M1 Core v 0.93

This remains a preliminary version.
Many of the patents I refer to I think I have the essential ideas correct, but that's the result of a quick scan and analysis, not a thorough reading or a tracking down of all the related patents.

We're all in this together, trying to understand. However I'm exhausted and others probably also have good ideas. I hope others can contribute, so we can continue the great work.
I will keep updating this and maybe publishing new updates every six months or so. Who knows when it will be done? When I started I had no idea it would become such a deep dive, or that so much could be (somewhat reliably) established.

Getting Started

Introduction

This document was written for myself, to remind me of, and to organize, my investigations into the M1. These investigations took the form of experiments, and reading many Apple patents, all tied together by a reasonable knowledge of the academic literature. The audience is anyone who is interested in technical details of the M1. The level is somewhat choppy, but assume a substantially higher degree of knowledge than ye average internet opinionator on CPU's. I've included a huge number of references to papers and patents -- if you want to understand this stuff, read them. Yes it takes works. Yes, you have to do that work; no-one else can do it for you.
I expect the presentation will be to pretty much no-one's tastes. What can I say – skip over the parts that don't appeal to you, whether that's how an experiment was designed, how it was interpreted, a description of the literature surrounding a point, or a patent dump.
There is some repetition, in part because some material naturally fits into more than one place, and because reading the same thing in different contexts helps with understanding.

Obviously I've done my best to make this accurate. Even so, there are probably multiple errors, whether of experimental design, implementation, analysis, my understanding of a patent, or anything else. Technical corrections are welcomed.

In addition to the other various references below, you may want to look at Apple's recently released CPU Optimization Guide, https://developer.apple.com/download/apple-silicon-cpu-optimization-guide/
You will need a (free) developer ID to access this. Mostly it confirms what Dougall and I have already established, though it does contain a few surprises. To me the most non-obvious items were
- it does not refer to the A17 at all (though it has constant references to the A16 and M3)
- it occasionally bundles the A16 together with the M3, which doesn't seem to make sense (most obviously because the M3 is 9-wide whereas the A16 is 8-wide). Presumably(?) the A17 is likewise 9-wide?
- there's been some degree of shrinkage ("optimization") especially in the A chips over the years. For example the A14 had 16MiB SLC, the A15 had 32 MiB, the A16 has 24MiB.
Likewise for L1D TLB M1 had 160 entries, A14 had 256 entries (as did M2, M3, and A15) while A16 moves to 160 entries.
This matches Dougall's investigation that the A15 seemed to have a few of the micro-architectural queues shrunk relative to the M1 (and presumably A14).
I expect this is hardly nefarious, deliberately making the chip worse; rather it's likely reflecting real world experience [many more testing hours than are possible in the simulator before the chip is fabbed], that transistors used in these structures can be re-allocated elsewhere with minimal performance reduction.

The document also provides a list of rather more performance monitor counters than previously available, and some explanation as to what exactly they count.
All in all it's what you want IF your goal is "how can I tweak my code to make it run faster on an Apple P- or E-core".
However it's NOT what you want (these documents are!) if your goal is to understand how these CPU's (along with the competition from ARM, AMD, Intel, etc) work at a fairly technical level. It also says nothing about the rest of the SoC (GPU, ANE, etc); possibly similar documents, with similar limitations, for those IP blocks will come.

Experimental Setup

To run the experiments you will need a test setup. I used the one created by Dougall Johnson here https://gist.github.com/dougallj/5bafb113492047c865c0c8cfbc930155, and https://gist.github.com/dougallj/c9976a52d592af24960ea7989cf652b1.
(Right now the above asm.py code fails with XCode 13.3, returning blank lines instead of disassembly. If it hasn't been fixed by the time you download it, look at the python code and, on line 16, change [7:-1] to [6:-1])

Dougall is the true hero of all this M1 investigation, doing the hard work of creating a useful test harness, especially all the low-level OS nonsense required to create JIT'able pages, set up the CPU counters, and so on, along with a python script to convert lines of ARMv8 assembly into machine code (required to make any interesting modifications to the tests).
Dougall also created the M1 instruction cycle counts web page at https://dougallj.github.io/applecpu/firestorm-int.html, and it's worth reading his investigations into the M1 at https://dougallj.wordpress.com/author/dougallj/.
Having said all that, much of my technique deviates substantially from both what Dougall did and what (Travis Downs, and, earlier, Henry Wong did). My technique is probably less numerically precise than Henry Wong's technique, but I found it easier to modify and change, something essential for this sort of open-ended research where one has no idea quite what to expect.

Note something extremely important and not at all obvious.
In Dougall's code there is a fragment of code called add_prep() that zero's out as many of the integer or FP/SIMD registers as feasible. This seems like an optional flourish that might be important for certain specialty measurements (like the size of the physical register file). Not so!
What is important is not the zero'ing of these registers (any value will do), it is marking them as "being used by this app". If this is not done then any code that reads these registers will run substantially slower than expected. For example, while the throughput of basic integer `ADD` is 6-wide, if your code is reading a register like x5 (eg `ADD x0, x5, x5`) that has not been "claimed" it will run at something more like 4-wide, with constant weirdness and results that don't make sense.

I mention this because, of course, I hit this very issue, occasionally switching this off (in my version of the code), forgetting to switch it on, and then wanting to cry as nothing I did from that point on made any sense or matched my earlier results.

(Note that little of this will make sense, especially if you're not yet an OoO CPU expert, till you have read the entire report.)
What is actually going on here?!? I have no idea, but much much later, after we understand many more features of the M1, we will revisit the issue. My *guess* is that we are colliding with a security feature that is supposed to prevent access to "unauthorized" registers.

Recall the SPECTRE exploits some years ago in all their various forms; the common theme was a fear that, under the right circumstances, generally speculation beyond an inappropriate point, a piece of code could read or perhaps even "influence the value of" machine structures that were not appropriate to it.
Apple has a specific patented solution for this for the branch predictor; essentially every entry in the branch predictor is tagged with a value that incorporates at least some bits from every sort of indicator that might be violated by an exploit, so the tag includes an exception level (hypervisor, OS, user), a processID, even some high address bits (to catch JIT'd code spying on the rest of the process). We will discuss this much later when we cover branches.

However this same scheme is very general, and is almost certainly being applied to registers. The idea would

be have a security tag for each mapping in the architectural to physical address mapping. Each time the mapping is referenced, the tag is compared with the current security tag and, in the event of a mismatch, various things happen which presumably include the CPU
- providing some value (zero, or random) that is not the architected value (and the value in the physical register), meaning that an app cannot, eg, read the registers used by the OS across a system call
- noting the mismatch somewhere, incrementing some register
- providing, probably, some debug mode (maybe only available within Apple) that would force an exception at this point.

Normal code should never find itself in a position where it is reading a register that it, itself, did not previously write (at which point thee security tag was set). It's only either malicious code, buggy code, or weird code (like ours, which cares only by timing `ADD`s, not the fact that the `ADD` is reading a random register value!) that would ever read a register with a mismatching security tag.

So, put it bluntly, every time my code tries to read x5, the core will notice that x5 is currently "owned" by some other thread (probably the OS or an interrupt), and this incurs some substantial additional cost. By overwriting every register at the start of our code, we “claim ownership” of that register and avoid this security violation and, in particular, its overhead.

We can probe this further. What if we use something like `MOV xn, xn`, rather than the current `MOV xn, #0`, to force the overwrite? That does *not* work! Either that instructions is mapped to `NOP` earlier in the pipeline, or the zero-cycle rename stuff kicks in, without ever validating the thread ownership of the source register.
(My bet is on `MOV xn, xn`, being mapped to `NOP`.)
It's worth noting that I can't generate the same sort of disaster by not forcing ownership of the FP registers. If the "loss of ownership" of the registers is occurring at an OS-call or interrupt boundary, and the OS/interrupt does not touch FP registers, this makes sense.

There is an alternative possibility, that we are seeing a mechanism for hardware context switching, to be discussed much later. However the facts seem a better fit with the security violation explanation.

As an aside, at this point might I point out how much I absolutely *LOATHE* XCode. Everyone associated with the Debug side of the product should be deeply ashamed of themselves. It is beyond pathetic that this multi-gigabyte behemoth provides a worse debugging experience than freaking Macsbug from 1981, let alone Think C or Metrowerks. In particular, the inability to *IMMEDIATELY* display a pointer as an array of that kind is beyond incomprehensible.
If I never engage in any more of this work for further Apple SoCs, a desire never again to repeat the XCode experience will be a large part of the reason.

Mathematica Setup (not relevant if you’re reading the PDF)

Need to use a conditional on “printing to PDF” to hide this!

This writeup was all done in Mathematica. If you have access to Mathematica, you can download the companion notebook and look at the actual numbers, draw your own graphs from those numbers etc. But most people don't have Mathematica, so for you I've printed the notebook to a PDF.

If you use Mathematica, we need a way to paste results data from the command line apps into Mathematica.
Easiest solution appears to be
http://szhorvat.net/pelican/pasting-tabular-data-from-the-web.html

When you first open this notebook, say **yes to "Allowing Dynamic Content"**. You will need this to activate the two UI elements ("Show Input" and "Outline") at the top of the document.
Next **choose Evaluate Initialization Cells** from the Evaluate menu (this will take a few tens of seconds to execute). This will load all internal variables (specifically all the many arrays of measurement data) into Mathematica, allowing you, if you want, to plot the graphs in different ways, or otherwise interact with the data.

Note the "Show Input" button at the top of this notebook; toggle it if you want to see the (sometimes copious!) input data for any particular graph.
The Mathematica code below adds that functionality to this notebook (not shown when "Show Input" is untoggled.

Also remember ctrl-clicking on a graph brings up a contextual menu, one of whose items, "Get Coordinates" is often useful in getting a quick, reasonably accurate feel for the coordinates of a point .

Theory of a modern OoO machine

Introduction

Around the 2000's up to the 2010's a number of nice articles were published giving overviews of how current OoO CPUs worked. However 20, even 10, years is a long long time in CPU design (think what you would do if you had 2^5 or 2^{10} x resources for a project, how differently you would proceed!), and ideas that were state of the art back then are baseline today.
You should be able to read something like David Kanter' s Nehalem overview, (2008) https://www.real-worldtech.com/nehalem/ and understand all the terms introduced, know what a ROB is, know what instruction scheduling is, know why register renaming exists, and so on. For a very simple overview of some aspects of a modern design, you should read (2008) https://carrv.github.io/2020/papers/CAR-RV2020_paper_15_Zhao.pdf *SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine.*
Our goal is to move to a level substantially beyond that.
You will encounter a lot that is unfamiliar, but I will try at every stage to explain the reasons why things are done, or not done, as they are.

For obvious reasons, some of this article has to be speculation. But it is informed speculation. We have three types of sources available.

- There are academic papers, and I will frequently reference these, which explain that something can be done, at least one way of doing it, and how well it works. They don' t prove that anything is implemented, in the M1 or anywhere else, anywhere, but they do explain the details of a technique, and that it is feasible.

- There are Apple patents, which explain in detail the precise innovation that is to be patented, and that often, as part of their explanation, include other interesting details of the design. It is always possible to claim that a patent, and the details that it contains, tell us nothing about how Apple actually does things; that the patent was simply filed as a good idea that Apple may eventually want to use but not yet. That certainly has happened – Apple has a large collection of patents filed around an architecture called Macroscalar
(2004) https://patents.google.com/patent/US8412914B2 *Macroscalar processor architecture,* based on very flexible indefinite length vectors, and which have not yet turned into a product. (It' s unclear whether SVE was in part inspired by Macroscalar; many of the ideas seem similar…)
However when a patent is narrowly defined, makes clear sense, and fits in with everything else we know, it' s sheer stubbornness to insist that it does not "prove" anything; our goal here is understanding, not impossible standards of certainty that will never be attained until perhaps the relevant designers write their memoirs .

To put the patents in context, remember a few dates:

- Sept 2012 is the release of Swift/A6.	3-wide OoO, first (visible...) Apple core	1st gen
- Sept 2013 is the release of Swift/A7.	6-wide OoO, add 64-bit	2nd gen
- Sept 2017 is the release of A11.	8-wide OoO, drop 32-bit, clusters (P and E) as the basic unit	3rd gen
- Sept 2021 [reckless speculation!]	10-wide?, ARMv9?, virtual registers?	4th gen
- Sept 2025 [doubleplus speculation!]	12-wide? drop x86 support?	5th gen

- Finally there are code experiments. One can write carefully designed programs, measure their timing, perhaps augment that information with readings from Performance Monitor Counters, and try to figure out from the results what' s going on. This is the only way to establish quantitative information – but one has to be careful.
Ultimately all the program tells you is how long it took; it does not tell you why. Benchmarks without understanding gives you Phoronix, but we subscribe to the Richard Hamming viewpoint: *the goal of computing is insight, not numbers.*
Much of the early information about the M1 is not so much incorrect as extremely incomplete, because people have been running these programs without a good model of the machine, interpreting it as much like a standard x86 machine. Our goal here is to go beyond that, to explain how these programs work, what they measure, and how to interpret what they measure.

I would hardly expect you to be able to read the papers or patents referenced right now. But hopefully, if you get to this end of this document and want to learn more, you' ll be in a position to work your way through them. Be patient! The first few times reading a paper or a patent is very difficult. The trick is to accept that you don' t need to understand everything.
Read the parts that you understand, skip the parts that don' t interest you.
But take time to struggle through the parts that do interest you, but are unfamiliar – that' s where the value is!

