CSE 141: Introduction to Computer Architecture

Advanced Pipelines

Part I: Branch Predictors, how do they *actually* work?

- Sometimes it’s easier to understand when you trace all the real pieces

Branch Target Buffer

aka, how to know it's a branch before you know it's a branch

- Keeps track of the PCs of recently seen branches and their targets.
- Consult during Fetch (in parallel with Instruction Memory read) to determine:
  - Is this a branch?
  - If so, what is the target

What about jumps?

- How many stalls/flushes are required for each of the following situations:

	Jump Register, has BDS	Jump Immediate, has BDS	Jump Register, no BDS	Jump Immediate, no BDS
A	1	1	2	2
B	0	0	1	1
C	1	0	1	0
D	1	0	3	0
E	*None of the above*			

Jump Immediate, Jump Register – *with* BDS

- What parts of our MIPS machine makes this stall, hazard free?

Jump Immediate, Register – with no BDS

- What parts of this machine gets us to 1 stall / flush (which one, why?)

Can we eliminate the flush for jumps?

- (I mean, would I ask if we couldn't?)
- What is the difference between jump immediate and jump register here?

Wait, if a jump is just a ‘control flow operation’ that we always take, can’t we just re-use the BTB?

- We could, but there are some reasons it’s not a great idea
  - (why not?)
  - Waste of space
    - … not hard to predict whether a jump will be taken…
  - Aliasing
    - Lots of “taken” predictions…

What information do we need to mitigate different types of control flow hazards? How well can we do?

	Need to learn in instruction type before decode?	Need to record history of last destination?	Control flow change prediction accuracy?	Destination prediction accuracy?
Jump Immediate	Yes	Yes	100%	100%
Jump Register	Yes	Yes	100%	???
Branch	Yes	Yes	???	???

What is this about?

	Need to learn in instruction type before decode?	Need to record history of last destination?	Control flow change prediction accuracy?	Destination prediction accuracy?
Jump Immediate	Yes	Yes	100%	100%
Jump Register	Yes	Yes	100%	???
Jump Register to ________	Yes	No	100%	**~100%**
Branch	Yes	Yes	???	???

To support all these different needs, we build custom structures for each case [caveat: names vary!]

	Need to learn in instruction type before decode?	Need to record history of last destination?	Control flow change prediction accuracy?	Destination prediction accuracy?
Jump Immediate	Jump History Table			
Jump Register	Jump History Table			
JR to `$ra`	Return Address Stack			
Branch	Branch History Table			

The best way to keep track of all of this is to reason out what is needed to support various features

- What must ____________________, that handles jump immediate, look like?

The best way to keep track of all of this is to reason out what is needed to support various features

- What must ________________, that handles jump register, look like?

The best way to keep track of all of this is to reason out what is needed to support various features

- What must ________________, that handles jump to `$ra`, look like?

The best way to keep track of all of this is to reason out what is needed to support various features

- What must ________________, that handles branches, look like?

Pulling it all back together: For our MIPS machine without BDS, but with JHT, RAS, and BHT…

- Workload is 50% arithmetic, 5% jump immediate, 10% jump to GP register, 15% jump to $ra, and 20% branches.
  - Jumps to GP registers go to the same destination 90% of the time
  - Branches are predicted with 80% accuracy
  - Assume no aliasing
- What is the CPI?

Reviewing the branch predictors we have learned about

- Single-bit predictor
- Two-bit bimodal
- Two-level local

Rank the physical size of the following control hazard mitigation hardware elements

i. 1024-entry JHT
ii. 1024-entry BHT with 1-bit predictors
iii. 512-entry BHT with bimodal predictors
iv. 256-entry BHT and a 2-level local predictor with 7-bit patterns and 1-bit predictors

What are all these entries worth anyway?

- Assume the following branches are encountered in a loop such that each branch is seen once each loop
- If a machine has a 256-entry BHT with 1-bit predictors, what is the prediction accuracy for each branch?

Inst Addr	**Branch Pattern**
`0x400`	`T T T T`
`0x600`	`T N T N`
`0x800`	`N N N N`

	0x400	0x600	0x800
A	100%	0%	100%
B	0%	0%	0%
C	100%	50%	100%
D	33%	33%	33%
E	None of these		

Part II: Exceptions

- This is the last piece of what’s needed to make a “real” CPU useful

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?

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

Exceptions and Interrupts

The terminology is not always 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…

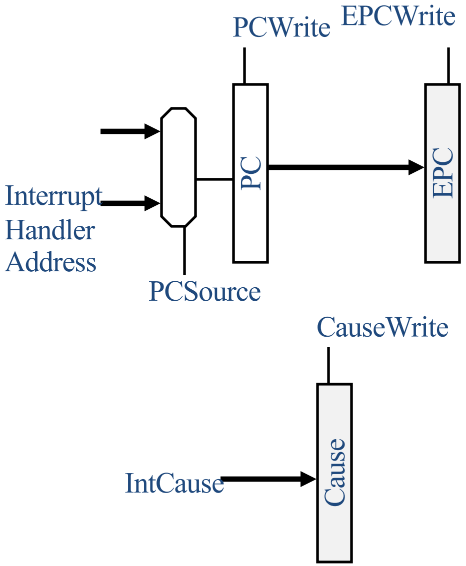
- The machine we've been designing in class can generate two types of exceptions.
  -
  -

For now…

- The machine we've been designing in class can generate three types of exceptions:
  - arithmetic overflow
  - illegal instruction
  - illegal memory address
- On an exception, we need to
  - save the PC (invisible to user code)
  - record the nature of the exception/interrupt
  - transfer control to OS

