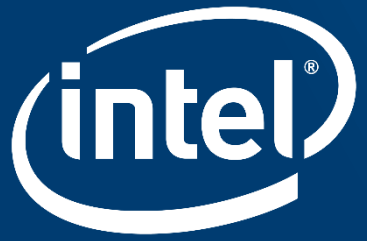


CRASH COURSE ON COMPUTER ARCHITECTURE

Areg Melik-Adamyan, PhD

Engineering Manager, Intel Developer Products Division

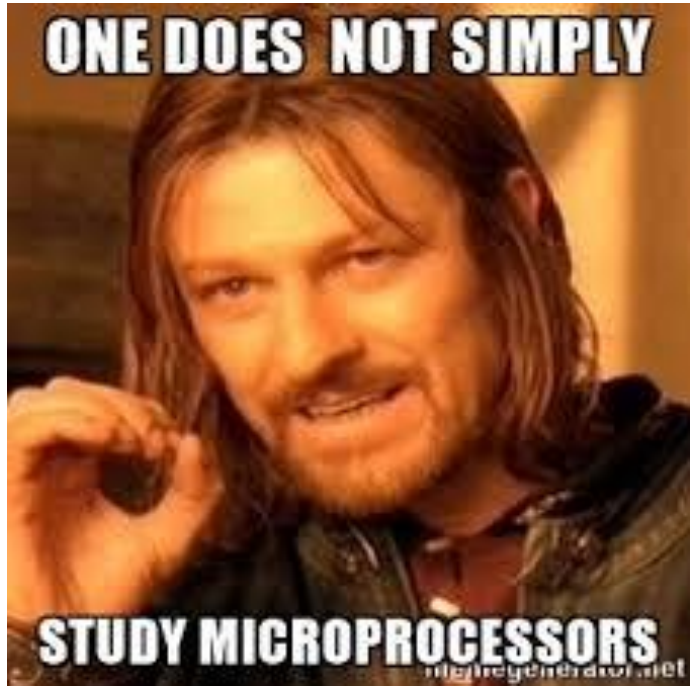
Introduction

Who am I?

- 7 years at Intel, 17 years in industry
- Managing compiler teams (GCC, Go)
- 10 years teaching

Why we are here?

- To better understand how CPU works



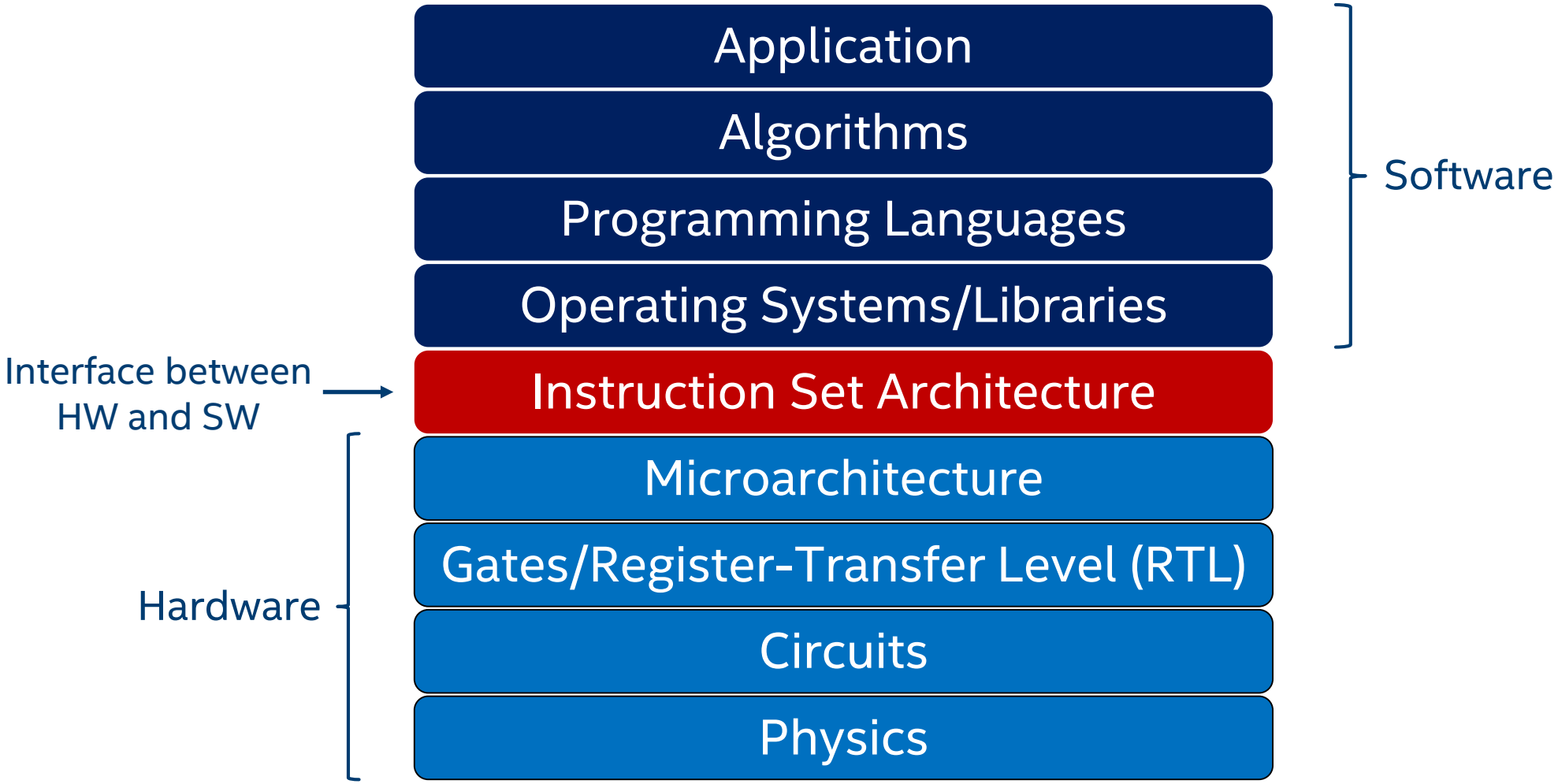
Texbooks and References

- 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

Lecture Outline

- Pipeline
- Memory Hierarchy (Caches: +1 lecture later)
- Out-of-order execution
- Branch prediction
- Real example: Haswell Microarchitecture

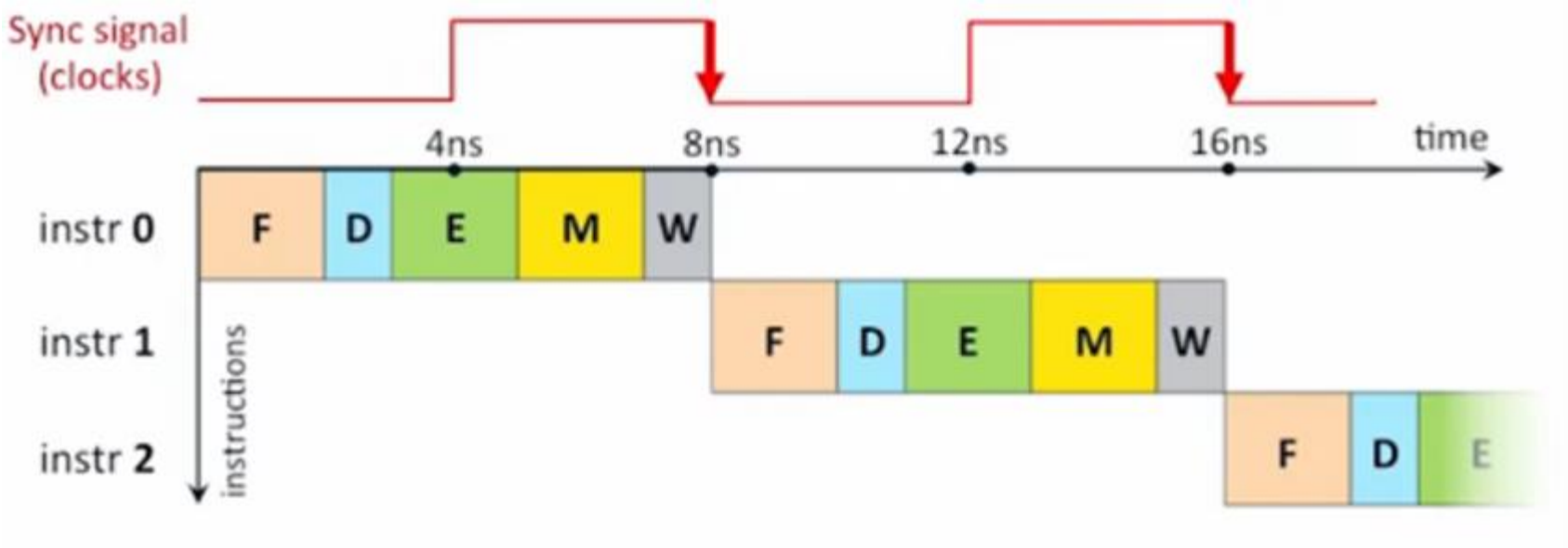
Layers of Abstraction



Basic CPU Actions

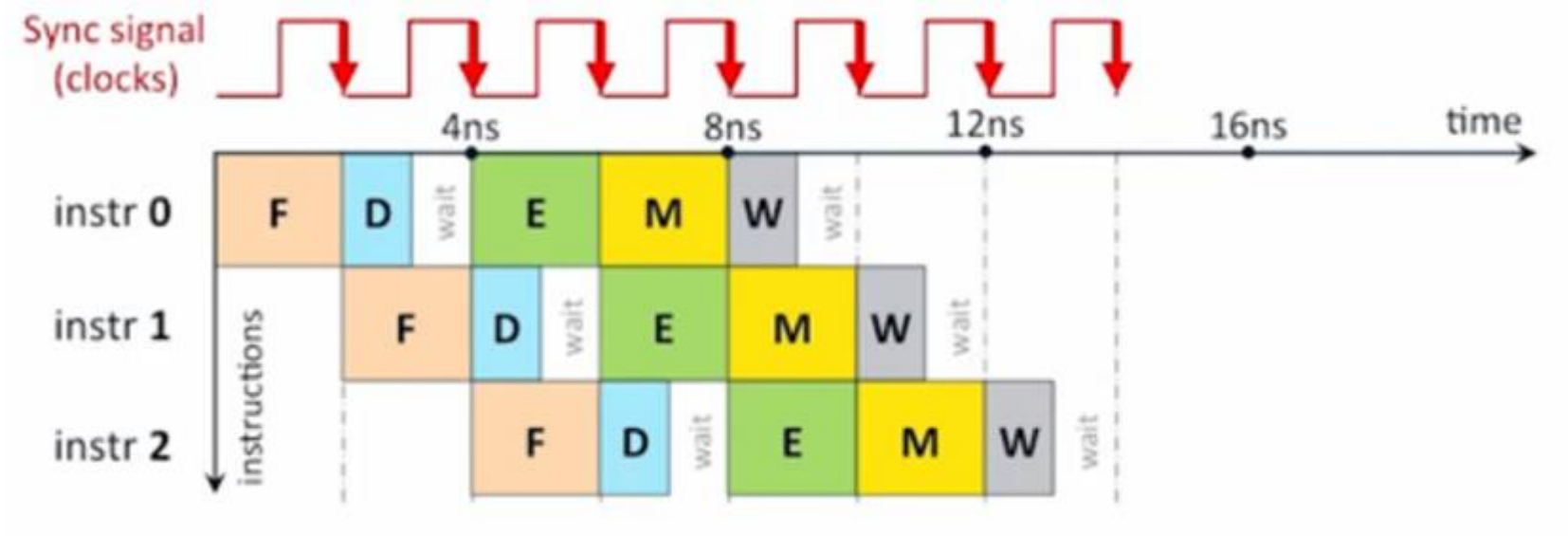
1. Fetch instruction by PC from memory
2. Decode it and read its operands from registers
3. Execute calculations
4. Read/write memory
5. Write the result into registers and update PC