You will find, after repeating the exercise five or ten times, you have begun to understand the structure of patents and papers, at which point you can be a lot faster, knowing what can be skipped over and immediately heading for the good stuff. I tend to skim the diagrams, looking for those that represent the part I care about, then looking for the explanation of the diagram in the DETAILED DESCRIPTION OF EMBODIMENTS section (this is most easily done by search for one of the numbers that appears on the diagram). Lawyers care especially about the Claims section, which is the legally binding part, but I find there tends to be nothing interesting there; Whereas the Detailed Description part tends to be fully of statements like "in one embodiment of", which is almost certainly going to be the way Apple actually do it in their implementation. Sometimes what' s described in the patent may seem petty, or obvious, but in part becoming "one skilled in the art" is being able to look past the petty, obvious, parts to pick out the one gem, the one part that' s new or interesting.

The Basic Speculative Superscalar OoO Machine

Consider the basic out of order superscalar machine, as of say around 2000. Once we understand the general idea of this machine, we can consider all the many dimensions along which to improve it.

The CPU pipeline starts with a mechanism for deciding the address from which to load the next few instructions. This is non-trivial!
Normal'ish code contains about a branch every six or so instructions. The numbers vary depending on the type of code, but we can assume around a half of them are taken (meaning that the instruction pointer changes in some discontinuous way after the branch). So we are talking a change in PC around every ten instructions or so. If you're trying to run at around 8 instructions per cycle, you need to be able to handle a new PC discontiguous with the previous run of instructions, every cycle.
Clearly you cannot wait for the last branch instruction *executed* to tell you what the new PC is. Even under the best of conditions, that would result in a delay of maybe five or so pipeline stages between the fetch of a branch and when it is executed – five cycles while your fetch stage and most of your CPU is waiting, until it can jump to the next run of instructions. Clearly unacceptable!

And so we use branch prediction. The CPU guesses (based on past history) what the new PC will be at this point in execution (ie if the previous 64 times we encountered this branch, it jumped to address X, it's a good bet that it will do the same thing this time). For now let's ignore the details of how branch predictors work beyond accepting that they continually inform fetch of a (generally very good guess) as the address of the next few instructions in the execution stream.
The consequence of this prediction is that almost every instruction on the machine is executed in a speculative state, meaning that the machine needs to hold every result generated (including all stores) in temporary storage until the point at which it's clear that all the branches that affect a given instruction were correct, and so that instruction can Commit its state (that is, convert it from something temporary into something permanent).

We also perform instructions out of order. It's a somewhat surprising fact of real world programs that there's a lot of independence between successive instructions. This takes two forms. There is "immediate" parallelism, where successive stages of a chain of execution each require two or three independent operations.
If we bundle those together as a single "macroinstruction", we then tend to have chains of sequentially dependent macroinstructions, each about two to three instructions wide.
Your intuition might be that this means most code can only run about two to three wide, but that's not the case!

What most code looks like is that it consists of short chains of sequentially dependent macroinstructions (say 5 to 7 macroinstructions, 10 to 20 instructions long in total) which store their result to memory or a register, and that memory or register is not accessed until many (hundreds) of cycles later.
This means that while each sequentially dependent macroinstruction has to execute one after the other, you can execute many of the chains in parallel...
That sounds good but you need a variety of machinery to track which instructions are independent of previous instructions, and to track the program order of instructions so that as branches are resolved as correct, you know which of the instructions in program order now resolve as correct.
(This fact is why so many people's intuition about the value of superscalarity is so flawed. Most people

hone their assembly optimization skills on long stretches of sequentially dependent instructions; but such code is actually unrepresentative of most of what runs on a CPU.
This fact is also why OoO superscalarity works so well, whereas most attempts to create static wide machines have been problematic. All the pieces -- out of order, prediction, and superscalarity -- work synergistically. In particular most of these chains that are running in parallel come from different basic blocks [ie are separated by some sort of if() statement that the compiler can't see past] and so are impossible to aggregate statically.)

So the basic machine fetches instruction in a guessed instruction order, allocates resources to each instruction in order, throws the instructions into a large pool from which they Execute as soon as they can (ie once their Dependencies are satisfied), and then Retires the instructions in complete program order.
Retirement is the point at which all the guesses along the way are tested for correctness (and if they fail, we flush everything after the wrong guess and start again). Retire (because it happens in order) also undoes all the confusion caused by the OoO execution.

So let's think about the consequences of all this. What sort of state needs to be held as tentative until it can be Committed?
One obvious set of state is stores.
There are less obvious issues surrounding loads.
There are possible exceptions that were raised (eg by loads or stores that had invalid addresses).
And there are register values that need to be restored to whatever the programmer view of the registers was at the point of restoration.
(Even less obvious is the state that is used to inform predictors. You can update your branch predictors, or your prefetchers, as soon as the relevant instructions are executed. But you may be updating them with flawed data...
It turns out, another non-intuitive result, that on the data side, eg for data prefetch, this is mostly not a big problem, whereas on the instruction side, if you want quality accuracy for both your branch predictors and your instruction prefetchers, you need to ensure that they are not polluted by incorrectly speculated paths. [You also need to ensure that they are not polluted by interrupts, which throw in behavior that doesn't actually represent the flow of control you are trying to model.])

So we need a variety of structures to keep track of all this. Traditionally the largest of these structures is known as the ROB. This stands for Reorder Buffer which is not an especially helpful name. A better way to think of it is as Retirement Buffer, and to think of retirement as the point where *two* tasks are performed
- every instruction is given its last chance to either Commit its results to programer state (ie we agree that the instruction is not speculative at this point, and has raised no problematic issues like an exception)
- all resources that were allocated to the instruction can be returned to the machine for reuse.
(We can ignore the second point for now, but will return to it in time.)

So the basic flow of instructions (and remember, under ideal circumstances, each of these stages is dealing with around 8 instructions during the same cycle, and all stages are happening in parallel on a

different 8 instructions!) is:
• Fetch (all the branch prediction machinery)
• Decode
• Map
• Rename
• Coarse Scheduling
• Fine-grained Scheduling
• Execution
• Retirement
• Commit

What do each of these do?

Decode transforms instructions from their representation in memory (ie 32 bits) into something that's more convenient for the machine. Of interest to us, at this stage pairs of instructions may be fused, or meaningless instructions (mainly NOP) can be thrown away as irrelevant to the rest of the pipeline. NOP might seem a special case, not worth special treatment. Why bother with an instruction that does nothing, and why bother to treat it efficiently?

In the first place, much of the low-level machinery of modern operating systems is based on dynamic linkers and position independent code. This code is created in multiple pieces (separate compilation) and joined together by a linker. This joining process (the details of creating code that is position independent and can act as a library that can be called simultaneously by multiple programs) requires that all the calls in the code (app or library) that look like they might cross from one piece of code to another need to have a particular structure that might consist of two or three specific instructions. But when the linker actually stitches the code together, many of the calls that seemed like they might need to be "international" are in fact only local. Hence the two or three instruction slots that were reserved for an "international call" to a separately maintained piece of code, can be filled in with a local call (single instruction) and one or two NOPs.
Thus real world code contains a surprising number of NOPs, so why not make them as cheap as possible?

ARM also defines a family of HINT instructions (basically NOPs with different bits set in the instruction) might act as various hints for prefetching, branch prediction or other such control. A machine may understand a particular hint, in which case it will be treated as a real instruction, or it may not understand this particular hint, in which case it will be treated as a NOP and just ignored.

Map handles register renaming. (Yes the naming seems wrong, be patient.)
Recall that part of the machinery of speculative OoO is that the logical registers of the machine (the programmer visible registers, let's say 32 for ARM integer registers) are mapped onto a much larger pool (hundreds) of physical registers.
The idea is that at any particular time in the CPU, there is a map saying that logical register rN is

mapped to physical register pM.
So consider an instruction like

```
ADD r2, r1, r0
```

This instruction has two input registers (r0 and r1) so we need to consult the mapping table to figure out that r0 is currently mapped to physical register p7, and r1 is mapped to p45.
The add also takes a destination register, r2, so we need to create a new mapping for r2 (a new mapping is created every time a register is overwritten). This "single-write" rule means that the physical register can holds temporary state as long as the instruction is speculative; no other instruction is allowed to reuse a physical register until we can be absolutely certain that the temporary result it stores won't ever be needed again.

This concept of renaming registers should be familiar, and you should take some time to think about exactly how it operates, how the mapping table would be updated, when it is safe to reuse a register, and so on. But the most difficult part of the problem is one you probably didn't think of!
Remember, the machine may have to remap (source registers, plus allocation of new physical registers for all destination registers) up to eight instructions in a cycle. The tricky part of this is that often the same registers are used in many of these instructions. For a trivial case consider just the two instructions

```
ADD rA, rB, rC
MUL rD, rA, rF
```

The rA physical register must be allocated for the Add, before the mul is handled because the physical register looked up in the mapping table for rA (and rF) must reflect the mappings *after* the add instruction.
So you can't just remap all eight instructions independently! You have to figure out the successive name dependencies between all the instructions and treat that appropriately. That's the most difficult job of Map, and why it has a separate pipeline stage. It's an interesting fact that Apple is very clear, in multiple patents, that they use this separate Map stage, whereas most other companies do not mention such a stage. Being willing to move the most difficult part of the job to a separate stage may be part of why Apple has been able to maintain such spectacular CPU widths?

Rename is the traditional name given to a stage that should really be called Allocate. This is the stage where each instruction is given the resources it requires to perform its job as part of a speculative OoO machine.
What sort of resources need to be allocated?

The most basic is a slot in the ROB. The ROB is a queue of instruction in program order (this order is speculative, as best can be guessed by the branch predictor, but assuming that's correct, instruction are stored in the ROB in program order with no OoO weirdness). Each cycle the instruction that have just arrived in Rename are allocated a slot at the end of the ROB, while (as a separate stage) the instructions at the head of the ROB are tested to see if they have completed, and completed correctly.
If the instructions at the head of the ROB have completed correctly their resources are returned to the system and the head of the ROB queue moves down a few slots.

If they have completed incorrectly, corrective action is taken (maybe flush for a branch misprediction, maybe call into the OS for an invalid load or store address).
If the instruction at the head of the ROB has not completed, it remains at the head of the ROB until it completes.

The consequence of this is that suppose an instruction begins to be executed that can take a long time (say a square root). This instruction may stay at the head of the ROB for many cycles, while other easy instructions after it get their slots in the ROB marked as complete. By the time the square root completes, there may be 80 instructions directly after it in the ROB that have all been marked complete. The CPU will only then start retiring instructions from the ROB, as fast as it can. And it will keep retiring as fast as it can until either the queue runs dry or it hits another instruction that isn't yet marked as complete, at which point it will again wait until the head instruction is marked complete.

The most extreme version of an instruction that takes a longtime to complete is a load that misses all the caches and has to go to DRAM. This can take hundreds of cycles. Since the load cannot exit the ROB until it completes, instructions pile up behind it. At some point every slot in the ROB is full (or some other resources has been used up), and the machine halts until the load completes.
Thus the primary significance of the ROB's size is that it represents how well the machine can cope with a load that misses to DRAM. In time we will explore exact numbers, but assume a CPU that is 8-wide with a ROB that can hold 640 instructions. That would mean that if a load blocked the head of the ROB, the OoO part of the CPU could keep processing instructions for at least 80 cycles (assuming a full 8 instructions can be executed every cycle which is probably a little optimistic). That's good enough that the machine will not have to halt on a miss to L2 cache (around 15 cycles for M1) and will usually cover the delay to the System Cache (around 90 cycles for M1).
But when this 640-entry ROB machine misses to DRAM, (taking about 100ns, so about 300 cycles), eventually all the slots in the ROB will be filled (waiting for the head of the queue, the load from DRAM, to retire). Rename will not be able to allocate a slot to its instruction so they will not move down the pipeline. So the instructions in the Map stage will not be able to move into Rename, those in Decode will not be able to move into Map, and so on. At this point no new instruction can enter the machine until the load completes.

In addition to ROB slots, there are other resources required by different instruction types.
- Any instruction that generates a result needs a destination physical register to hold the result, and traditionally that would be allocated here (Map decided on an appropriate ID for the physical register, but the actual allocation of the register could be delayed till this stage.)
- Load and Store require slots in the Load and Store queues (for reasons we will describe).