First steps towards supporting exceptions

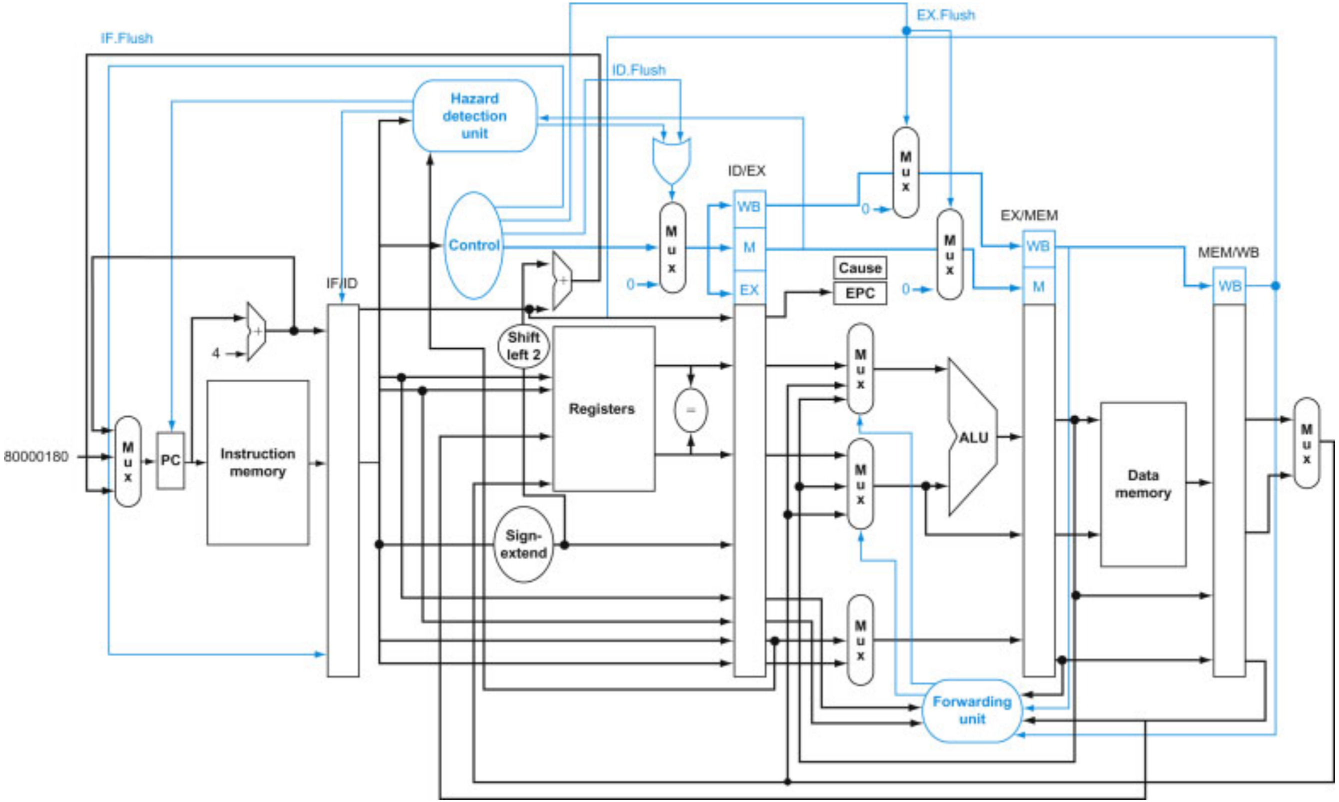
- For our MIPS-subset architecture, we will add two registers:
  - EPC: a 32-bit register to hold the user's PC
  - Cause: A register to record the cause of the exception
    - we'll assume undefined inst = 0, overflow = 1
- We will also add three control signals:
  - EPCWrite (will need to be able to 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.
  - Q: What is the first stage that can change 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*
  - (different words, same idea: ARM has *asynchronous / synchronous exceptions*)

Pipelining and Exceptions – The Whole Picture (except not really)



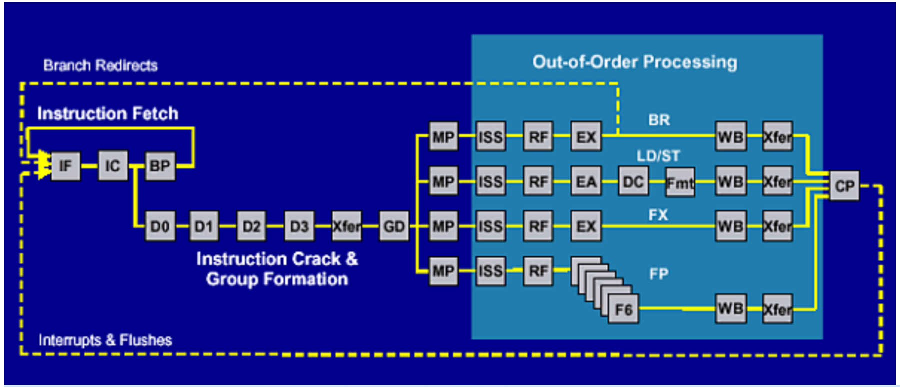
Part III: The Fancy Stuff in Real (Fast) Machines

Pipelining in Today's Most Advanced Processors

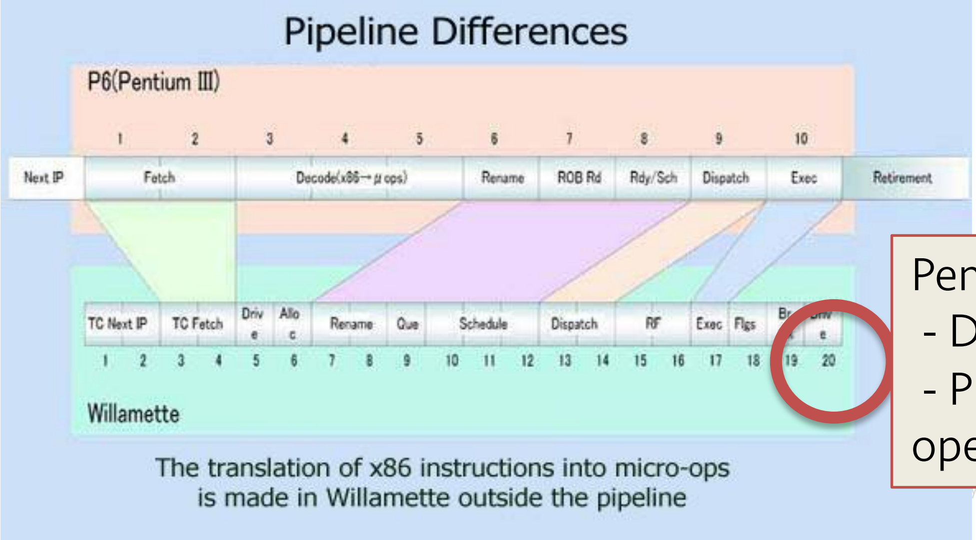
- Not fundamentally different than the techniques we discussed
- Deeper pipelines
- Pipelining is combined with
  - superscalar execution
  - out-of-order execution
  - VLIW (very-long-instruction-word)

