consumers
PR0			
PR1			
…..			

Physical Registers

- Enter:

```
    PRm = free physical register;
    R3.PR# = PRm;
    R3[v] = false;
    PRm.v = false;
    Enter instruction and PRm into ROB[n]; (Invariant 5)
    if (R1.v) then
            {ROB[n].slot1 = R1.value; (Invariant 4)
             Mark instruction as ready;}
    else  if (R1.PR#.v) then
            {ROB[n].slot1 = R1.PR#.value; (Invariants 5 and 6)
             Mark instruction as ready;}
    else
            Add #n to R1.PR#.consumers; (Invariant 7)
```

- Retire:

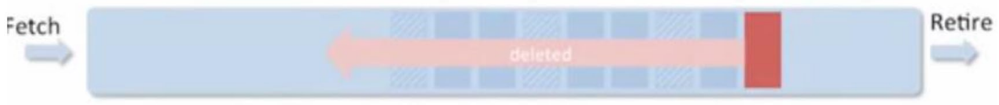
```
    R3.value = R3.PR#.value;
    R3.v = true; (Invariant 3)
```

- Execute:

```
    Dispatch to free functional unit;
    pr = ROB[n].PR# //physical register associated with r3
    Write result to pr.value;
    pr.v = true; (Invariant 6)
    Notify all ROB entries in pr.Consumers; (Invariant 7)
```

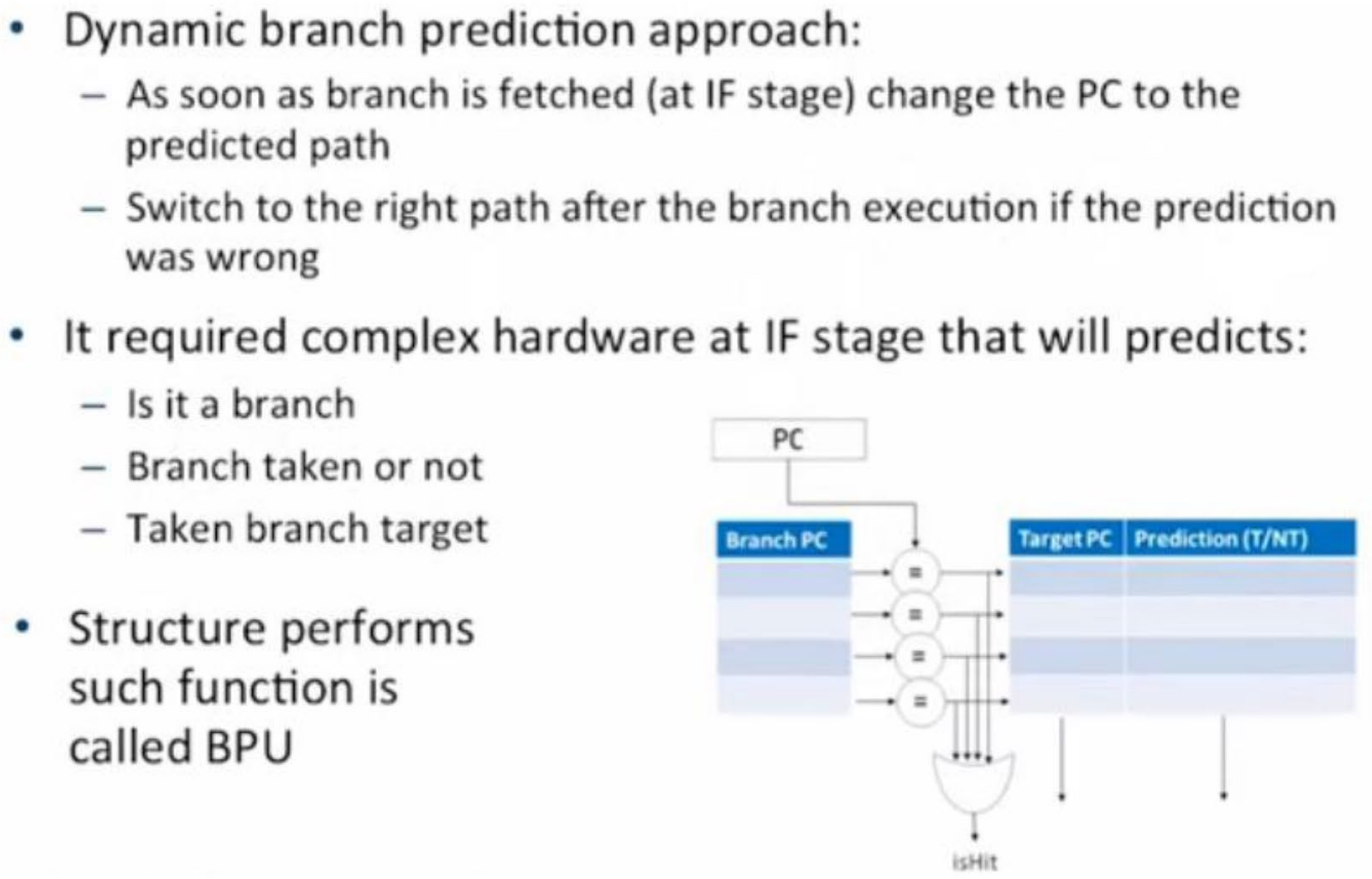
How large should ROB be?

- In principle, larger the better
  - Find more independent instructions
  - Hide longer memory latencies
- Example
  - Many CPUs have ROB of size ~200



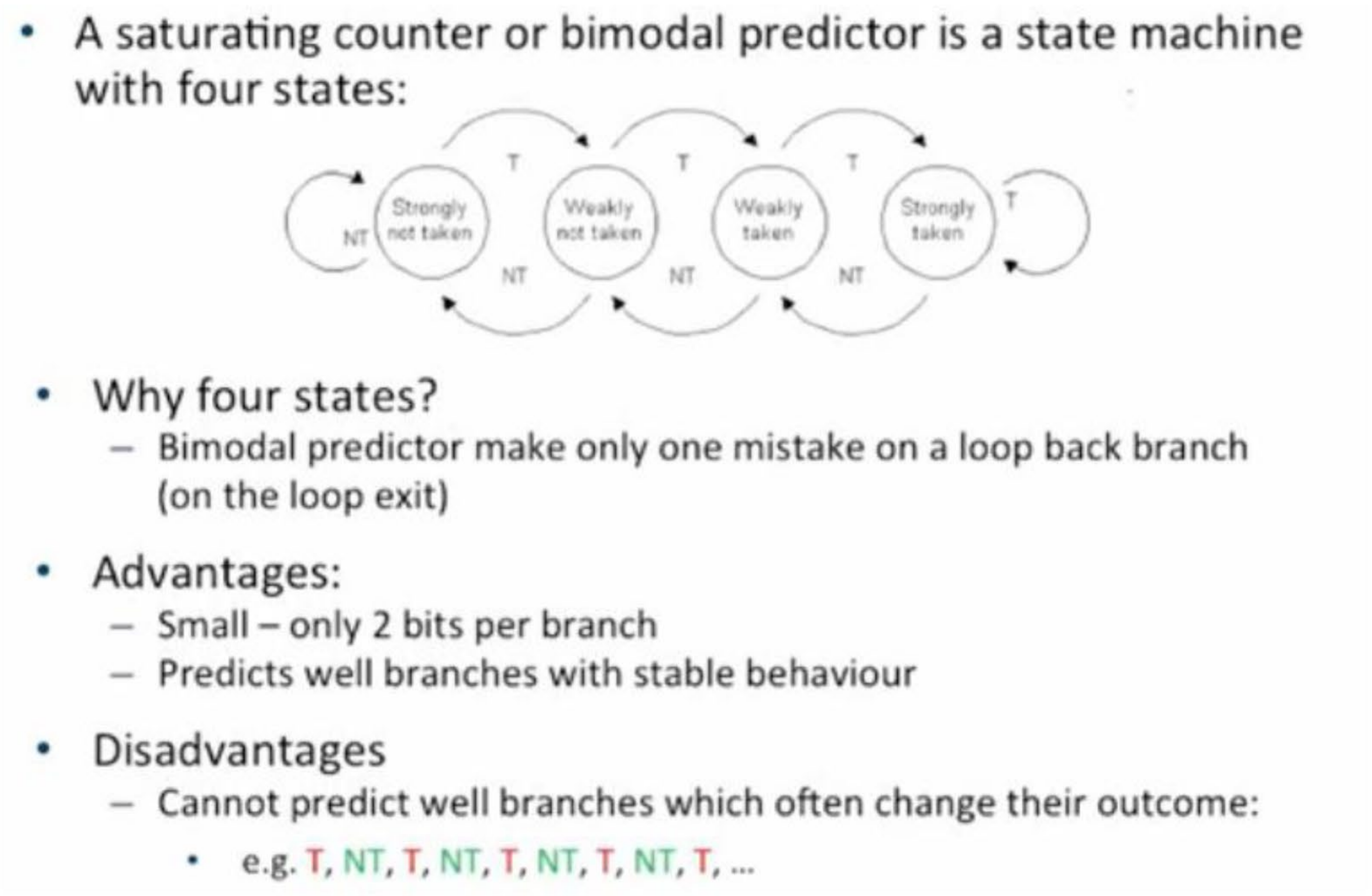
- Main limitation: branches
- On average, 1 in every 5 instructions is branch
- How to fetch instructions into ROB when branch has not been resolved?
- One solution: guess randomly which way branch will go
  - Probability of getting one branch right: 50%
  - Probability that 100th instruction in window will be executed is (0.5)^20 = 0.0001%

Dynamic Branch Prediction



- Dynamic branch prediction approach:
  - As soon as branch is fetched (at IF stage) change the PC to the predicted path
  - Switch to the right path after the branch execution if the prediction was wrong
- It required complex hardware at IF stage that will predicts:
  - Is it a branch
  - Branch taken or not
  - Taken branch target
- Structure performs such function is called BPU

How To Predict Branch?



- A saturating counter or bimodal predictor is a state machine with four states:

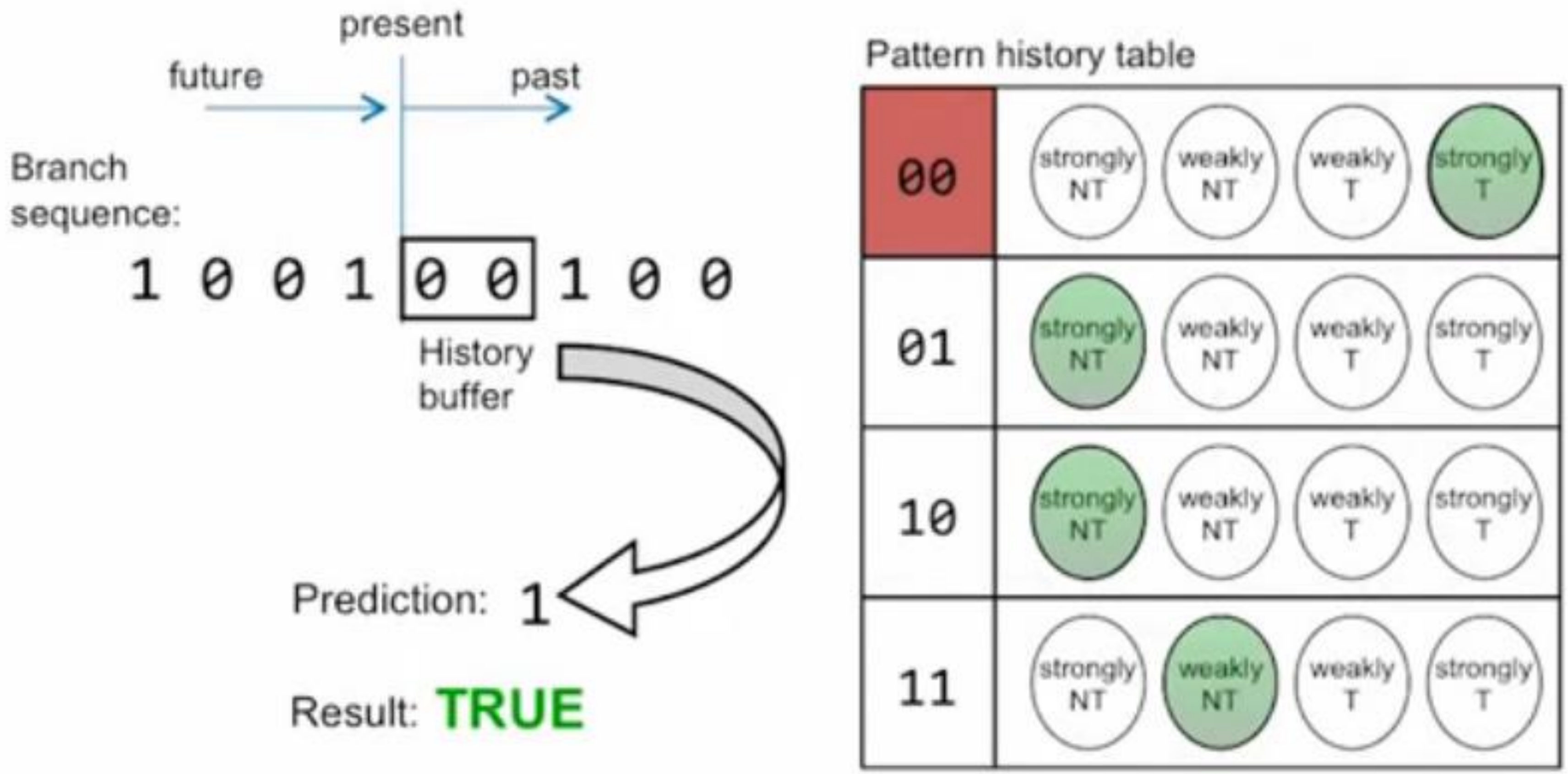
Strongly not taken
Weakly not taken
Weakly taken
Strongly taken
T
T
T
T
NT
NT
NT
NT

