Exceptions, then Advanced Pipelining

Exceptions

or

Oops!

Exceptions

- There are two sources of non-sequential control flow in a processor
  - explicit branch and jump instructions
  - exceptions
- *Branches* are synchronous and deterministic
- *Exceptions* are typically asynchronous and non-deterministic
- Guess which is more difficult to handle?

(*control flow* refers to the movement of the program counter through memory)

Exceptions and Interrupts

the terminology is not consistent, but we'll refer to

- *exceptions* as any unexpected change in control flow
- *interrupts* as any externally-caused exception

So then, what is:

- arithmetic overflow
- divide by zero
- I/O device signals completion to CPU
- user program invokes the OS
- memory parity error
- illegal instruction
- timer signal

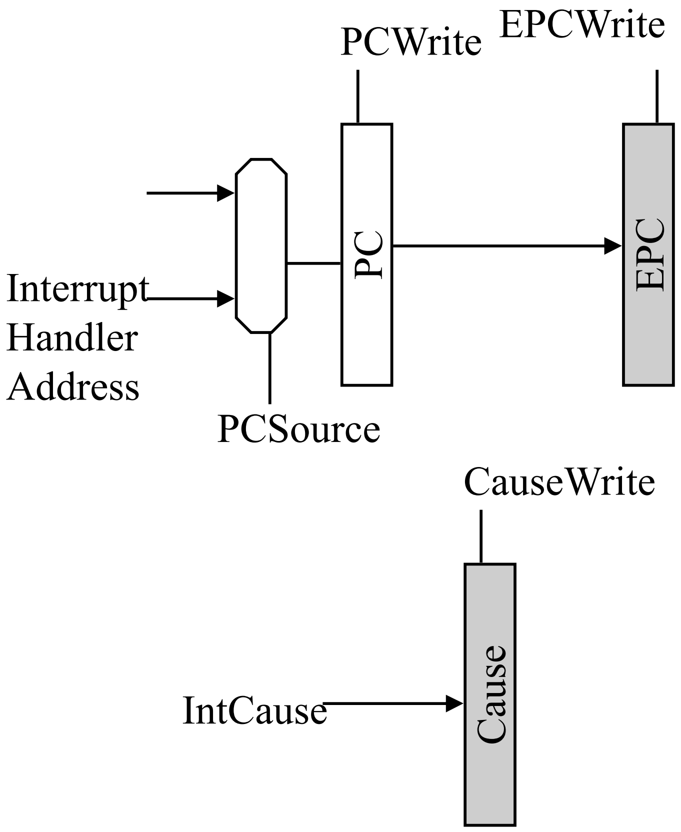
For now...

- The limited machine we’ve been designing in class can only generate two types of exceptions.
  - arithmetic overflow
  - illegal instruction
- On an exception, we need to
  - save the PC (invisible to user code)
  - record the nature of the exception/interrupt
  - transfer control to OS (what does that entail? Set PC to a new address and maybe flush some instructions)

- Let’s think a little about how we’d implement exceptions on our processor…

Supporting exceptions

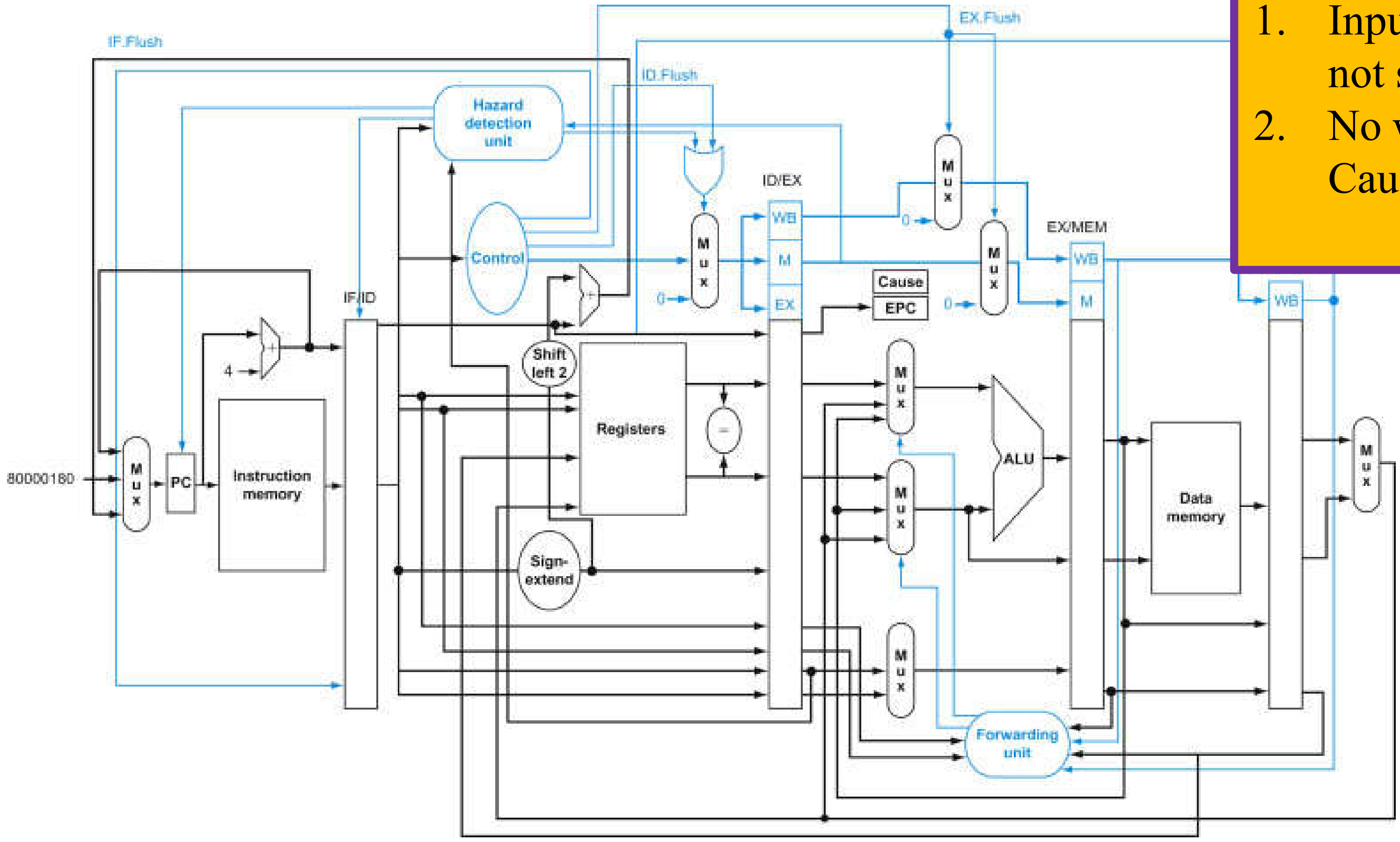
- For our MIPS-subset architecture, we will add two registers:
  - EPC: a 32-bit register to hold the user's PC
  - Cause: A register to record the cause of the exception
    - we'll assume undefined inst = 0, overflow = 1
- We will also add three control signals:
  - EPCWrite (will need to be able to subtract 4 from PC)
  - CauseWrite
  - IntCause
- We will extend PCSource multiplexor to be able to latch the interrupt handler address into the PC.



Pipelining and Exceptions

- Again, exceptions represent another form of control flow and therefore control dependence.
- Therefore, they create a potential branch hazard
- Exceptions must be recognized early enough in the pipeline that subsequent instructions can be flushed before they change any permanent state.
- We also have issues with handling exceptions in the correct order and “exceptions” on speculative instructions.
  - Exception-handling that always correctly identifies the offending instruction is called *precise interrupts*.

Pipelining and Exceptions



Notice that this figure from book is simplified in the following ways!

1. Inputs to "Cause" not shown
2. No way to read Cause or EPC.

Things to look for here:

1. Can record cause of exception (sort of)
2. Can record PC of last inst executed.
3. Can introduce a hardcoded exception handler PC
4. Can flush instructions in EX stage, ID, IF stages.