Deeper Pipelines

- Power 4



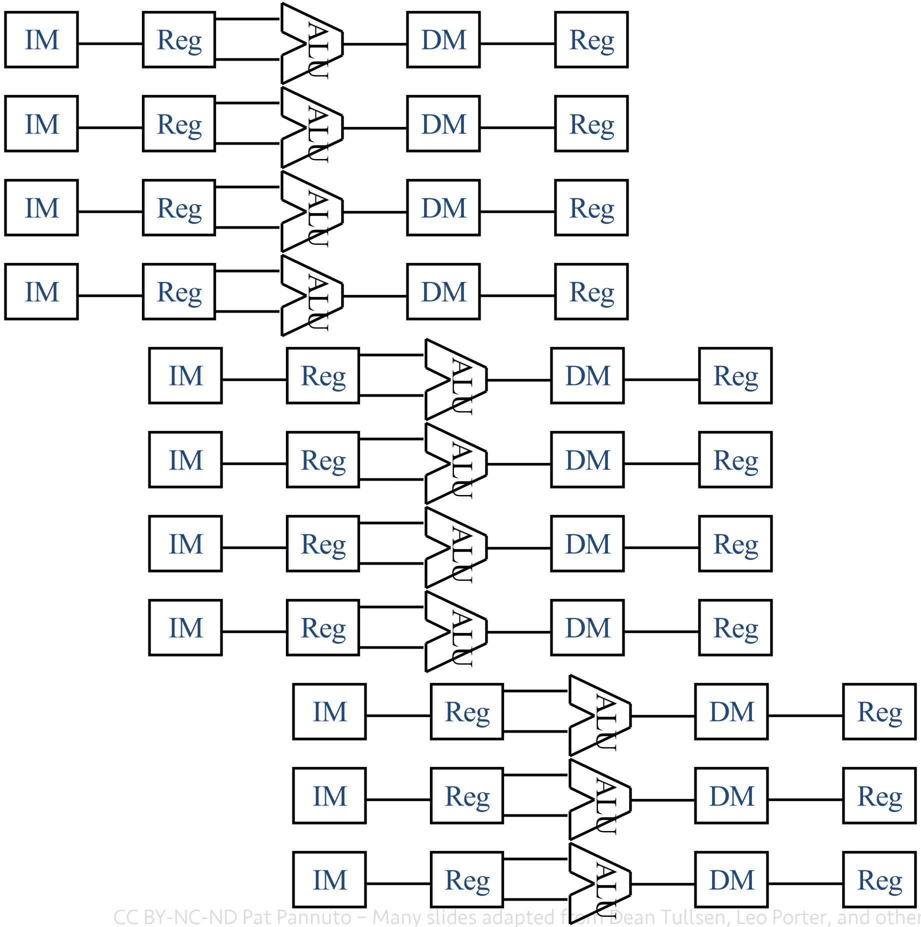
- Pentium 3
- Pentium 4


Pipeline Differences

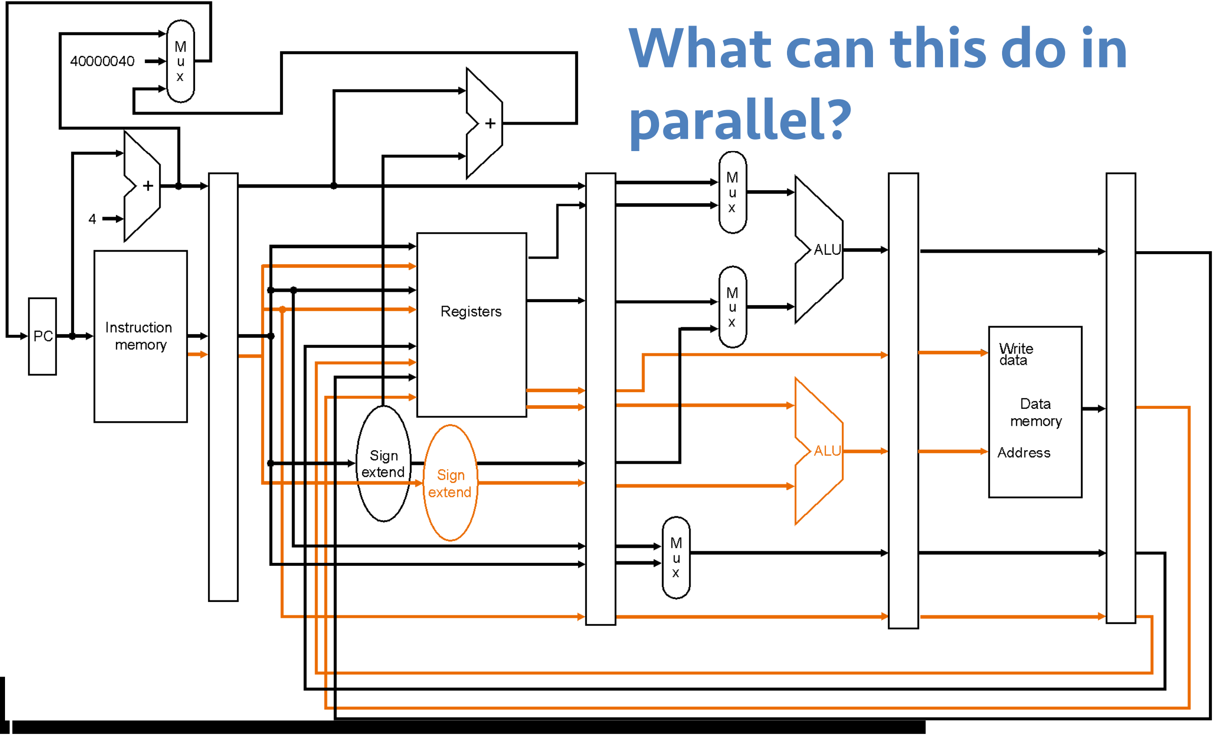
The translation of x86 instructions into micro-ops is made in Willamette outside the pipeline

Pentium 4 "Prescott"
- Deeper still: 31 stages!
- Planned for up to 5 GHz operation! (scrapped)

Superscalar Execution

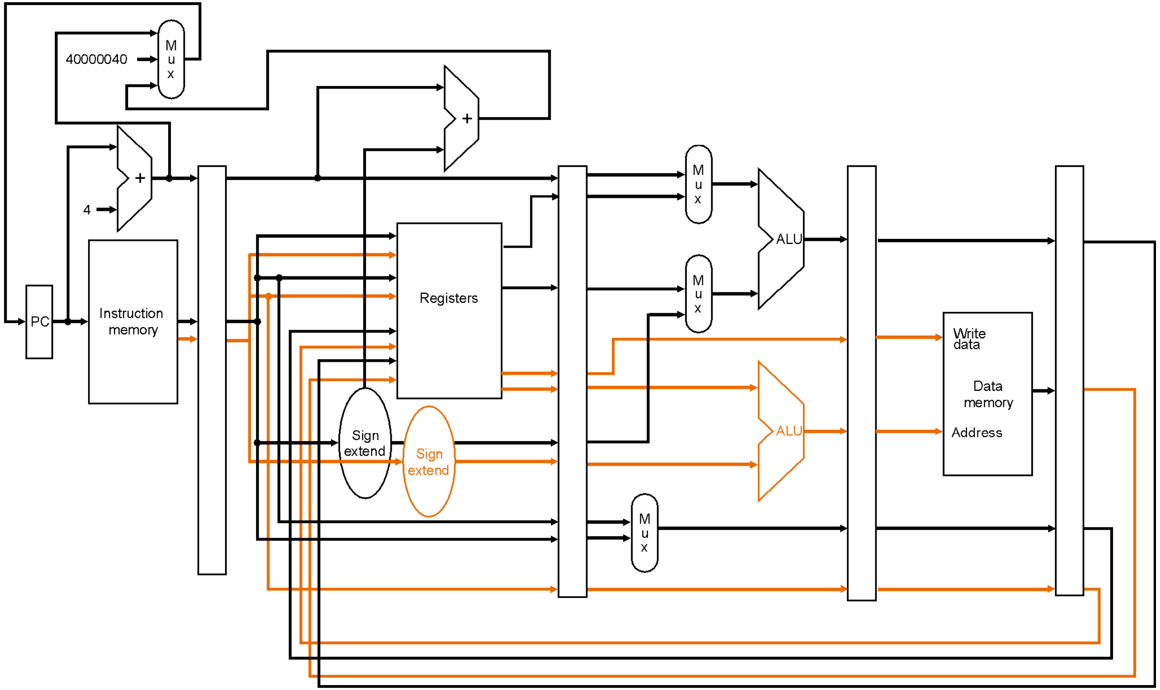


What can this do in parallel?