- Why four states?
  - Bimodal predictor make only one mistake on a loop back branch (on the loop exit)
- Advantages:
  - Small – only 2 bits per branch
  - Predicts well branches with stable behaviour
- Disadvantages
  - Cannot predict well branches which often change their outcome:
    - e.g. T, NT, T, NT, T, NT, T, NT, T, ...

More elaborate branch predictors exist: (e.g.) Perceptron-based branch predictor (Calvin Lin)

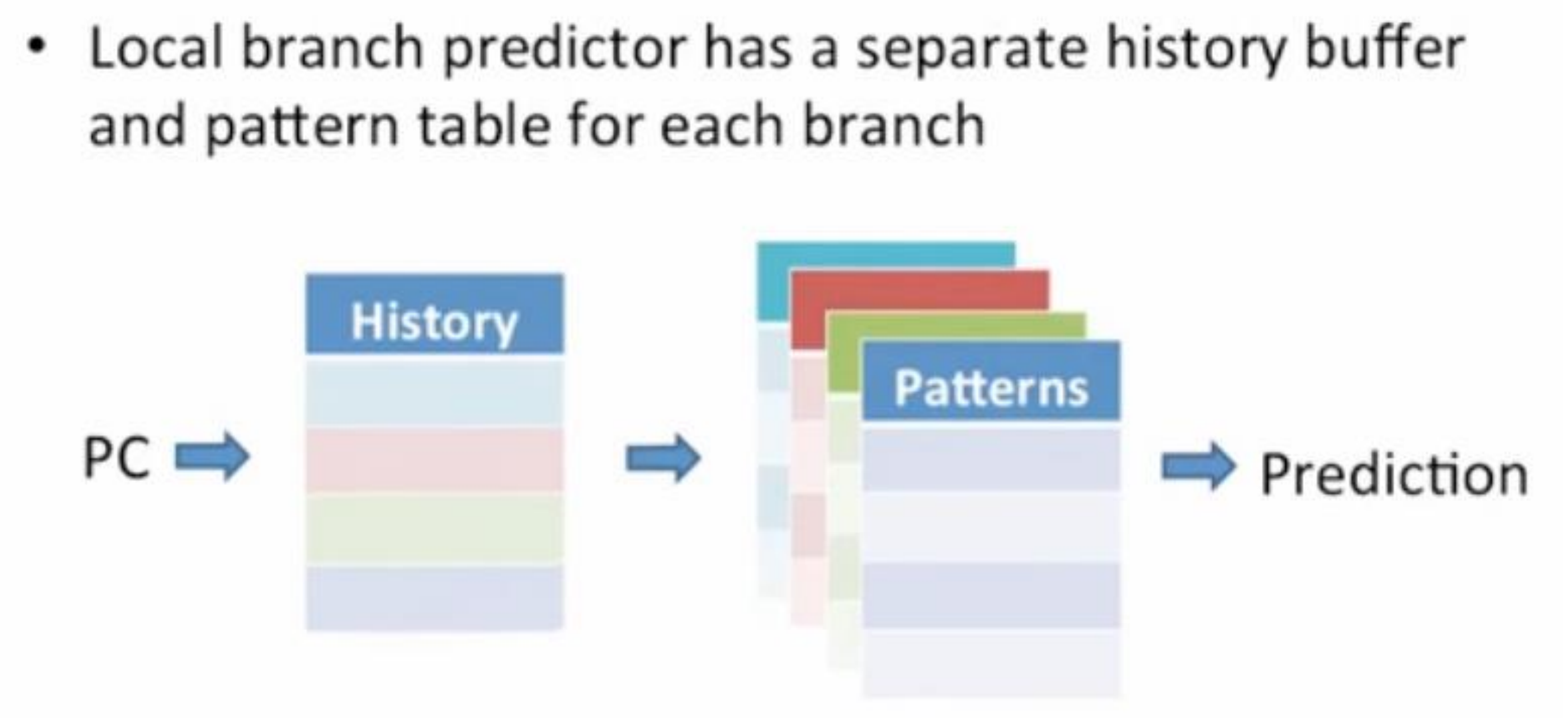
Using History Patterns

- Remember not just most often outcome, but most often outcome after certain history patterns



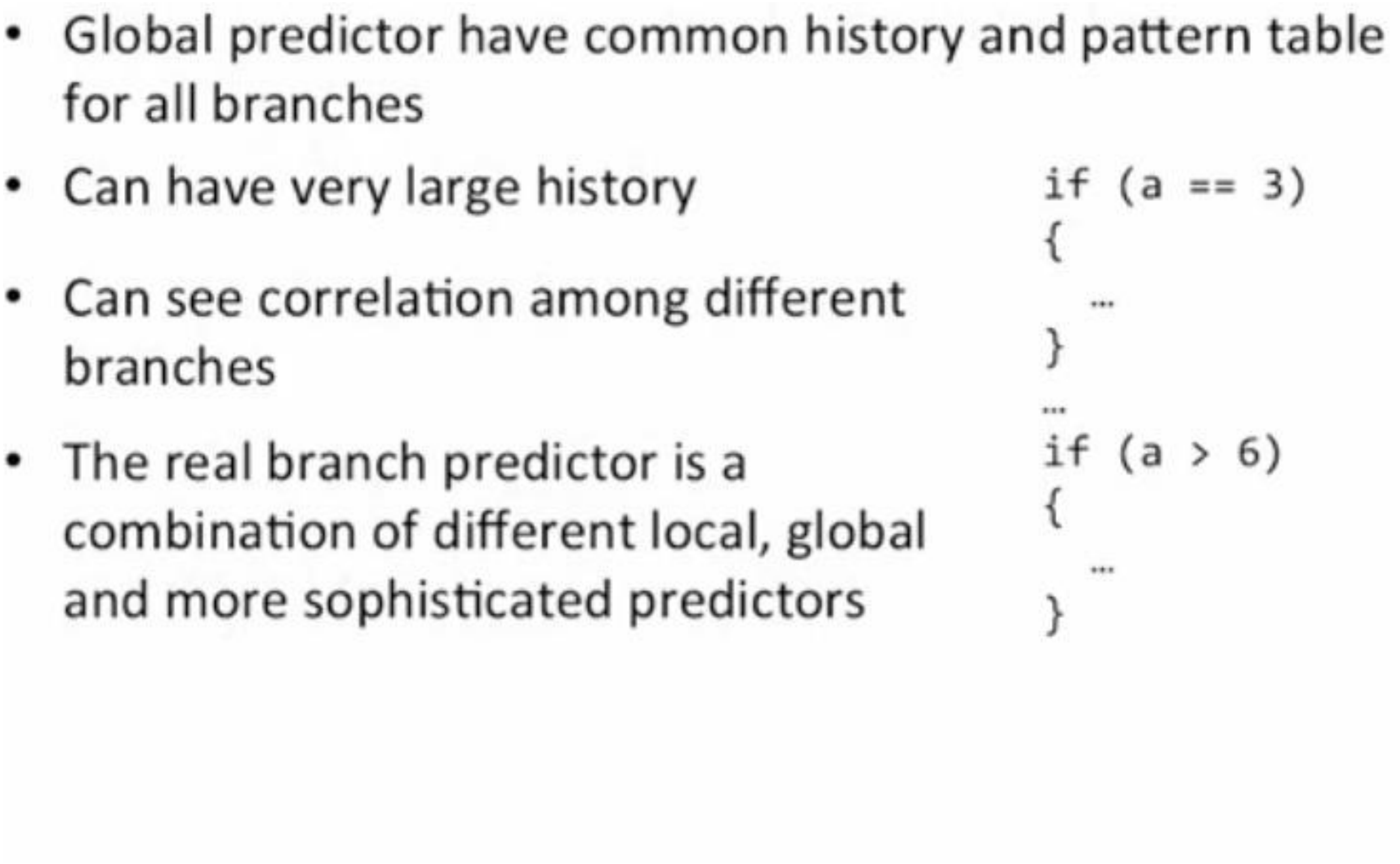
00	strongly NT	weakly NT	weakly T	strongly T
01	strongly NT	weakly NT	weakly T	strongly T
10	strongly NT	weakly NT	weakly T	strongly T
11	strongly NT	weakly NT	weakly T	strongly T

Local Predictor



- Local branch predictor has a separate history buffer and pattern table for each branch

Global Predictor



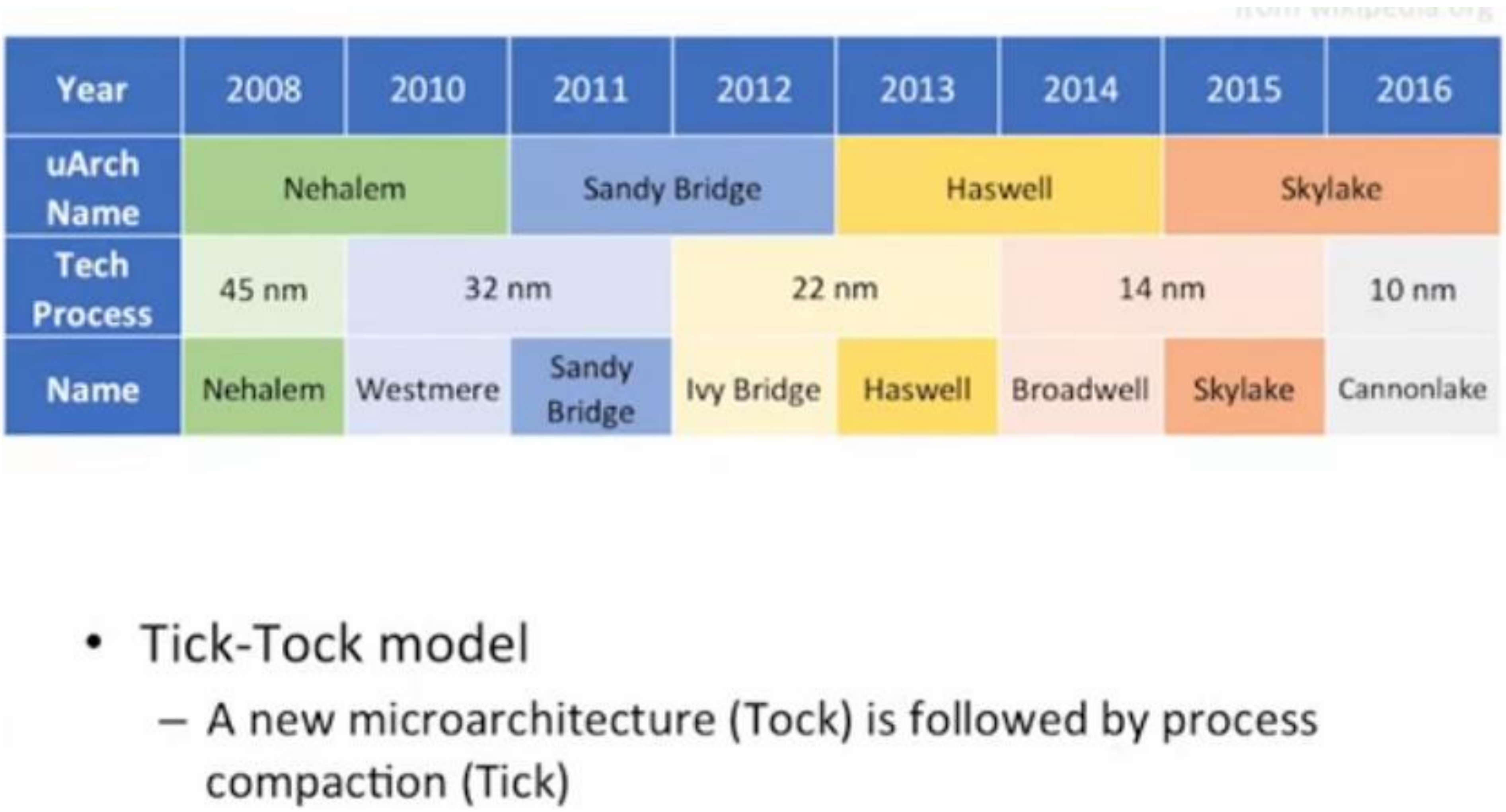
- Global predictor have common history and pattern table for all branches
- Can have very large history
- Can see correlation among different branches
- The real branch predictor is a combination of different local, global and more sophisticated predictors

```
if (a == 3)
{
   …
}
…
if (a > 6)
{
   …
}
```