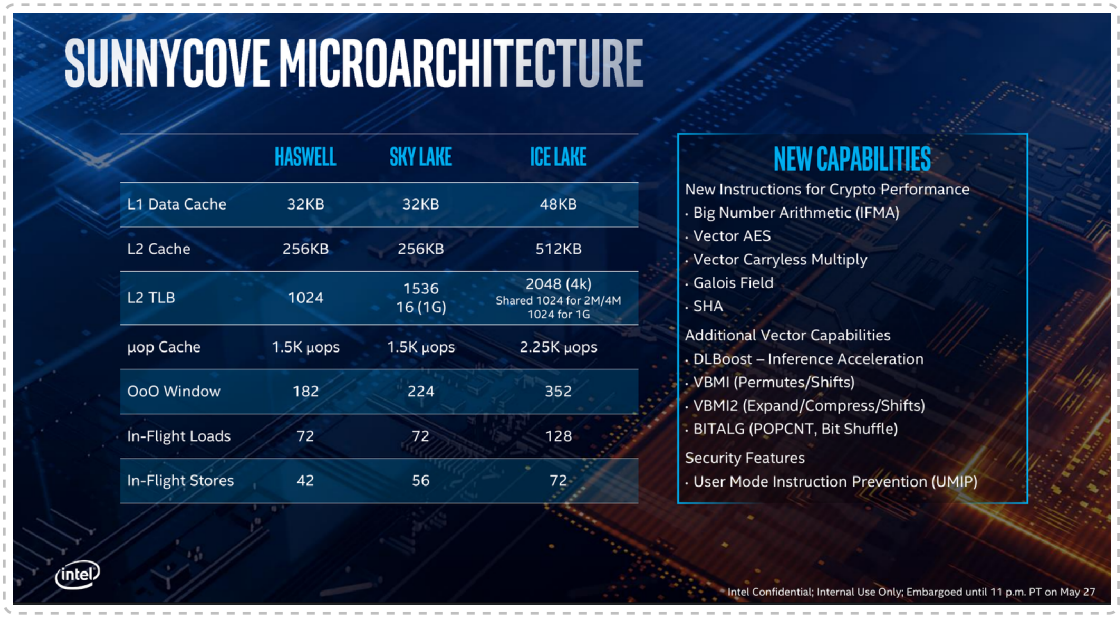
If any of these resources (physical register, load slot, store slot) are unavailable, again the the flow of instructions halts at this point, until some instructions in the ROB retire, and release the appropriate resource.

The extent to which execution can keep going after a load misses to DRAM depends on the size of the ROB. But the ROB is essentially just a queue, a low power structure that can easily be made larger. So why not make the ROB thousands of entries in size, so that we can sustain load misses all the way to DRAM?
Because a large ROB is no help if we run out of other resources along the way and so our machine grinds to a halt when only a few hundred ROB slots are occupied...

The problematic resources as far as Rename is concerned are the physical register file and the load and store queues (usually referred to as a single object, the LSQ). Both of these are large in area, power hungry, and difficult to grow .(Even if you are willing to pay the area and power costs, as they grow larger they grow slower, and if they become slower than can be accessed in a single cycle, performance falls off a cliff).
So you generally grow the physical register files and the LSQ as large as you can (given your area, power, and clock budget) and then scale the ROB to a size that seems to make sense given these resource limits.

You want LSQ to be as large as possible because, while the head of your ROB is blocked behind a load that is missing out to System Cache or DRAM, the instructions behind that load might consist of a large number of other loads (most of which hopefully hit in L1), or stores, or instructions that write a result to a register, and you'd like to do as many of these as possible (hundreds if necessary). Given that you could have (as we said, say 640 instructions piling up in the ROB this suggests that you might want to have access to many hundreds of physical registers, and perhaps a few hundred load or store queue slots. Those are large numbers!
Compare with the competition:

In[ ]:=



We see that Intel are running at a ROB of 352, with a load queue size of 128, and a store queue size of 72, also 180 integer physical registers and 168 fp physical registers.

(More details here if you want: https://www.hardwaretimes.com/intel-sunny-cove-vs-amd-zen-2-core-architectures-10th-gen-ice-lake-vs-ryzen-3000/)

The above, as I have described it, is that traditional role of the ROB, along with the constraints imposed by various resources. As we will see, one reason Apple can do so much better is that they substantially rethink this traditional design.

Scheduling. Instructions are processed in-order up through Rename (ie Allocate). After this they are placed in a scheduling queue.
The point of the scheduling queue is to provide a buffer until the “resources” required for instruction execution are available.
Every instruction describes what it needs to execute. Some of these resources (eg ROB slot, or destination register) have already been discussed, but to execute the instruction also requires its data inputs, and an execution unit. Consider, for example,

```
ADD rA, rB, rC
```

This requires an execution unit that can perform ADD, and the data value for rB and rC, which may not yet have been calculated.
So the instruction sits in the scheduling queue and, essentially, every cycle the scheduler checks "has rB become valid? has rC become valid? is an adder free?" Once all three are true, then the instruction gets moved on to the execution unit.

A scheduling queue is another very area intensive and power hungry structure. Instructions are moved into and out of it at random places, and testing (for whether the instruction can now execute) has to be redone every cycle for every instruction in the queue.

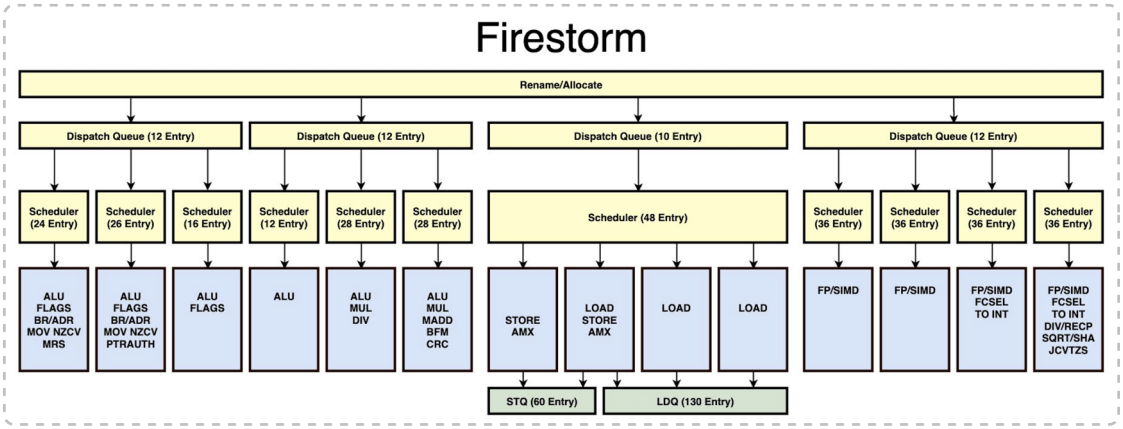
The scheduling queue is yet another constrained resource in the CPU. In any cycle, an instruction may not be able to execute because its dependencies have not yet been calculated, or an execution unit may not be available. It is possible for more and more instructions to pile up in the scheduling queue until it is full, at which point, once again, yes, everything grinds to a halt until one of the instructions in the queue has all its requirements satisfied and can be fed to an execution unit, freeing up its slot in the queue.

There are a variety of technical details of how exactly one might design a scheduler queue to try to reduce the power and area. But we are interested in a higher level design issue:
Intel has traditionally used a single large scheduling queue, which is expensive for the reasons given, but can be used by any instructions.
Almost everyone else uses multiple scheduling queue (for example maybe one for integer operations, one for load/store, and one for FP). This allows each queue to be shorter (lower area, lower power, easier to make it fit cycle time constraints), but it can mean that your integer queue has filled up, while your FP queue is sitting empty, and there's no way to use that FP queue space.

One can then go to the opposite extreme of, instead of one queue for integer operations, one has say a

separate queue in front of every integer unit; and this is what Apple does, they have multiple functional units (eg six integer units, four FP/SIMD units) with separate queues in front of each. Each scheduling queue may sound small (36 entries compared to say Intel's 97) but when you sum them all up, Apple has many more scheduling queue entries. Here's a diagram (edited slightly, from https://twitter.com/-dougallj/status/1373973478731255812)

In[ ]:=



You can work around the queue imbalance problem to some extent if you have a two level scheduling system. The way this works is you have a "coarse buffer" that accepts instructions out of Rename, and then, as slots open up, moves them to an appropriate scheduling queue in a load-balancing fashion. In the diagram above, this is called the Dispatch Queue (though that's incorrect; a better name would be Dispatch Pool, because these pools do not attempt to preserve instruction order, unlike queues).

How many scheduler slots does one want? Lost, sure, but how many exactly? What do more slots get you?
Remember the goal of an 8-wide OoO machine is to execute 8 instructions every cycle. What sort of things prevent that?
Issues that we will cover later include
- ensuring that 8 instructions are available every cycle,
- that branch prediction is rarely incorrect, and
- that loads can usually be found in cache.
Let's ignore the fetch and branch issues for now, and consider a given stream of instructions.

What most instruction streams look like is, first, at a rough level, about 15% branches, 10% stores, 25% loads, leaving 50% integer operations.
FP tend to pull stores and branches down some, and integer quite a bit, leaving space for fp operations.
You can see the sorts of analyses people do here: (2018) https://tosiron.com/papers/2018/SPEC2017_IS-PASS18.pdf *A Workload Characterization of the SPEC CPU2017 Benchmark Suite.*
So a standard instruction stream will consist of branches (we ignore, that's Fetch's job) stores (we ignore because they are fire and forget) and a whole lot of loads and integer instructions.

Mostly these instructions occur in small clumps (they are separated by a branch every five or six cycles,

and there are limits to how much a compiler can structure code across branches), with maybe two or three independent instructions, followed by another two or three instructions that each dependent on the previous instructions. To make this run fast we need to be able to queue up instructions that cannot yet execute because they depend on results from prior instructions (which may be executing, or in turn waiting for earlier instructions). And we need to be able to look in this pool of instruction to find every cycle at least eight that are ready to execute. The larger your pool of instructions waiting to be scheduled, the more instructions you can have look over every cycle, hoping to find eight or more that are executable.

People frequently confuse ROB size with scheduling queue size.

- As a practical matter, ROB size determines *how many instructions the machine can continue to execute after a load misses to DRAM*.

At any given time, many to most of the instruction the ROB have completed, they are simply waiting their turn to be Retired because Retirement happens in order.

- As a practical matter, Scheduling Queue size determines *how far ahead in the instruction stream the CPU can look for instructions to execute that are independent of the instructions currently executing*. At any given time every instruction in the scheduling queue has not yet executed, and it is waiting to execute as soon as everything it requires (dependency data and execution unit) becomes available.

So if we assume as rough rule that a dependency chain is about ten instructions long, and we'd like to sustain eight instructions per cycle, we'd like our Scheduling queue to be at least 80 instructions in size; and of course, like all resources, unevenly balanced situations may arise (much longer than usual dependency chains, for example) so we'd like as much larger than 80 as our design, power, area, and timing allow. Apple's techniques (many, shorter, Scheduling Queues, kept balanced by Dispatch Pools) allow them to do extremely well, achieving a lot better than Intel (looking further ahead in the instruction stream, at lower power, while issuing many more instructions) than Intel.

Be aware that the language related to scheduling is somewhat confused.

IBM (and Apple) use the terminology that moving instructions into the Scheduling queue(s) is called Dispatch, and moving them out of the queues (to an execution unit) is called Issue. But Intel tends to use those words with the reverse meaning. So generally don't get too obsessed about which is used for what, look at the context to understand exactly what is meant.

Also, Apple and IBM use the term Reservation Station for what I have been calling a Scheduling Queue, because this is the Intel, and thus the internet default, terminology.

An interesting point here is how are dependencies tracked. I have described this as the instruction

```
ADD rD, rA, rB
```

depends on rA and rB; rA is mapped to physical register pM, rB is mapped to physical register pN; and the instruction cannot be Issued to Execute until pM and pN are marked valid.

This seems so obvious and natural (and appears to be how Intel do it) that most people think it's the only way to do things. But there are alternatives.

In particular rather than tracking the dependency on rA through physical register pM, what if you track a dependency on the instruction, call it iR that will calculate pM? So we say that the ADD depends on

instructions iR and iQ, and won't execute until those have both completed. Logically this is the same as a dependence on registers, but how would you implement it, and why?
Implementing is easy – every pending instruction has a unique, unchanging number, namely the which slot in the ROB it occupies, and which is allocated early in the pipeline. So we can use that as an instruction identifier.
Why do things in this way that seems unnatural? Well, where do instructions get their operand values from? From the register file, sure, that's what you think, but how exactly is this done?
There have been many possibilities but let me just describe two.
One possibility is to copy the value from the register file into scheduling queue when the instruction is placed in the queue. This assumes the value in the register file is already valid, and requires extra storage in the scheduling queue. It seems (IMHO) kinda pointless except in light of the historical evolution of the technology where it was one of these easy extensions of the even earlier method.
Another possibility is to read the value from the register file as the instruction is issued for execution. That gives a little more time for the value in the physical register file to be valid, and doesn't waste space replicating the value in the Scheduling Queue.
But both these suffer from the question of: what if the previous cycle created one or both of the register values required by this instruction? Do I have to wait a cycle for the result(s) to be written to the register file before I can read them back out of that same register file? That's clearly not ideal, and so we have the concept of the bypass bus: the busses on which results flow from each execution unit back to the register file can be read by each execution unit, so that the operand is immediately available without having to read it from the register file. And it's another of these facts about real code that most instructions feed their results into an immediate successor, and most results have a very short lifetime before their register is overwritten.

All this means that, in fact, when it comes to considering implementation rather than considering the programmer's viewpoint, it's closer to optimal to say "my two inputs are instruction iR and instruction iQ" than to say they are "register pM and register pN". Of course there are still long-lived inputs that come from registers, but the common case (and the desirable case, for latency and power) is to grab as many inputs as possible directly off the bypass bus.

Apple patents explicitly say that registers are read at Issue time, not early; and they strongly suggest that scheduling is based on instruction numbers (called SCH#'s); examples of the latter are https://-patents.google.com/patent/US9940262B2 or https://patents.google.com/patent/US8555040B2.