Advanced Pipelining

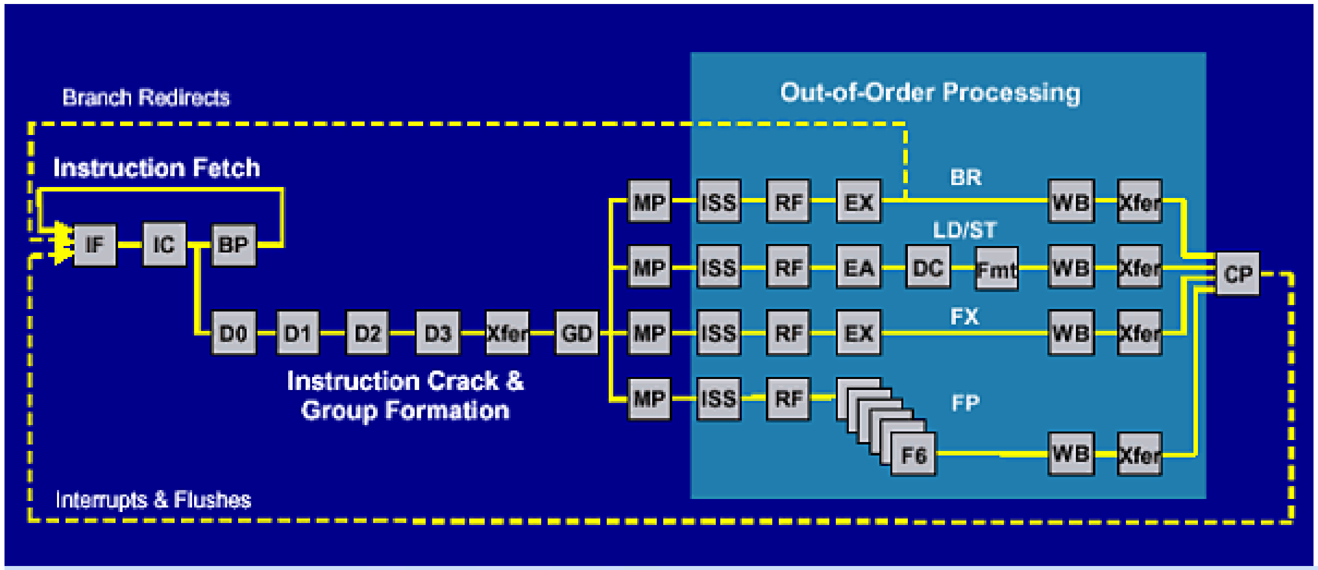
Pipelining in Today's Most Advanced Processors

- Not fundamentally different than the techniques we discussed
- Deeper pipelines
- Pipelining is combined with
  - superscalar execution
  - out-of-order execution

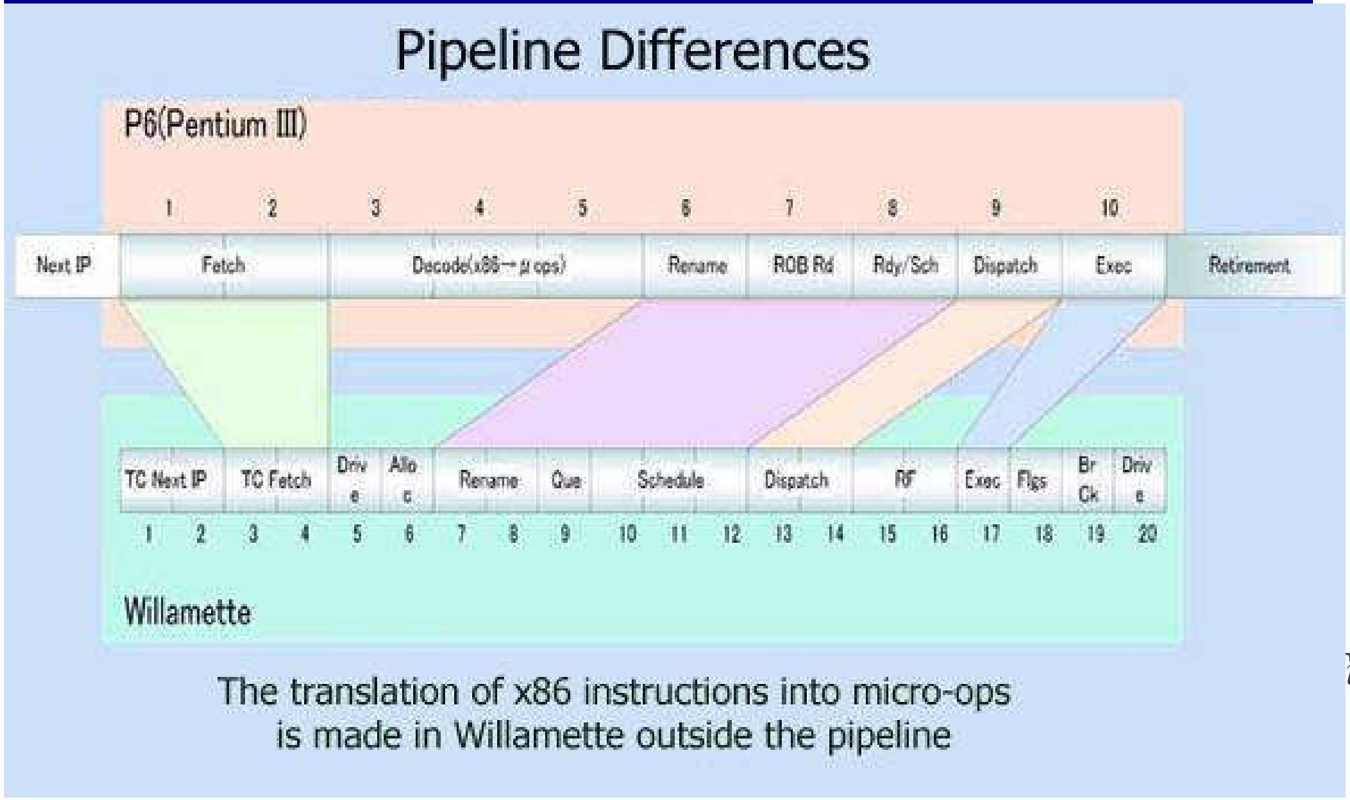
    or possibly…
  - VLIW (very-long-instruction-word)