Topics covered in lecture

- Instruction level parallelism (ILP)
  - Pipelined execution
  - Superscalar
  - Out-of-order execution
  - Limit on ILP: data and control dependences between instructions
- Out-of-order execution implementation
  - Out of order execution, in-order completion (commit)
  - RAW dependences: forwarding
  - WAR and WAW dependences: register renaming
  - Control dependences: branch prediction and speculative execution

Intel Processor Roadmap

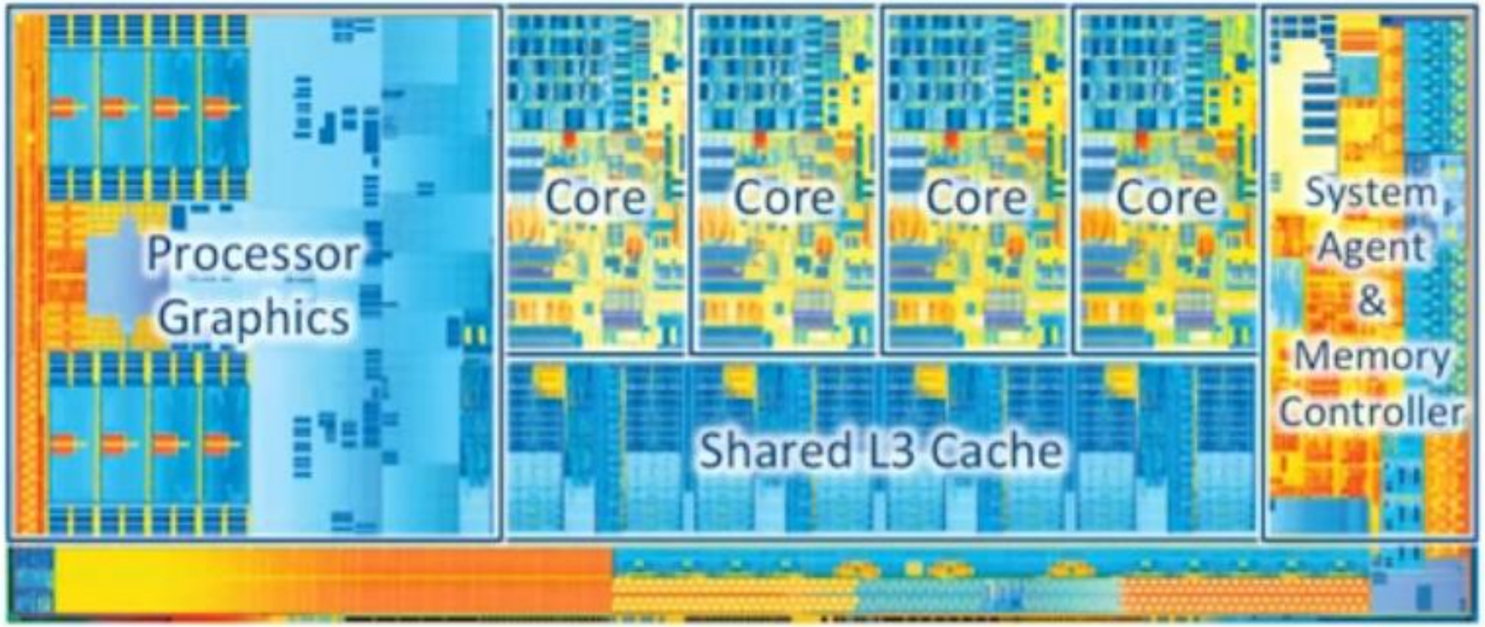


from wikipedia.org

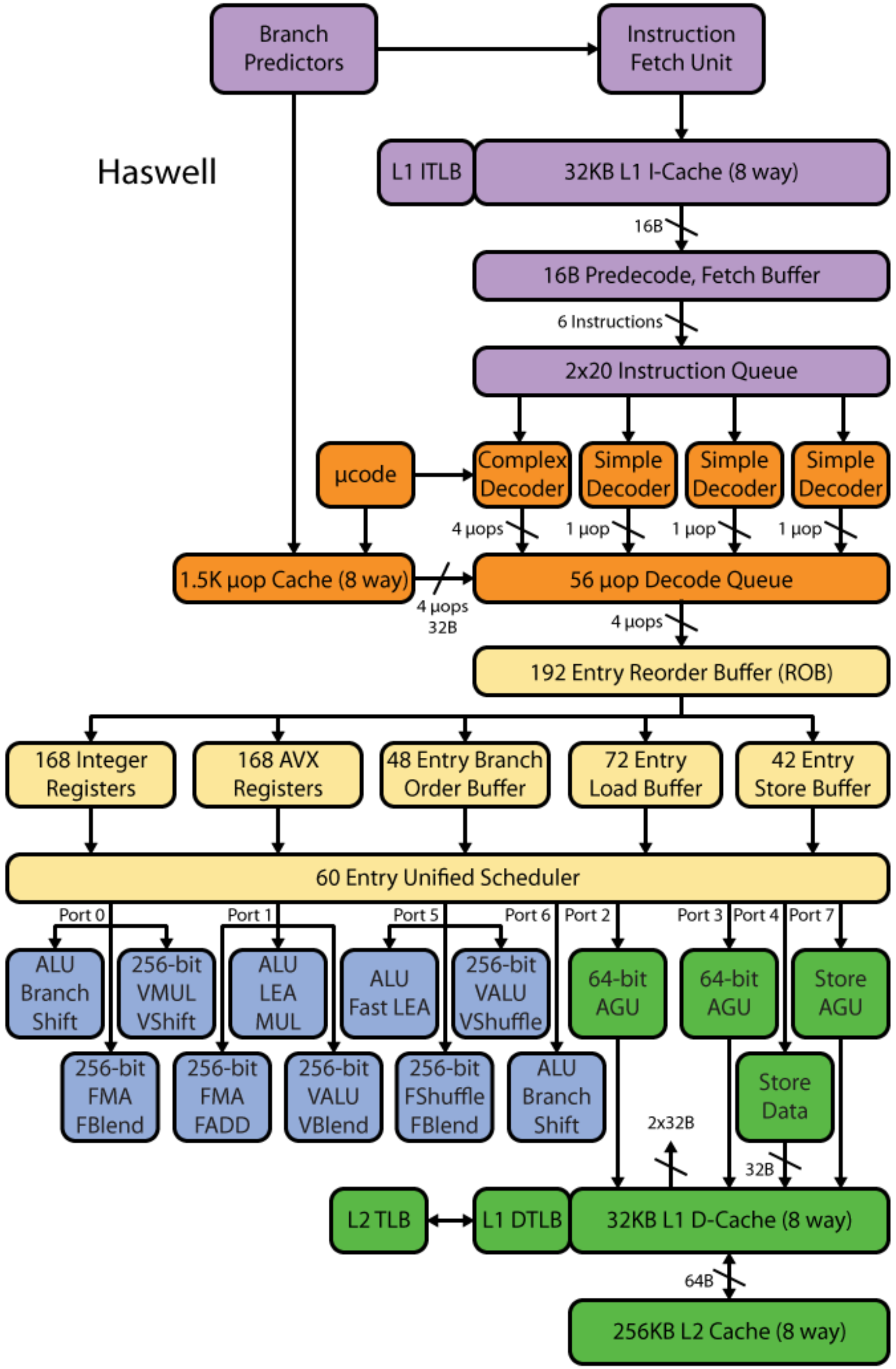
Year	2008	2010	2011	2012	2013	2014	2015	2016
uArch Name	Nehalem		Sandy Bridge		Haswell		Skylake	
Tech Process	45 nm	32 nm		22 nm		14 nm		10 nm
Name	Nehalem	Westmere	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Skylake	Cannonlake

- Tick-Tock model
  - A new microarchitecture (Tock) is followed by process compaction (Tick)

Haswell Floorplan



- 22nm process
- 1.4 Billion transistors
- Die size: 160 mm2



Haswell

Branch Predictors

Instruction Fetch Unit

L1 ITLB

32KB L1 I-Cache (8 way)

16B

16B Predecode, Fetch Buffer

6 Instructions

2x20 Instruction Queue

µcode

Complex Decoder

Simple Decoder

Simple Decoder

Simple Decoder

4 µops

1 µop

1 µop

1 µop

1.5K µop Cache (8 way)