We've described the obvious instruction dependencies on previously calculated registers. But there are more subtle instruction dependencies. For example consider a load instruction. The obvious dependencies are the registers used to calculate the load address. But the next stage of execution is to load the result from the L1D cache. What if the data is not in the cache? This gives rise to a situation where you can't clear the instruction (it hasn't fully executed!) but you don't want it hanging around, blocking the execution unit until the value is available. This gives rise to a concept known as *replay*. Every CPU vendor does this differently, but the common themes are that

- you need to be able to keep the instruction in the Scheduling Queue for a few more cycles (so you can't automatically just move instructions out of a Scheduling Queue, not until their execution sets a flag saying "OK, we're done, it's all over")
- you also need to be able to mark the instruction in some way as "try again later". In the dumbest sort of Replay, you just try again a fixed time later, say every 5 cycles. More ideal is to add a new type of dependency of some sort to the instruction, so that it doesn't try again until the additional dependency is satisfied. Now that sounds good in theory – but given what I have said about dependencies being either based on physical register being marked valid, or instructions in the ROB marked as completed, how exactly will you implement these additional new types of dependency (like "cache line has been deposited in the L1D")? We will eventually see one possible answer.

Execution: There's actually not much to say about execution. Execution resources tend to be clustered into a "unit" which can perform a variety (but not the full range) of operations.
Our current best knowledge of the M1 is that it contains 14 units as described here: https://-dougallj.github.io/applecpu/firestorm.html.
In one sense execution units are the point of the CPU, and the numbers that are associated with execution units (what's the latency of operation X, how many of operation Y can I perform per cycle, what operations share what units [because if divide and multiply share a unit, I can't do both operations in the same cycle) are something that has traditionally been obsessed over by a certain type of enthusiast.
But in another sense, execution units are just no longer where the action is. Yes, you want as many execution units as possible, and you want each operation as fast as possible; if FP multiply is reduced from 5 cycles to 4, that's a good thing. But on a CPU like the M1, very little of the performance of most code can be understood by thinking about the execution units. The M1 is fast because it is extremely wide, and can run extremely out of order, not because each execution unit behaves in some way that is exceptional compared to other ARM or to x86.

One amusing way to capture this fact is to ask what is meant by the "width"of a machine.
- Amateur enthusiasts will reach for the number that looks most impressive, which is generally the Issue number, ie the number of instructions that can be fed to an execution unit. For the case of the M1, there are fourteen units, each has an independent scheduling queue, so under ideal circumstances, the M1 could fire off fourteen instructions, one to each instruction unit, in the same cycle.
- Serious enthusiasts will answer this as the maximum sustained possible throughput. In the case of the M1, this is 8, not 14.

Why this difference?
In the old days a machine that was, say, 4 wide, was built with each stage 4 wide, so fetch would deliver four instructions, decode would decode 4, there would probably be a single scheduler queue that could make a maximum of 4 scheduling decisions (even though there might be 5 or 6 execution units), and retire could likewise remove 4 instructions from the head of the ROB per cycle.
This sort of design is still based in the original microprocessor days where a single instruction moved

from one execution phase to the next; with the implicit assumption that the machine works by moving blocks of four instructions from one stage to the next per cycle. But this has become an ever less appropriate model as CPUs have ever more transistors available.
What you should think of is that a CPU consists of a number of jobs to be done, with queues connecting each job to the next. So Fetch does its job and dumps fetched instructions into the Instruction Queue. Decode/Map/Rename do their job, then dump instructions into (multi-level) Scheduling queues. After execution instructions proceed to Retirement via the ROB queue.
Under ideal circumstances, each of these queues should be extremely dynamic (they should constantly be growing very full, then shrinking to empty). The point of a queue is to buffer between stages, so that if any stage is temporarily slowed down (eg Fetch misses in the I-fetch cache) the next stage can keep going for a while just by draining its feeder queue.

With this sort of design philosophy, the width you make each stage is no longer constant because you have no vision of a single block of four instructions proceeding together through each stage of the pipeline. Rather you make each stage as wide as it can be, given power budget and timing. A wider stage will drain its feeder queue faster, ensuring that that particular stage is never the bottleneck.
This is most obvious in the case of the Execute stage. It is fairly easy to run this extremely wide (14-wide for the M1, as we see) because the Scheduling queue and execution units are essentially independent. (There are technical details, like the result buses between units, that mean we can't go absolutely wild, but we can go fairly wild).
Another stage that can easily be made wider than the sustained width of 8 is Fetch. As we'll see when we discuss Fetch, Fetch can probably pull in a maximum width of 16 instructions (one cache line) in one cycle. But in most cycles that's not possible because the basic block [distance to the next branch point] is shorter than 16, or because the basic block runs over into the next cache line, and only one I-cache line is accessed per cycle. Thus to sustain an *average* of 8 in the face of these frequent problems means you want to be able to pull in a lot more than 8 when you have the chance.
Likewise it is ideal to be able to retire wider than 8 so that once an instruction that has been blocking the head of the ROB retires, you release any resources being held by successor instructions as rapidly as possible.

Another way to look at this is that, in the older, resource-constrained days, a reasonable way to design a CPU was to target the mean properties of code, to design around the 15% branches, 10% stores, 25% loads, 50% integer operations already mentioned. But once you have the resources to do better, you should try to deal not just with the average behavior of code but also with extremes. On average, yes, 25% of the instructions are stores, but this is not the same thing as saying every fourth instruction is a store; in fact you may encounter long runs of stores (or load+stores) followed by almost no stores. So a better design target tries to hit not the averages but a metric that captures this variation. Once again this is where design as a sequence of connected queues really pays off. By providing deep queues between every conceptually different stage, Apple can rapidly shunt instructions into the appropriate queue, where they can wait to eventually execute while not blocking other instructions. The goal should not be to flow the same N instruction instructions from the beginning of the machine to the end;

it should be to have enough reserve capacity in *every* queue that all reasonable surges (a run of 50 successive FP instructions?) and dips (no instructions fetched for 8 cycles while we wait for a new cache line from L2?) can be handled without pausing.

Retire: We've pretty much covered everything related to Retire in the earlier discussion. Retire as a general concept refers to three different sort of ideas.

- Completion. This means that the sum has been added, the load has been delivered from cache, whatever it is, the result is available for someone else to use. Completion mainly means that a flag gets set in the ROB entry (saying this instruction has generated a result), but under the traditional design all the resources acquired by the instruction at Rename are still held onto.
- Retire is the point at which the instruction is checked in-order, to make sure that everything went correctly; whatever speculation occurred was correct, no exceptions or faults occurred, etc.
- Commitment refers to somehow moving the (now non-speculative and absolutely correct) state from speculative storage to non-speculative storage.

The exact order in which these tasks are done has tended to be very variable, and mostly undisclosed because most of what gets written up about CPUs tends to be to help developers optimize code, and pretty much nothing on this backend has any relevance to code optimization. It has tended to be treated as boring cleanup work that has to be done after the main event of the instruction execution. This is shortsighted, and the M1 shows why.

Note that there are distinct tasks to be done here. And whenever you have distinct tasks to be done, you can disaggregate them into distinct data structures with distinct timings. The way Apple has done this with the ROB and the structures surrounding Retire and the release of resources is, as we will see, substantially different from anyone else (though has some similarities to IBM).
- Ideally you'd like to release resources as soon as an instruction completes. But that may not be possible, especially if the resource is the speculative storage that holds the instruction result, because we can't go non-speculative until we pass through Retire. Commitment may be feasible in the same cycle as Retire, or it may be an additional cycle (usually invisible because almost nothing depends on it).
- Moving to non-speculative storage (ie Commit) for store instructions takes the form of moving instructions from the Store Queue to the L1D cache, but while (for x86) there are good reasons to do this as fast as possible after Retire, whereas for ARM (and POWER, in general for weakly ordered memory models) there are good reasons to delay this write out for some time.
- For instructions that write to registers, in principle Commit could be copying the result from the physical (speculative, out-of-order) register file to the architected (non-speculative) register file; but this model seems to have fallen out of favor, Instead now Commit means only flipping some bits associated with the physical destination register of the instruction; so that there is no single register file that represents the architected state of the machine at a given time, but you can recover this state by going through the physical register file and the logical to physical mapping indexes.

In particular, it is *possible* (if you're willing to make the effort) to
- Commit stores to L1D (thus giving early release of LSQ entries, and early informing other CPUs for multi-threaded code) as soon as a store becomes non-speculative. Even if the head of ROB is blocked by a long-executing instruction, if there are no pending branches (branches not known to be successfully predicted) between the head of ROB and the store, then there is nothing preventing the store from being committed. Apple does this with the M1.
- Early release registers. If a physical register's architected value has been overwritten, and there are no dependencies, and the register is no longer speculative (same as above, no pending branches) you could reuse the physical register. Apple does *not* do this (nor, as far as I know, does anyone else). Like every innovation, it requires some additional book-keeping, and I expect that book-keeping machinery is not present in the M1 -- but is an obvious extension to future CPUs.

So the basic flow of instructions after decode is
- [in-order] Rename (which should be considered more generally as "Resource allocation")
- [likely mixed order] Dispatch (move the instruction together with identifiers for all its resources into a scheduling queue)
- [OoO] Execute the instruction (which will involve lots of waiting around in various queues while predecessors execute)
- [in-order] Retire the instruction

Design Principles

Once you are committed to designing an OoO CPU and want to make it go faster, what are you are options? You can understand what Apple has done better if you appreciate some general design principles.

Faster

Obviously three trivial mechanisms to go faster are "run at higher frequency", the not -exactly-equivalent "generate less power per operation", and the easy "use more cores".
These are valid mechanisms, but are primarily the province of the silicon process (TSMC) and specific circuit design techniques. Apple has plenty of patents in these areas, specific ways to run some low-level design element at lower power, or at higher frequency; but these are not the level which we are going to explore.
An important qualification is that these are valid *within reason*. In particular the optimal frequency is a contentious point. Both Intel and IBM overshot the point of sensible frequency (Intel with the Pentium 4, IBM with the POWER6). It's unclear that either fully learned their less. Higher GHz is always easy to justify -- it sounds cool, it sounds heroic, it's easy to market. But it's a bad path to go down for many reasons.
- Higher GHz requires physically larger (substantially larger) transistors and standard cells. This means substantially lower (like half to a third lower) transistor density than if your design was based primarily on the lowest power transistors. And designing with only half the transistors otherwise available is a

severe handicap. Are you sure the additional frequency (say 5GHz rather than 3GHz) is worth the IPC cost?
- Of course higher GHz uses more power – a lot more power. Once again, is this a sensible tradeoff, if you could hit the same performance at lower GHz but higher IPC?
- Higher GHz requires much more human intervention in the circuit design. It's like choosing to write your code in assembly rather than in Swift. With the same sort of consequences. Working at a very low level makes you focus on very local optimizations, but it's hard to see the big picture to try for more powerful optimizations. Doing anything takes far far longer, which less help from tools. And it becomes terrifying to contemplate a total restructuring of your design.

More

Only slightly less trivial is "do more of what you're already doing". Use more cache! Provide more physical registers! Make all queues deeper! Go wider!
Even apart from the practical issues associated with this (using "more stuff" always costs more power and increase clock cycle length) it's not generally appreciated how impotent this design strategy is when pursued in isolation.
This Intel paper, (2019) https://arxiv.org/pdf/1906.08170.pdf *Branch Prediction is not a solved problem*, is concerned with a different topic, but the very first graph shows the important point. Even truly massive (32x) scaling of OoO resources delivers a substantially less than 2x performance increase.

In[ ]:=

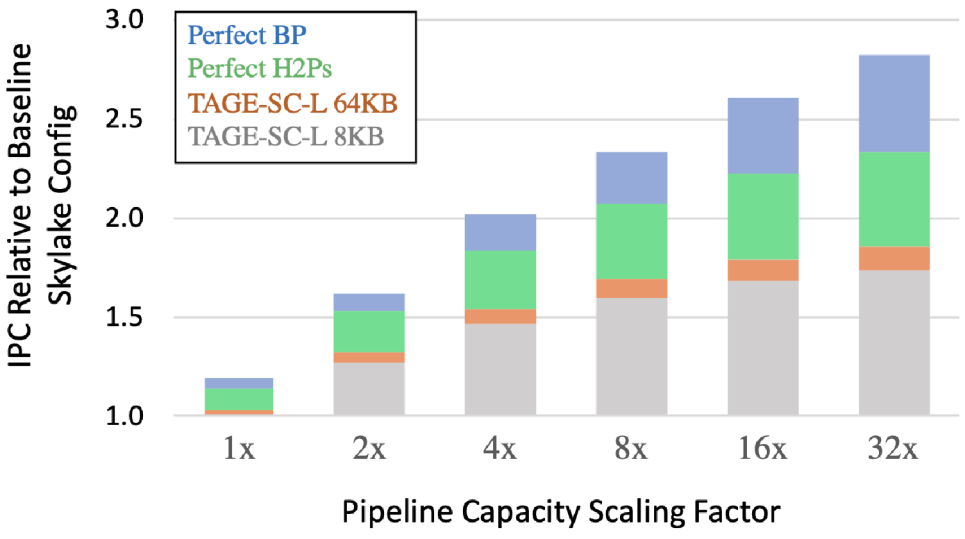


Fig. 1: Without better branch prediction, scaling the pipeline capacity of an Intel Skylake configuration will produce diminishing returns in single-threaded IPC for SPECint 2017.

The point is not that more resources are bad, it's that blindly increasing resources is not enough, not even close. You need to understand everything in your processor that is holding you back, you need to

attack *every single one* of the pain points, and you need to keep redoing it every few years as many more design options open up with more transistors.