Non-Pipelined Processing



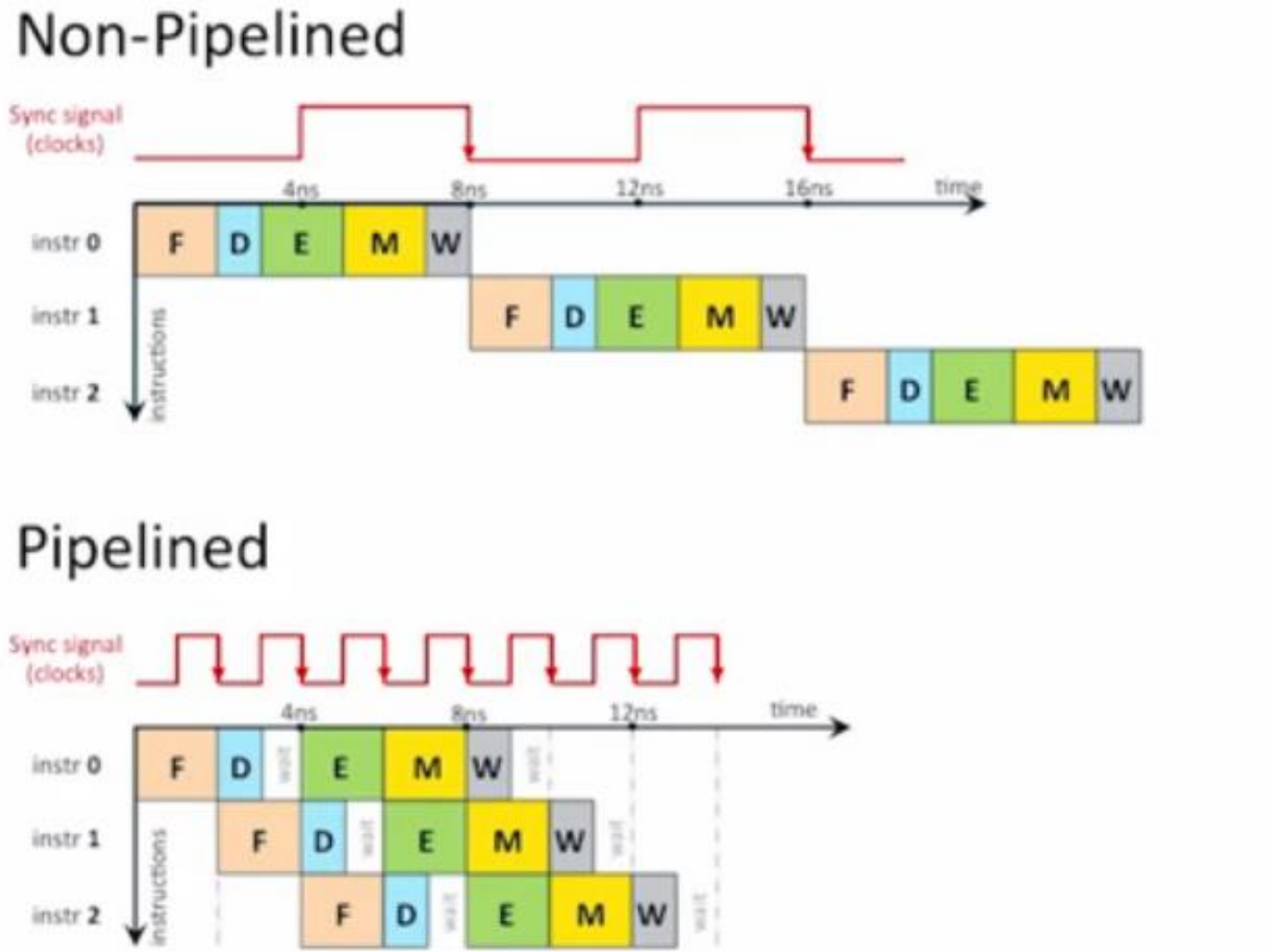
- Instructions are processed sequentially, one per cycle
- How to speed-up?
  - SW: decrease number of instructions
  - HW: decrease the time to process one instruction or overlap their processing. i.e. make pipeline

Pipeline

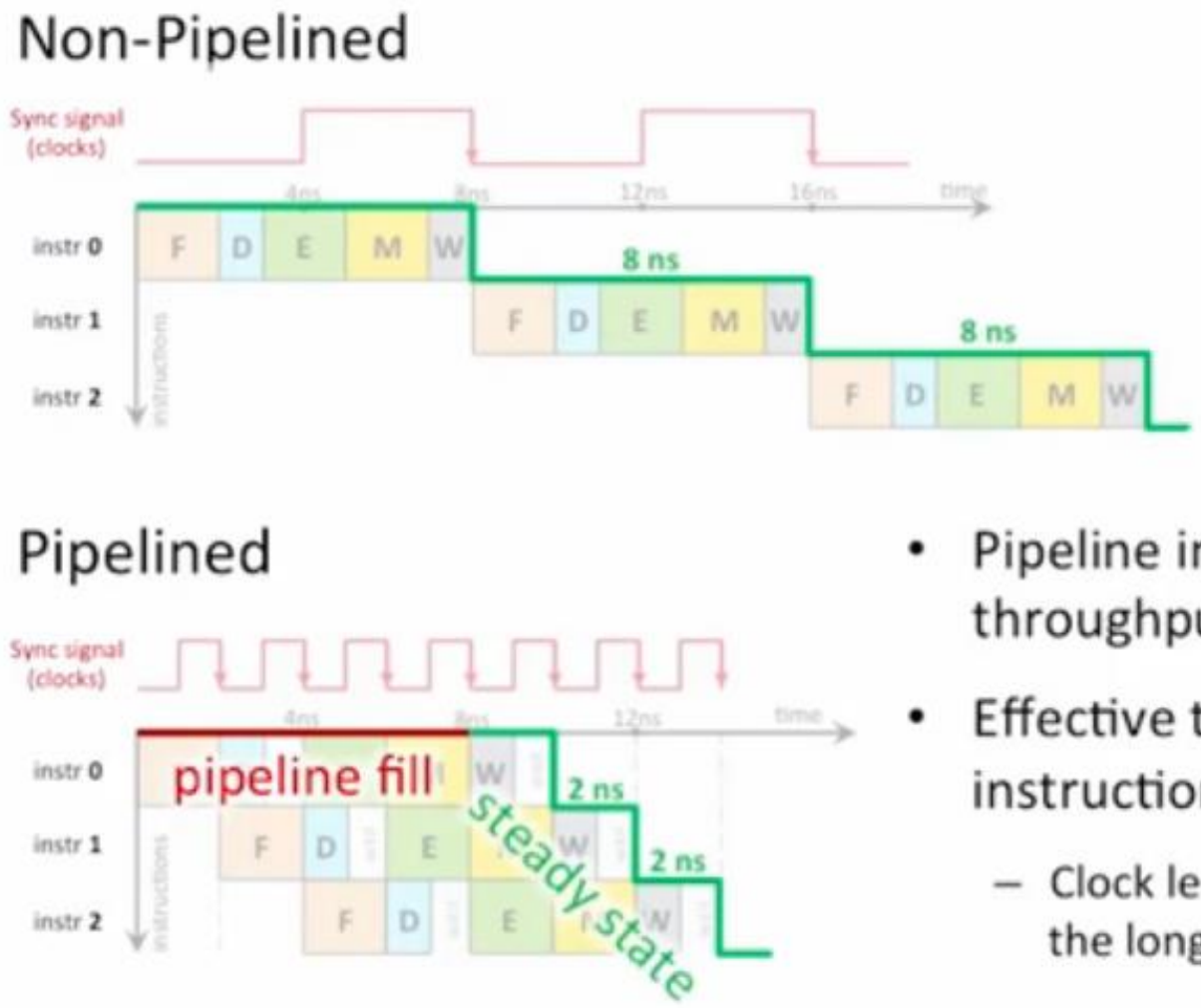


- Processing is split into several steps called “stages”
  - Each stage takes one cycle
  - The clock cycle is determined by the longest stage
- Instructions are overlapped
  - A new instruction occupies a stage as soon as the previous one leaves it