4 µops 32B

56 µop Decode Queue

4 µops

192 Entry Reorder Buffer (ROB)

168 Integer Registers

168 AVX Registers

48 Entry Branch Order Buffer

72 Entry Load Buffer

42 Entry Store Buffer

60 Entry Unified Scheduler

Port 0

Port 1

Port 5

Port 6

Port 2

Port 3

Port 4

Port 7

ALU Branch Shift

256-bit VMUL VShift

ALU LEA MUL

ALU Fast LEA

256-bit VALU VShuffle

64-bit AGU

64-bit AGU

Store AGU

256-bit FMA FBlend

256-bit FMA FADD

256-bit VALU VBlend

256-bit FShuffle FBlend

ALU Branch Shift

Store Data

2x32B

32B

L2 TLB

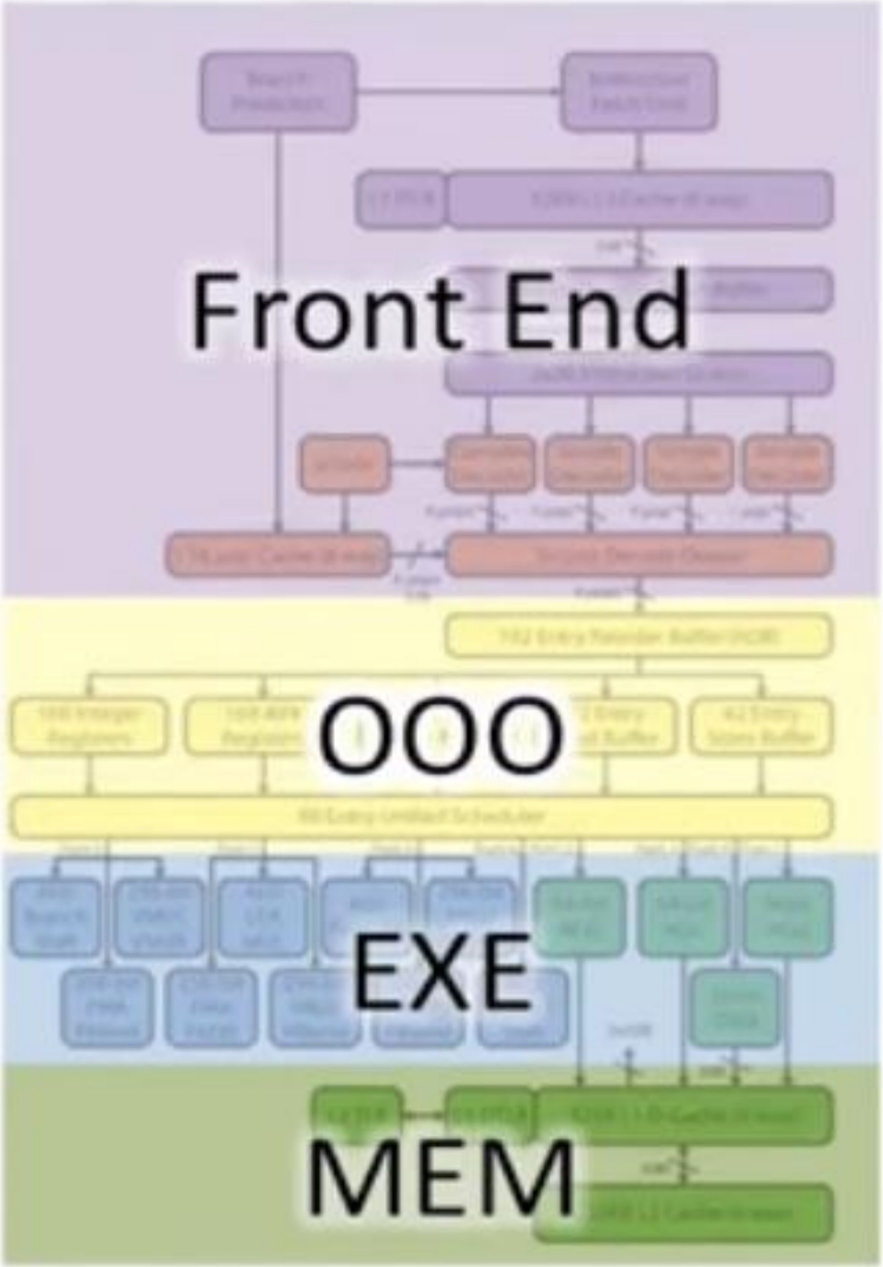
L1 DTLB

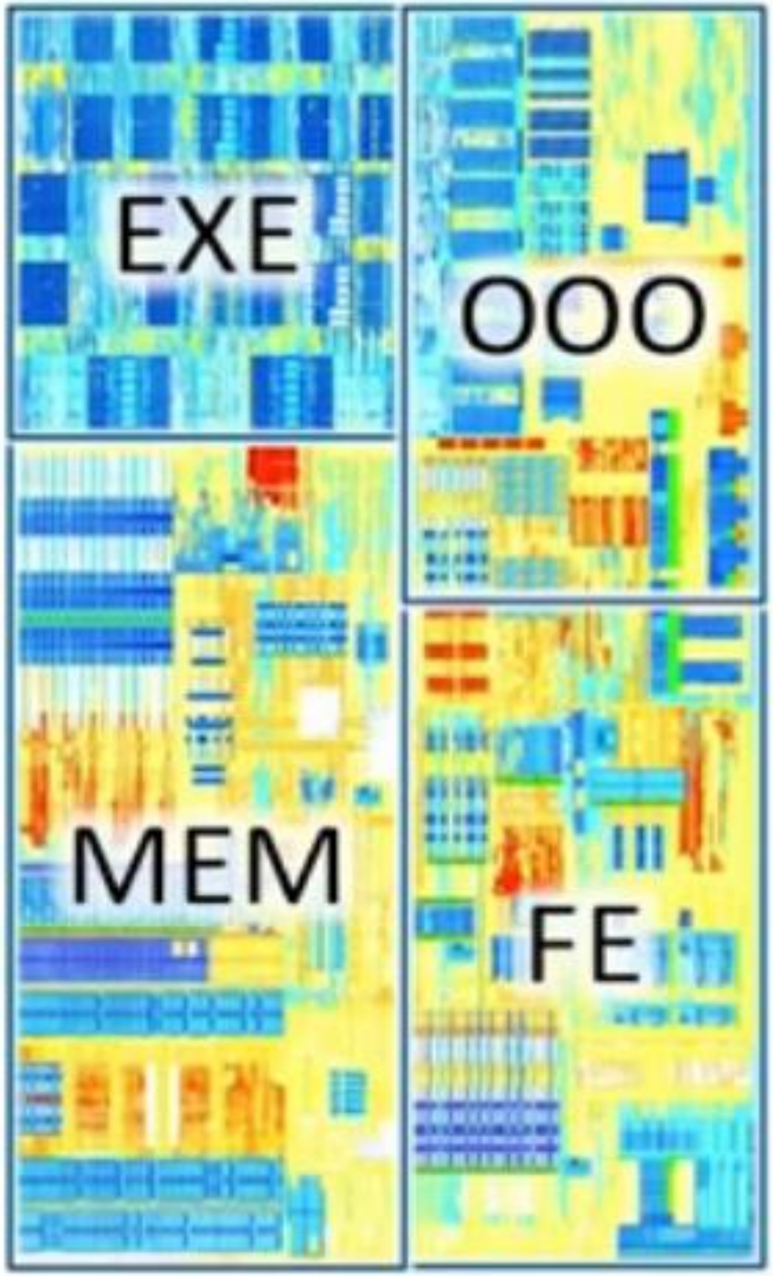
32KB L1 D-Cache (8 way)

64B

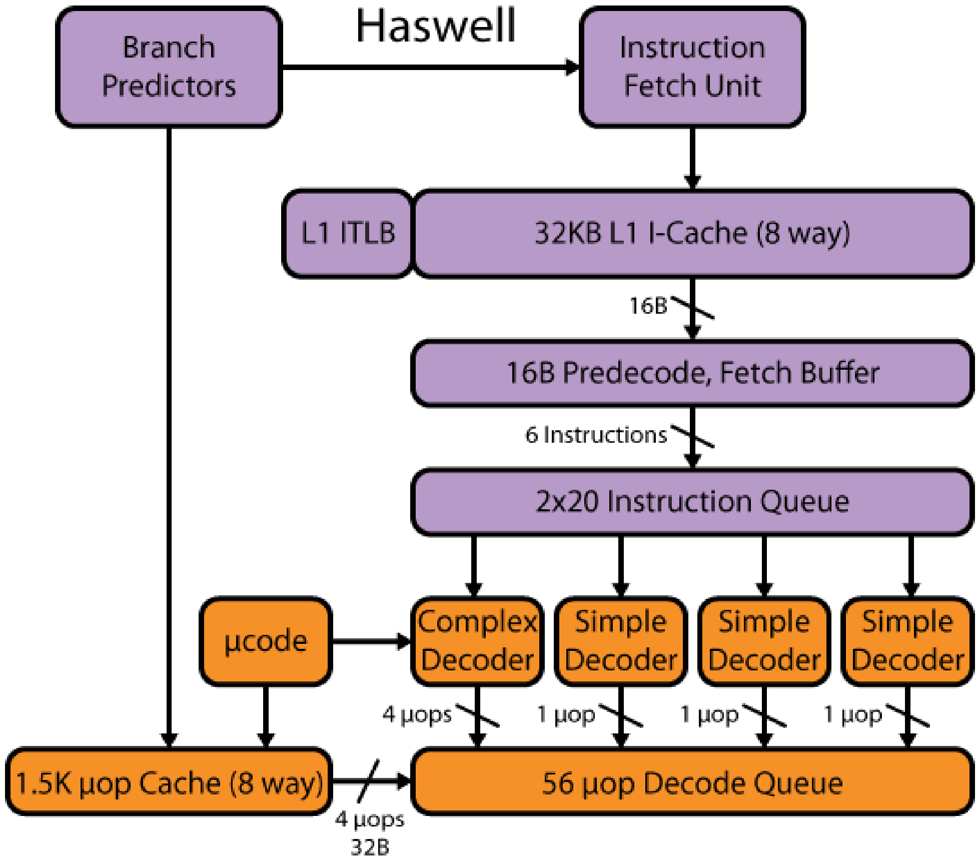
256KB L2 Cache (8 way)

Optimization Notice


Front End
OOO
EXE
MEM


EXE
OOO
MEM
FE

Block Diagram



Haswell

Branch Predictors

Instruction Fetch Unit

L1 ITLB

32KB L1 I-Cache (8 way)

16B

16B Predecode, Fetch Buffer

6 Instructions

2x20 Instruction Queue

µcode

Complex Decoder

Simple Decoder

Simple Decoder

Simple Decoder

4 µops

1 µop

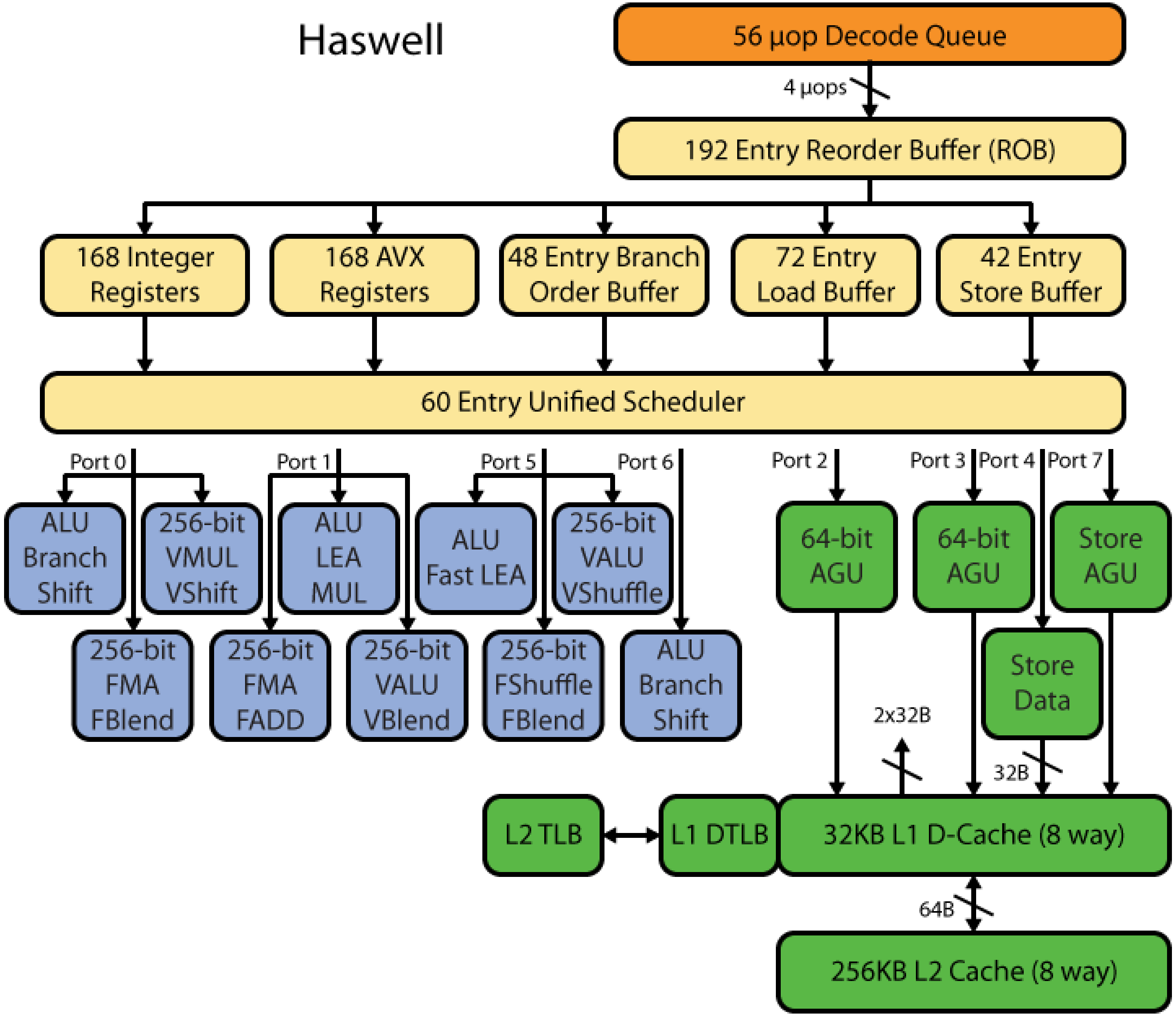
1 µop

1 µop

1.5K µop Cache (8 way)