This is, in a way, very depressing news for CPU designers. As you can imagine, designing a CPU from scratch is not easy! What you would prefer to do every year is make some tweaks, provide more resources. But not rewrite from scratch!
Samsung have given us a rare look into the sorts of annual evolutionary updates to a CPU in (2020) https://conferences.computer.org/isca/pdfs/ISCA2020-4QlDegUf3fKiwUXfV0KdCm/466100a040/466100a040.pdf *Evolution of Samsung Exynos,* but it's clear if one looks at any CPU family that this sort of unambitious evolution is the norm.
Apple had the good fortune (or good sense) to begin with a design that looks like it's from scratch as of the mid-2000s (when many of these good ideas were already known), and to design with the expectation that what future processes would give them would be many more (but not substantially faster) transistors; so their initial design incorporated a variety of flexible "communication channels" from one part of the pipeline to another, which allowed them in turn to insert various good ideas with each new design. Other machines that have not been designed with these communication channels in mind can see the value in many of Apple's ideas, but cannot easily retrofit them without substantial redesign. Or to put it differently, "more, but without new algorithms, doesn't get you much".
In other words while it is interesting to see, for every successive CPU in a family tree, how many more resources were provided of each class, you can learn this mainly because it's what's easiest to probe, not because it's what's most important. Far more important is any sort of modification in how this resource class is managed, and that most of what we will be discussing.

Guess Smarter

Much more interesting is more, and better, predictors. You should know (and should understand how it is implemented) that modern CPUs engage in aggressive branch speculation, and doing this ever better remains an active area of research. But many many other things are or can be speculated in a modern CPU:
- there is load-store address speculation (which guesses as to whether a load can be executed early, before pending stores),
- there is scheduling speculation (which guesses as to how long an operation, usually a load, will take, and schedules dependent instructions based on this guess), (2015) https://hal.inria.fr/hal-01193233/document *Cost-Effective Speculative Scheduling in High Performance Processors*.
- there are way prediction and drowsy caches (which are techniques for saving power rather than increasing performance).
- in the past Apple had predictors for whether loads might partially overlap with recent stores, or whether loads might be misaligned, though now both these issues are solved by more powerful techniques. They also have a patent (maybe still relevant) that’s essentially predicting how congested the NoC will be (and thus how long it will take for data being transferred from DRAM to reach the CPU).
In every case what you’re trying to do is discover a pattern that is common in real execution, and try to make that common pattern faster or lower power; at the expense of making uncommon patterns more

expensive.

Along with predictors one should track the *confidence* of a predictor. Suppose that recovering from a mispredict is cheap; in that case go ahead with the prediction even if it is low confidence. But if recovery is expensive, you may want to delay an operation until it is no longer being speculated, until you know for certain.

A new concept (which appears occasionally in the literature of the past fifteen years, but which does not yet seem to have been productized, including by Apple, is criticality, or the similar idea of urgency: (2009) https://www.cs.cornell.edu/~bracy/resources/pubs/hpca2009-lcp.pdf *Criticality-Based Optimizations for Efficient Load Processing.*
(Apple does use urgency in some recent NoC patents, so who knows, maybe we'll see it in the CPU soon?)

Of course closely related to predictors are the concepts of *prefetching* and *cache management*, both huge subjects.

Work Smarter

Most interesting of all, and very important is two, somewhat related design principles, which I will call *resource amplification* and *task disaggregation*. (It's interesting to note, as an aside, that while resource amplification is the common design pattern in almost every decent micro-architecture design paper, task disaggregation is far far more rare. It's something one sees all over the place in the Apple design – and almost nowhere else... Not in the literature, not in other designs.)

Resource amplification is about making existing resources (which are always in short supply!) go further.
For example you have access to a fixed size L1D cache. You may think that's the end of the story, but not even close! For example -- what algorithm are you using to replace lines in your cache? A better algorithm (which holds onto useful lines longer) will make your cache effectively larger. This is amplification – using better algorithms to make a small resource provide the value of a large resource (or, in a slight twist, providing a large resource that mostly costs the power of a small resource).
We will an astonishing level of this in Apple's load/store/TLB/cache pipeline, where Apple gets most of the performance of a 4-wide pipeline while paying the costs of a 1..2-wide pipeline.

Resource Lifetimes

It has been traditional in CPUs to Allocate resources in one place (in a pipeline stage called Rename) and to Deallocate them in one place (in a pipeline stage called Retire or Commit or something similar). Like many things, this was an obvious solution, and probably optimal at the time it was invented, but far from optimal today.
This pattern means that any resources that is allocated (think for example of a destination register) is reserved, and locked up unavailable for use, from the Rename stage forward. This is so even if the

register is the target of a very long latency operation (maybe a load that missed to DRAM), and even though the register is only required at the very end of this operation, at the point where the result needs to be saved in a register.
A similar situation holds, for example, with slots in the load and store queues.

Why was it done this way? The reason is ordering/dependency requirements. The pattern of register allocation and renaming establishes the true data dependencies that allow an OoO machine to reorder operations for maximum speed while still being correct; the ordering of load/store slots allows loads and stores to operate out of order while still ensuring that if a load tries to read data from a store address that is pending, but not yet executed, the load will delay until the store provides the data.

These seem like non-negotiable requirements, but not so! The trick is to realize that the allocated resource is performing two tasks.
One of these is expensive (storage, of an execution result, or of a load/store address), the other task is cheap (providing an ID that's ordered relative to other, equally cheap, IDs).

Turning this rough insight into an implementation is done via Virtual Registers, first explained here: (1999) https://upcommons.upc.edu/bitstream/handle/2117/101362/00809456.pdf *Delaying Physical Register Allocation Through Virtual-Physical Registers*
or Virtual Load Store Tags, described here (2007) http://citeseerx.ist.psu.edu/viewdoc/download?-doi=10.1.1.140.3533&rep=rep1&type=pdf *Late-Binding: Enabling Unordered Load-Store Queues.*
Apple and IBM both use Virtual Load Store Tags. No-one (not yet, as far as I can tell) uses the full virtual register idea.

So one can delay the allocation of resources, thus having them reserved for a shorter time.
An alternative is to tackle the other side, to try to free resources sooner. Once again, it has been traditional to deallocate at Retire because that's the point at which everything related to the instruction is over, but it is frequently possible to retire a physical register much earlier. (2004) https://pages.cs.wisc.edu/~rajwar/papers/taco04.pdf *An Analysis of a Resource Efficient Checkpoint Architecture* talks about many interesting things (including why one wants to keep increasing resources, especially via amplification mechanisms), but among other things, in section 4.3 it discusses ways to more rapidly reclaim physical registers.
Apple are using one (limited) form of early register reclamation: (2017) https://patents.google.com/patent/US10691457B1 *Register allocation using physical register file bypass.*

One slight variant of this idea (less resource amplification, more energy minimization) is how Apple manage both their registers files and their L2. Both of these are nominally large structures that could take lots of power and be slow. But in both cases they are in fact split into multiple independently managed smaller pieces. So, as much as possible, the machine runs using eg one quarter of the physical register file, or one third of the L2, with the rest powered down and taking no energy; the extra resources are only powered up and used once certain metrics indicate that they would provide real

Fusion

The most obvious form of resource aggregation is instructions that do more than one thing. So you pay

the cost of a single instruction, at least for some stages of the pipeline, while achieving more than one result.

Much of the ARM64 instruction set provides *pre-fused* instructions that do this – a single instruction that performs one task (add, select, move, ...) with a small additional tweak (shift an input by a few bits, increment or negate the output, …). The trick, of course, is to ensure that the tweak is indeed so lightweight that it can be performed as part of the single instruction without compromising the design of the entire CPU.

Alternatively, the decode stage of the CPU can fuse together common lightweight instruction pairs. x86 does this for compare+branch instruction pairs, and so does every high end ARM CPU, but many additional instruction pairs can be fused when it makes sense.
The idea of fusing instructions has been around in various forms for years: (1996) ftp://ftp.cs.wisc.edu/-sohi/papers/1996/micro.collapse.pdf *The Performance Potential of Data Dependence Speculation & Collapsing* is worthless for its performance projections, but is interesting in its categorization of the most common feasible fusion sequences. You'll see that the most common patterns (and the ones that have been implemented by everyone) fuse with a branch, and so only require a standard ALU; most of the patterns require a 3-input ALU, which we know how to design and which is known to be feasible in current cycle times, but which has (as far as I know) still not yet been implemented in generic form.
Right now what Apple does, though more aggressive than any other vendor, is still limited to the obvious pairs, as listed here: https://dougallj.github.io/applecpu/firestorm.html.
(For people who are curious, the reason some non-fused pairs are given is that they are, in fact, obvious fusion candidates at some later point:
`ADRP+ADD` is used to construct large offsets for loading global data, strings, floating point constants, and suchlike
`MOV+MOVK` is used to construct large immediate values (ie large integer constants)
`MUL+UMULH` (or `SMULH`) is used to multiply two 64-bit values to create a 128-bit value
`UDIV+MSUB` is used to calculate A mod B)

Industrially, instruction fusion is still somewhat unexplored territory.
On the public side, we mostly only know the fusion patterns that have been published by vendors (eg in ARM or x86 per-CPU optimization guides), or that have been guessed at and tested. It's possible there are more fusion pairs on the M1 not yet discovered.
Cases that are still open for exploitation (even by Apple) include
- A fused instruction pair may allow the second instruction to reuse some work that was done by the first instruction. There are a few ARMv8 pairs of crypto instructions that do take advantage of this, but it would be possible in theory for 128-bit multiply (`UMULH` giving the upper 64 bits, along with `MUL` giving the lower 64), or for the pair (`DIV+MSUB`) giving the quotient and remainder of an integer division, to likewise share work across the two instructions, which is why they are obvious test candidates.
However it's not always easy! There is a tricky element to the latter two cases, namely that they return two results. So you're kinda asking for a new type of instruction that overwrites not one but two regis-

ters, and so requires the allocation of two physical registers.
When we get to register allocation, we will see why this is a non-trivial issue, but Apple is in a position to deal with it (they already have to allocate a register pair for Load Pair instructions). But if you look at the various cases that have been fused so far, they either involve branches (no second register) or involve overwriting an intermediate register, so avoid this issue.

- A different type of case is if the next instruction immediately overwrites a register, only a single register allocation may need to be performed. Consider for example
`add rD, rA, rB`
`add rD, rD, rC`; which could in principle fuse to
`add3 rD, rA, rB, rC`
This would require only one destination register allocation, and, for ARMv8, which already has integer instructions that take three source registers, would not require a substantial redesign of the internal data flows. Would it be worthwhile? The pattern is probably common, and if the two additions could be performed in a single cycle...? Of course multiply-add is already a pre-fused version of the same *sort* of pattern, where we remove the use of an intermediate register.

A particularly cute form of fusion is performed by ARM Ltd (Apple appears not to do this, at least not as of M1).
We've described why, because of linker realities, even real world code has the occasional NOP scattered through it. What should the Decoder do when it encounters a NOP? You'd think just dropping it would be fine, but apparently the instruction tracking machinery seems to really want to track every instruction, more or less, so Apple don't do that, they apparently allocate a ROB slot to each NOP, which seems a waste, albeit not a terrible one.
What ARM Ltd do in their newest chips is fuse the NOP with whatever the next instruction is, so the NOP is still accounted for in the same way that all fusions are accounted for, with the fused instruction+NOP occupying a single ROB slot. Neat!
https://chipsandcheese.com/2022/05/29/graviton-3-first-impressions/

Instruction fusion is not a panacea; it only amplifies resources if there is in fact some sort of common resource whose utilization can be removed by the fusion. But there are other ways it can be useful.
Fusion can also be considered a way to remove a cycle of latency.
In modern CPUs, especially a brainiac CPU like Apple's CPUs, cycle time is defined by the book-keeping stages of the pipeline like Rename or Schedule, not by the time it takes the ALU to perform a simple instruction. Which means that if you fuse together two simple instructions, eg rD=rA+(rB&rC), and design the ALU appropriately, you can execute both operations in a single cycle.

In a way this is a return to (and overall more efficient way to perform) what Intel called the Rapid Execution or Double-Pumped ALU in the Pentium 4,
https://www.anandtech.com/show/604/4 .
A variant of this idea (using timing slack in earlier stages) is claimed to be performed by Intel in Golden Cove,

which executes some immediate additions in Rename. In a way this is the logical extension of the progression Zero Cycle Moves, to Zero Cycle Immediates, to Zero Cycle “trivial ALU operations”, and one suspects Apple will at some point do the same thing with similar “easy” immediate ALU ops.
https://www.anandtech.com/show/17047/the-intel-12th-gen-core-i912900k-review-hybrid-performance-brings-hybrid-complexity/5
I’m extremely curious as to exactly how this is done. If you think about it, Register Rename is about
- lookup logical registers in a logical→physical map for source operands
- allocate a new physical register for the destination result
- add an entry in the logical→physical map for the destination entry.
You can easily fit move elimination into this by fiddling an entry in the logical→physical map.
You can easily fit zero’ing into this by having a permanent zero physical register, and fiddling an entry in the logical→physical map to point to that.
You can (with a little more difficulty) fit immediates into this by having a path that can move the immediate into the physical register fast enough within the Rename cycle.
But the Golden Cove claim is that we can do all the standard work above AND read the value of a source operand AND calculate a sum AND store the result all within the Rename cycle. Possibly(?) Rename has been extended to cover two cycles (the work they wanted to do with a wider core, and the higher frequencies, meant perhaps it was going to take say 1.25 cycles, so they extended it to two cycles and now have time to do some minor arithmetic?
Some details of this are discussed in https://www.computerenhance.com/p/the-case-of-the-missing-increment .