Deeper Pipelines

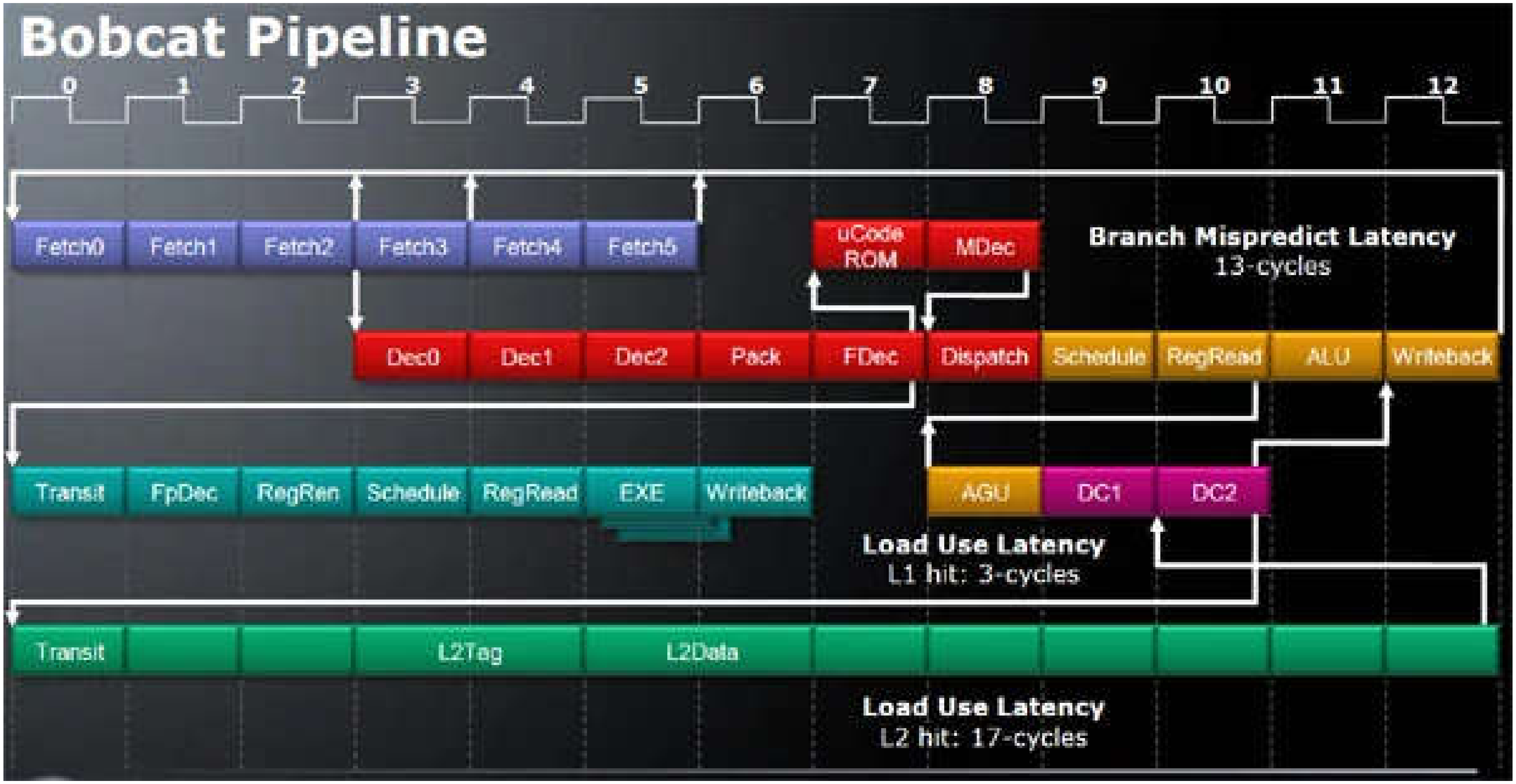
- Power 4



- Pentium 3
- Pentium 4



AMD Bobcat

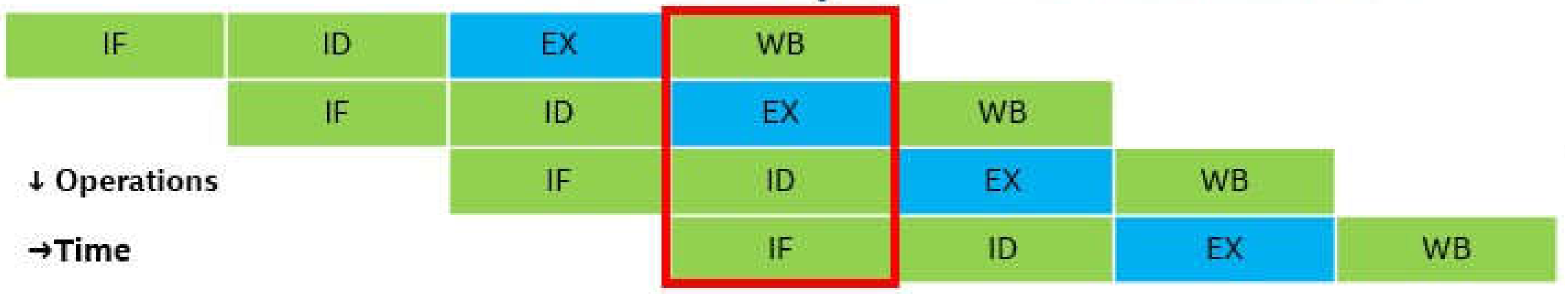

Bobcat Pipeline
0
1
2
3
4
5
6
7
8
9
10
11
12
Fetch0
Fetch1
Fetch2
Fetch3
Fetch4
Fetch5
uCode ROM
MDec
Branch Mispredict Latency
13-cycles
Dec0
Dec1
Dec2
Pack
FDec
Dispatch
Schedule
RegRead
ALU
Writeback
Transit
FpDec
RegRen
Schedule
RegRead
EXE
Writeback
AGU
DC1
DC2
Load Use Latency
L1 hit: 3-cycles
Transit
L2Tag
L2Data
Load Use Latency
L2 hit: 17-cycles

Pipelining in Today's Most Advanced Processors

- Not fundamentally different than the techniques we discussed
- Deeper pipelines
- Pipelining is combined with
  - superscalar execution
  - out-of-order execution

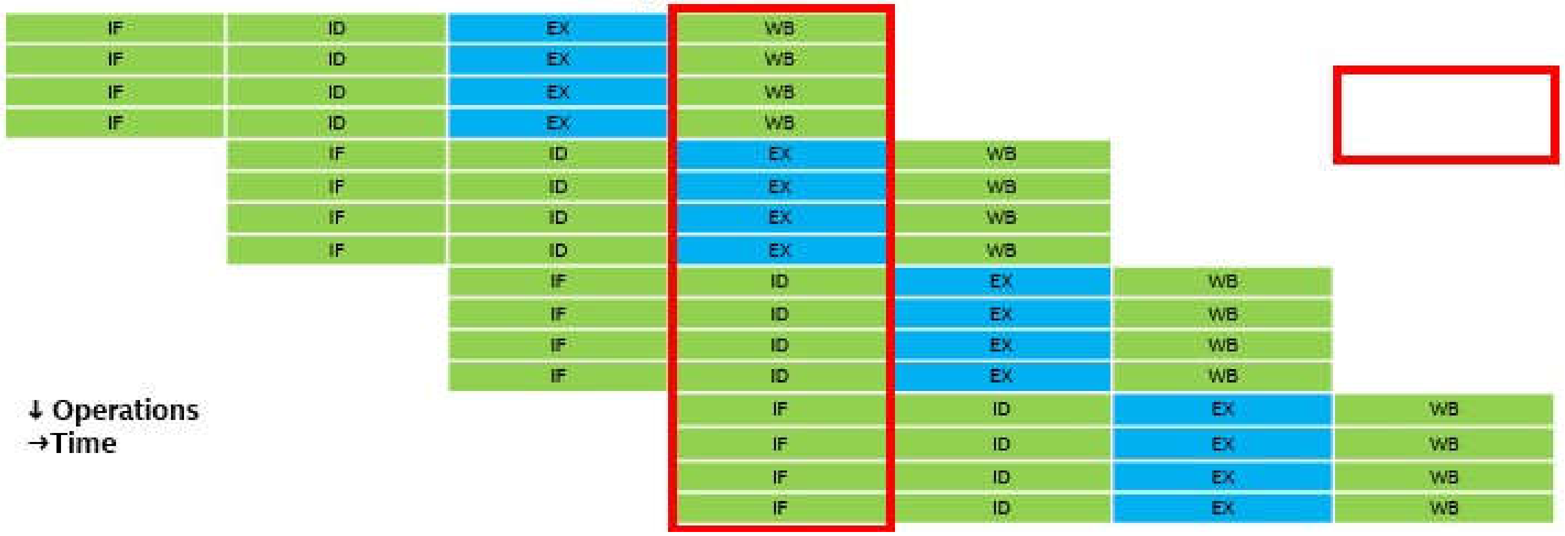
    or possibly…
  - VLIW (very-long-instruction-word)