4 µops 32B

56 µop Decode Queue



Haswell

56 µop Decode Queue

4 µops

192 Entry Reorder Buffer (ROB)

168 Integer Registers

168 AVX Registers

48 Entry Branch Order Buffer

72 Entry Load Buffer

42 Entry Store Buffer

60 Entry Unified Scheduler

Port 0

Port 1

Port 5

Port 6

Port 2

Port 3

Port 4

Port 7

ALU Branch Shift

256-bit VMUL VShift

ALU LEA MUL

ALU Fast LEA

256-bit VALU VShuffle

256-bit FMA FBlend

256-bit FMA FADD

256-bit VALU VBlend

256-bit FShuffle FBlend

ALU Branch Shift

64-bit AGU

64-bit AGU

Store AGU

Store Data

2x32B

32B

L2 TLB

L1 DTLB

32KB L1 D-Cache (8 way)

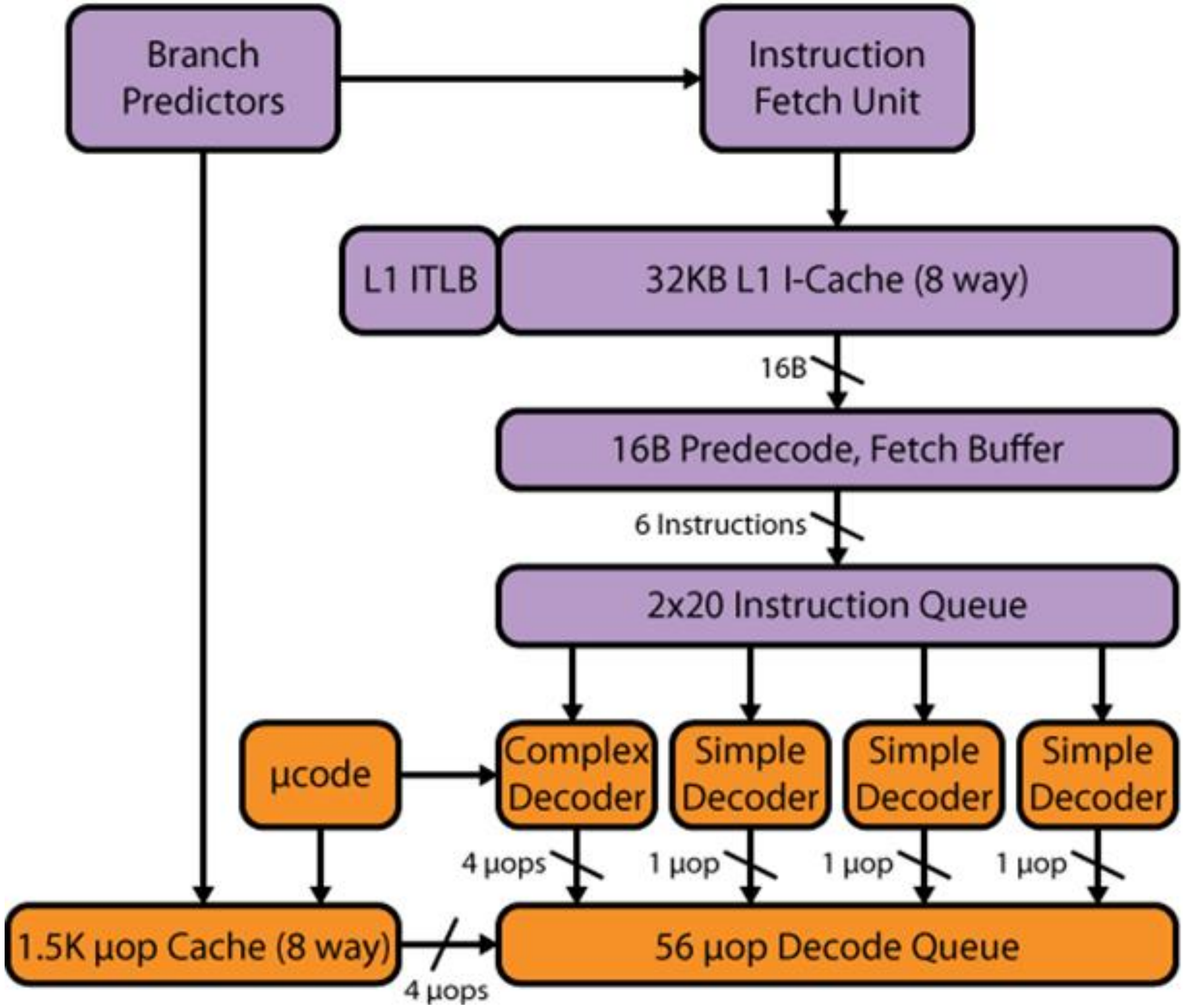
64B

256KB L2 Cache (8 way)

Optimization Notice

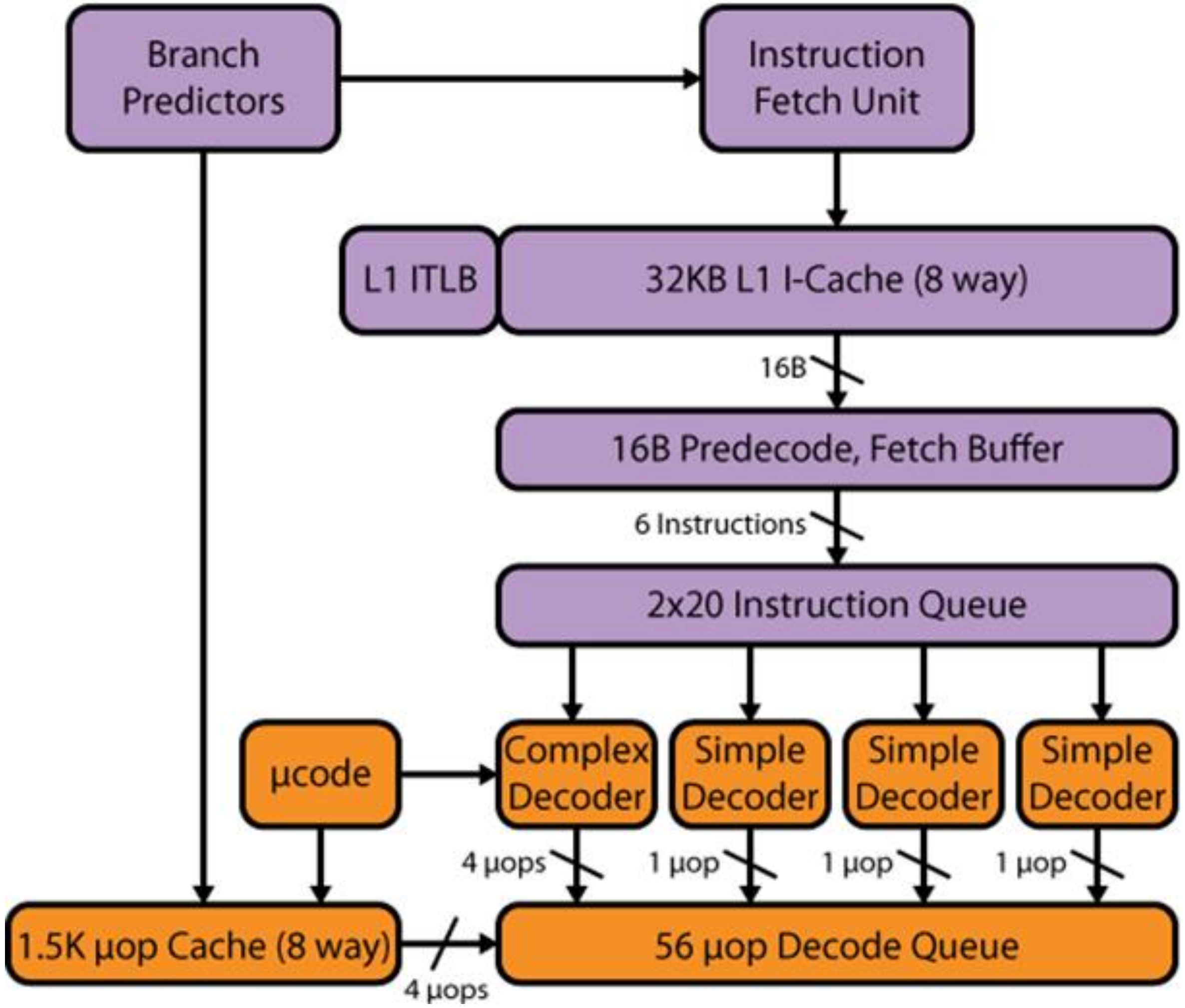
FrontEnd

- Instruction Fetch and Decode
  - 32 KB 8-way Icache
  - 4 decoders, up to 4 inst/cycle
  - CISC to RISC transformation
  - Decode Pipeline supports 16 bytes per cycle



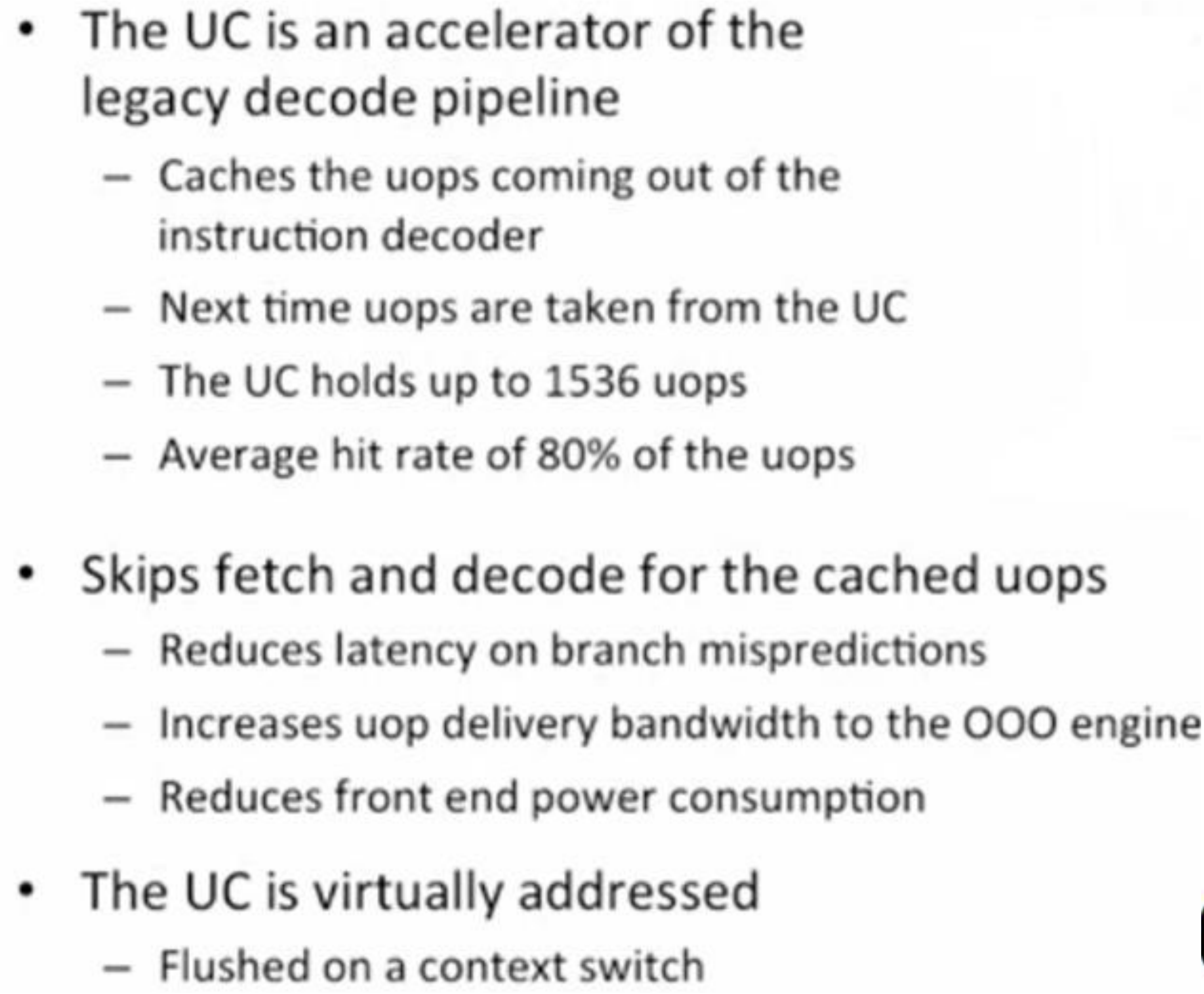
FrontEnd: Instruction Decode

- Four decoding units decode instructions into uops
  - The first can decode all instructions up to four uops in size
- Uops emitted by the decoders are directed to the Decode Queue and to the Decoded Uop Cache
- Instructions with >4 uoops generate their uops from the MSROM
  - The MSROM bandwith is 4 uops per cycle