Similarly, if one wants to be even more ambitious, there’s probably timing slack to allow at least some cases of op+Store and Load+op to be fused and executed in the LSU (using a dedicated ALU), again shaving off a cycle here and there.

There are a number of potential fusions that Apple does not appear to have implemented in the M1, but the following LLVM checkin is very interesting.
https://github.com/llvm/llvm-project/blob/main/llvm/lib/Target/AArch64/AArch64.td
gives Apple’s official claim for LLVM as to supported instructions and performance tweaks (most important fusions)
FeatureFuseAddress
FeatureFuseArithmeticLogic
FeatureFuseCCSelect
FeatureFuseLiterals

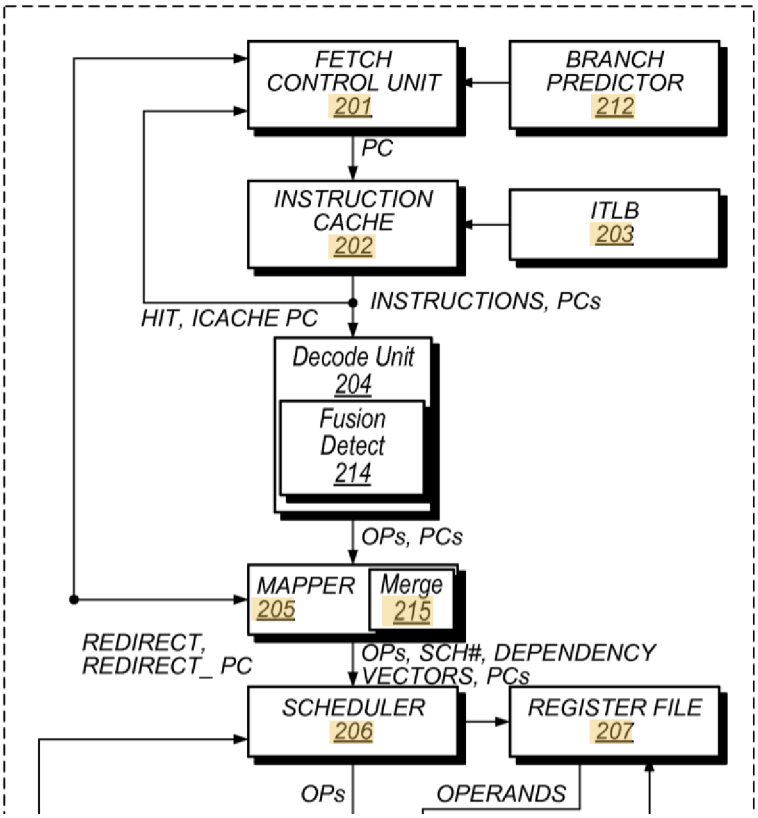
The patterns implemented by these fusions are described in https://github.com/llvm/llvm-project/blob/main/llvm/lib/Target/AArch64/AArch64MacroFusion.cpp
and https://github.com/llvm/llvm-project/blob/main/llvm/lib/Target/AArch64/AArch64.td states that the A14 has them.
(Look for isArithmeticLogicPair in AArch64MacroFusion.cpp. The compiler also makes no attempt to force register reuse for the fused pairs, but any reasonable implementation will probably require register reused! So

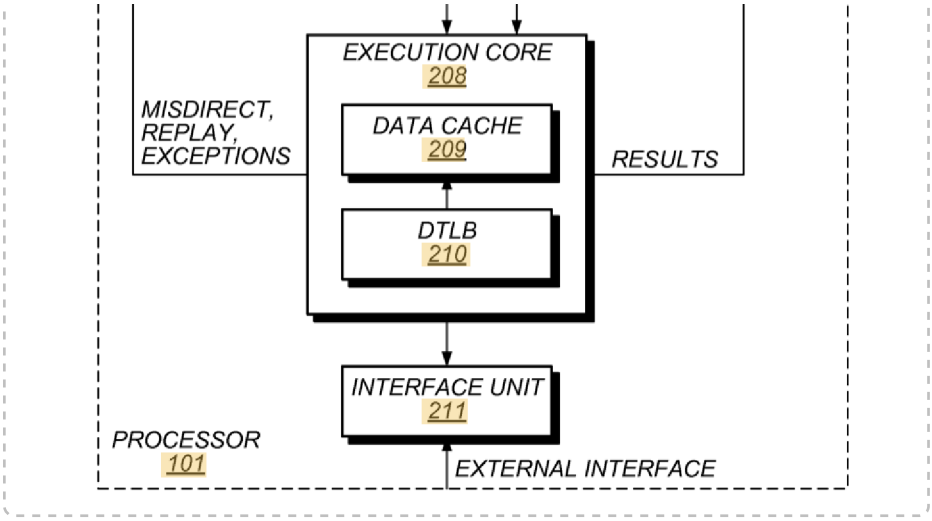
it looks like a first pass at compiler modifications for the future was made (perhaps experimentally, to see how often this case triggers?) but no serious productization has yet been done.)
It's unclear whether these fusions are present in A14/M1 (or in later chips). People investigating latency or throughput report no obvious improvement from these fusions but the implementation, at least for now, could be purely about resource amplification. (That is, as we will see, perhaps the most common constraint as to how aggressively the M1 can reorder instructions is the size of the History File, which records the allocation of destination physical registers. If these fusions allow the "reuse" of a register, ie one final destination register is allocated for the fused pair of instructions – think of eg, rD = rA op1 rB; rD = rD op2 rC → rD = op1op2(rA, rB, rC) – then they would amplify the effective size of the History File; but this would only be visible if you actually tested for this effect, and as far as I know no-one has yet done that. Similarly, though it's less pressing, such fused pairs could fit two operations into a single ROB slot, or into a single Scheduler Queue entry, or extract better value from various other resources.)

My guess is that Apple is preparing the way for a future design. There's really no downside to ensuring that the relevant pairs of instructions are placed adjacent to each other by the compiler, and doing so ensures that a future CPU that does implement the fusions will immediately take advantage of them.

We have one Apple patent that covers fusion, namely (2013) https://patents.google.com/patent/US9672037B2 *Arithmetic branch fusion*, which discusses the pair arithmetic op+compare result. (The obvious cases of interest are `SUB+CBZ` and something like `OR+TBZ`). The patent includes these diagrams:

In[ ]:=





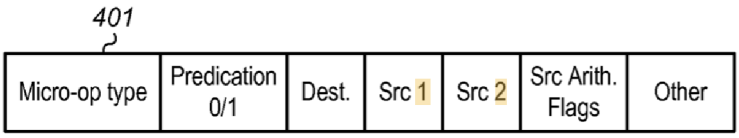
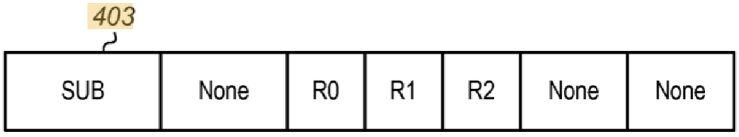


FIG. 4A

FIG. 4B



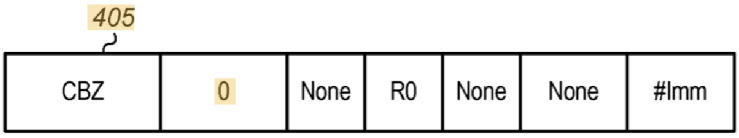
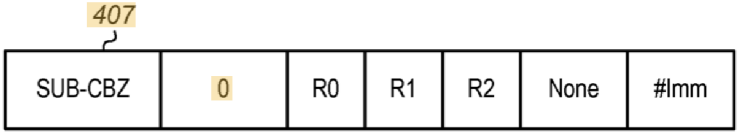


FIG. 4C

FIG. 4D



Of particular interest is that

- fusion detection occurs at Decode and is implemented at Map (essentially the Rename/Resource

Allocation stage)
- it is implemented by changing one of the two instructions in the instruction stream as seen in the second diagram (from an ARM CBZ to an Apple-invented SUB–CBZ instruction) while the other instruction is discarded.
Consequence is that some resources (in particular a ROB slot) are already allocated; we see something similar in the way that NOP and NOP-equivalents also take up ROB slots though they execute at full 8/per cycle, presumably limited by Decode. However only one Scheduling slot is required, not two, and the result is available in once cycle, not two.

This implementation is not ideal, but still seems to be the implementation structure today. More ideal would be earlier fusion, something like
- mark instructions of possible fusion classes in pre-decode (ie when the instruction is pulled into the I1-cache from the L2)
- perform the actual fusion (ie tie the two instructions together, and delete NOPs, and other cleanup) before Decode, while the instructions are in the Instruction Queue.
Among other things, this would allow higher Decode throughput (a SUB–CBZ would count as a single Decode instruction, so in many cases Decode would be performing the work of nine or ten "current-style" Decodes in a cycle); and we'd save a few ROB slots.

Note also the power implications: that is, the fusion is repeated every time it re-Decodes and re-Maps. Power can be saved slightly with a loop buffer, and one could imagine a CPU that, at the point where it decides to utilize the loop buffer, makes a second pass over the instruction stream looking for a wider collection of possible fusion pairs (or the similar sort of exercise for CPUs that utilize a micro-op cache). However, in addition to energy savings, what I suggested above about doing this work as early as possible in pre-Decode and in the Instruction Queue also allows the fusion to be made more complex (spend more time looking for candidate pairs) while it benefits both straight line code and can amortized over a wider range of loops.

We do know that Apple performs pre-decode. Examples include:
- an (interesting, but obsolete) case here, related to 32-bit ARM THUMB processing: (2013) https://-patents.google.com/patent/US9626185B2 *IT instruction pre-decode*
- (2014) https://patents.google.com/patent/US20160011875A1 *Undefined instruction recoding* detects all the possible undefined instruction encodings in an instruction, and replaces them with a single "undefined instruction" value
- (2014) https://patents.google.com/patent/US20160085550A1 *Immediate branch recode that handles aliasing* precalculates a portion of the target address of a branch (adds the branch offset to the lowest 14 bits of the PC)
- (2016) https://patents.google.com/patent/US10747539B1 *Scan-on-fill next fetch target prediction,* and a few other branch/fetch patents mention how cache lines already have the branch instructions in a cache line marked by branch type

One final interesting thing about fusion. Through all generations so far, Apple's cores have followed the traditional pattern of having Decode scaled to the same size as Rename, so that insofar as a core is

described as 6- or 8-wide, this refers to having a maximum Decode or Rename width of 6 or 8. The advantage of these being the same size is that you can transfer each instruction “directly” from Decode to Rename without any intermediate routing. But as fusion becomes ever more important, this also becomes more of a constrain, and you find that more and more cycles you are decoding say 8 instructions, but these collapse down to maybe only five operations that pass through Renaming. (Maybe one is a NOP, and two pairs [four instructions] fuse down to two operations.)

In parallel with this, Rename has to handle the worst possible case of every set of instruction patterns, whether it’s 8 integer instructions, 8 stores, or 8 FP instructions, even though these all require very different items to be allocated by Rename.

One can solve both these problems by inserting a queue between Decode and Rename giving flexibility to both ends. Now Decode (easy) can scale up to 10 (or even wider if that makes sense), packing the excess operations (after dropping NOPs, and fusion) in the queue. Meanwhile Rename can now run, far as is practical, all the different allocators in parallel so that, for ideal instruction patterns maybe 12 or more instructions can pass through Rename in a single cycle (some branches, some integer, some stores, some FP, ...)

Such a design is a substantial change from what we have today and, like the introduction of any queue between two stages, also comes with some cost, a small amount of power and area, and an extra cycle for those (hopefully rare) cases when one has to recover from branch mis-prediction. But it also comes with the huge win that you can keep scaling the machine wider up to perhaps (nominally...) even 10 or 12 wide because you don’t actually have to scale up the hardest parts of scaling wider, namely worst-case Rename; all you have to do is allow for a flexible Rename that does multiple “types” of Rename in parallel and reap the benefit for the case of most code which mixes a variety of different operation types in every run of 10 or 12 instructions. The win perhaps wasn’t worth the work until aggressive fusion became feasible, the point we’re at now, at which point, unless you decouple Rename from Decode, you’re only reaping half the benefit of your fusing!

Task Disaggregation

This refers to splitting a task that' s traditionally considered unitary into multiple pieces.

For example suppose you have a load in the load store queue that is behind a memory synchronization of some sort. In other words, the program semantics are that no memory operations are allowed to happen until the memory barrier is complete. That sounds like it is game over, no optimization possible.

But a load consists of multiple pieces! One piece is the actual "load value into register" part; but another piece is getting the actual value into the cache. What' s to stop you generating a prefetch that optimistically starts that loading into the cache now, maybe even before the synchronization begins to execute? (Of course you have to be careful about the precise semantics of each synchronization primitive, but the idea is possible.)

You can do similar things with stores. Again, of course, you cannot write a store to the cache too early. But you can preload the target line of the store into the cache, and simultaneously inform other CPUs of your Exclusive capture of the line, even before the store instruction becomes non-speculative.

Once you appreciate this point, you can do this kind of thing in multiple places. For example Apple disaggregates Instruction Retire into one part that handles freeing up the ROB (and can be extremely aggressive, freeing up to 56 ROB slots in a single cycle), and a second part that frees registers, a more difficult task that can only run at 16 registers per cycle.