Pipeline vs Non-Pipeline



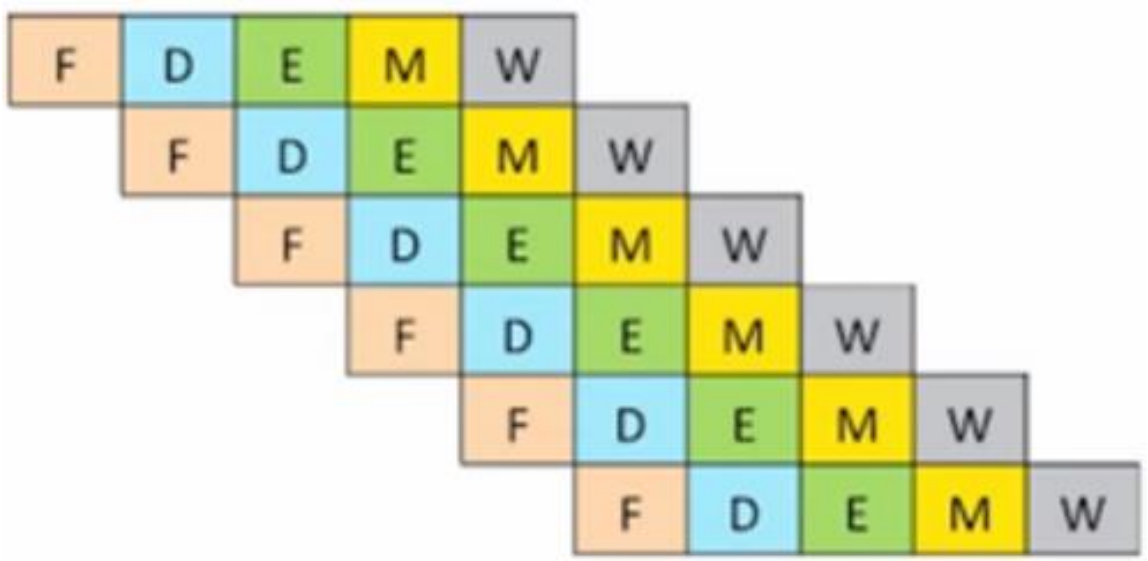
Pipeline vs Non-Pipeline



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

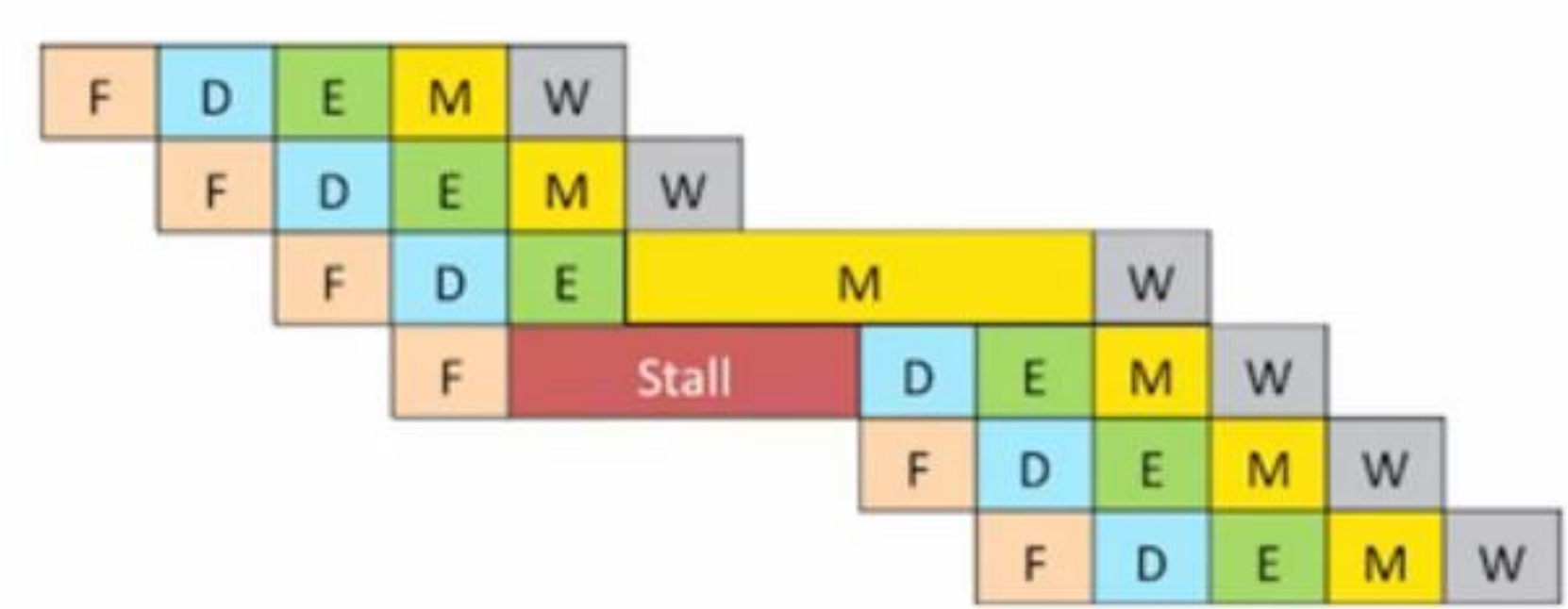
Pipeline Limitations

- 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

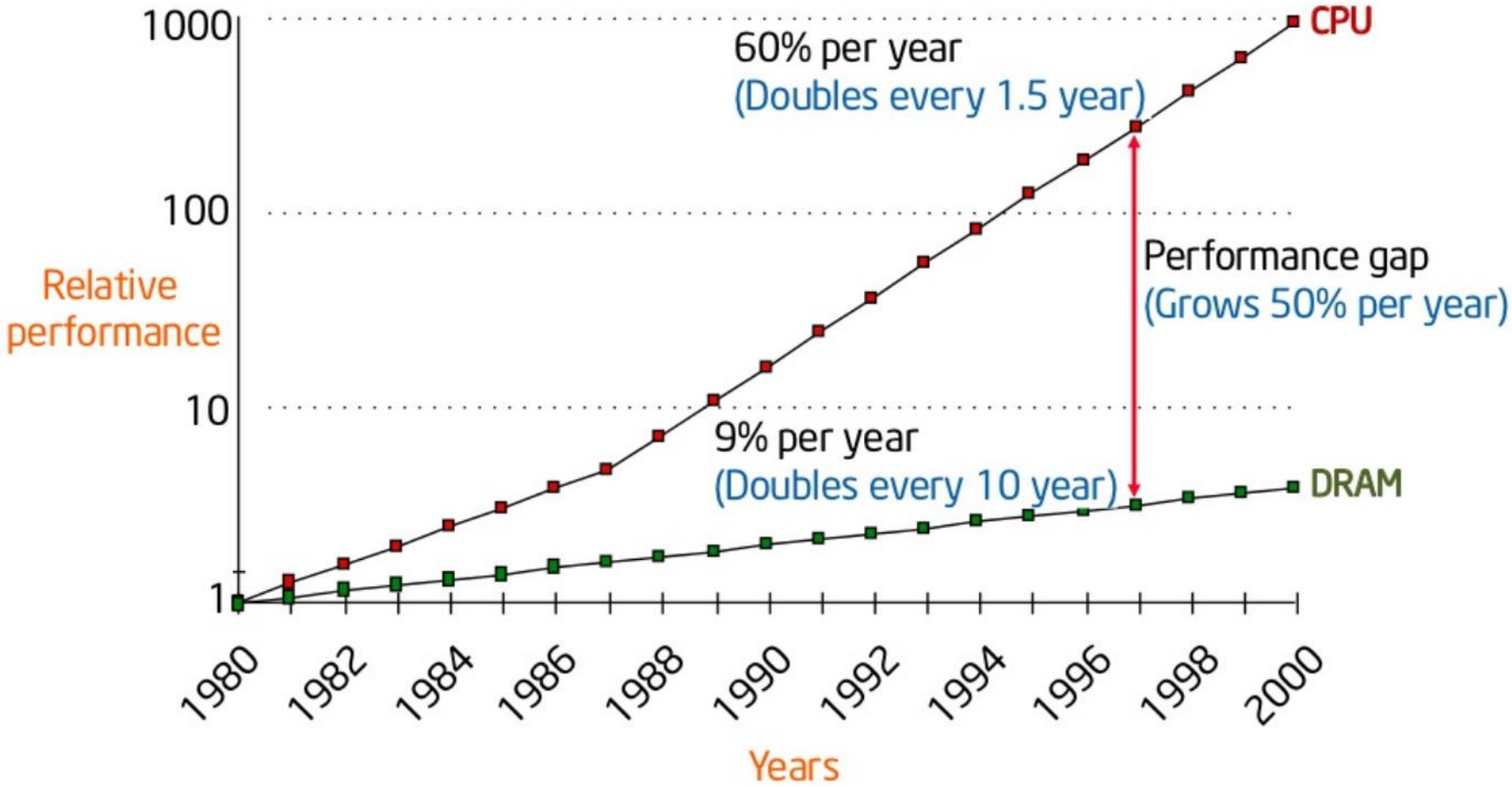


Pipeline Limitations

- Various types of hazards:
  - read after write (RAW), a *true dependency*
  - write after read (WAR), an *anti-dependency*
  - write after write (WAW), an *output dependency*

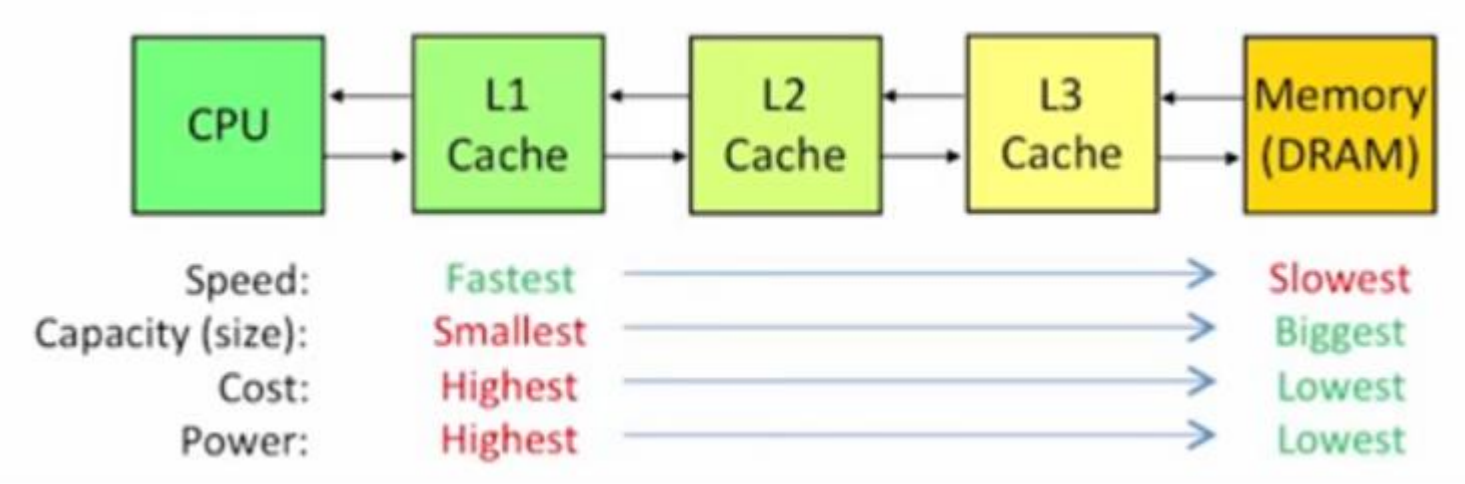


Motivation for Memory Hierarchy



Memory Tradeoffs

- Large memories are slow
- Small memories are fast, but expensive and consume high power
- **Goal:** give the processor a feeling that it has memory which is fast, large, cheap and consumes low energy
- **Solution**: Hierarchy of Memories



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?

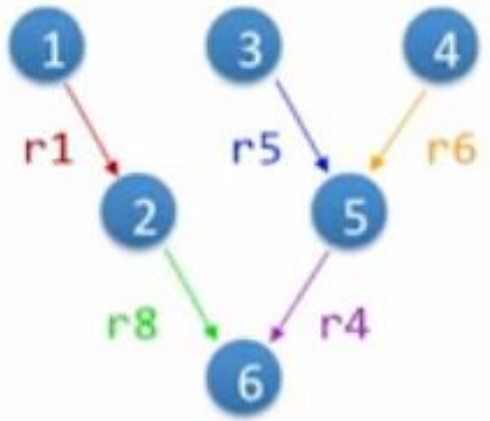
- Theoretically can execute multiple instructions in parallel
  - Wide pipeline => more performance
- But…
  - Only independent subsequent instructions can be executed in parallel
  - Whereas subsequent instructions are often dependent
  - So the utilization of the second pipe is often low
- Solution: **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

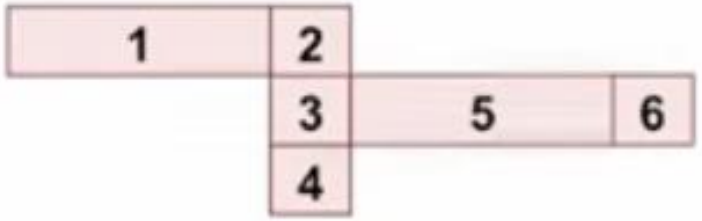
Example:

```
(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
```