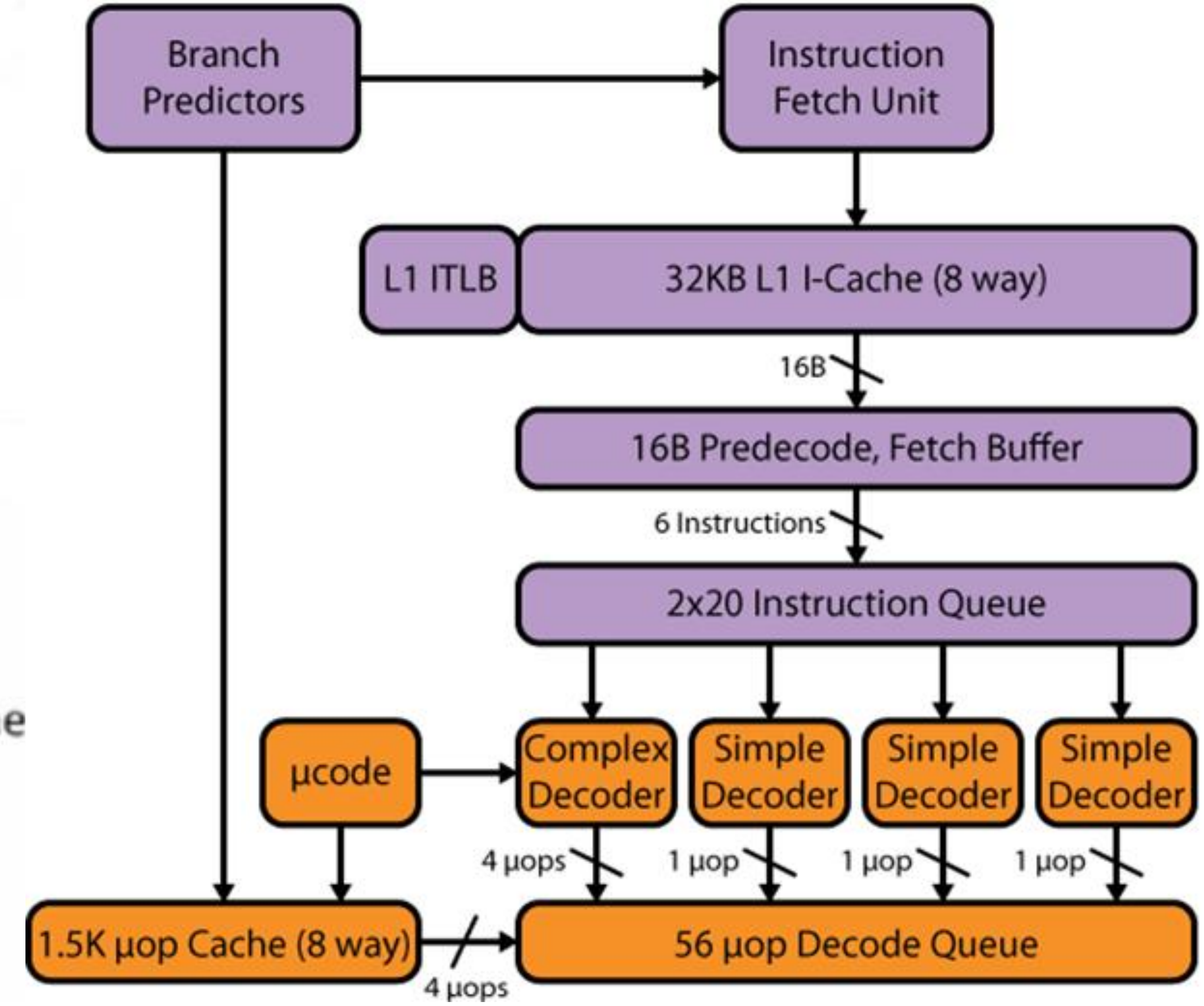


Optimization Notice

FrontEnd: Decode UOP Cache



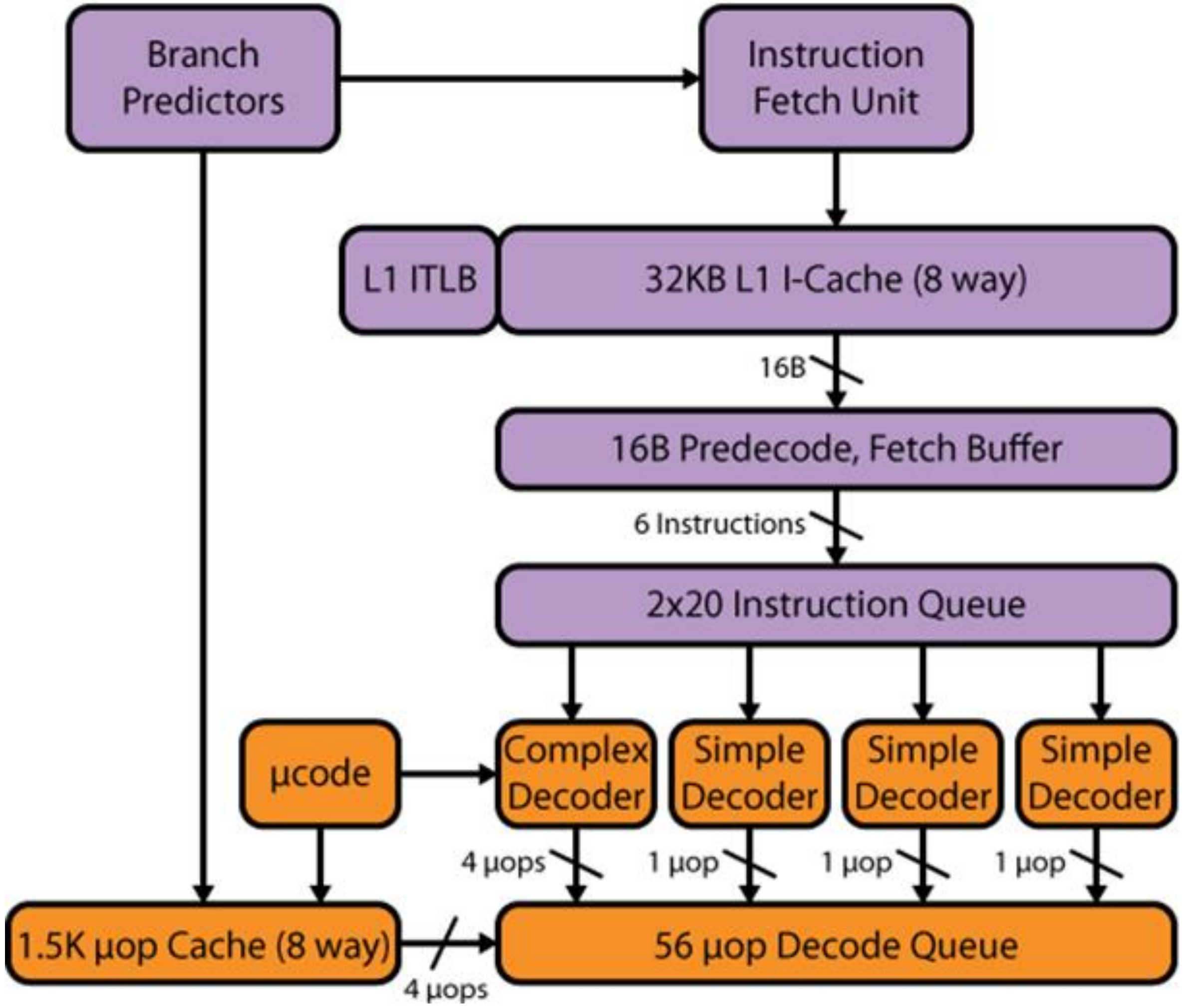
- The UC is an accelerator of the legacy decode pipeline
  - Caches the uops coming out of the instruction decoder
  - Next time uops are taken from the UC
  - The UC holds up to 1536 uops
  - Average hit rate of 80% of the uops
- Skips fetch and decode for the cached uops
  - Reduces latency on branch mispredictions
  - Increases uop delivery bandwidth to the OOO engine
  - Reduces front end power consumption
- The UC is virtually addressed
  - Flushed on a context switch



Optimization Notice

FrontEnd: Loop Stream Detector

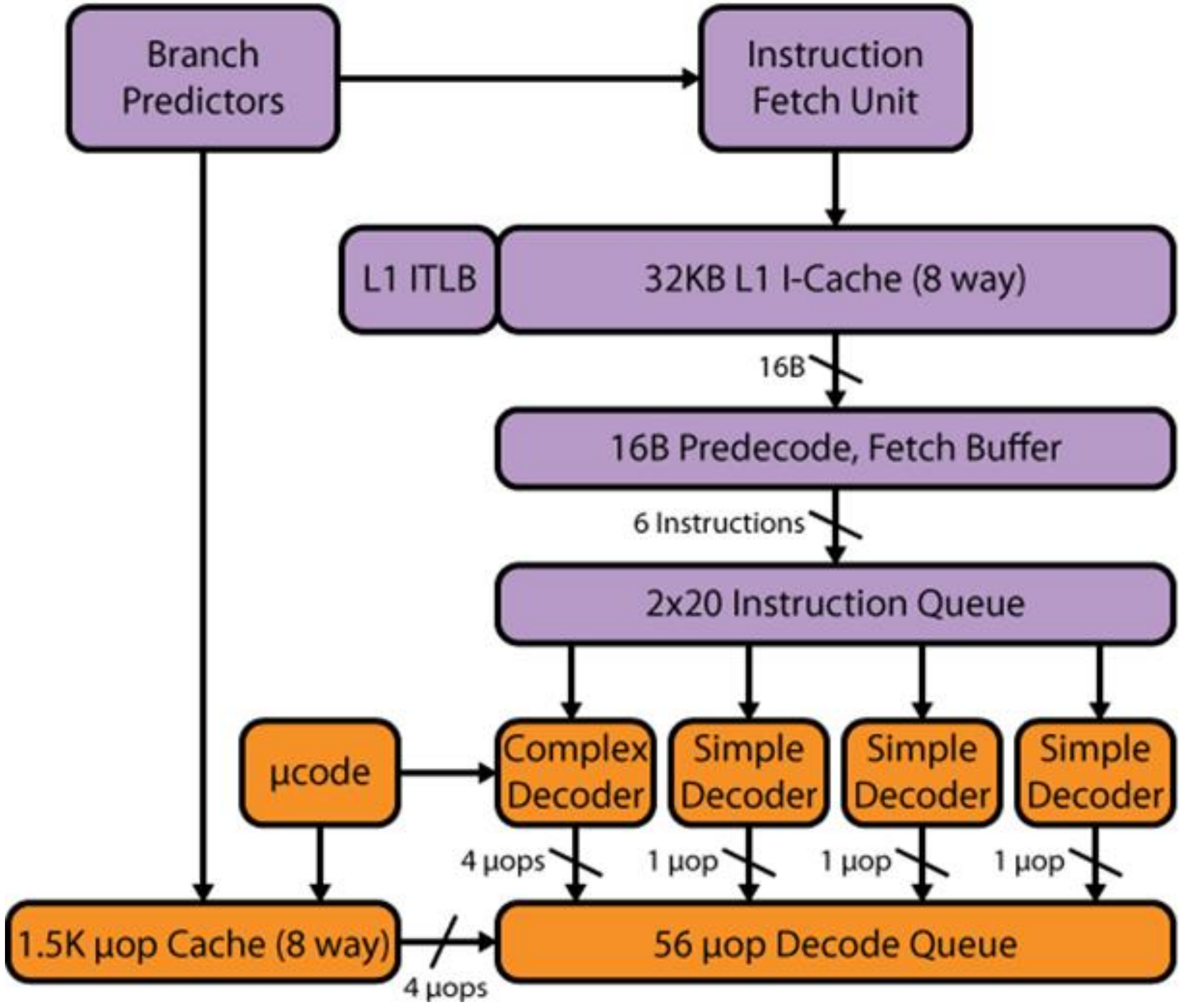
- LSD detects small loops that fit in the Decode Queue
  - The loop streams from the uop queue, with no more fetching, decoding, or reading uops from any of the caches
  - Works until a branch misprediction
- The loops with the following attributes qualify for LSD replay
  - Up to 56 uops
  - All uops are also resident in the UC
  - No more than eight taken branches
  - No CALL or RET
  - No mismatched stack operations (e.g. more PUSH than POP)