Another place is: consider that instructions are sometimes split into two for the purposes of resource allocation (for example the implementation of an instruction may require allocating an implicit second destination register along with the obvious destination register). This sort of splitting an instruction into two, called cracking, is common; everyone does it.
What' s not common is that Apple then sometimes joins the two pieces back together. The resource allocation task was one step – best handled by two instructions – but Scheduling and Execution are different steps, best handed by one instruction! Again, disaggregation of the different parts of "performing the instruction".

Theory of the experiments

Now that we have an idea of some of the complexity in a modern CPU, how do we go about investigating what's there?
It's an imperfect science! One has to maintain in one's head a number of simultaneous possibilities, till the correct option becomes clear (and sometimes it never does). It requires patience and creativity!

One direction of attack is obvious.
You can learn the latency (how many cycles before you can use the result of an instruction) of an instruction by creating a chain of instructions that each feed their result to the next element in the chain, something like

```
op r2, r2, r0
op r2, r2, r0
op r2, r2, r0
op r2, r2, r0
```

...
Create 1000 successive instances of this assembly, loop it 10 times, use the CPU's cycle counter to tell you how long it took. and you have your result.

Alternatively you can learn the throughput (how much independent instructions of this type you can execute per cycle) via something like

```
op r3, r2, r0
op r4, r2, r0
op r5, r2, r0
op r6, r2, r0
```

...
Create 1000 successive instances of this assembly, loop it 10 times, use the CPU's cycle counter to tell you how long it took. and again you have your result.
These are, obviously, the essence of what Dougall used to create his latency/throughput website.

This is fine as it goes; important first step information. But note what it doesn't tell you (even in the big picture). It won't tell you about instructions that fuse together. Or clever tricks that may allow some instructions to exit the pipeline without requiring a step in each stage. Or how functionality is grouped together into different units. Or resource limits like the size of the ROB of scheduler queues. or other aspects of the OoO design.

The basic pattern of an OoO machine is that
- at the front of the machine, IN-ORDER, instructions are Decoded, Mapped (establish dependencies, establish the remapped registers) and Renamed (allocate additional resources, like a destination register).
- One of these stages (I assume Decode) also allocates an In-Order ROB slot.

- Between Rename and Retire the OoO machinery takes over
- At Retire, and after all speculation has been tested and found correct, resources are deallocated as each instruction retires.

If a resource cannot be allocated, then the machine will stall (ie no instructions will move past the in-order stage that cannot allocate). The OoO part of the machine will continue to execute, and eventually the back end will make enough progress to retire some instructions and free some resources, at which point the in order front end will resume execution.

The consequences of this pattern are that if you create a delay block (so that the head of ROB cannot clear until the last instruction of the delay block execute), then you can prevent the deallocation of resources until that head of ROB clears. Which means that you can stall the machine at the front-end in-order stage until the head of ROB clears. It takes a few cycles for instructions to then proceed from the stall down the pipeline to execution; and we hope to be able to see this glitch, this transition from one performance regime to another.

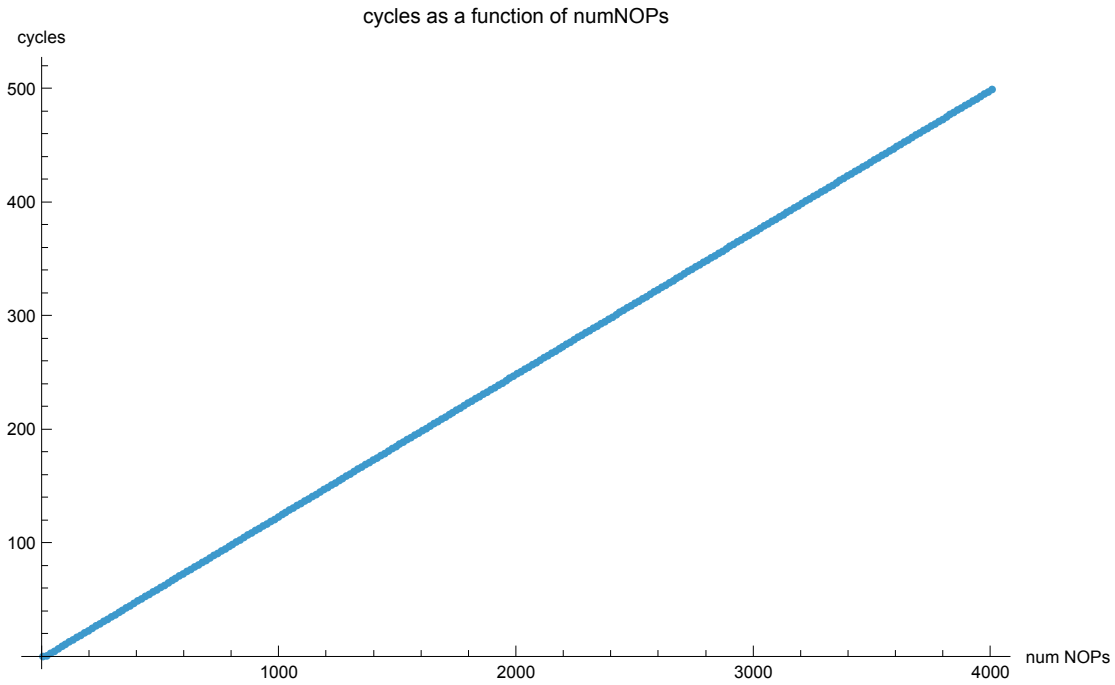
Life is complicated (as we will see going forward!) by the fact that, to make a machine more performant, you can, if not break the above rules, at least substantially bend them. So we will find that some resources are in fact allocated beyond the in-order front end, or that some resources that we imagine as unitary are in fact composed of multiple facets.

Let’s see a simple example of the above idea:

ROB size (NOPs)

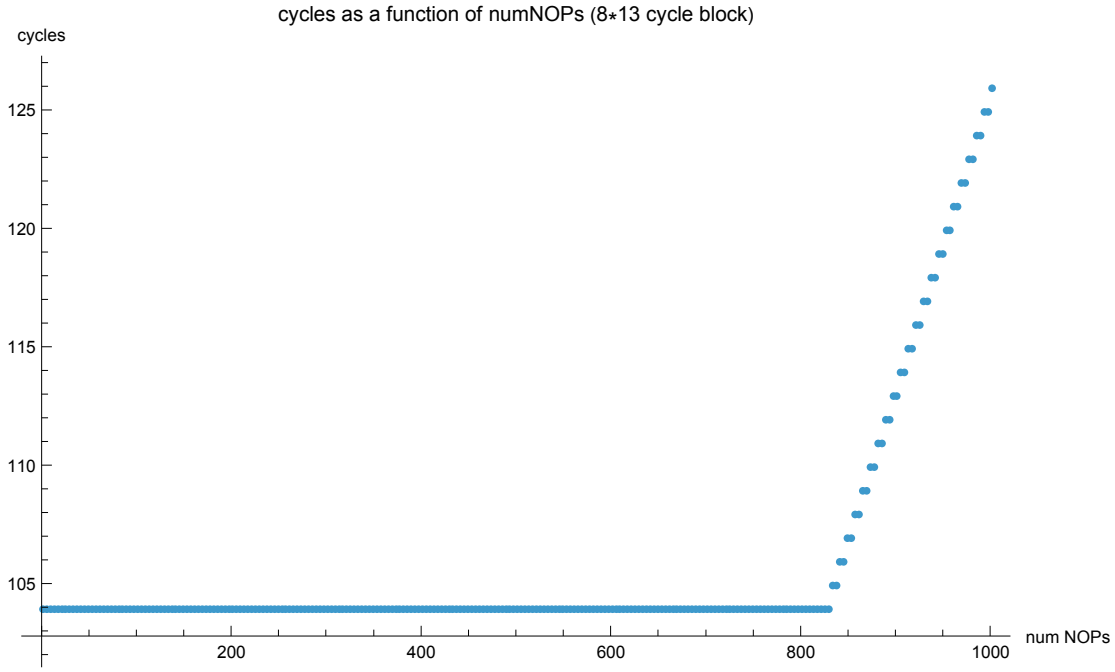
To get calibrated, our first code is nothing but a series of NOPs. We would expect that this should run at 8 NOPs/cycle, and that’s exactly what we see.

Out[450]=



Now add a delay block of 8 chained FSQRT.D, each 13 cycles long.

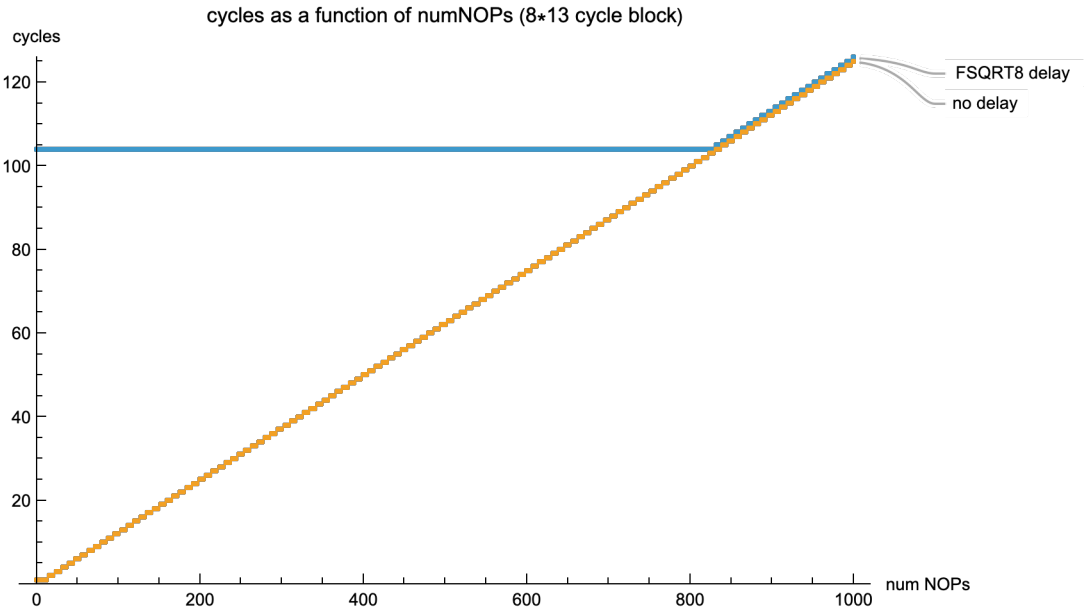
Out[453]=



Nothing too difficult to understand here; for fewer than (8*13*8=832 NOPs) the loop time is defined by the FSQRTs, for more than 832 NOPs the loop time is defined by the NOPs.

1. Delay block timing (blue) vs non-delayed timing (gold)

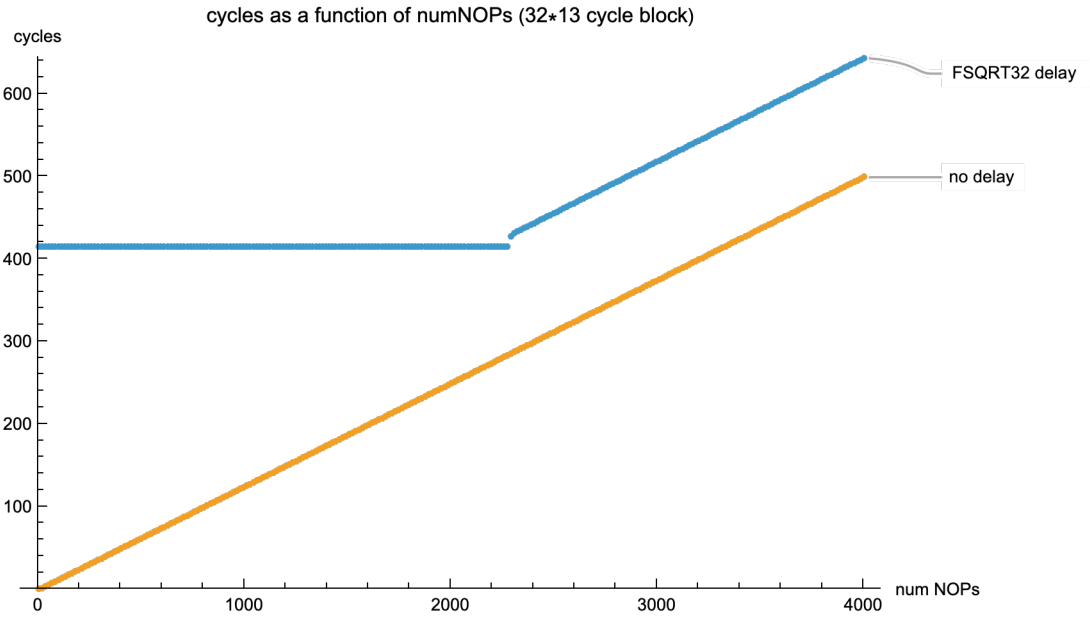
Out[454]=



We can force the FSQRTs to delay for a lot longer by chaining 32 of them together. Now we see something interesting, a glitch in the graph!

2. Delay block timing (blue) vs non-delayed timing (gold), with delay block and number of operations large enough to force a glitch.