Scalar Pipelined Execution



IF=Operation Fetch
ID=Operation Decode
EX=Execute
WB=Reg/Mem write back

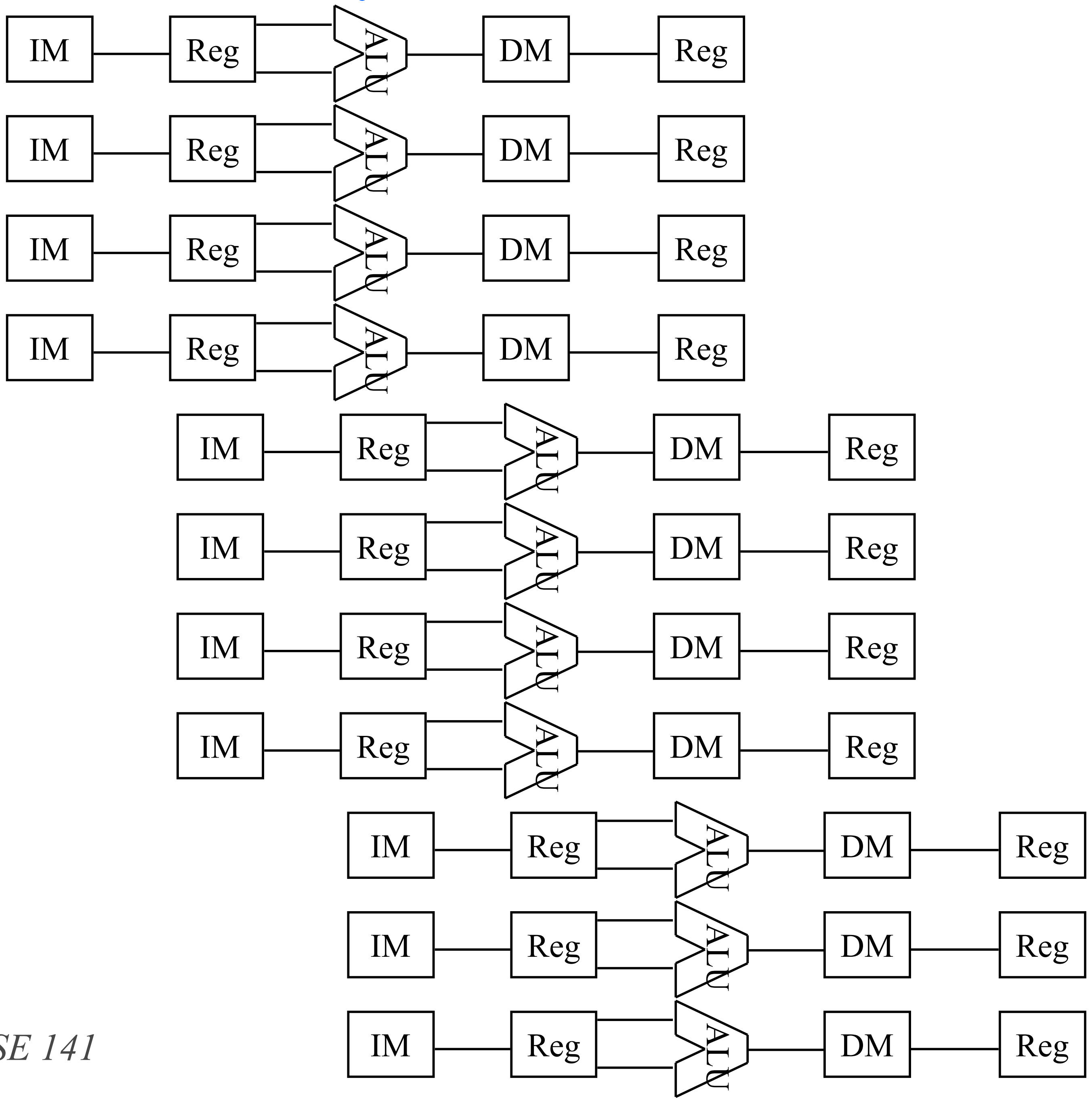
Superscalar Execution



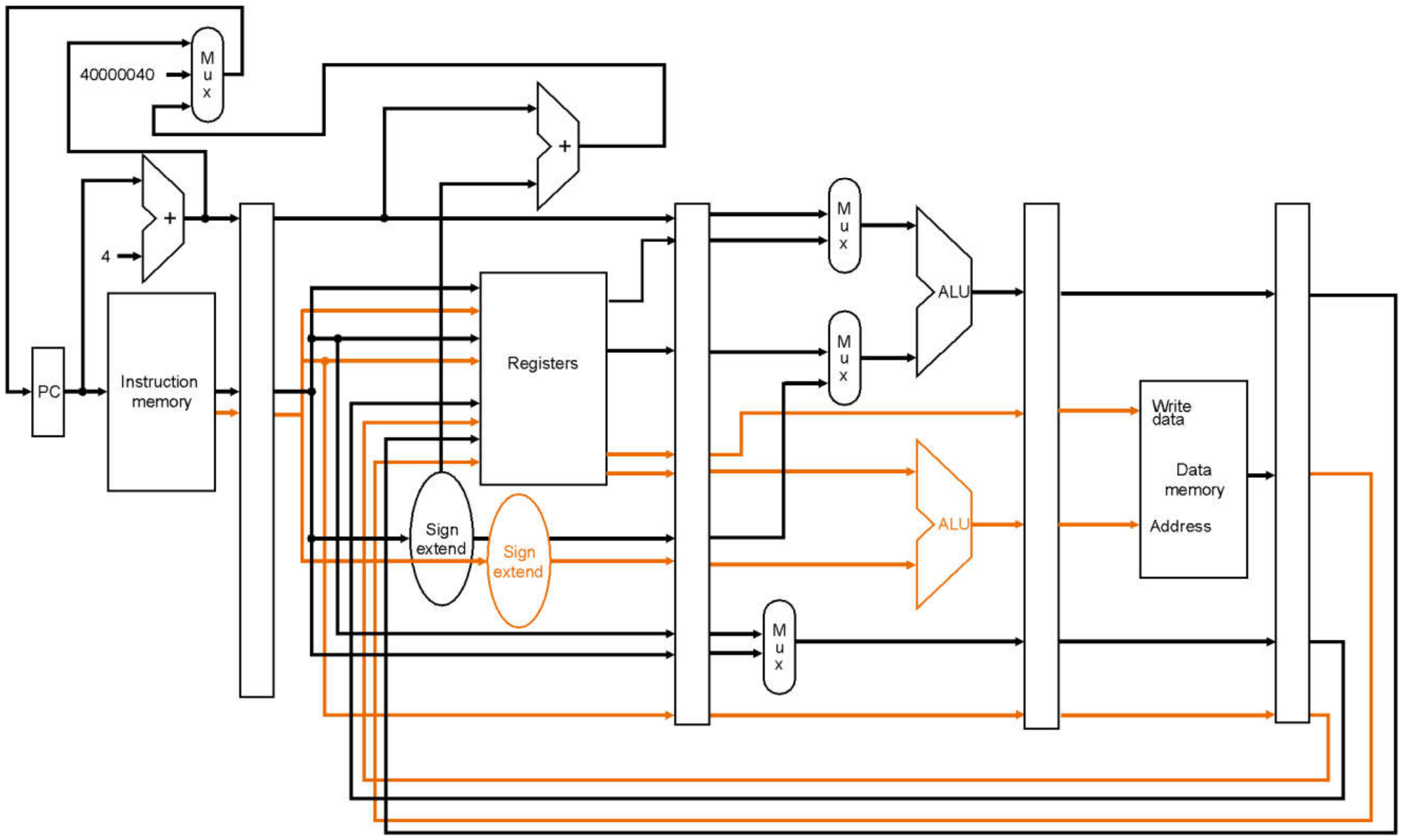
🟥 Operation-Level Parallelism

Figure copyright Intel

Superscalar Execution



A modest superscalar MIPS



- what can this machine do in parallel?
- what other logic is required?
- Represents earliest superscalar technology (eg, circa early 1990s)

Some historical perspective

- The earliest superscalar machines (see figures previous slides) were derived from scalar hardware, after noting that they already had distinct ALUs for various types of instructions – integer/load-store/FP – so they didn't have to add a lot of hardware to do 2 of those at once.
- Once we had superscalar processors, it didn't take long to measure the performance and say "if we had 2 (3, 4?) integer ALUs instead of 1…"
- So then there was a move to much more hardware replication. But not complete replication.
  - You may be able to do 4 integer adds, but not 4 load/store, not 4 FP multiplies, etc.

Models of Superscalar Execution

- To execute four instructions in the same cycle, we must find four independent instructions
- If the four instructions fetched are guaranteed by the compiler to be independent, this is a *VLIW* machine machine (e.g., Intel IA64/Itanium).
- If (up to) four consecutive instructions are only executed together if hardware confirms that they are independent, this is an *in-order superscalar* processor.
- If the hardware actively finds four (not necessarily consecutive) instructions that are independent, this is an *out-of-order superscalar* processor.
- What do you think are the tradeoffs?

Terminology note: VLIW machines are generally not considered superscalar, but a contrasting category.

Superscalar Scheduling

- assume in-order, 2-issue, ld-store followed by integer

```
lw $6, 36($2)
add $5, $6, $4
lw $7, 1000($5)
sub $9, $12, $5
```

- assume 4-issue, in-order, any combination (VLIW?)

```
lw $6, 36($2)
add $5, $6, $4
lw $7, 1000($5)
sub $9, $12, $5
sw $5, 200($6)
add $3, $9, $9
and $11, $7, $6
```

- When does each instruction begin execution?

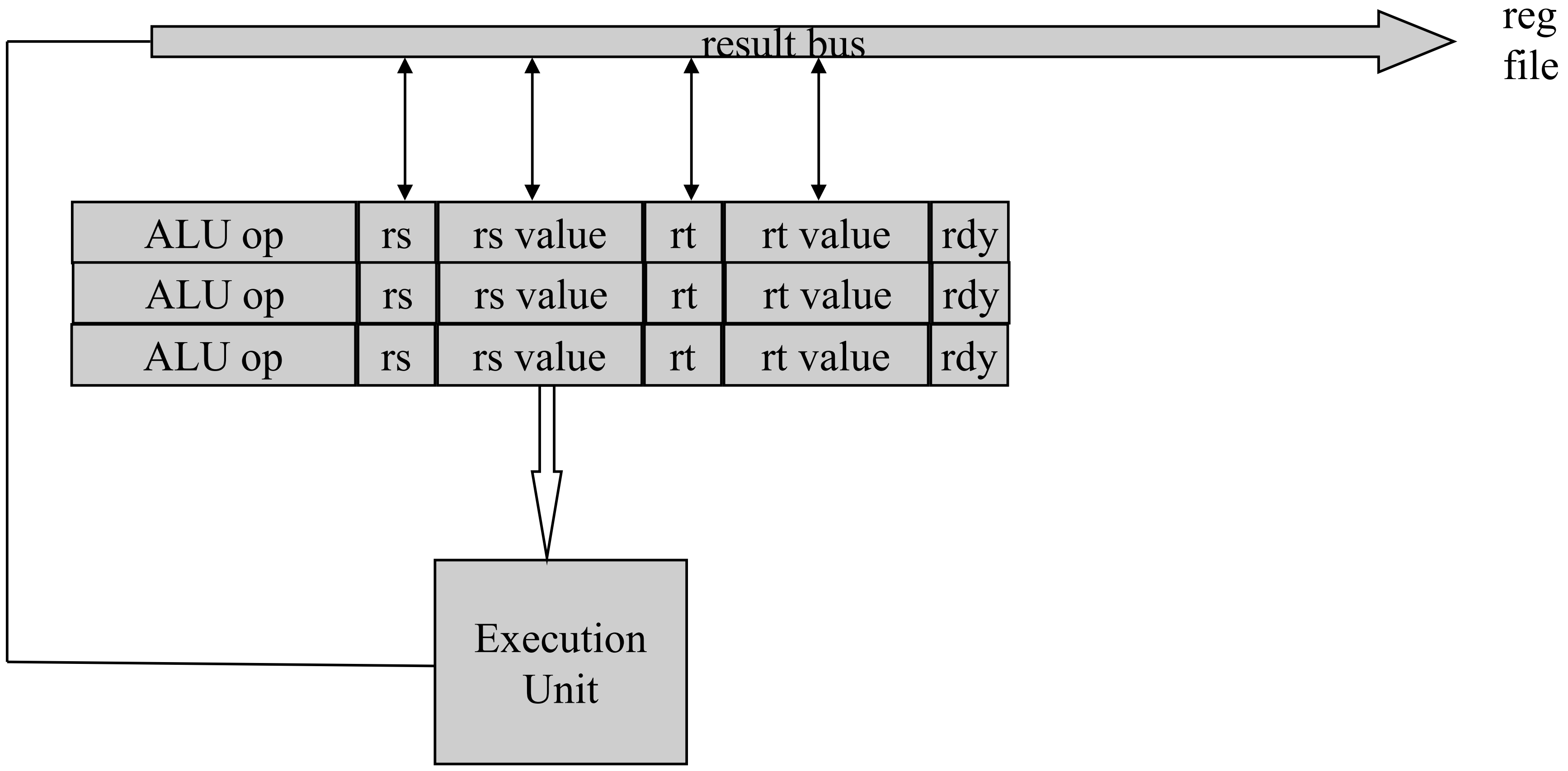
Dynamic Scheduling
a.k.a.
Out-of-Order Scheduling