Optimization Notice

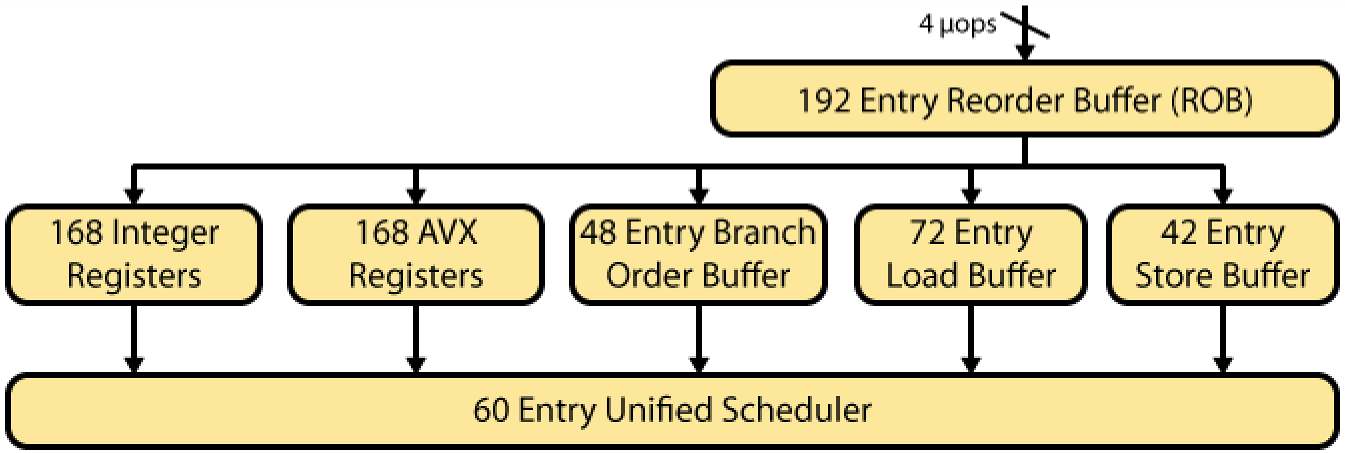
FrontEnd: Macro-Fusion

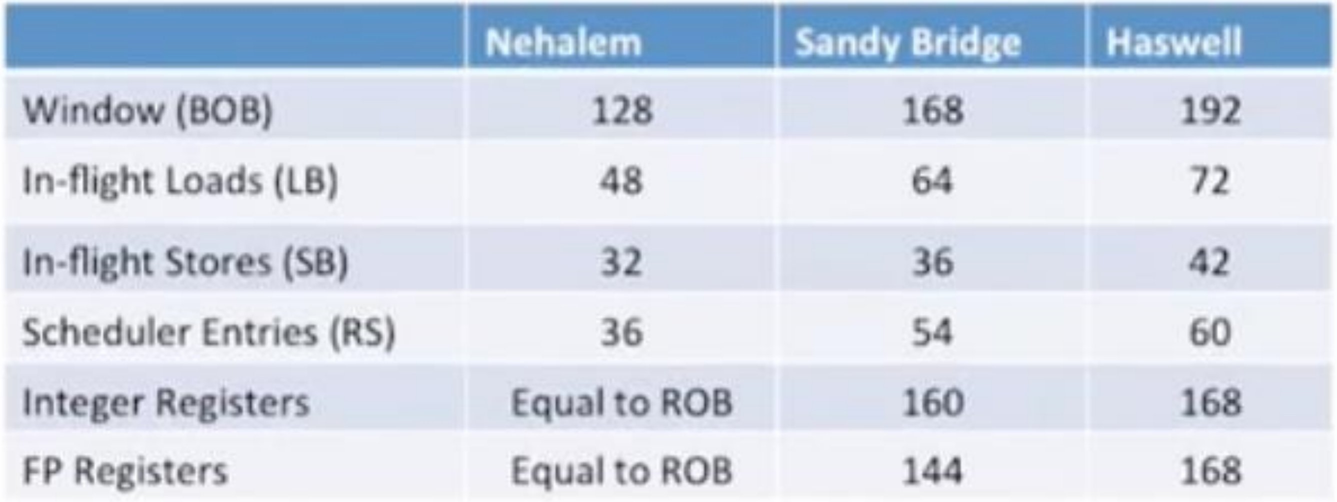
- Merge two instructions into a single uop
  - Increased decode, rename and retire bandwidth
  - Power savings from representing more work in fewer bits
- The first instruction of a macro-fused pair modifies flags
  - CMP, TEST, ADD, SUB, AND, INC, DEC
- The 2nd inst of a macro-fusible pair is a conditional branch
  - For each first instruction, some branches can fuse with it
- These pairs are common in many apps



Optimization Notice

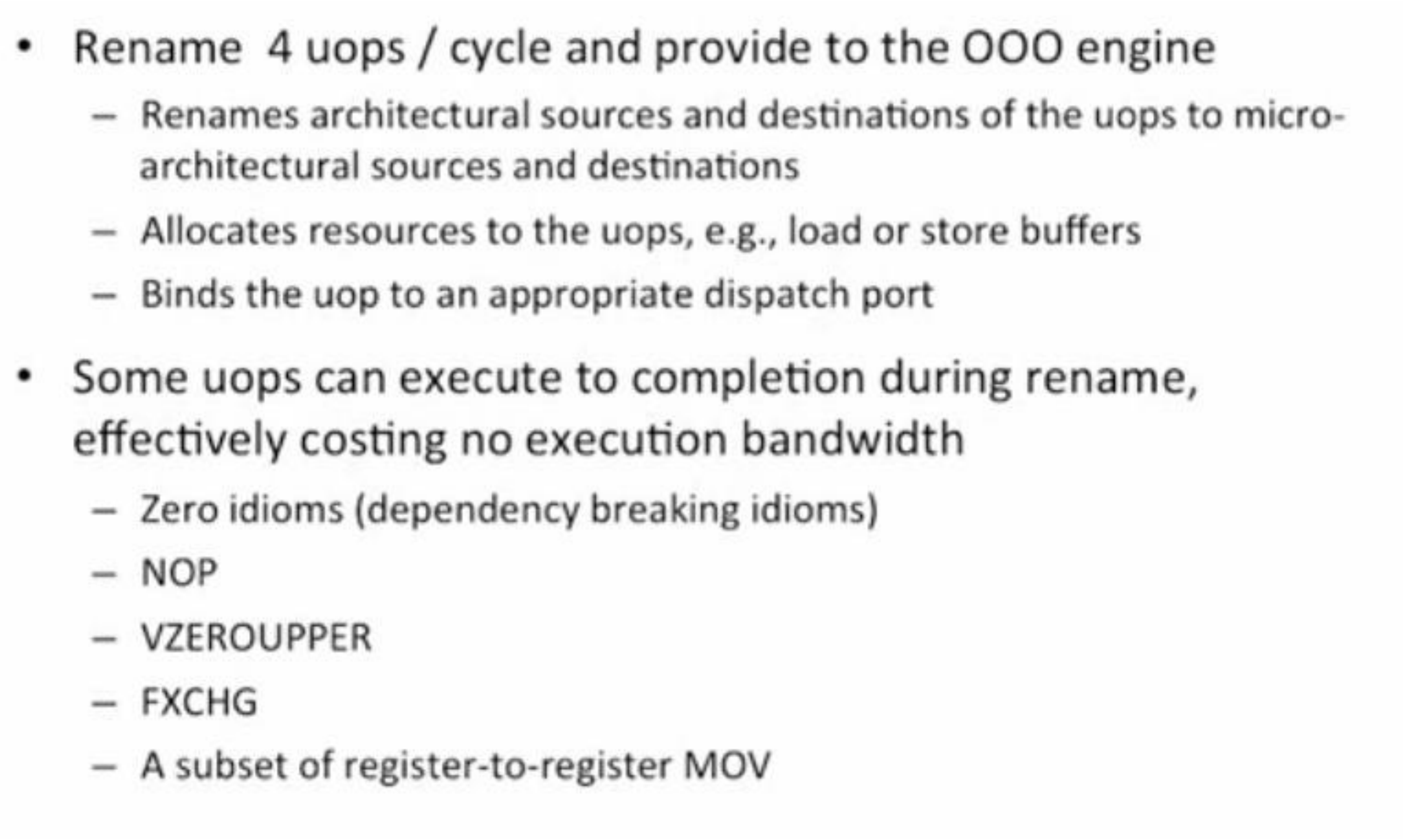
OOO Structures





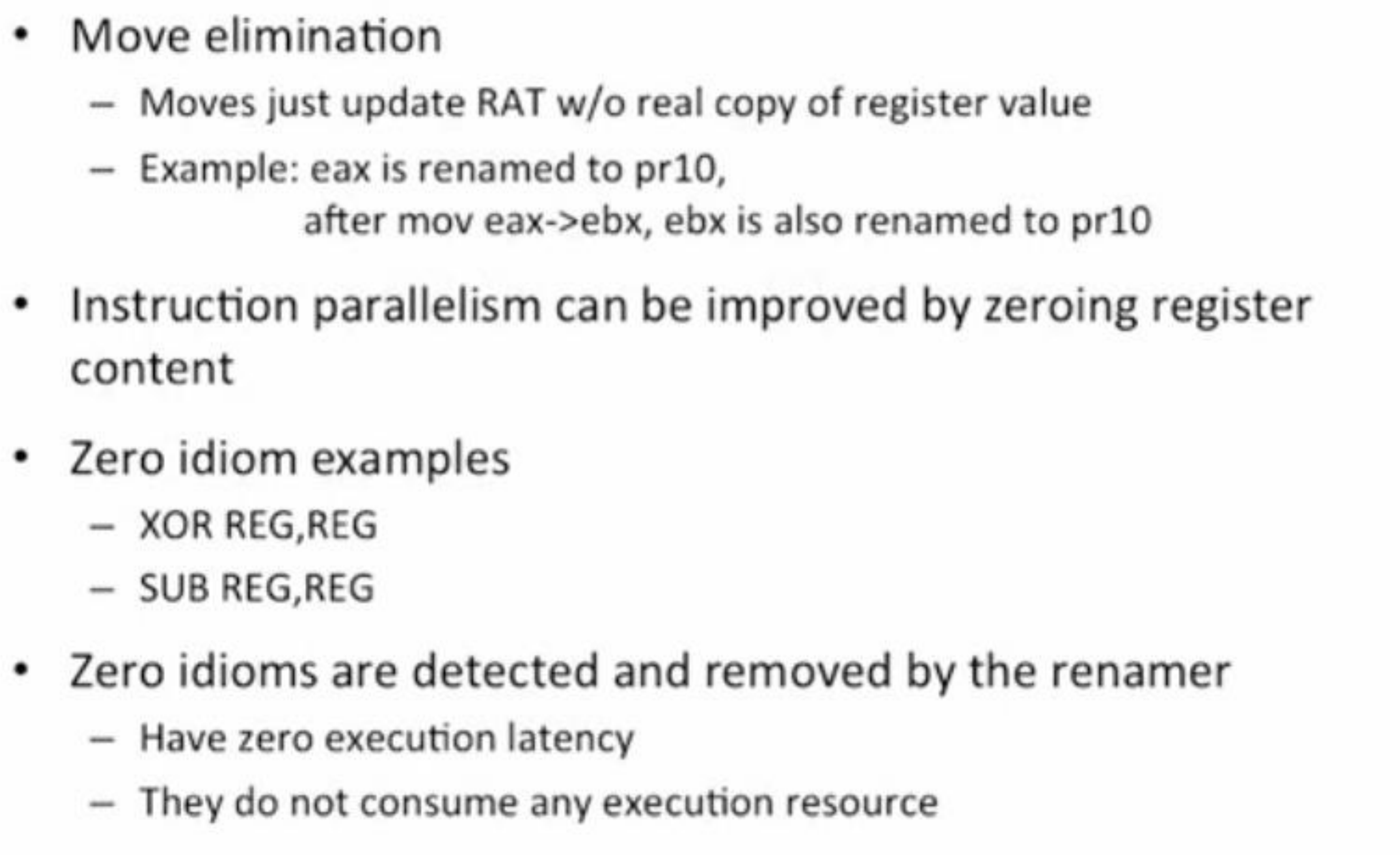
	Nehalem	Sandy Bridge	Haswell
Window (BOB)	128	168	192
In-flight Loads (LB)	48	64	72
In-flight Stores (SB)	32	36	42
Scheduler Entries (RS)	36	54	60
Integer Registers	Equal to ROB	160	168
FP Registers	Equal to ROB	144	168