Selection	
A	Any two instructions
B	Any two *independent* instructions
C	An arithmetic instruction and a memory instruction
D	Any instruction and a memory instruction
E	None of the above

A modest superscalar MIPS



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

Superscalar Execution

- To execute four instructions in the same cycle, we must find four independent instructions

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

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
- If the four instructions fetched are only executed together if hardware confirms that they are independent, this is an *in-order superscalar* processor.

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
- If the four instructions fetched 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.

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
- If the four instructions fetched 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?

Superscalar Scheduling

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

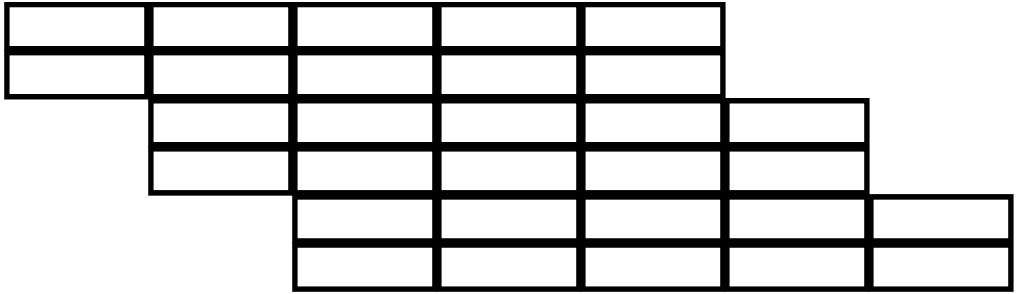
```
lw   $6,   36($2)
add  $5,  $6, $4
lw   $7, 1000($5)
sub  $9, $12, $5
```

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

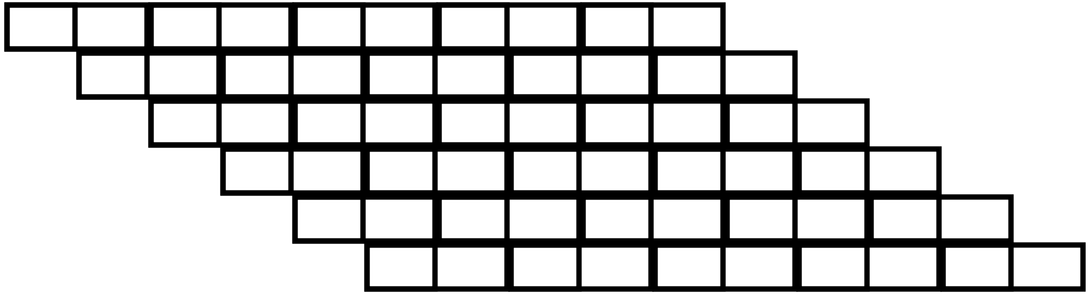
```
lw   $6, 36($2)
add  $5,  $6, $4
lw   $7, 1000($5)
sub  $9, $12, $5
sw   $5,  200($6)
add  $3,  $9, $9
and $11,  $7, $6
```

- When does each instruction begin execution?

Superscalar vs. superpipelined



(multiple instructions in the same stage, same clock rate as scalar)



(more total stages, faster clock rate)