- Issues (begins execution of) an instruction as soon as all of its dependences are satisfied, even if prior instructions are stalled. (assume 2-issue, any combination)

```
lw $6, 36($2)
add $5, $6, $4
lw $7, 1000($5)
sub $9, $12, $8
sw $5, 200($6)
add $3, $9, $9
and $11, $5, $6
```

Reservation Stations

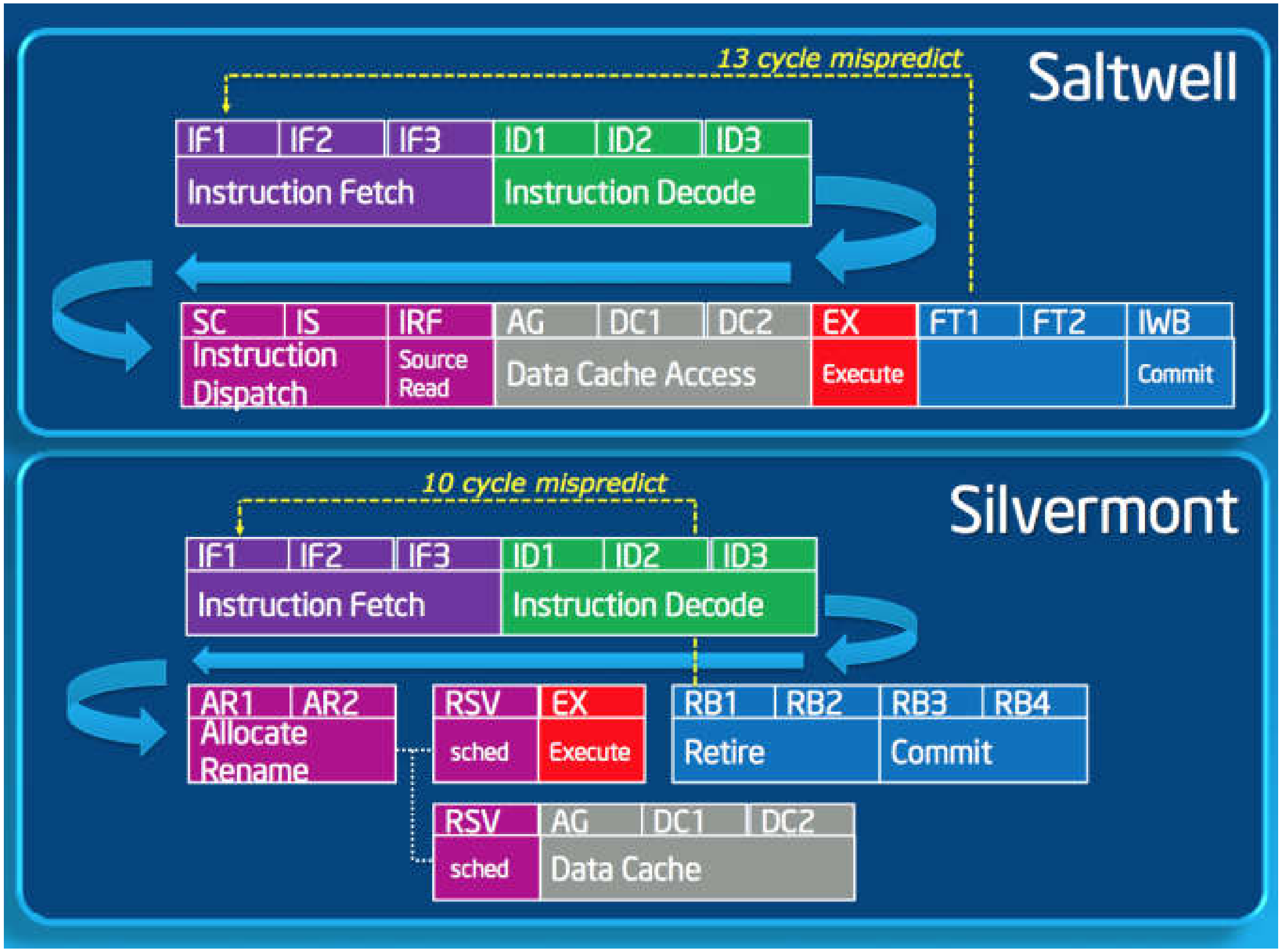
- are a mechanism to allow dynamic scheduling (out of order execution)



Okay

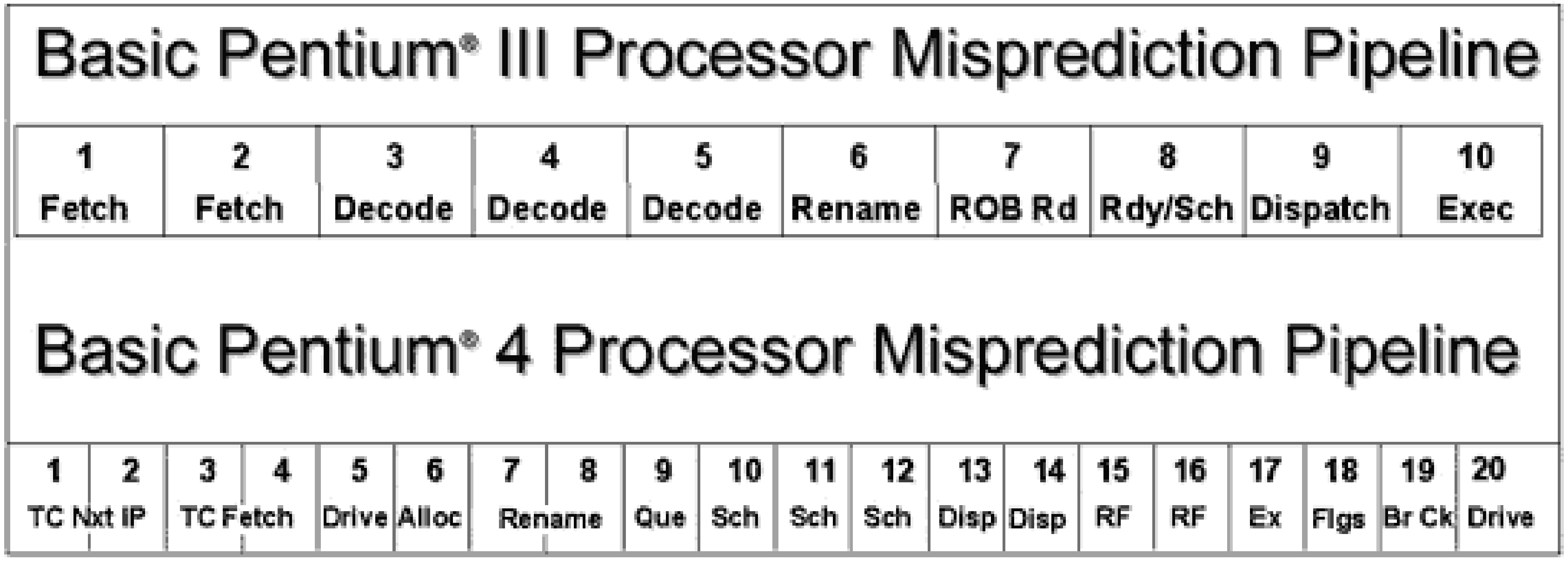
- Let’s look at some actual processors.