OOO: Renamer



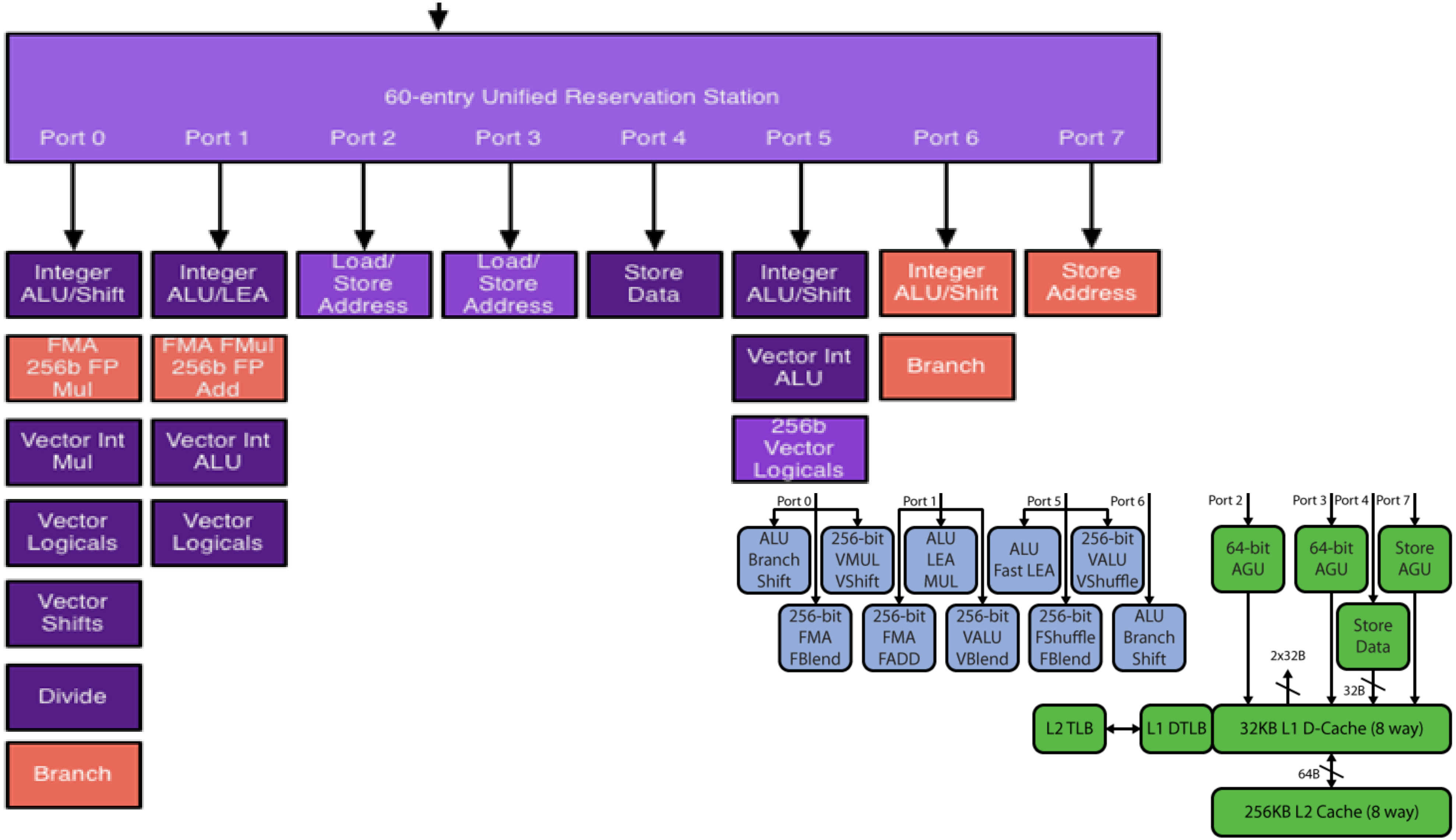
- Rename 4 uops / cycle and provide to the OOO engine
  - Renames architectural sources and destinations of the uops to micro-architectural sources and destinations
  - Allocates resources to the uops, e.g., load or store buffers
  - Binds the uop to an appropriate dispatch port
- Some uops can execute to completion during rename, effectively costing no execution bandwidth
  - Zero idioms (dependency breaking idioms)
  - NOP
  - VZEROUPPER
  - FXCHG
  - A subset of register-to-register MOV

OOO: Dependency Breaking Idiom

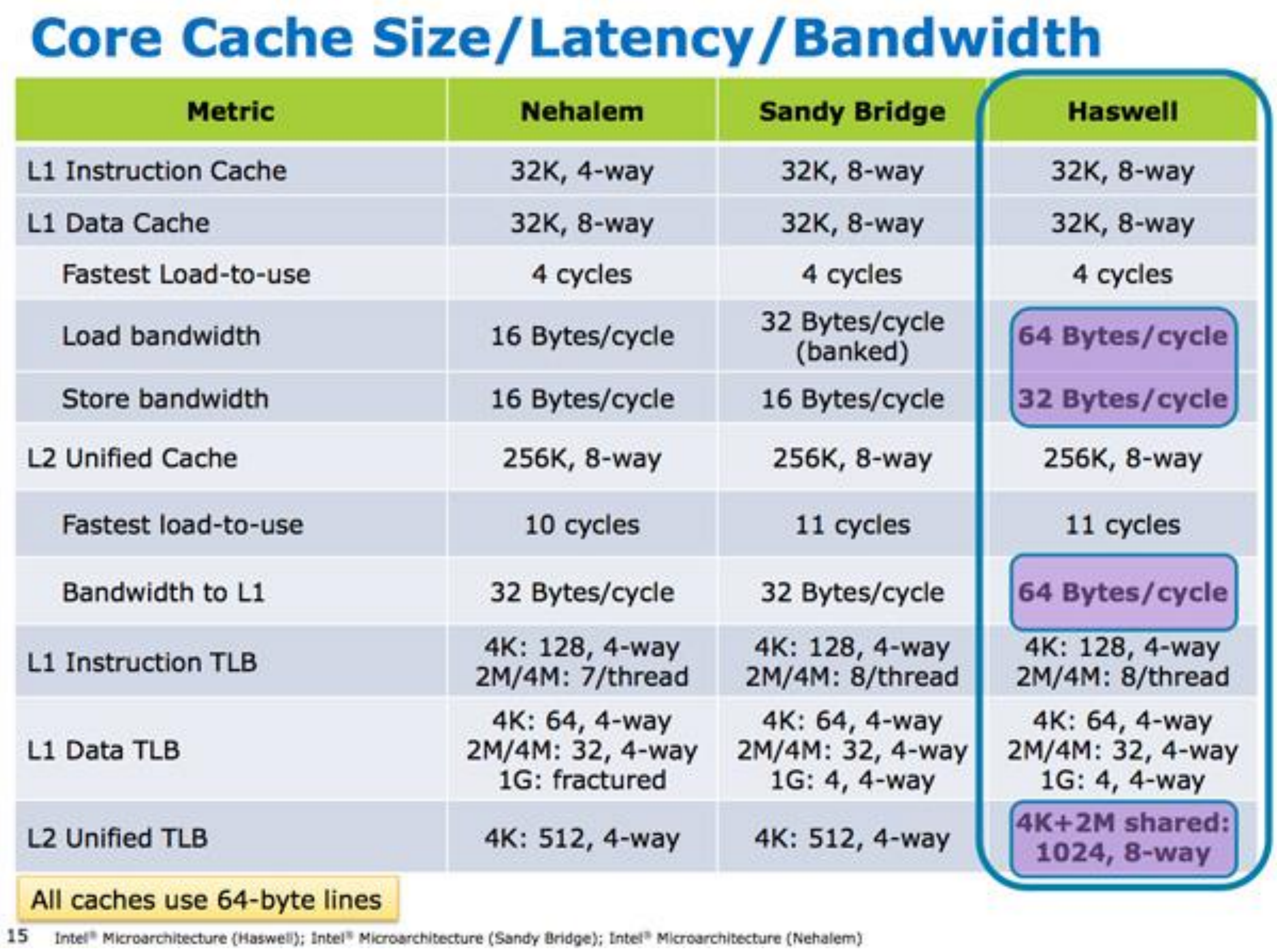


- Move elimination
  - Moves just update RAT w/o real copy of register value
  - Example: eax is renamed to pr10,
    after mov eax->ebx, ebx is also renamed to pr10
- Instruction parallelism can be improved by zeroing register content
- Zero idiom examples
  - XOR REG,REG
  - SUB REG,REG
- Zero idioms are detected and removed by the renamer
  - Have zero execution latency
  - They do not consume any execution resource

EXE



Optimization Notice



Core Cache Size/Latency/Bandwidth

Metric	Nehalem	Sandy Bridge	Haswell
L1 Instruction Cache	32K, 4-way	32K, 8-way	32K, 8-way
L1 Data Cache	32K, 8-way	32K, 8-way	32K, 8-way
Fastest Load-to-use	4 cycles	4 cycles	4 cycles
Load bandwidth	16 Bytes/cycle	32 Bytes/cycle (banked)	**64 Bytes/cycle**
Store bandwidth	16 Bytes/cycle	16 Bytes/cycle	**32 Bytes/cycle**
L2 Unified Cache	256K, 8-way	256K, 8-way	256K, 8-way
Fastest load-to-use	10 cycles	11 cycles	11 cycles
Bandwidth to L1	32 Bytes/cycle	32 Bytes/cycle	**64 Bytes/cycle**
L1 Instruction TLB	4K: 128, 4-way 2M/4M: 7/thread	4K: 128, 4-way 2M/4M: 8/thread	4K: 128, 4-way 2M/4M: 8/thread
L1 Data TLB	4K: 64, 4-way 2M/4M: 32, 4-way 1G: fractured	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way	4K: 64, 4-way 2M/4M: 32, 4-way 1G: 4, 4-way
L2 Unified TLB	4K: 512, 4-way	4K: 512, 4-way	**4K+2M shared: 1024, 8-way**

All caches use 64-byte lines

15 Intel® Microarchitecture (Haswell); Intel® Microarchitecture (Sandy Bridge); Intel® Microarchitecture (Nehalem)

Optimization Notice