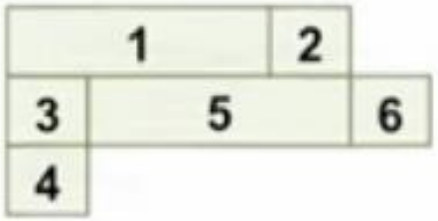
Data Flow Graph



In-order execution

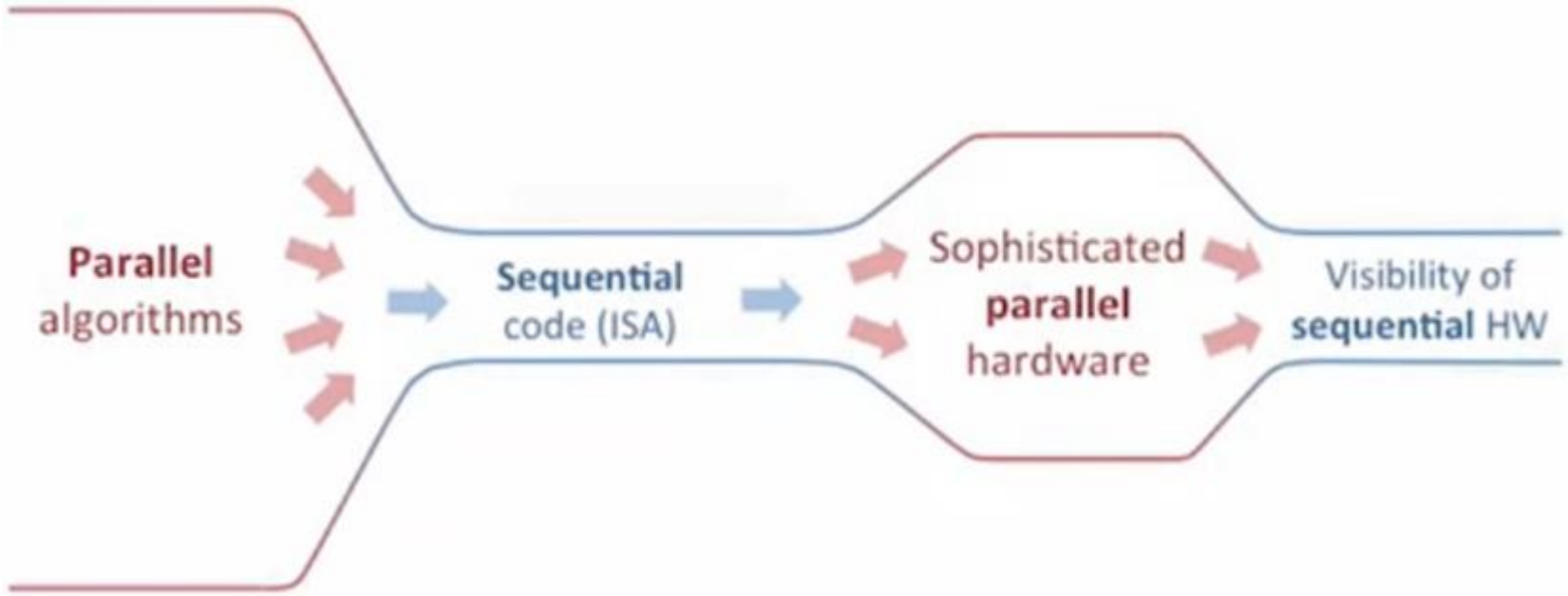


Out-of-order execution



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

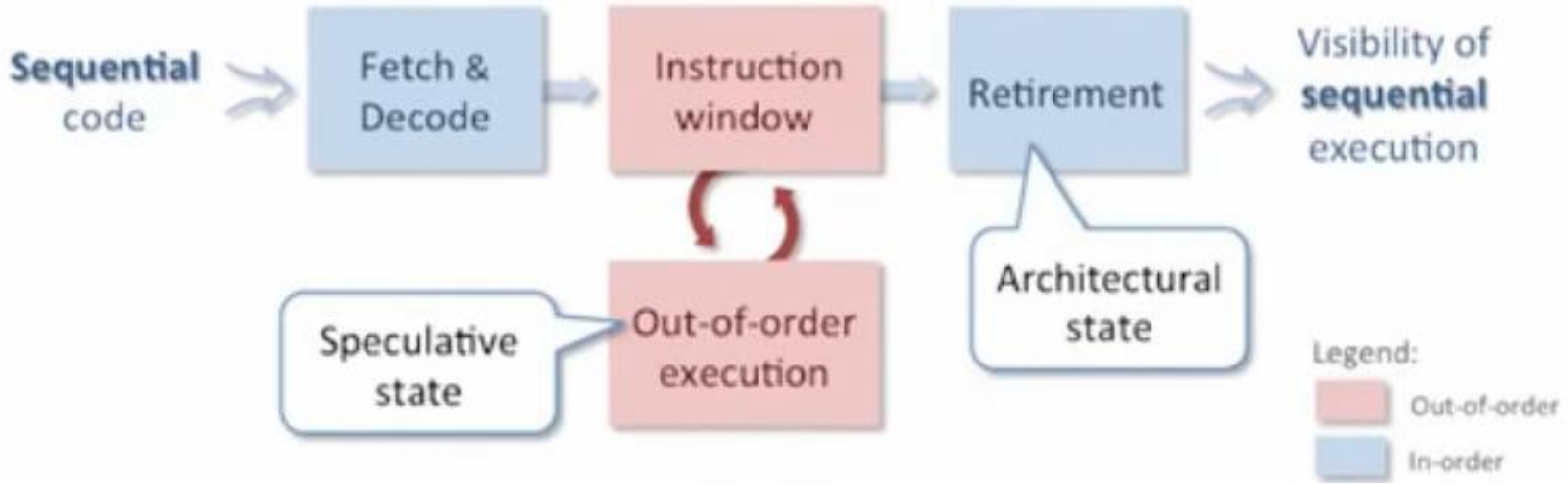
Load A returns new data, Load B returns old data = **NOT ALLOWED**

LD A	LD B
ST B	ST A

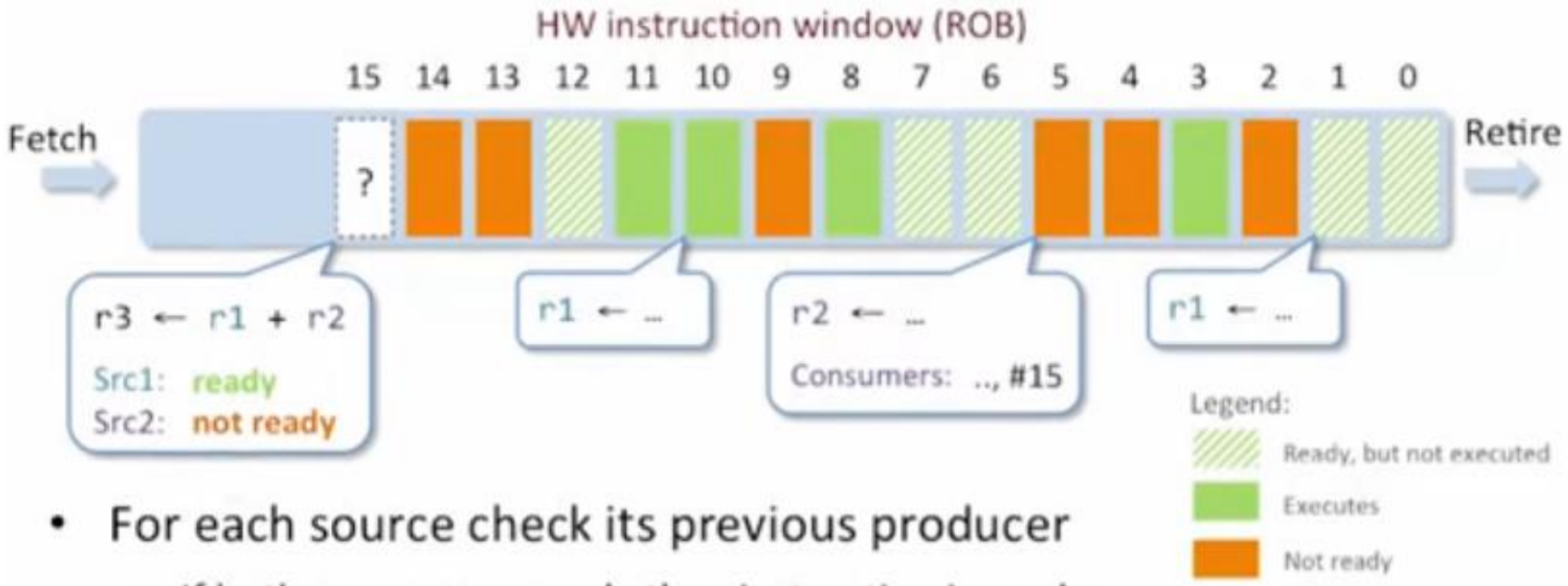
Both loads return new data = **NOT ALLOWED**

Maintaining Architectural State

- **Solution:** support two state, speculative and architectural
- Update arch state in program order using special buffer called ROB (**reo**rder **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



Dependency Check



- For each source check its previous producer
  - If both sources are ready then instruction is ready
  - If a source is not ready, write the instr# into the consumer list of producer
- When an instruction becomes ready, send a signal to all consumers that their sources become ready too
- For loads need also to check addresses of all previous stores

How Large Windows Should Be?

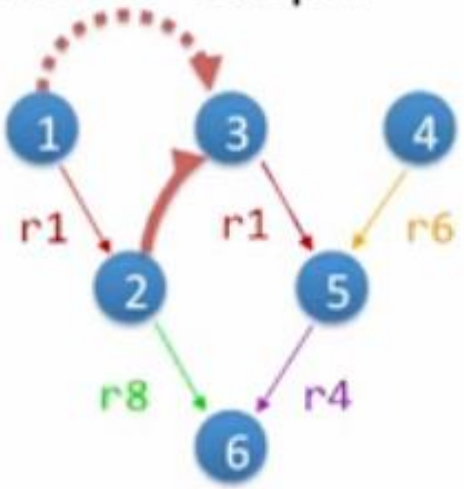
- In short, the large window → the better
  - Find more independent instructions
  - Hide longer latencies (e.g., cache misses, long operations)
- Example
  - The modern CPU has a window of 200
  - If we want execute 4 instruction per cycle, then we can hide latency of 50 cycles
  - It is enough to hide L1 and L2 misses, but not L3 miss
- But, there are limitation to find independent instructions in a large window:
  - **branches** and **false dependencies**

Limitation: False Dependencies

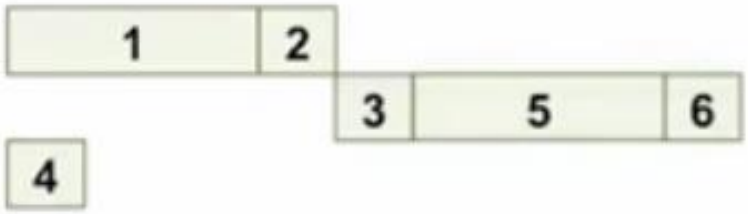
Example:

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

Data Flow Graph



Out-of-order execution



False Dependencies:

- Write-After-Write: (1) → (3)
- Write-After-Read: (2) → (3)

Register Renaming

- Redo register allocation that was done by compiler
- Eliminate all false dependencies

Example:

```
(1) pr10 ← r4 / r7
(2) pr11 ← pr10 + r2
(3) pr12 ← r5 + 1
(4) pr13 ← r6 - r3
(5) pr14 ← load pr12 + pr13
```

Renaming

```
pr10 ≡ r1
pr11 ≡ r8
pr12 ≡ r1
pr13 ≡ r6
pr14 ≡ r4
```

Register Aliases Table (RAT)

	r0	r1	r2	r3	r4	r5	r6	r7	r8
		pr12			pr14		pr13		pr11

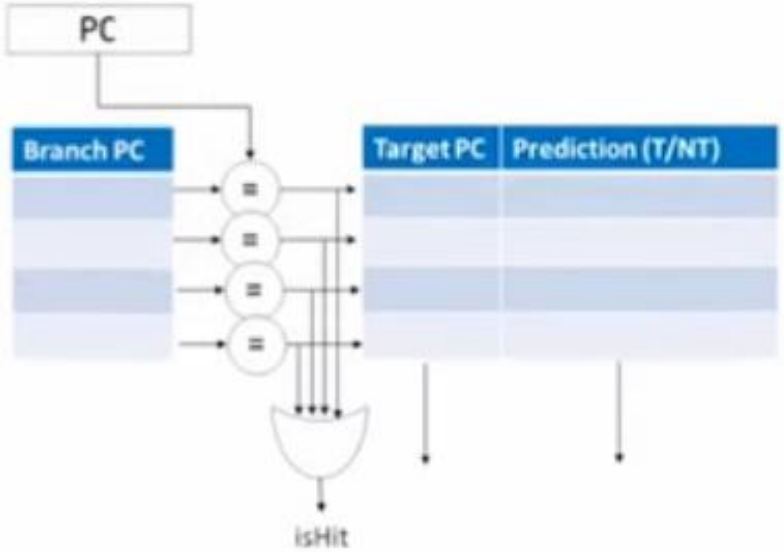
Limitation: Branches

- How to fill a large window from a single sequential instruction stream in presence of branches?

- How harmful are branches?
  - In average, each 5th instruction is a branch
  - If follow one branch path randomly, then accuracy is 50%
  - The probability that 100th instruction in the window will not be removed is (50%)^20 = 0.0001%
- Need significantly increase accuracy!

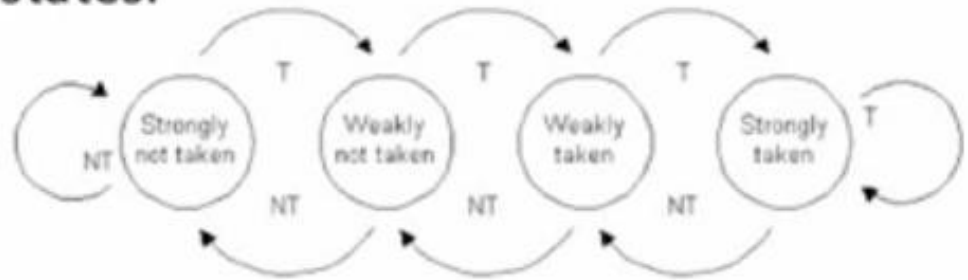
Dynamic Branch Prediction

- 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?