Dynamic Scheduling
aka, Out-of-Order Scheduling

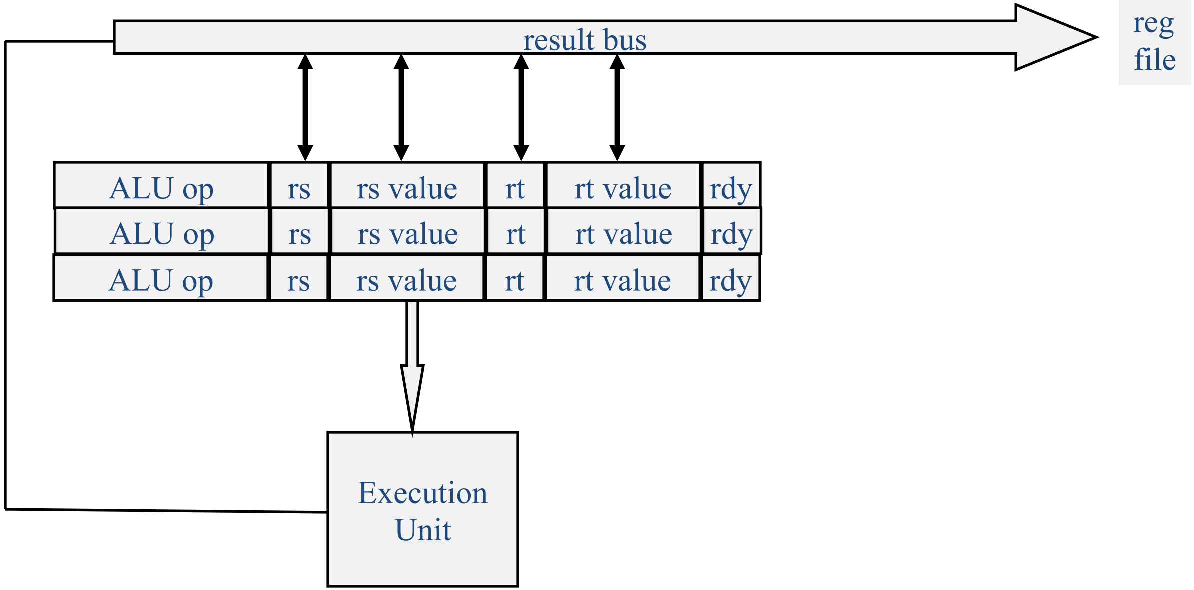
- Issues (begins execution of) an instruction as soon as all of its dependences are satisfied, even if prior instructions are stalled. (assume 2-issue, any combination)

```
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

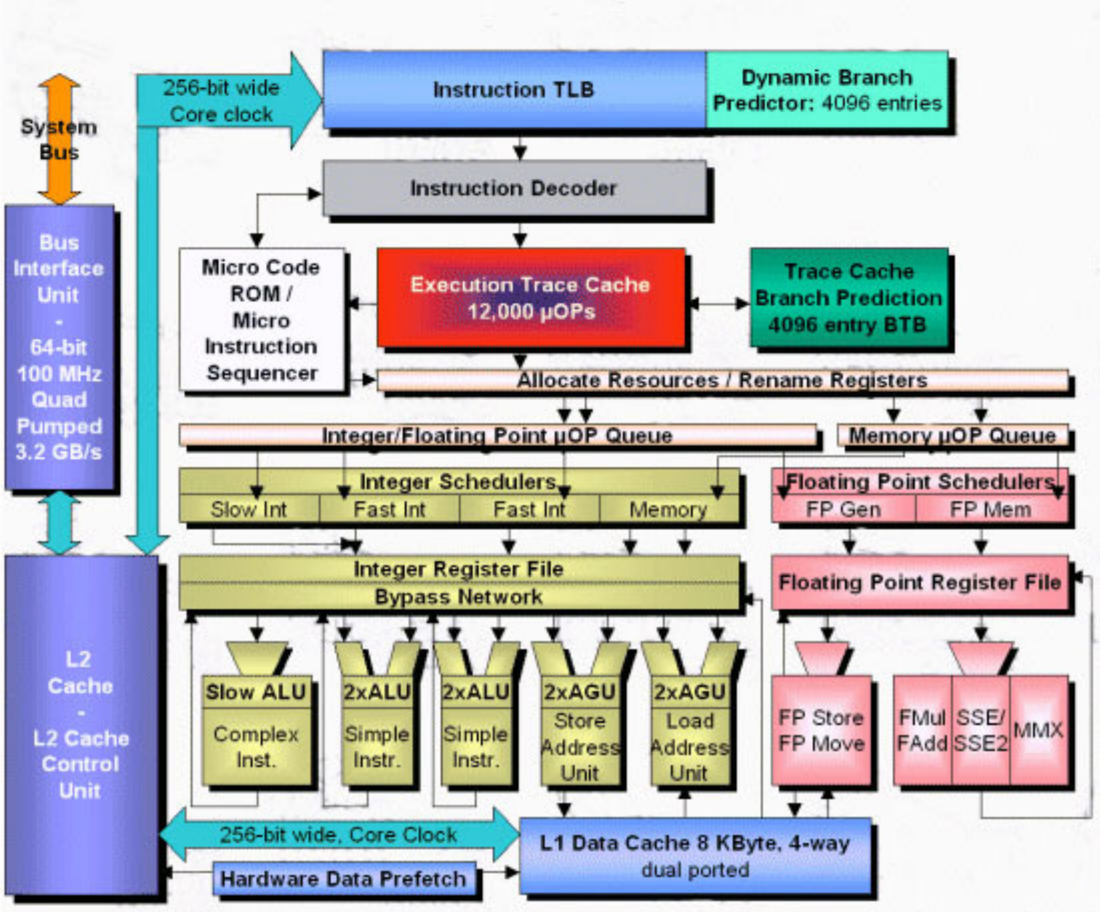
(other pieces: ROB, RAT, RRAT.. CSE 148 covers these!)

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



ALU op	rs	rs value	rt	rt value	rdy
ALU op	rs	rs value	rt	rt value	rdy
ALU op	rs	rs value	rt	rt value	rdy

Pentium 4


System Bus
256-bit wide Core clock
Instruction TLB
Dynamic Branch Predictor: 4096 entries
Instruction Decoder
Bus Interface Unit - 64-bit 100 MHz Quad Pumped 3.2 GB/s
Micro Code ROM / Micro Instruction Sequencer
Execution Trace Cache 12,000 μOPs
Trace Cache Branch Prediction 4096 entry BTB
Allocate Resources / Rename Registers
Integer/Floating Point μOP Queue
Memory μOP Queue
Integer Schedulers
Slow Int
Fast Int
Fast Int
Memory
Floating Point Schedulers
FP Gen
FP Mem
Integer Register File
Bypass Network
Floating Point Register File
L2 Cache - L2 Cache Control Unit
Slow ALU
Complex Inst.
2xALU
Simple Instr.
2xALU
Simple Instr.
2xAGU
Store Address Unit
2xAGU
Load Address Unit
FP Store FP Move
FMul FAdd
SSE/ SSE2
MMX
256-bit wide, Core Clock
L1 Data Cache 8 KByte, 4-way dual ported
Hardware Data Prefetch