A couple of Atom (Intel low-power) architectures



Pentium 4

- Deep pipeline
- Dynamically Scheduled (out-of-order scheduling)
- Trace Cache
- *Simultaneous Multithreading* (HyperThreading)



Basic Pentium® III Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium® 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Modern (Pre-Multicore) Processors

- Pentium II, III – 3-wide superscalar, out-of-order, 14 integer pipeline stages
- Pentium 4 – 3-wide superscalar, out-of-order, simultaneous multithreading, 20+ pipe stages
- AMD Athlon, 3-wide ss, out-of-order, 10 integer pipe stages
- AMD Opteron, similar to Athlon, with 64-bit registers, 12 pipe stages, better multiprocessor support.
- Alpha 21164 – 2-wide ss, in-order, 7 pipe stages
- Alpha 21264 – 4-wide ss, out-of-order, 7 pipe stages
- Intel Itanium – 3-operation VLIW, 2-instruction issue (6 ops per cycle), in-order, 10-stage pipeline

More Recent Developments – Multicore Processors

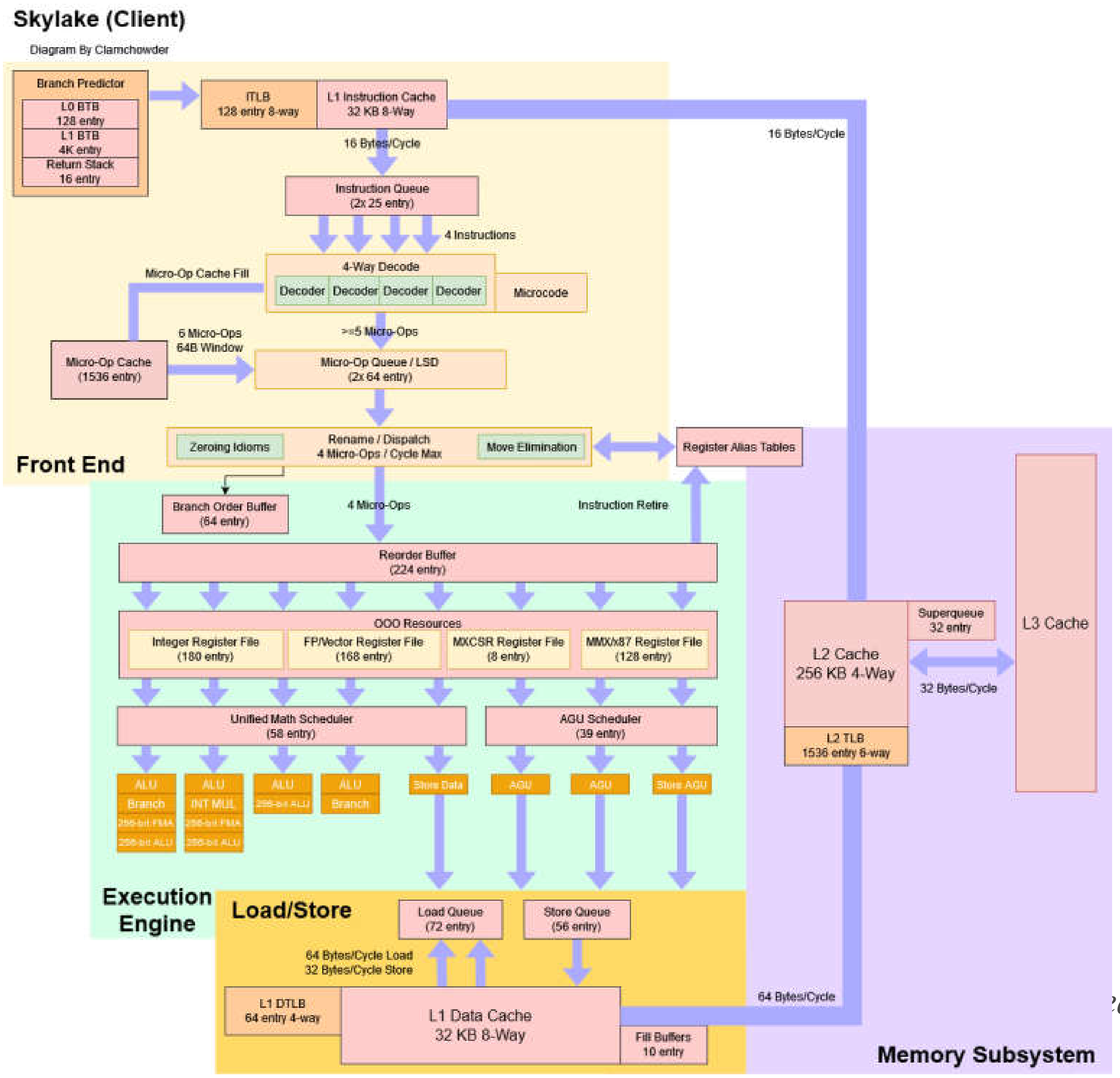
- IBM Power 4, 5, 6, 7
  - Power 4 dual core
  - Power 5 and 6, dual core, 2 *simultaneous multithreading* (SMT) threads/core
  - Power7 4-8 cores, 4 SMT threads per core
- Sun Niagara
  - 8 cores, 4 threads/core (32 threads).
  - Simple, in-order, scalar cores.
- Sun Niagara 2
  - 8 cores, 8 threads/core.
- Intel Quad Core Xeon
- AMD Quad Core Opteron
- Intel Nehalem, Ivy Bridge, Sandy Bridge, Haswell, Skylake, …(Core i3, i5, i7, etc.)
  - 2 to 8 cores, each core SMT (2 threads)
- AMD Phenom II
  - 6 cores, not multithreaded
- AMD Zen/Ryzen
  - 4-8 (mainstream, but up to 32) cores, 2 SMT threads/core, superscalar (6 micro-op/cycle)

Intel SkyLake

- Up to 4 cores (CPUs)
- Each core can have 224 uncommitted instructions in the pipeline
    - Up to 72 loads
    - Up to 56 stores
  - 97 unexecuted instructions in the pipeline waiting to be scheduled
  - Simultaneous Multithreading, meaning those 224/97 instructions can belong to two threads (processes, jobs)
  - Has 180 physical integer registers (used via register renaming)
  - Has 168 physical floating point registers
  - Executes up to 4 (?) micro-ops/cycle (think RISC instructions)
  - Has a 16-cycle branch hazard

- (note—Intel now hiding more and more architectural details)