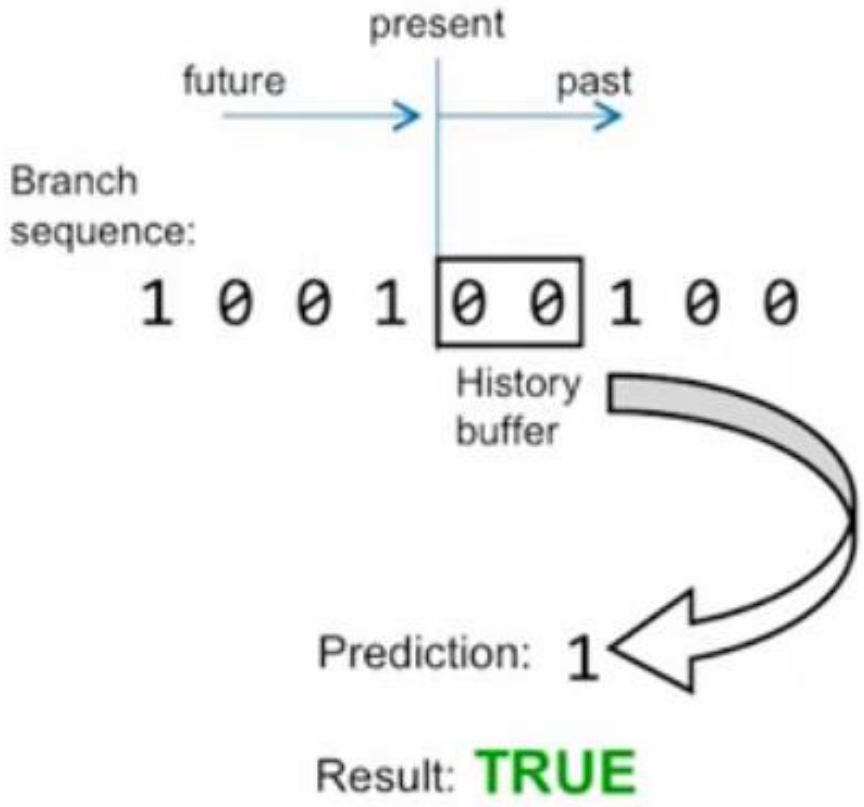
- A saturating counter or bimodal predictor is a state machine with four states:

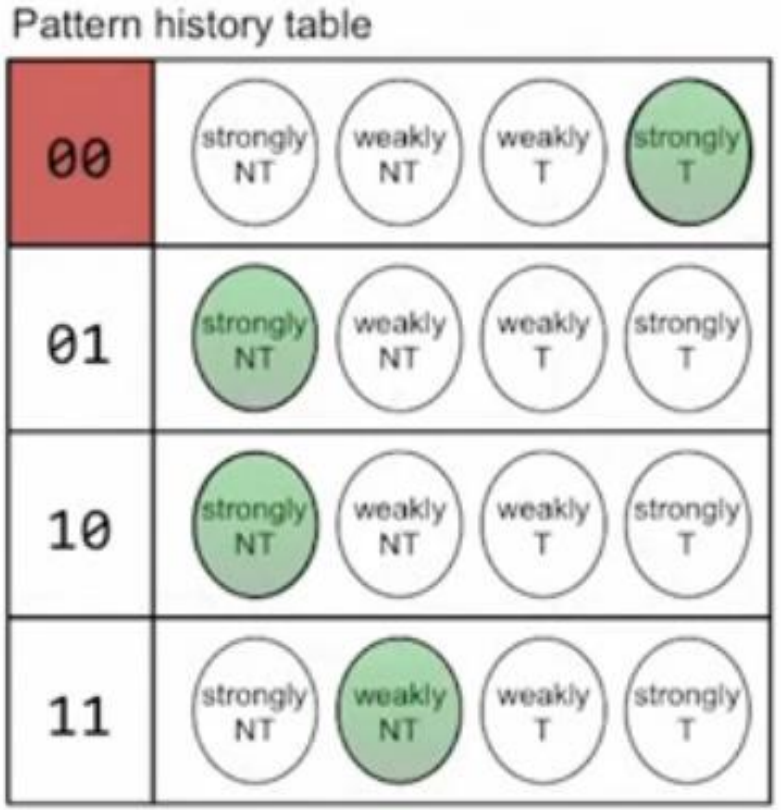


- Why four states?
  - Bimodal predictor make only one mistake on a loop back branch (on the loop exit)
- Advantages:
  - Small – only 2 bits per branch
  - Predicts well branches with stable behaviour
- Disadvantages
  - Cannot predict well branches which often change their outcome:
    - e.g. T, NT,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

Using History Patterns

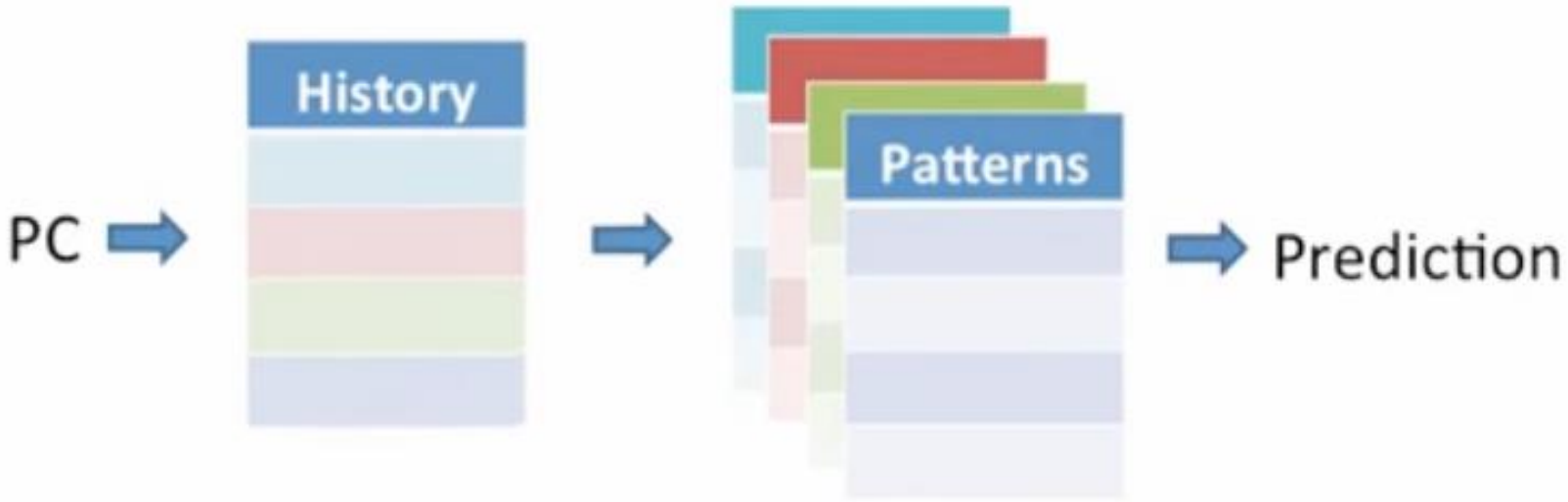
- Remember not just most often outcome, but most often outcome after certain history patterns