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
  - 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
    - 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

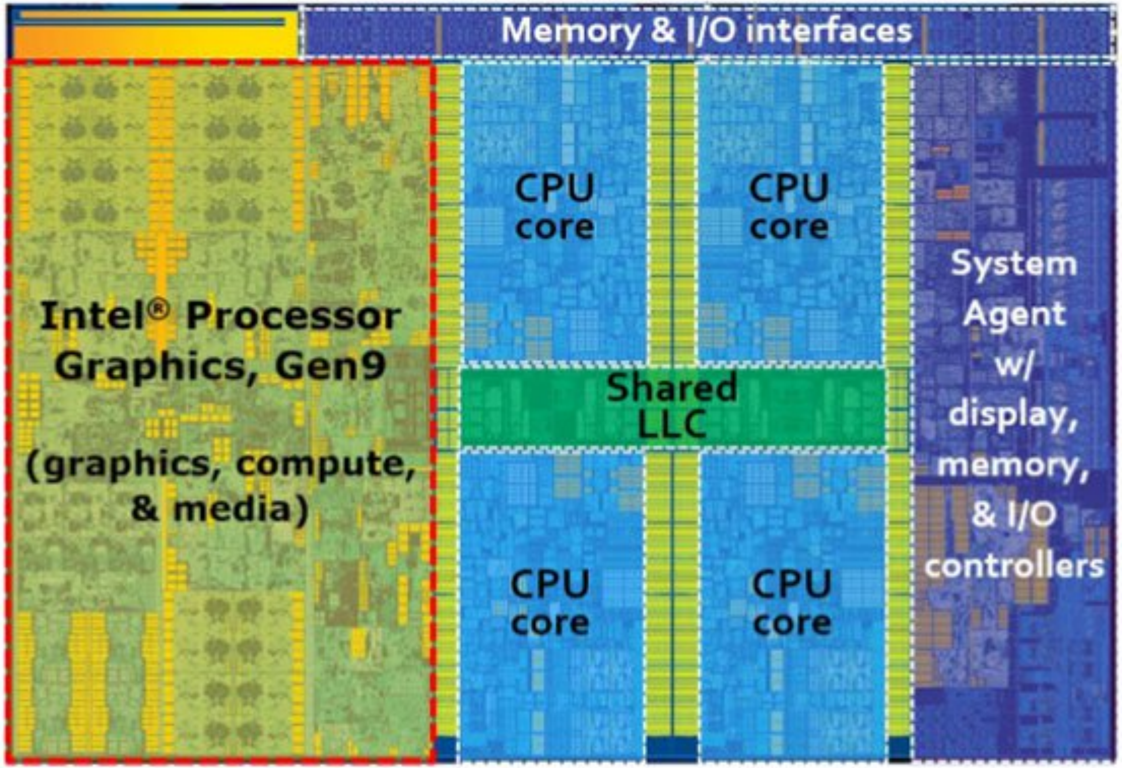
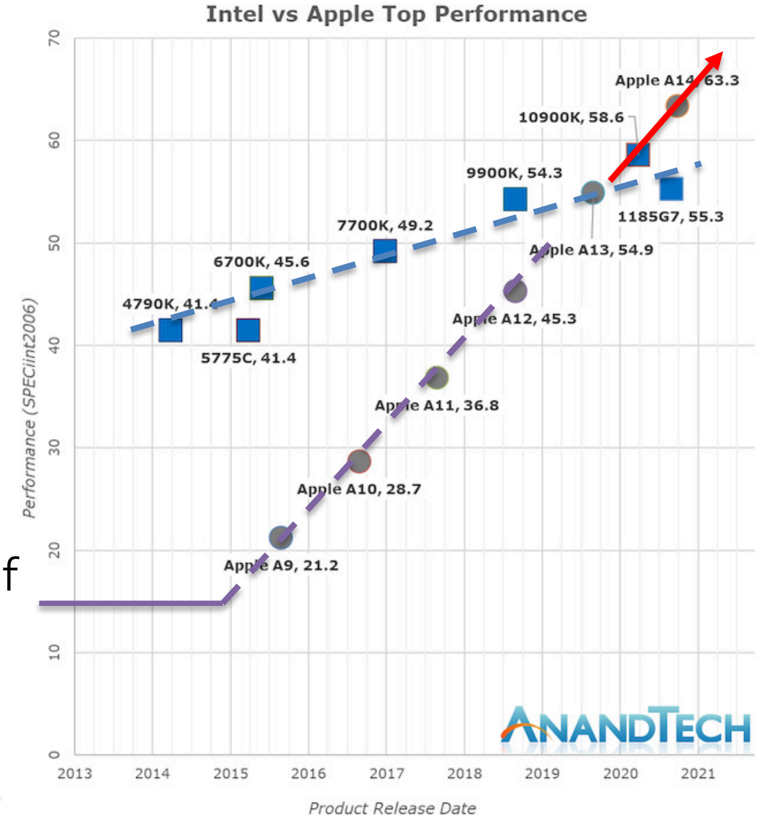


Figure 1: Architecture components layout for an Intel® Core™ i7 processor 6700K for desktop systems. This SoC contains 4 CPU cores, outlined in blue dashed boxes. Outlined in the red dashed box, is an Intel® HD Graphics 530. It is a one-slice instantiation of Intel processor graphics gen9 architecture.

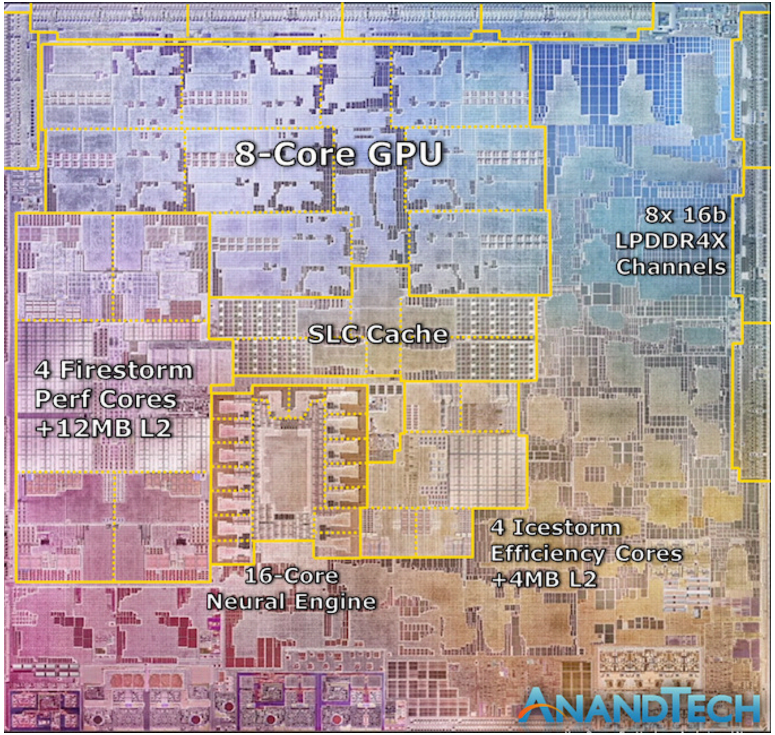
What do we know about the Apple M1? We can learn from the A14 (the M1 *may* be a rebranded, lightly enhanced A14)



This part implies they must be doing something different

This part makes a lot of sense for a new player

(Really this section should be what does Andrei Frumusanu know about the M1 – the AnandTech writeup is pretty good)

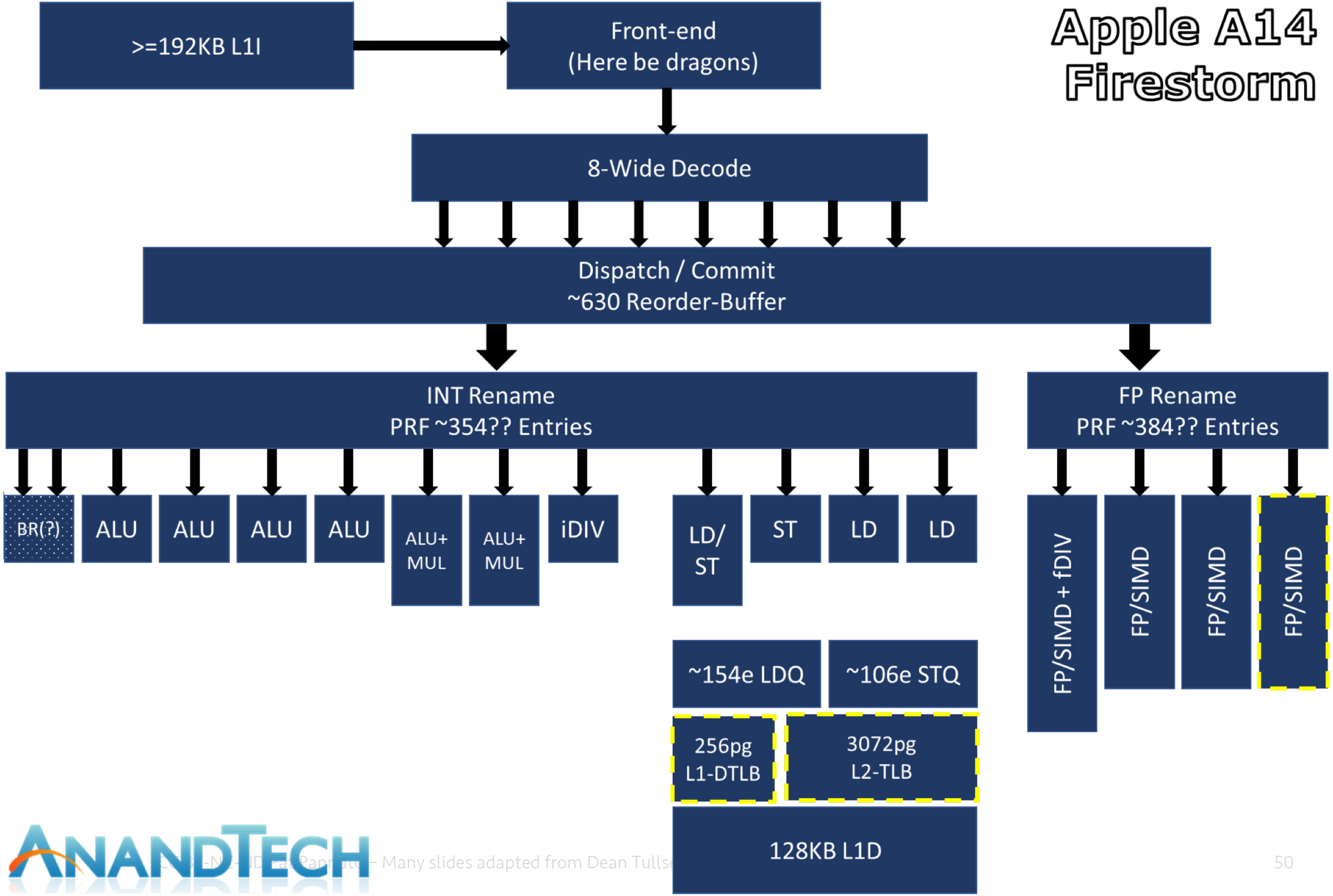


- 12 MB L2 cache [<u>this is huge</u>]
  - C.f. Intel Tiger Lake @ 1.25*4 = 5MB
  - C.f. Intel Cooper Lake @ 1*28 = 28MB
    - For $13,000
- Massive ILP
  - 8-wide instruction issue [SMT actual unclear]
  - C.f. Intel's 1+4 [CISC limitation??]
  - C.f. Samsung 6-wide [also ARM]
- Truly massive OoO window
  - ~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

The Internet's Educated Guess

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