Intel Skylake



Intel SkyLake

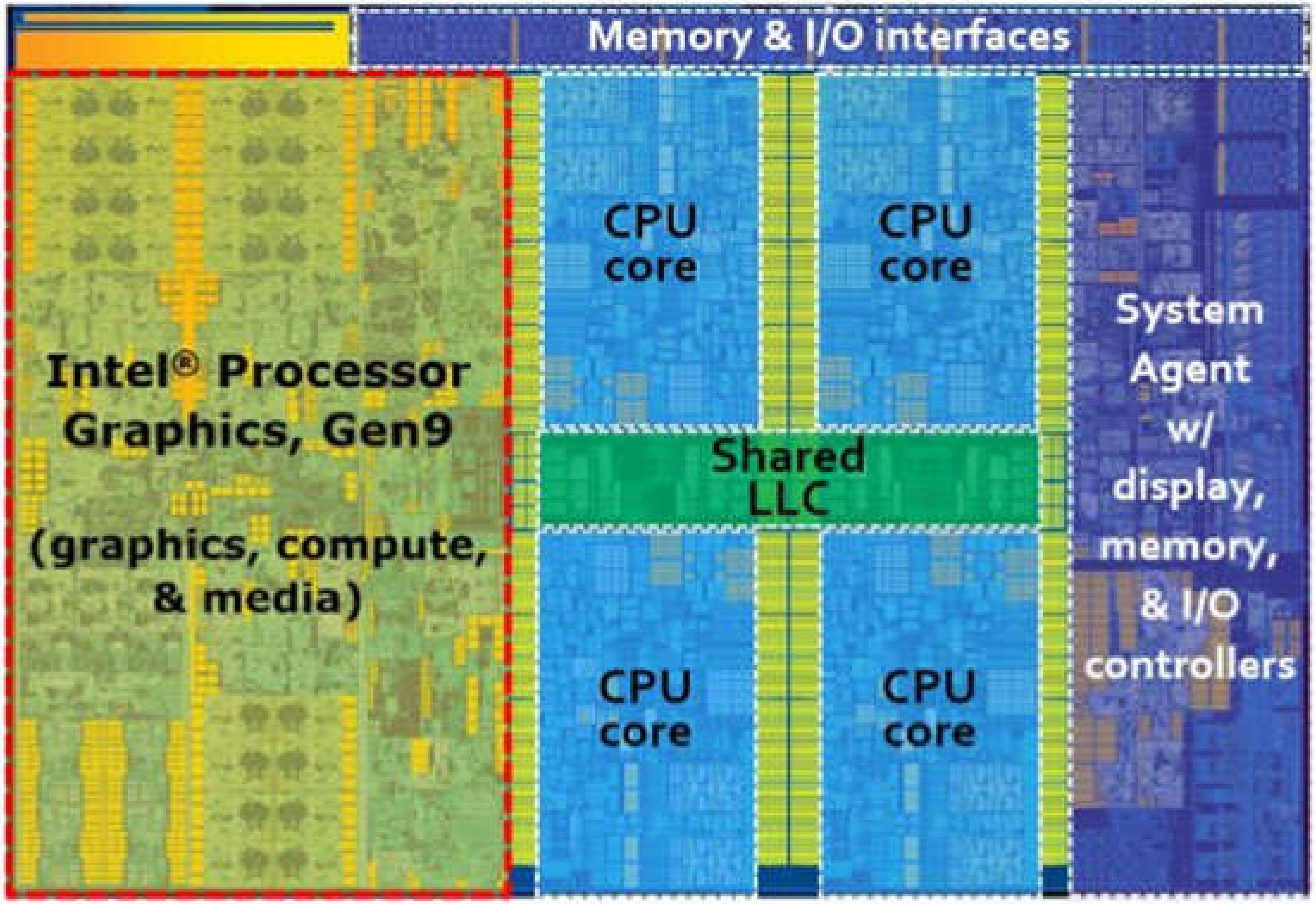
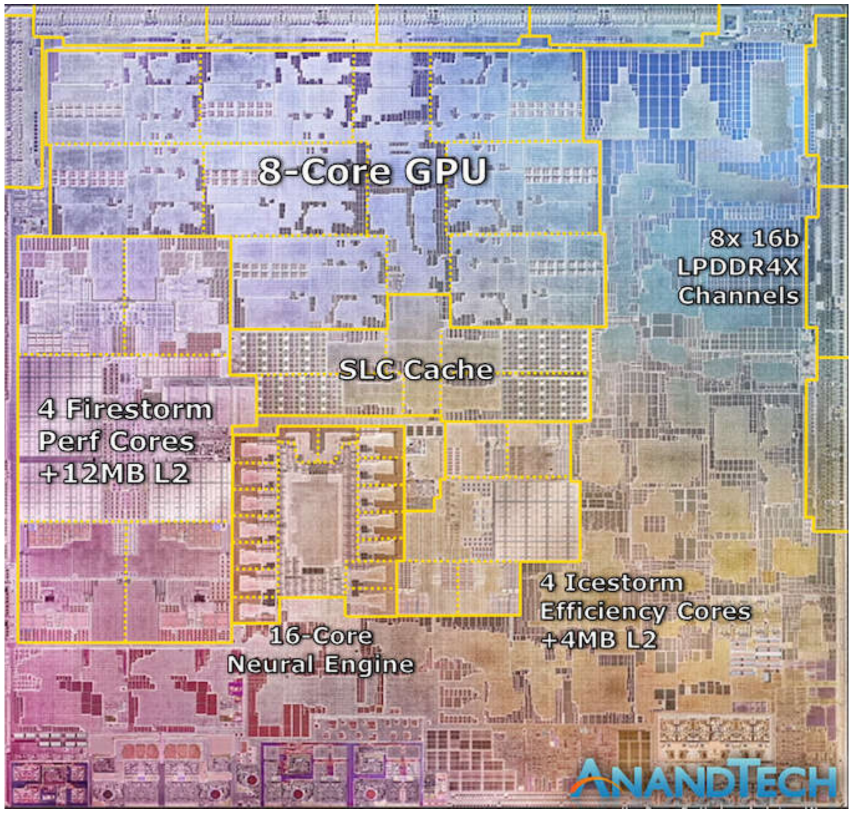

Memory & I/O interfaces
Intel® Processor Graphics, Gen9
(graphics, compute, & media)
CPU core
CPU core
Shared LLC
CPU core
CPU core
System Agent w/ display, memory, & I/O controllers

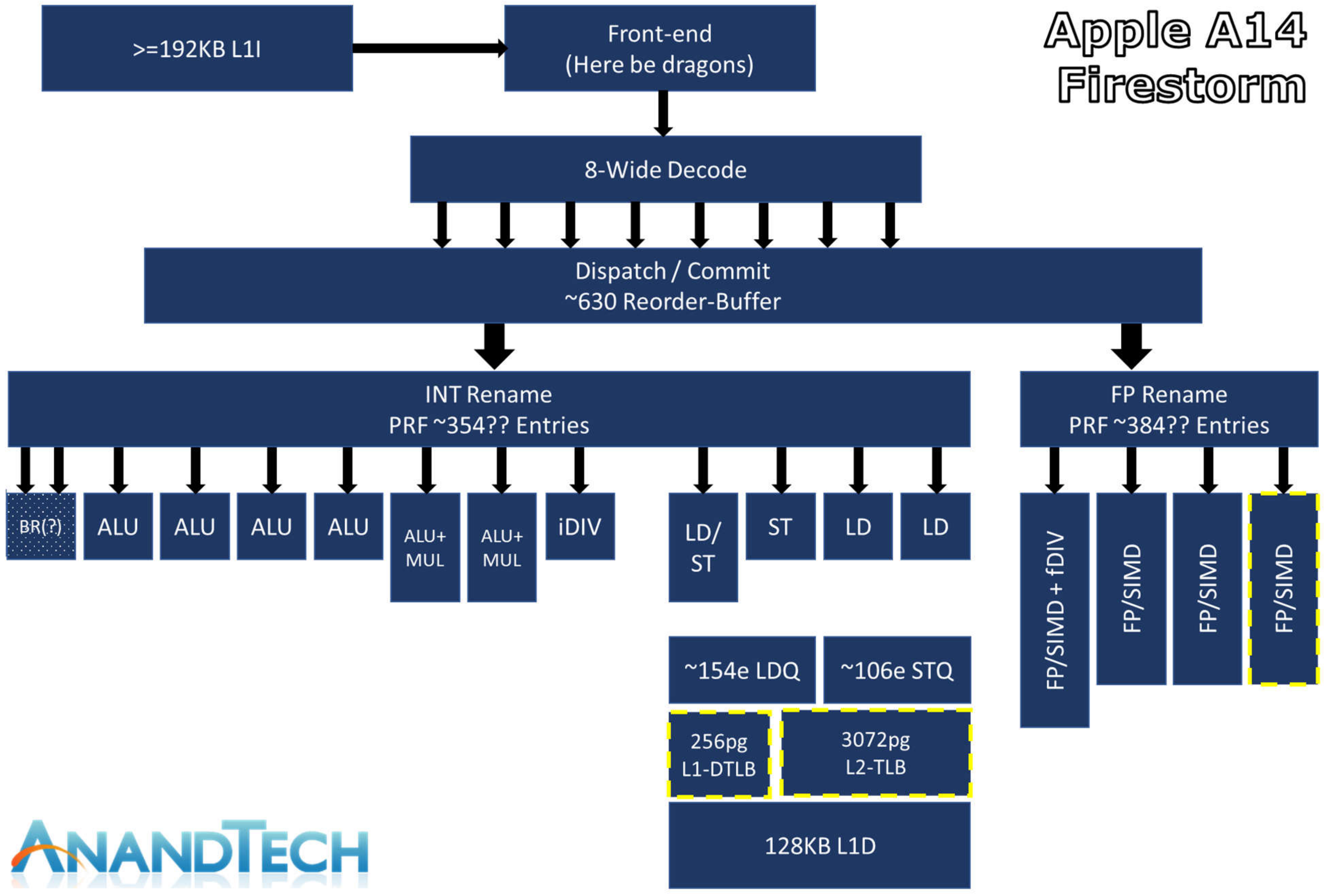
Figure 1: Architecture components layout for an Intel® Core™ i7 processor 6700K for desktop systems. This SoC contains 4 CPU cores, outlined in blue dashed boxes. Outlined in the red dashed box, is an Intel® HD Graphics 530. It is a one-slice instantiation of Intel processor graphics gen9 architecture.