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)
{
  …
}
```

Concepts Covered

- Advantages of OOO Execution
  - Help to exploit Instruction Level Parallelism (ILP)
  - Help to hide latencies (e.g., cache miss, divide)
  - Superior/complementary to the compiler

- Complex HW
  - Requires reconstruction of original order
  - Complex dependency check logic
  - Register renaming
  - Branch prediction and Speculative Execution

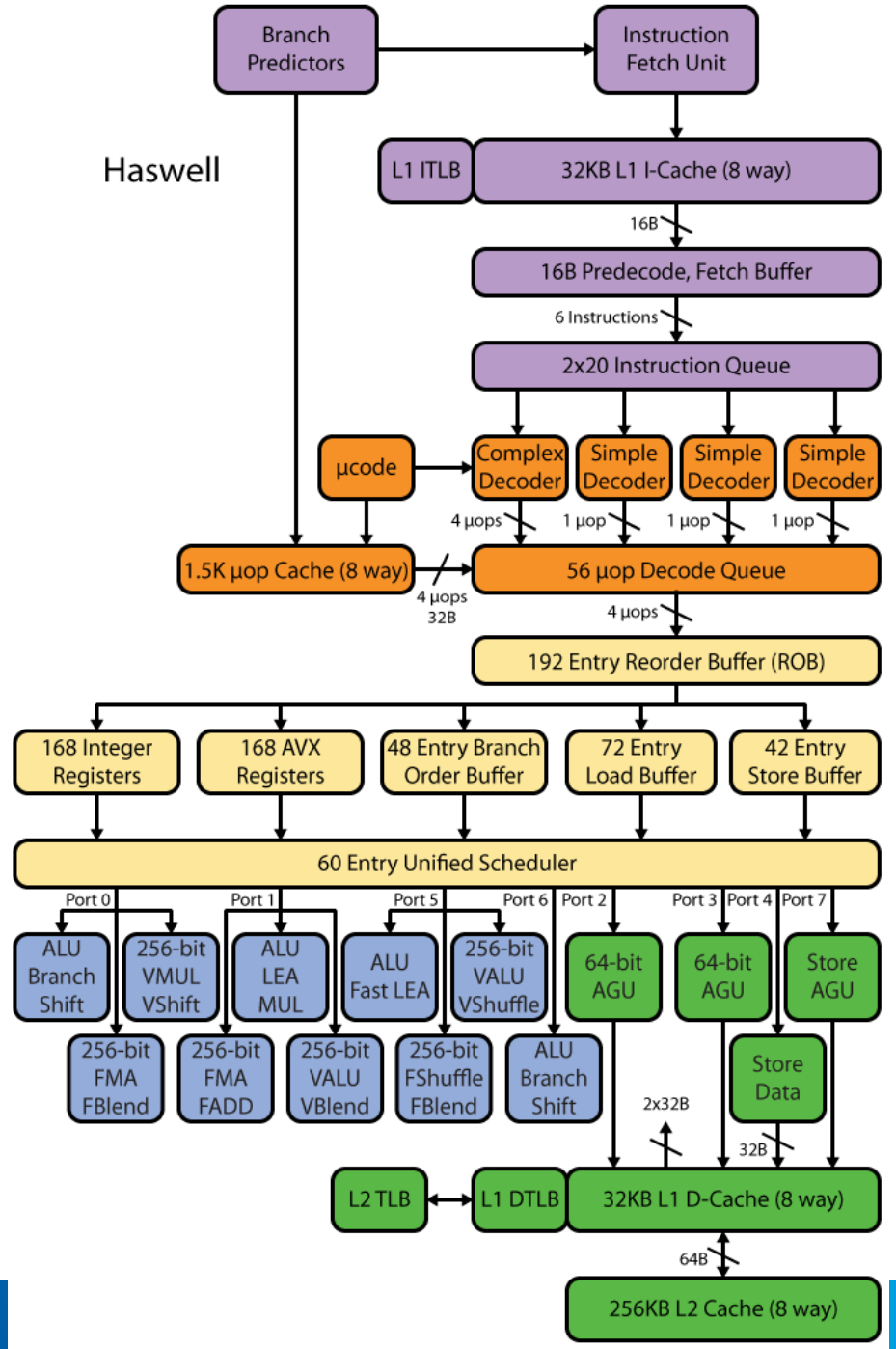
Intel Processor Roadmap

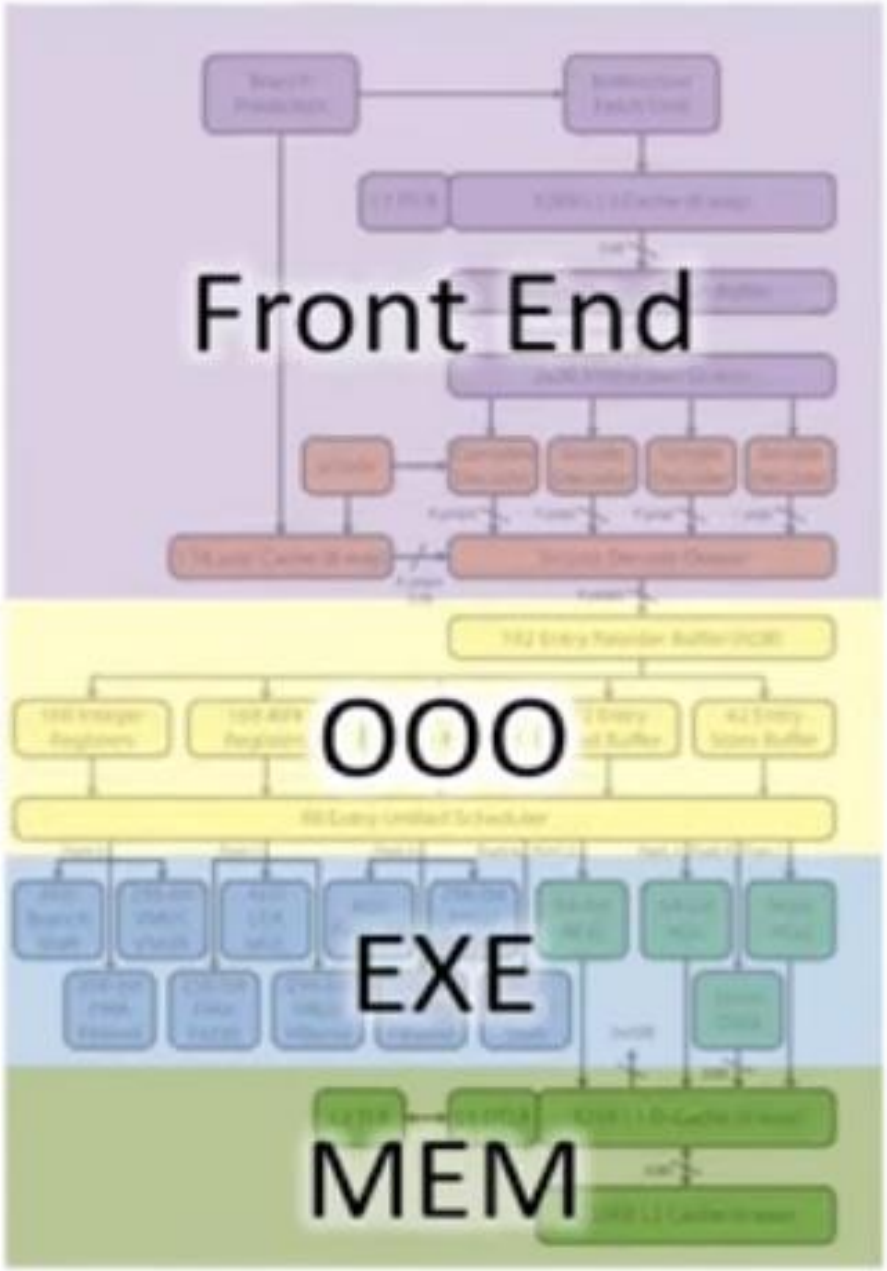
Year	2008	2010	2011	2012	2013	2014	2015	2016
uArch Name	Nehalem		Sandy Bridge		Haswell		Skylake	
Tech Process	45 nm	32 nm		22 nm		14 nm		10 nm