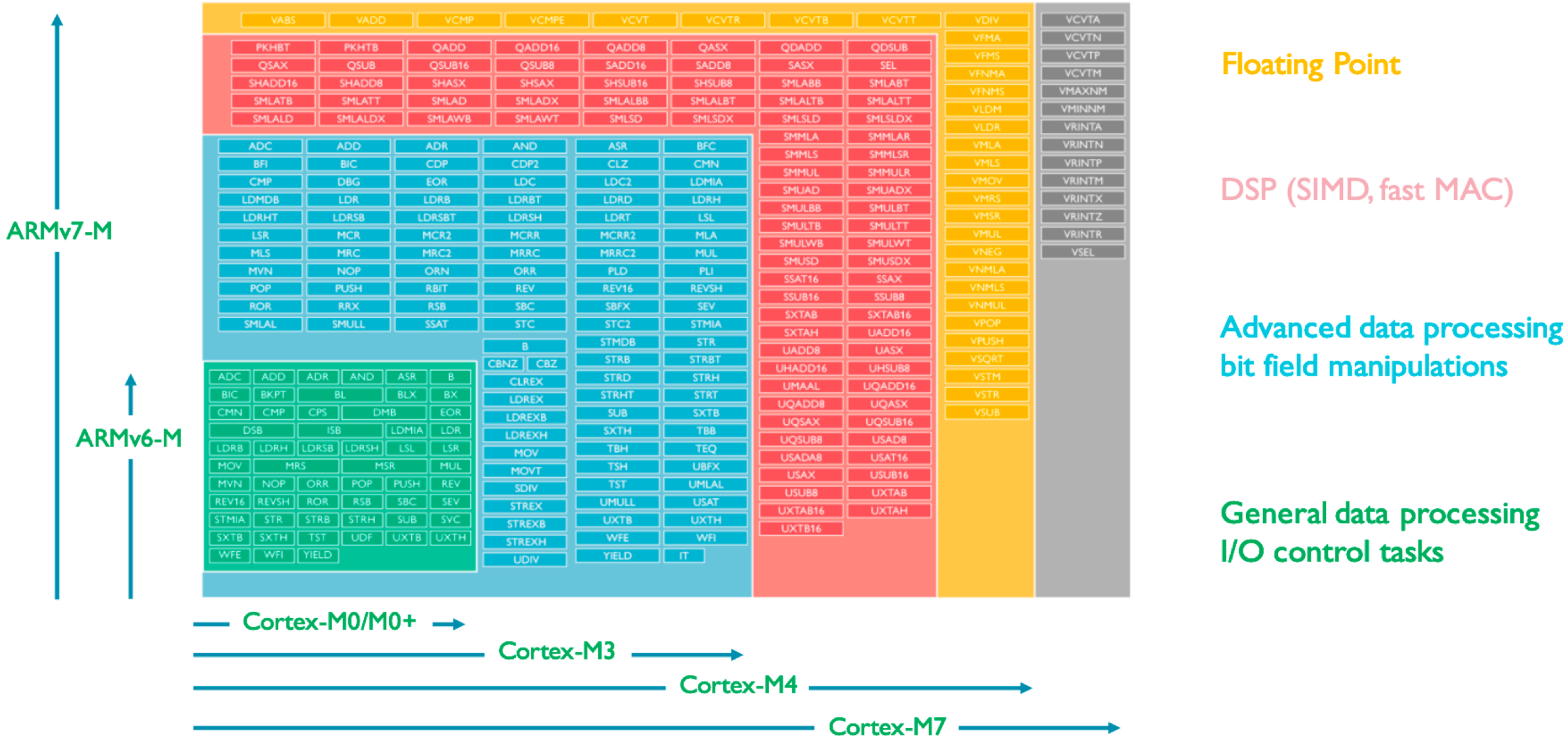
Part III: The Less Fancy Stuff in Real (Low-Power) Machines

- *How much of a real processor can we implement with CSE 141 alone?*

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

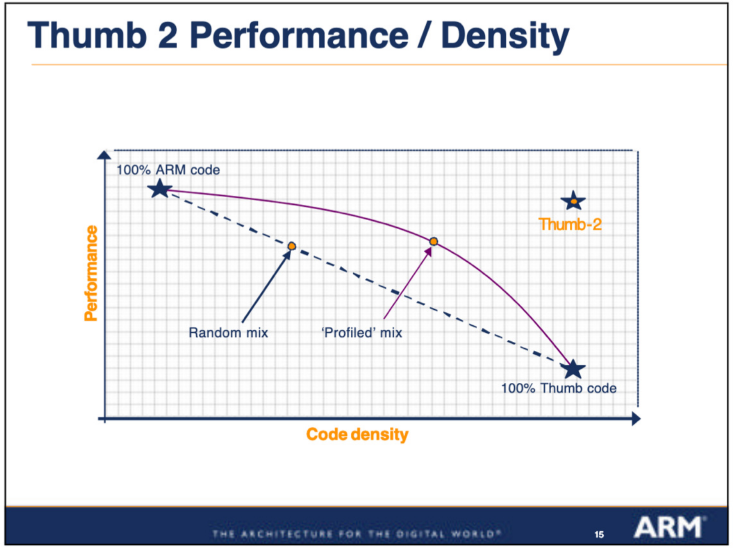
- Cortex A – “Application” processors
- Cortex R – “Real-Time” processors
- Cortex M – “Microcontroller” processors
  - (get it?)

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



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
  - Prefetchers help make unified memory designs fast
  - Q: How many instructions can prefetcher hold?



Thumb 2 Performance / Density

THE ARCHITECTURE FOR THE DIGITAL WORLD® 15 ARM

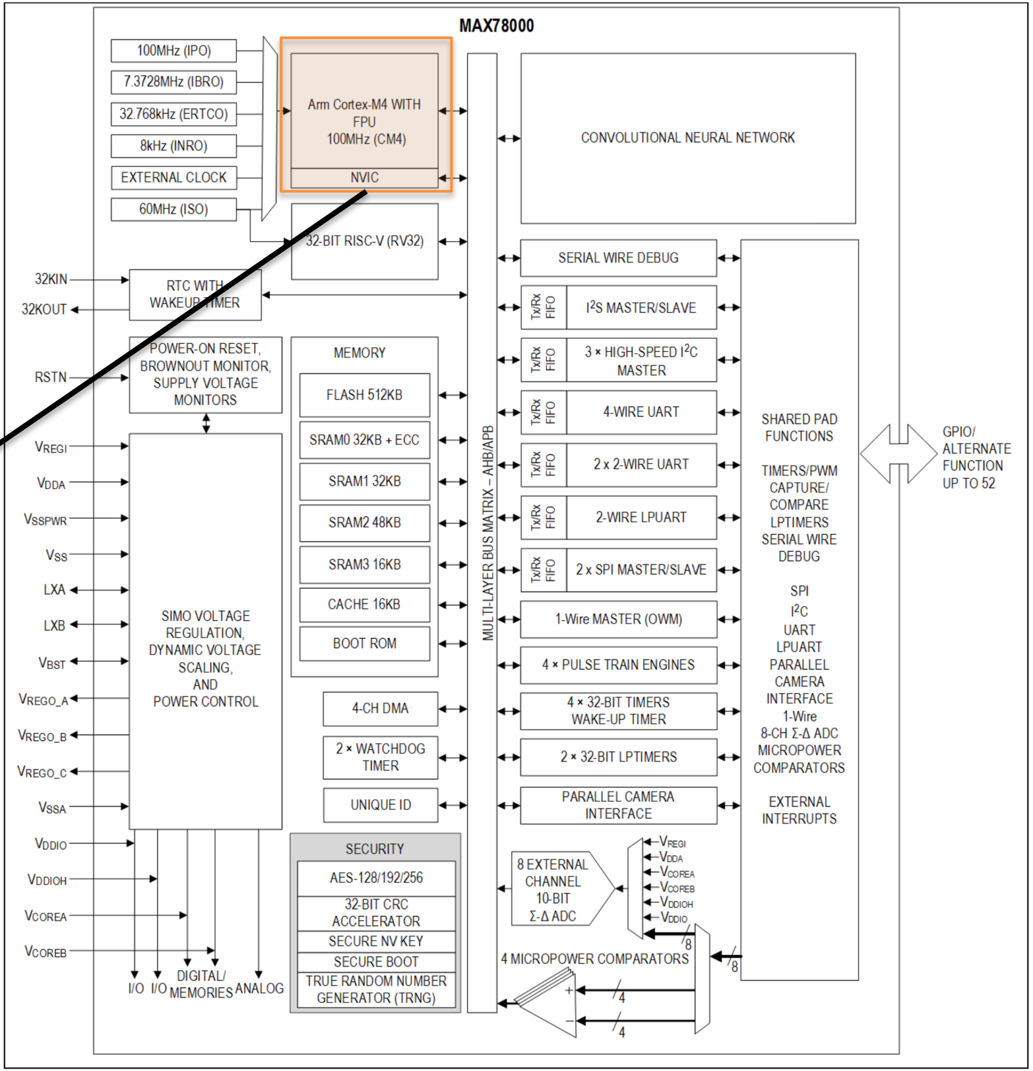
Implications of being area and energy constrained

- 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 200 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
- Q: How might Amdahl's law explain tradeoffs in embedded MCUs?
  - Embedded processors are *duty cycled*, modern ones run ~0.1% of the time
  - In embedded: Compute is not the bottleneck! New arch tradeoff opportunity!

How is ML at the edge changing the edge?

- Hot new chip: [MAX78000]
  - 22 uA/MHz Cortex-M4
  - + RISC-V Co-Processor
  - + CNN accelerator
  - + *many* peripherals

In this whole chip, this part is the processor



There are many, many more deeply embedded processors than high performance general purpose processors