Apple M1 – source: AnandTech blog



- ARM ISA
- 12 MB L2 cache [**<u>this is huge</u>**]
  - C.f. Intel Tiger Lake @ 1.25*4 = 5MB
  - C.f. Intel Cooper Lake @ 1*28 = 28MB
    - For $13,000
- Massive ILP
  - 8-wide instruction issue [SMT unclear]
  - 4 loads/stores per cycle
  - C.f. Intel's 1+4
  - C.f. Samsung 6-wide [also ARM]
- Truly massive OoO window
  - ~630 instructions in flight??
  - C.f. Intel Willow Cove at 352
  - C.f. AMD Zen3 at 256

Much more here: https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2


Apple A14 Firestorm
>=192KB L1I
Front-end (Here be dragons)
8-Wide Decode
Dispatch / Commit ~630 Reorder-Buffer
INT Rename PRF ~354?? Entries
FP Rename PRF ~384?? Entries
BR(?)
ALU
ALU
ALU
ALU
ALU+ MUL
ALU+ MUL
iDIV
LD/ ST
ST
LD
LD
FP/SIMD + fDIV
FP/SIMD
FP/SIMD
FP/SIMD
~154e LDQ
~106e STQ
256pg L1-DTLB
3072pg L2-TLB
128KB L1D
ANANDTECH

- Now that’s look at the other end…

That's the current state of the art for *high performance*

- We could title the next few slides either:

- State of the art for Power-conservative or energy-conservative processors.

- or

- Yes, there are real (and important) processors that look a lot like what you've done so far!

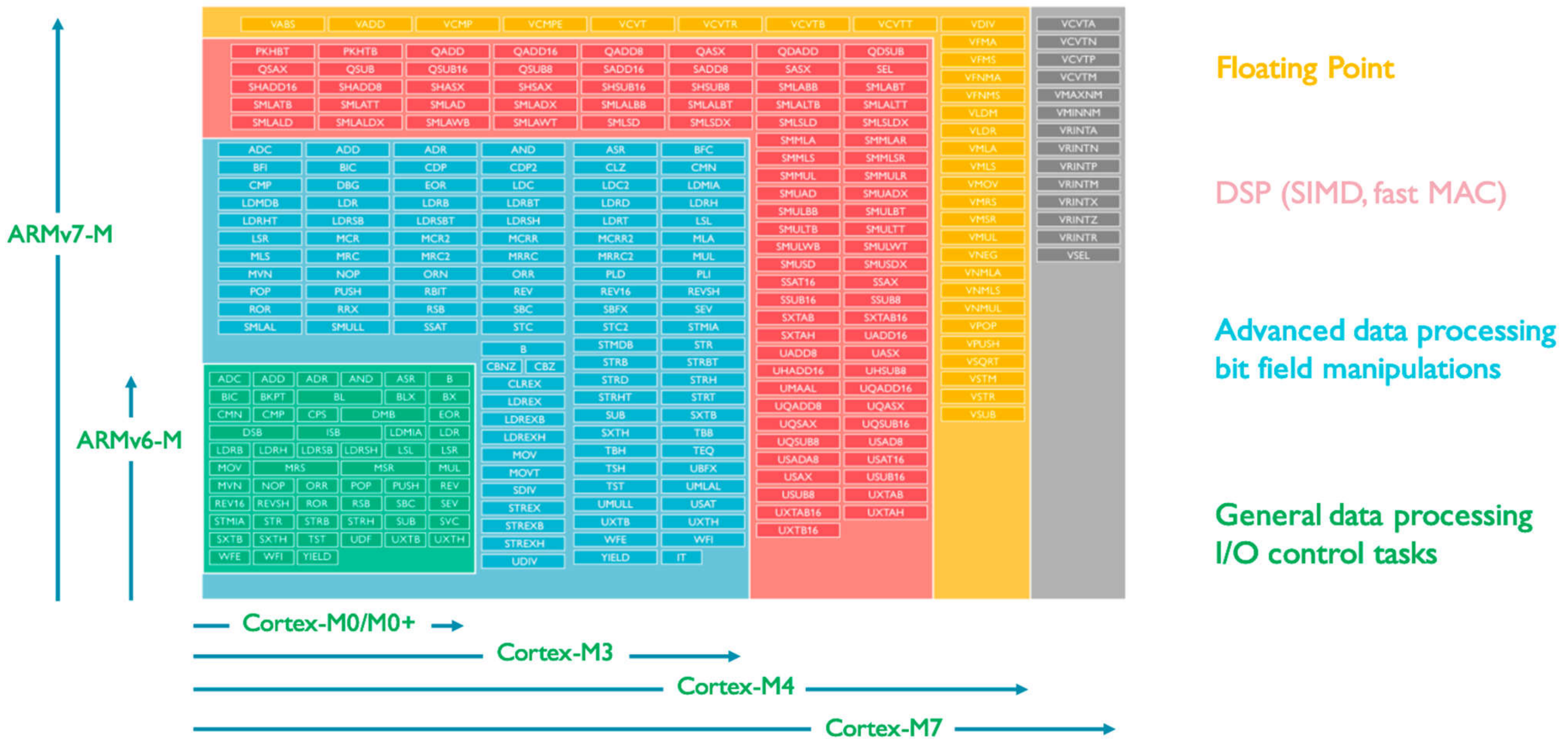
Thanks, Prof Pannuto

Acorn/Advanced RISC Machine (ARM) has three processor families

- Cortex A – "Application" processors
- Cortex R – "Real-Time" processors
- Cortex M – "Microcontroller" processors
  - (get it?)

Thanks, Prof Pannuto

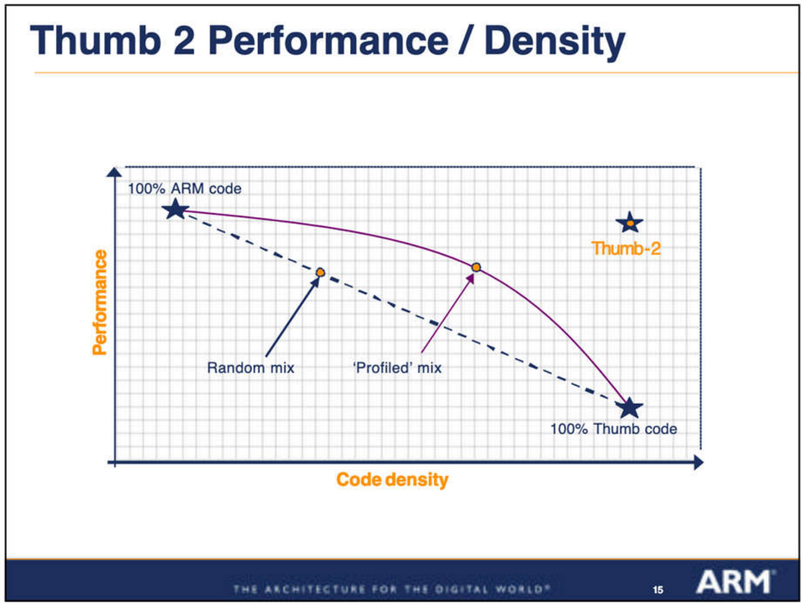
The Cortex-M family exposes a wide tradeoff of capability and cost – measured mostly in $$, Joules, and die area


Floating Point
DSP (SIMD, fast MAC)
Advanced data processing bit field manipulations
General data processing I/O control tasks
ARMv7-M
ARMv6-M
Cortex-M0/M0+
Cortex-M3
Cortex-M4
Cortex-M7

Thanks, Prof Pannuto

Let's look at the ARM Cortex-M3 in depth

- ISA: "Thumb2", specifically ARMv7-M
  - Mixed 16/32-bit instructions ["hybrid length' instructions]
  - Compromise: many instructions can be compact, why waste bits? Still simple (just two cases)
- 3 stage, in-order, single issue pipeline
  - With single-cycle hardware multiply!
- It has a branch predictor…
  - It predicts Not Taken!
  - 2 cycle mis-predict penalty
- It has a 3-word prefetcher



Thanks, Prof Pannuto

Implications of being area and energy constrained

- Performance / Watt >> than raw Performance
  - Latest designs are 22 **µA/MHz** (this is the measure that matters for IoT!)
- Fewer general purpose registers (There are 16)
  - Many of the smaller (16-bit) encodings can only access r0-r7
- Much slower core frequency (many in the 1-8 MHz, fastest M3's 48 or maybe 96 MHz)
- Much simpler microarchitecture
  - In-order design
  - Limited parallelism
- Tightly coupled memory -- No cache!
  - (well, a 3 word instruction cache)
  - **Just 1 cycle memory access penalty!** (i.e. `ldr` instruction takes 2 cycles, with no cache!)
  - ***VERY* different than traditional processors**

Thanks, Prof Pannuto

Pipelining -- Key Points

- ET = Number of instructions * CPI * cycle time
- *Data hazards* and *branch hazards* prevent CPI from reaching 1.0, but *forwarding* and *branch prediction* get it pretty close.
- Data hazards and branch hazards need to be detected by hardware.
- Pipeline control uses combinational logic. All data and control signals move together through the pipeline.
- Scalar pipelining attempts to get CPI close to 1. To improve performance we must reduce CT (superpipelining) or get CPI below one (superscalar, VLIW).