Name	Nehalem	Westmere	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Skylake	Cannonlake

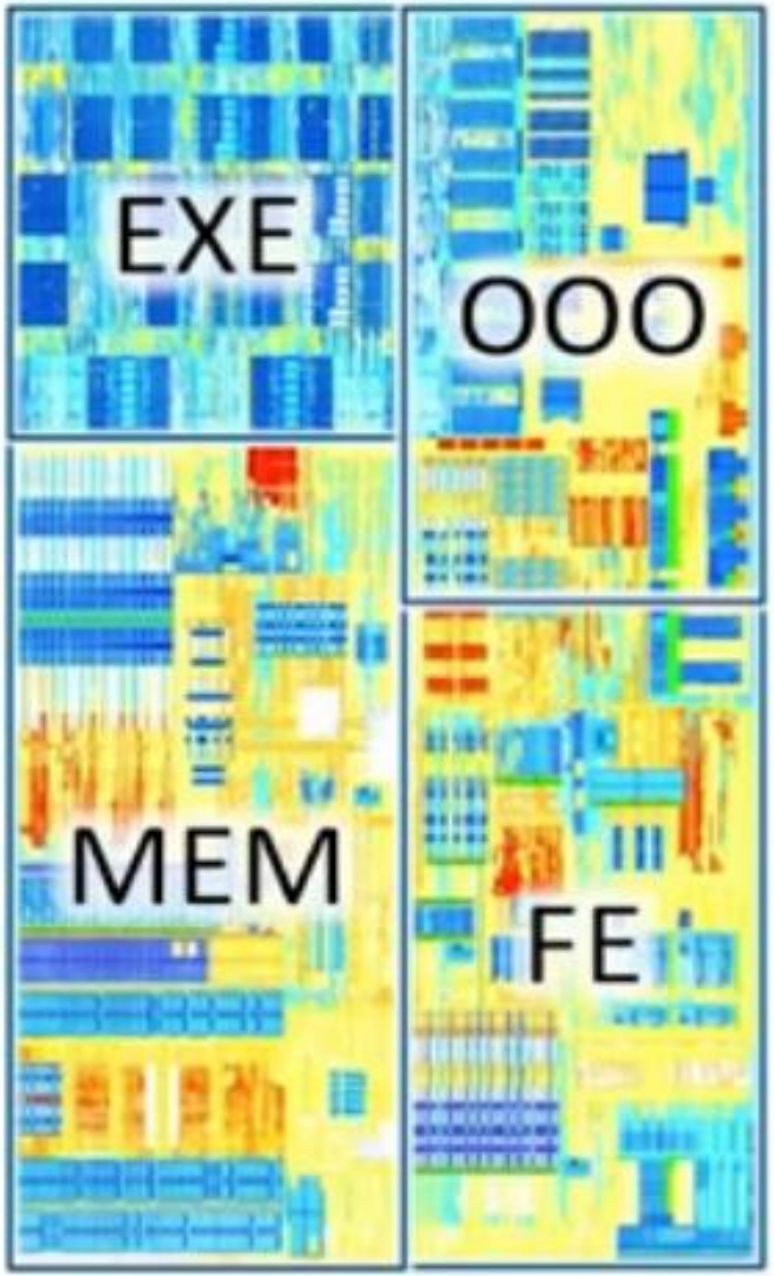
- Tick-Tock model
  - A new microarchitecture (Tock) is followed by process compaction (Tick)

Haswell Floorplan

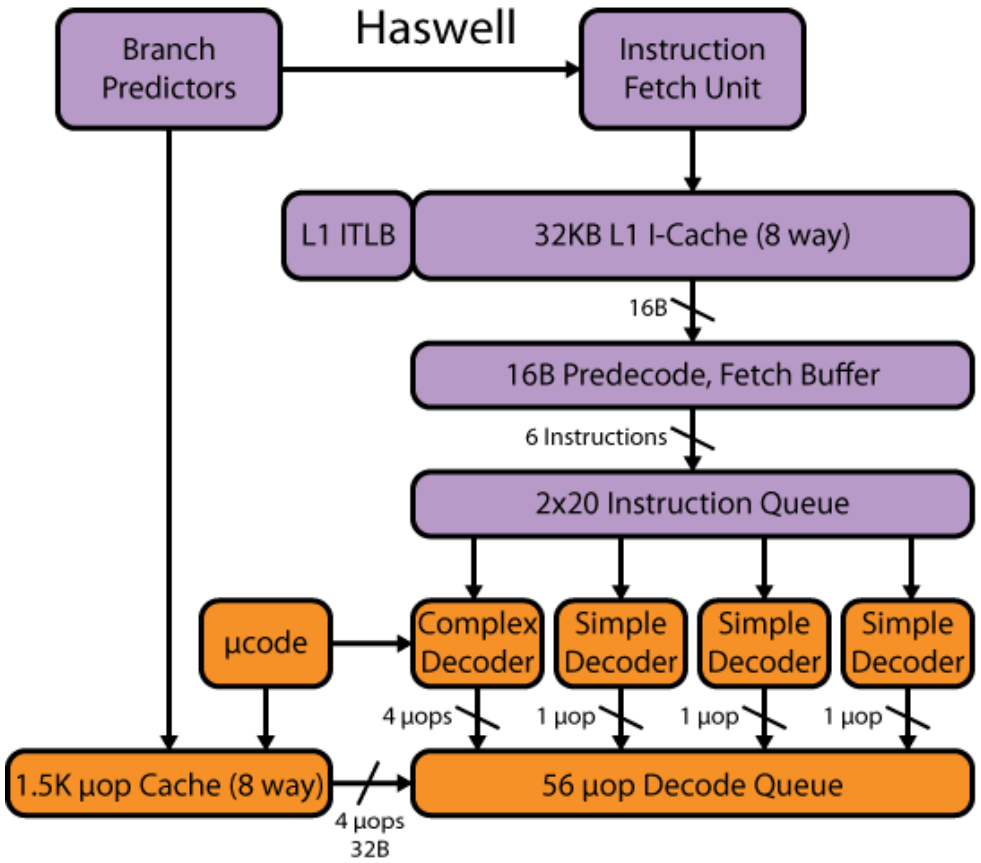
- 22nm process
- 1.4 Billion transistors
- Die size: 160 mm2

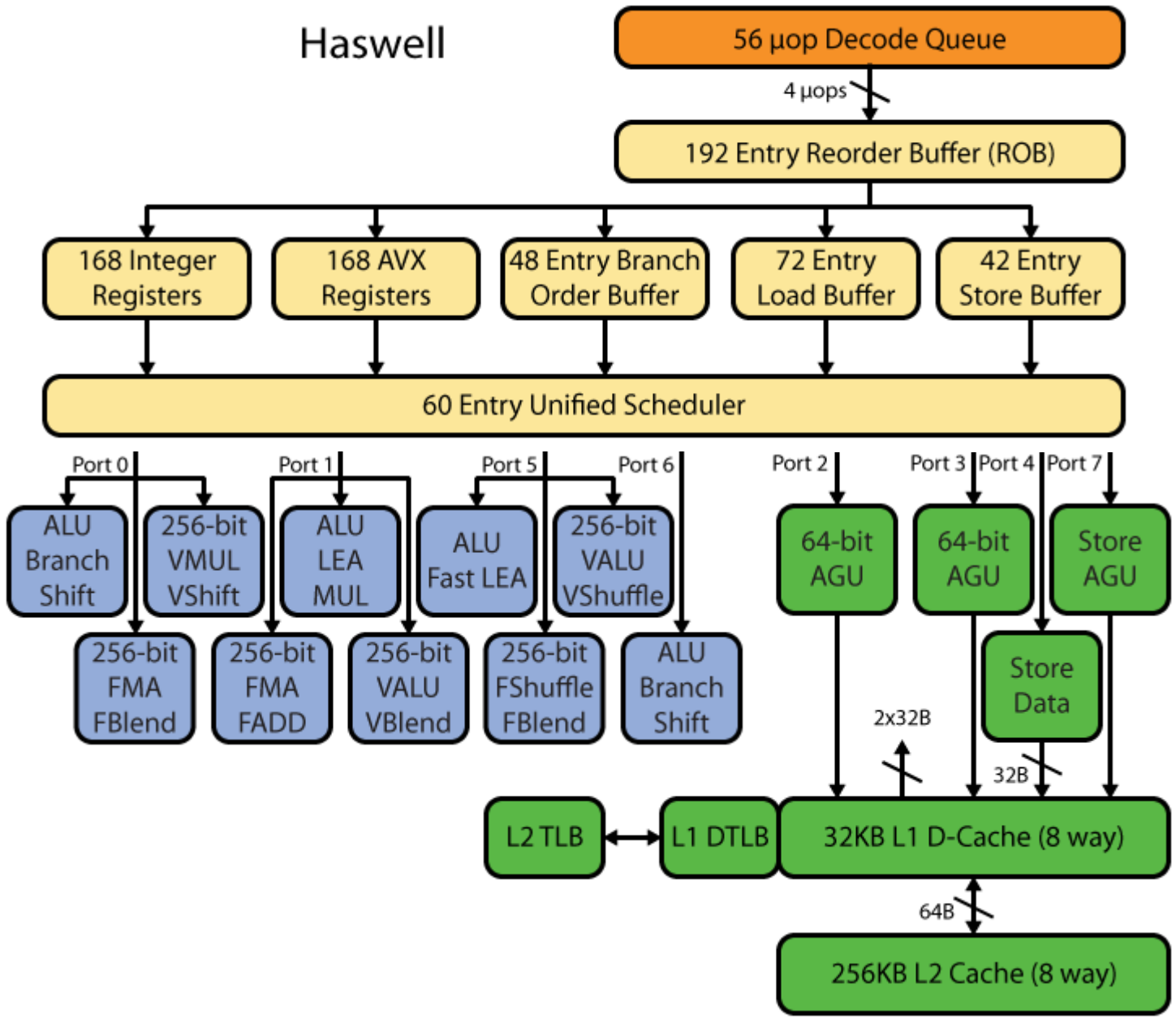






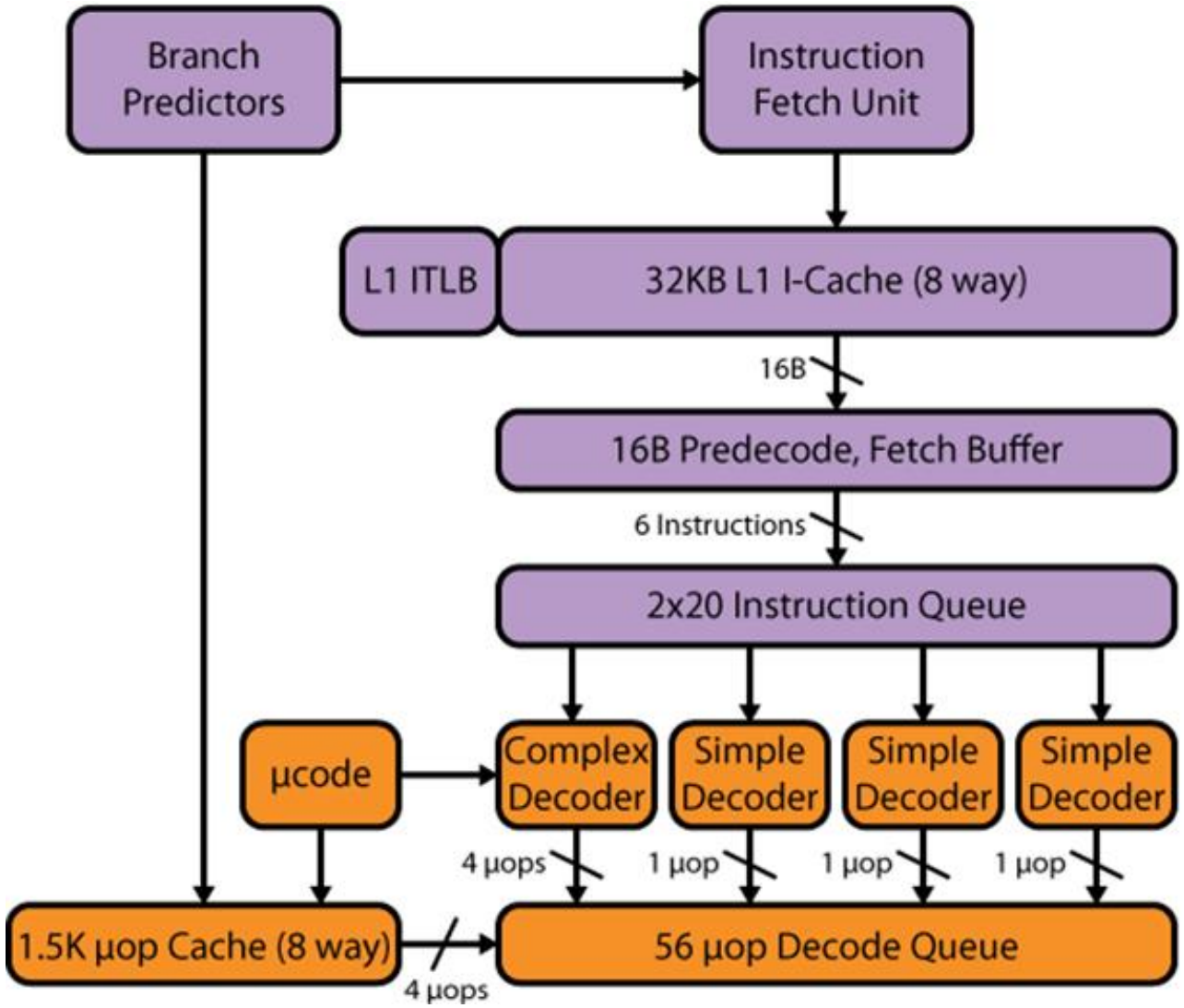
Block Diagram





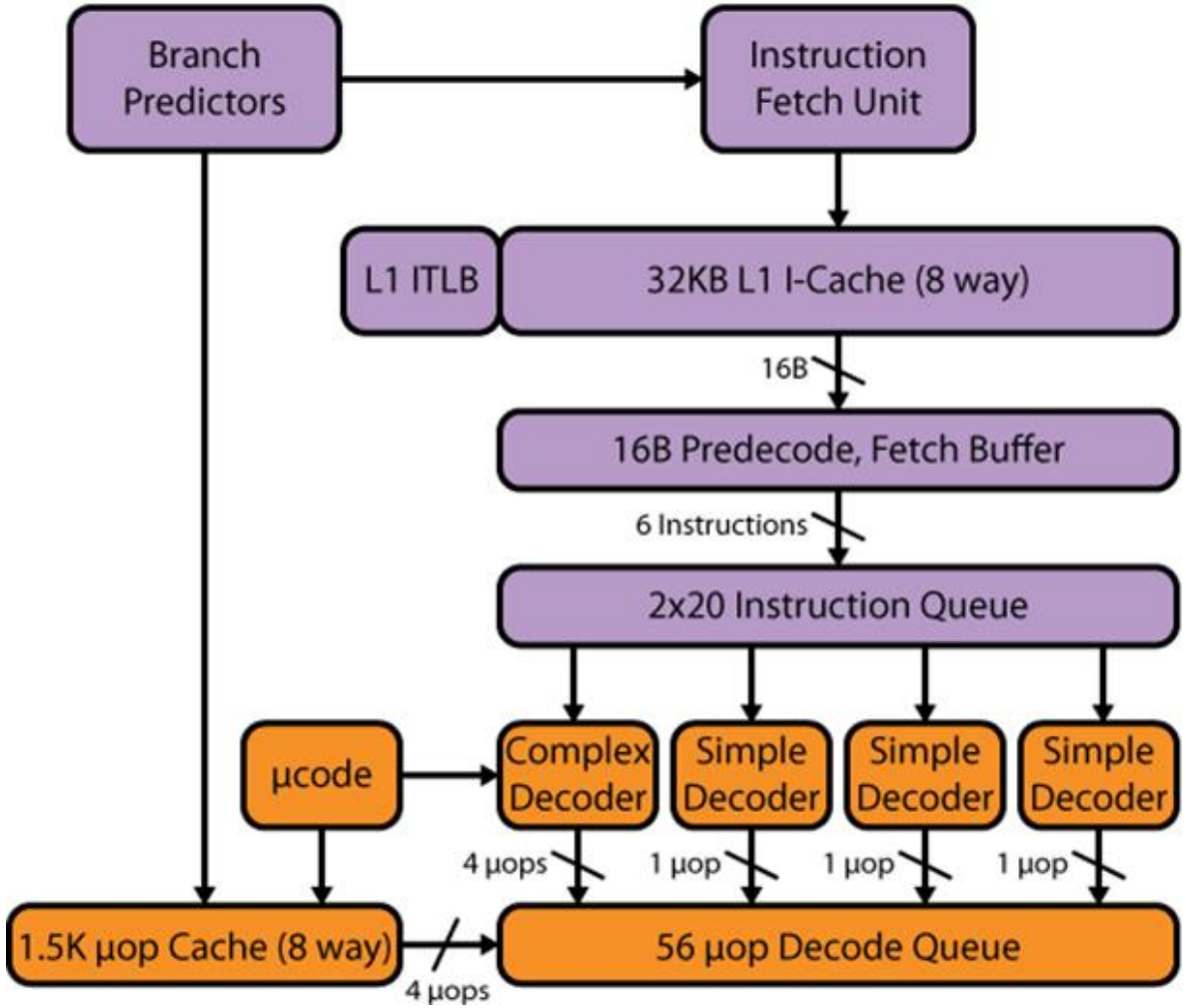
FrontEnd

- 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



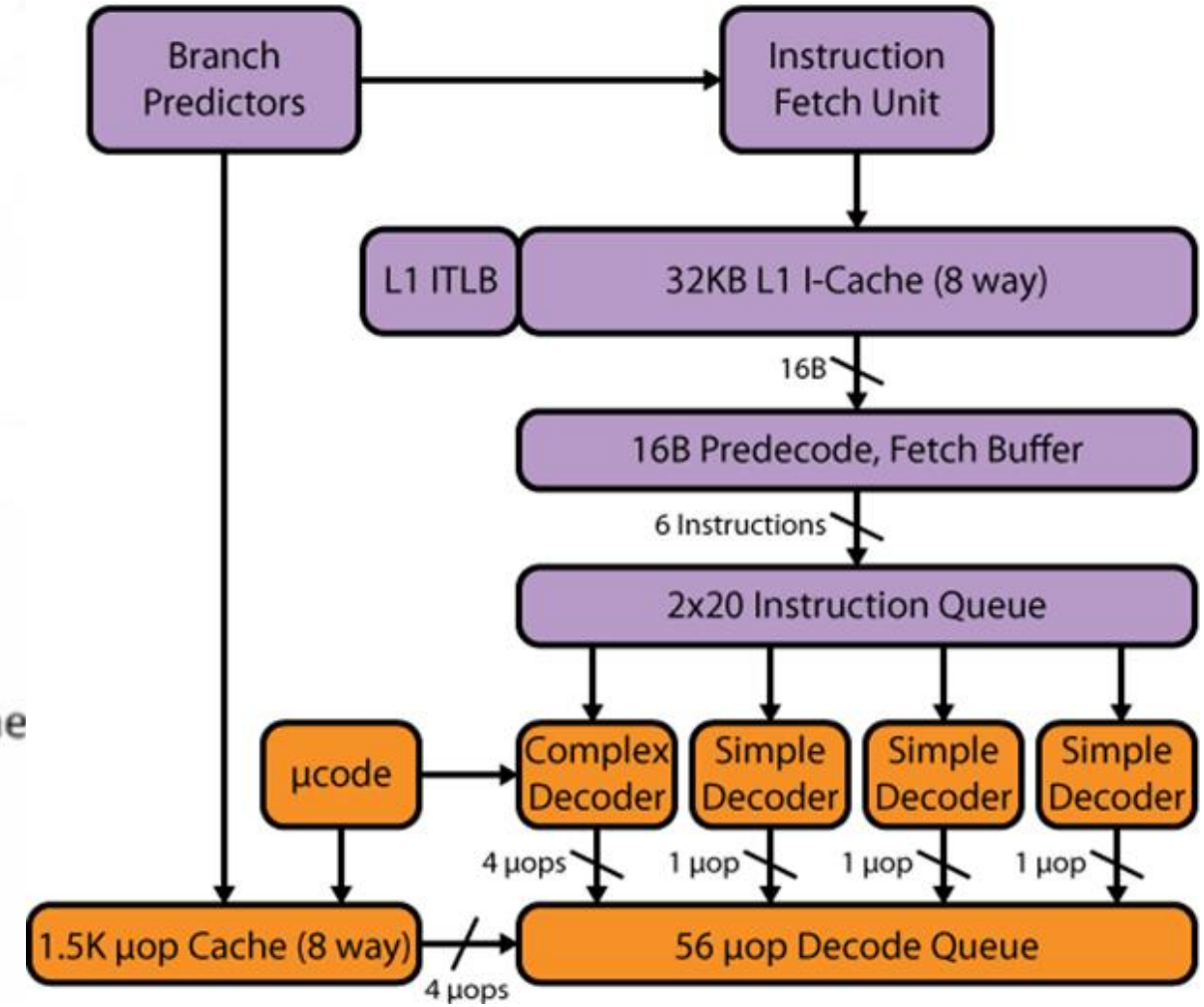
FrontEnd: Instruction Decode