- 0% of processors in the world are “high-performance” processors
  - Seriously, the number of Intel Core XXX and AMD XXX are a *rounding error* compared to AVR, MIPS (yes, our MIPS), PIC, ARM Cortex M’s, etc
- So why do we talk about the fancy machines?
  - Thought experiment: Which gives you the most aggregate processing power:
  - (Very) Coarse estimate: 1 trillion PIC-8’s in the world
    - Say, average 50 MHz, CPI of ~20 [for 32-bit math]
  - (Very) Coarse estimate: 120 billion ARM Cortex-M’s in the world
    - Say, average 24 MHz, CPI of ~1.25
  - (Very) Coarse estimate: 1 billion Intel Core i7’s in the world
    - Say, average 4 GHz, CPI of ~0.25

Advanced Pipelining -- Key Points

- Prediction takes real space, and structure informs operation
- Exceptions are another form of control flow
  - An "unexpected branch" or "unprogrammed branch" perhaps
- Scalar [fancy word for non-parallel] pipelining attempts to get CPI close to 1. To improve performance we must reduce cycle time (superpipelining) or get CPI below 1 (superscalar, VLIW).
  - What are the costs / problems of pipelining too deeply? When does better CT no longer improve ET?
- Modern processors are fast because they work on *many hundred* instructions at once
- Simple pipelines are valuable when raw performance is less important
  - Specialization can be more efficient, but only if you know workload!