- 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



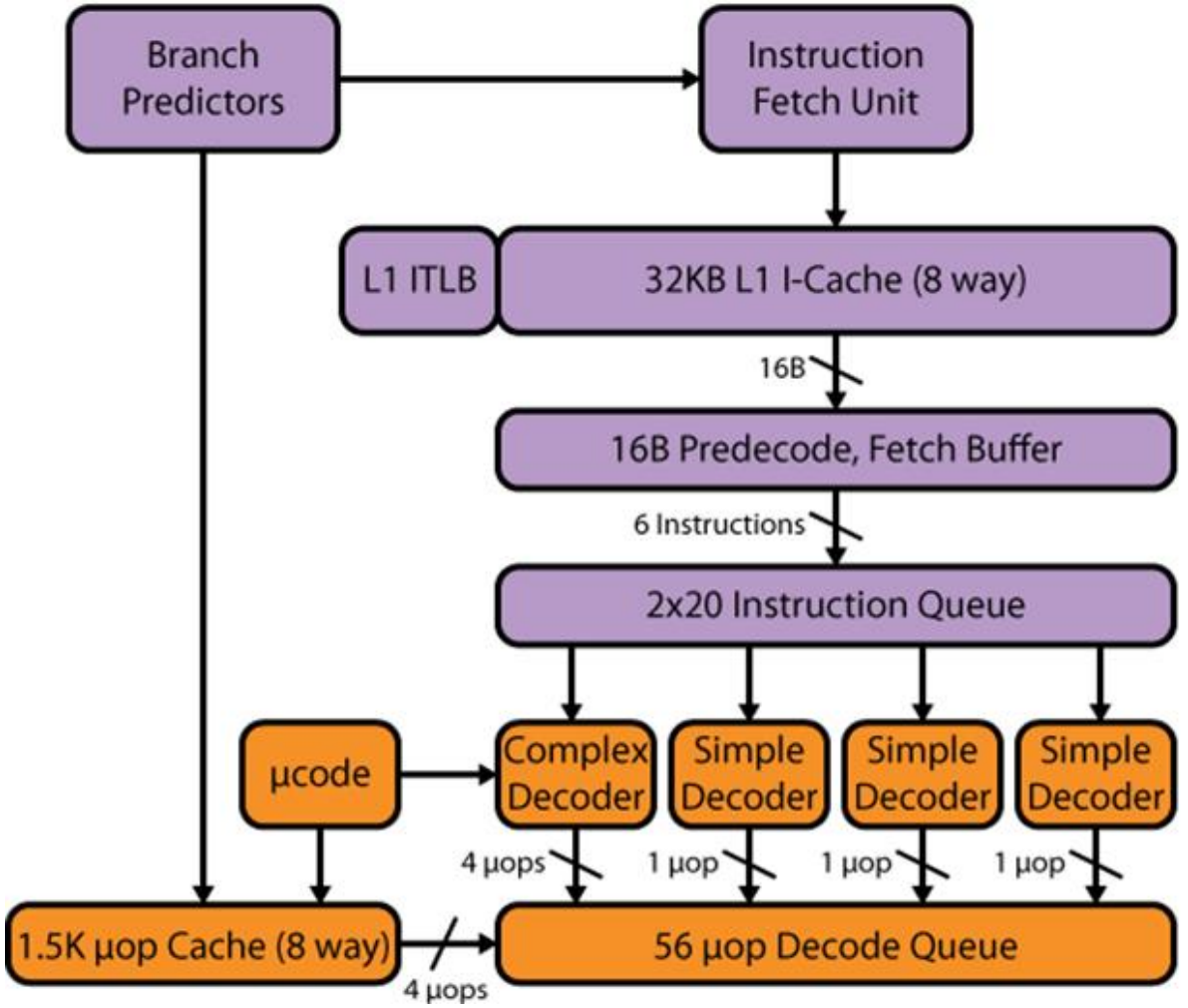
FrontEnd: Decode UOP Cache

- 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



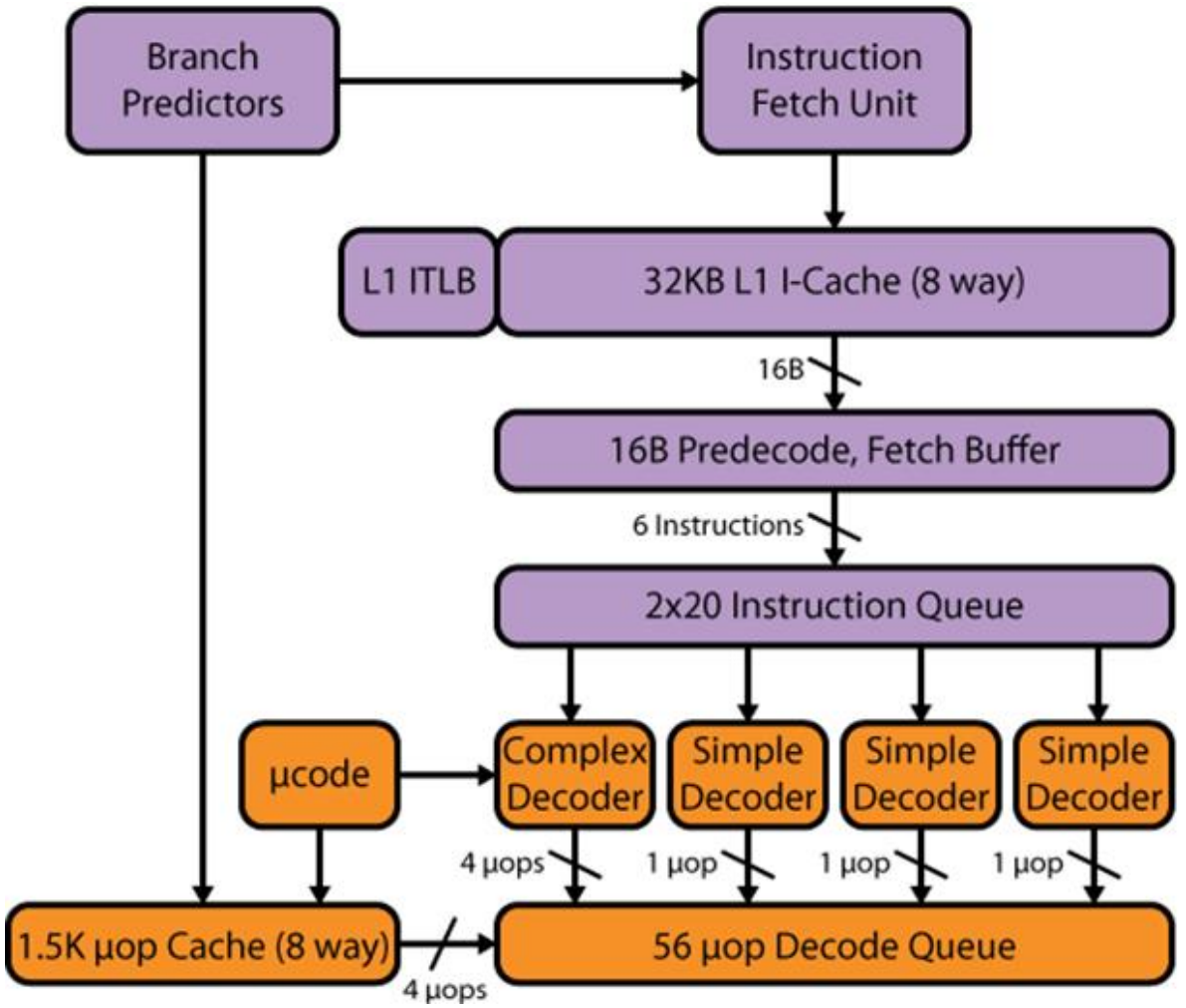
FrontEnd: Loop Stream Detector

- 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)

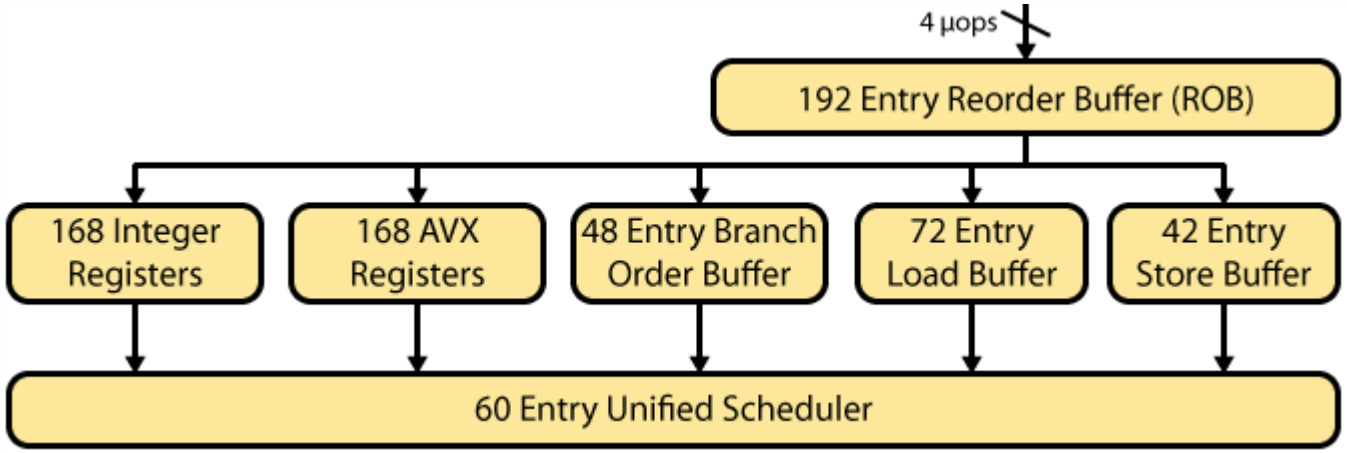


FrontEnd: Macro-Fusion

- 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



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

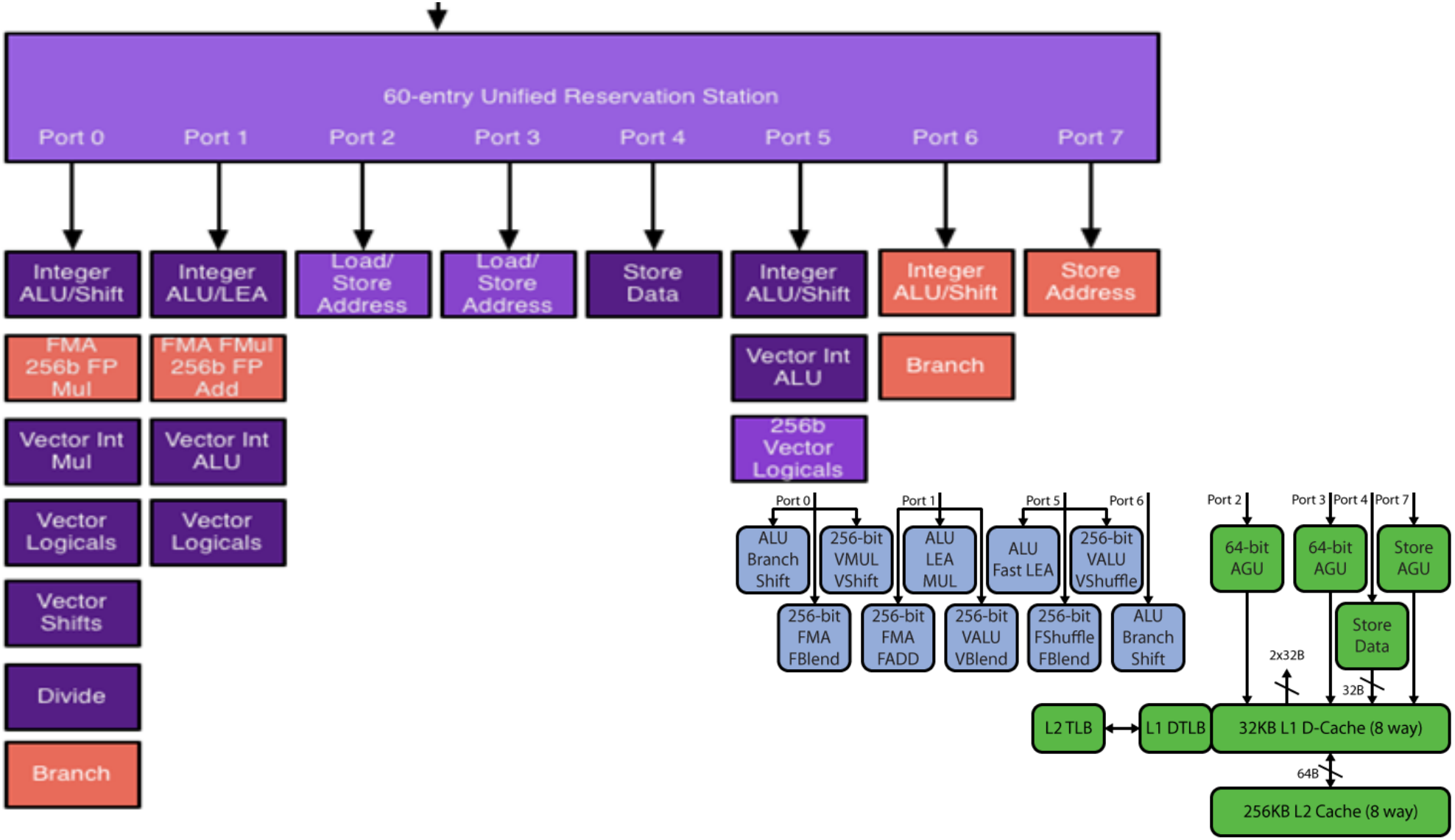
OOO: Renamer

- 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



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)

ST vs MT