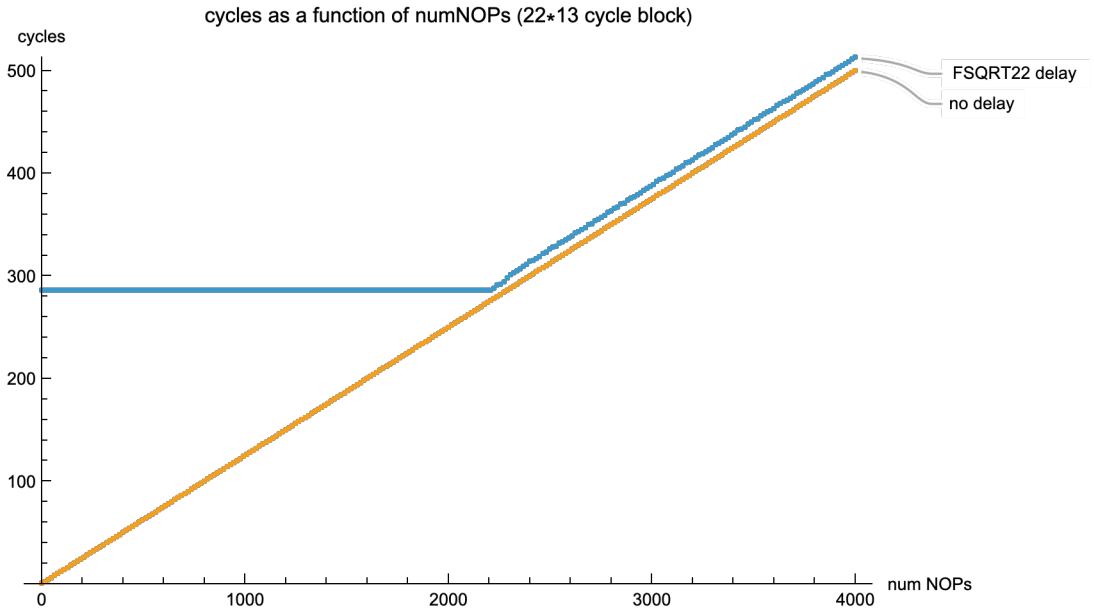
Out[456]=



What happens if we try to shrink the delay to the bare minimum?

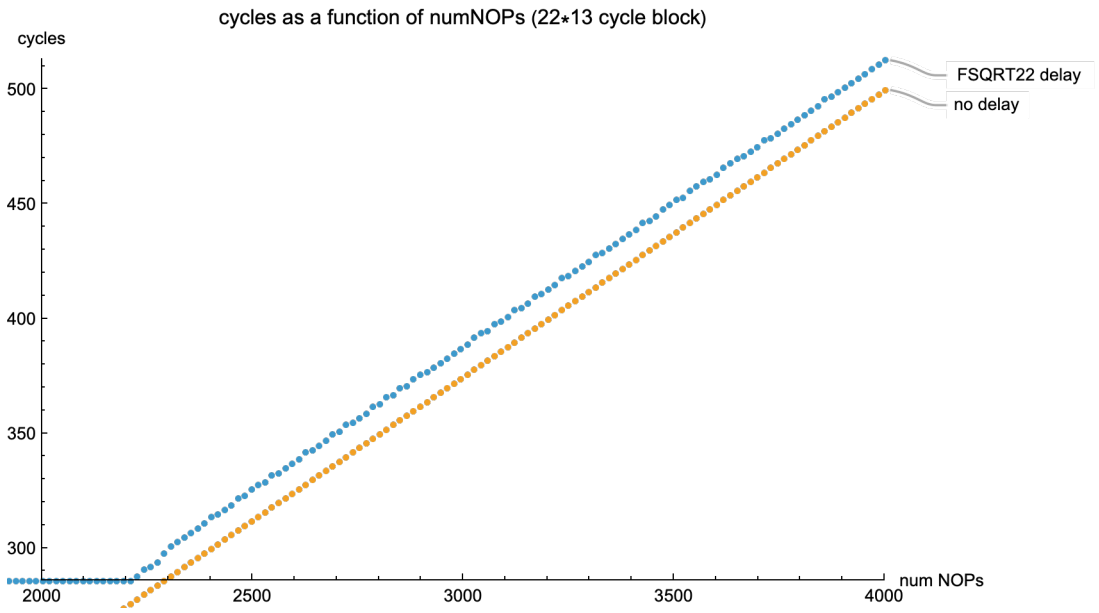
3. Delay block timing (blue) vs non-delayed timing (gold), with delay block and number of operations optimized for visible (but minimal) glitch.

Out[458]=

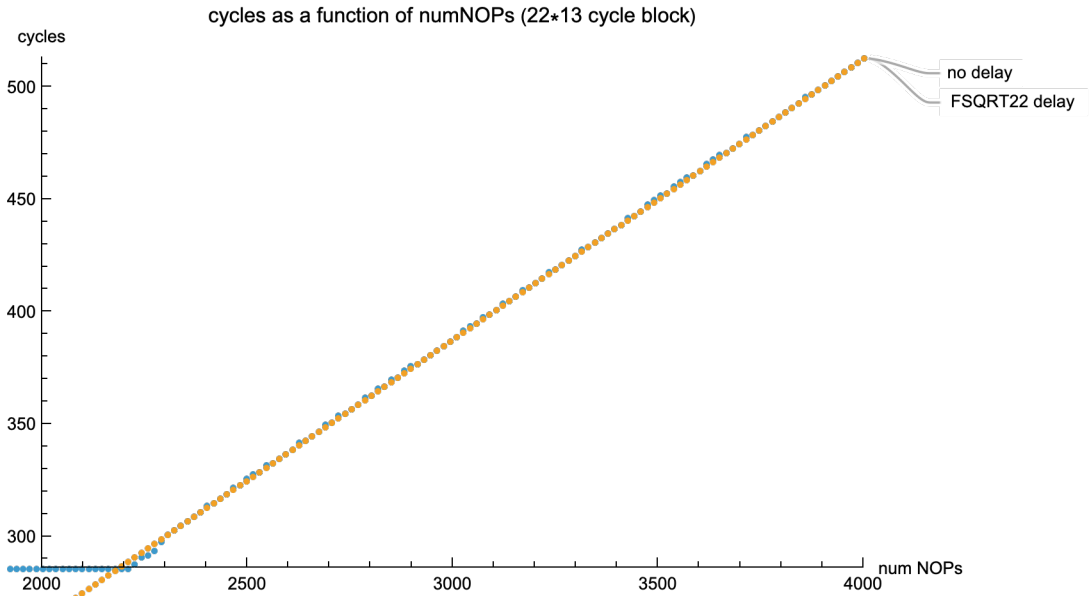


And zooming in

Out[459]=



Out[460]=



There are clearly two interesting numbers.
- At 4000 NOPs, the time taken is 500+13. ie one fsqrt was delayed each iteration.
- The delays starts at 2228 cycles.
How do we explain these facts?

In the case of NOP, the only resource that can possibly be relevant is the ROB. In other words, as a rough approximation, what's happening is:
- the fsqrt's block the ROB for long enough that around ~2200 NOPs can pile up in the ROB, unable to retire because the fsqrts are still going.

- at that point the next fsqrt (placed by looping around back to the start of the loop, is not able to even enter the machine because one of the requirements to pass Decode to the next stages is a ROB slot.
- so that fsqrt is delayed, and our timing switches from MIN(22*13, N/8) to something like MIN(21*13, N/8+13).

That gives the intuition (and approximates the numbers) but is off by a few cycles because it does not include the overhead of the fsqrts themselves. Can we do a little better?
We will assume that any type of operation can be decoded at 8/cycle and placed into the ROB at 8/cycle.
This assumption may need to be revised, but let's assume that for now.
At t=-1 the ROB holds nothing.
At t=0 the ROB holds 8 successive fsqrts.
At t=2 the ROB hold 22 successive fsqrts and 2 NOPs.
At t=13 the ROB holds 21 successive fsqrts and 2+ (13-2)*8 NOPs
until we loop to the second iteration:
At t=T the ROB holds (22-T/13) fsqrts and 2+(T-2)*8 NOPs

For T=21*13=274, the number of enqueued NOPs will be 2178
For T=22*13=286, the number of enqueued NOPs will be 2274
We see trouble at 2228 between these two, so our ROB size is somewhere between these two.
Essentially at T=~280, we reach a situation where
- 21 fsqrts have been processed
- the last fsqrt is still executing, with a few cycles to go (22*13-280=6)
- The ROB holds 1 fsqrt, ~2270 NOPs, and perhaps two instructions from the branch test+loopback
BUT it can no longer accept the first fsqrt from the next loop iteration. So we get a stutter at this point, a delay in the smooth flow by 6 cycles before that fsqrt exits head of the ROB, and Decode is no longer stalled.

Once one has a more accurate idea of the machine, one can attempt to perform precise measurements. But this entire document will be about trying to understand the overall machine, thus we'll be satisfied with approximate numbers, leaving precision for later workers.

So we've established a few initial facts here:
- The ROB is ~2274 instructions in size. Over time we'll see that I believe it's best treated as ~324 rows, each 7 slots in size, giving a total of 324*7=2268 entries.
These "rows" are the units of Retire.

Retire (and the ROB) have to handle the following tasks
- deallocate resources (physical registers, load/store slots, ...) as instructions are completed BUT
- maintain machine integrity in the face of misspeculation (eg branch prediction).
This means that temporary state (in physical registers) can only be deallocated once it is certain there's

no possibility that unwinding speculation may require the use of that state.

This leads to a delicate balancing act where one has to hold onto resources as long as necessary (for correctness) but wants to relinquish them (for reuse) at the absolute earliest moment that that is possible. Much of Apple's somewhat unusual ROB machinery is an attempt to be more aggressive in this reuse goal.

More Experimentation Theory

The ROB+NOP analysis gives some of the flavor of the enterprise going forward. The usual pattern is
- a delay block
- running in parallel with other operations
To void confusion, let's consider the delay block (which can be of variable length) to take duration D, and the Ops block to be comprised of N Ops that would be expected, under ideal circumstances, to take duration O.

(a) Once the "other operations" take longer than the delay block we get a diagram like diagram 1 above. The diagram to keep in mind is

```
| Delay<---------D---------->|| Delay<---------------------------------->|
|Ops<------------O--->|                    |Ops<---------------->|                    |
```

transitioning to

```
| Delay<---------D---------->|        | Delay<---------------------------------->|
|Ops<-----------O------------------------------>||Ops<------------------------------------------>|
```

(b) But once we make the ops block long enough, lack of some resource will stall the earlier stages of the machine (Decode, Map, Rename)and prevents all the operations from executing right away because all resources are locked up until the head of the ROB (the Delay block, retires .and so the Ops block encounters an additional delay, a delay caused by its inability to fully execute). This gives us a diagram like 2. with a notable glitch.

```
| Delay<---------------------------------->|                  | Delay<---------------------------------->|
|Ops<--------------------------xxxxxxxx---------->||Ops<--------------------------xxxxxxxx---------->|
```

In other words once the resource count used by Ops is exceeded, then the delay O_measured(N ops) is larger than O_expected(N ops) by the delay xxxxxxxx that occurred while the Ops were sitting idle, unable to proceed through the machine until the head of the ROB retired (at the end of the delay block) and resources were freed.

Note that for this condition to occur (presence of the xxxx) we require
- D must be large enough to block N ops partway through their execution
- N must be large enough that all resources are fully consumed.
However once we see a glitch, it's sometimes desirable to try to shrink D and N down to about the minimum values that will reproduce the phenomenon, the glitch in timing, as in diagram 3.

Other times it makes sense to push D rather larger than the minimum because you'll then catch an additional unexpected event!

In this case the number of interest was the minimal number of ops that had to be enqueued to generate a glitch. As our analyses become more sophisticated, we will be considering other features of these sort of graphs. For example the slope of the NOPs line tells us how many NOPs/cycle can be processed by the machine. We may structure things so that, after a glitch, the slope of the post-glitch line tells us something about how rapidly resources are freed (as opposed to what we have learned so far, namely how many resources can be used up).

Register Files

More OoO Theory

Remember why we distinguish between logical and physical registers: https://en.wikipedia.org/wiki/Register_renaming.
We use physical registers (the logical physical register perhaps mapped to different physical registers as execution proceeds) for two reasons
- to avoid having to pause execution because of false dependencies
- to maintain state in a temporary form for speculative execution. If we want to be able to execute further down a speculative path, we need to be able to hold more temporary state, ie more physical registers

Most machines distinguish three classes of registers: int, FP/SIMD, and flags; and all three are renamed and live in separate physical register files.

So consider a stream of instruction that each generate an integer result. Every such result requires a destination register which will be a newly allocated physical register. If the pool of such physical registers runs out, integer instructions cannot proceed until more physical registers become available. In fact *no* instructions can proceed, because register allocation occurs in the front end of the machine, where instruction flow is still in-order.

The flow of instructions is essentially
Fetch
Decode,
Map (mainly figure out the dependencies between the current group of eight newly decoded instructions)
Rename (allocate resources required by each instruction, eg a destination register
Dispatch

Schedule
Execute

Retire
Complete
Writeback

The initial set of stages all occur in-order – meaning that any block in one of those stages blocks all subsequent instructions.
Schedule and Execute occur out of order (which is where the performance win is!) The win has two

sources:

- some instructions will be delayed by random events the compiler can't predict, like loads that miss in cache. Other, independent instructions can occur while those instruction wait for whatever is they're waiting for.

- instructions tend to cluster as sequentially dependent instructions (C depends on B depends on A), but these dependency chains are usually surprisingly short (about 10 to 20 instructions) before they are terminated (usually by storing the result to memory), after which an independent chain starts. Thus if we can queue enough of these independent dependency chains in the machine, we can run many of them in parallel.

In *theory* the compiler could do this for us on an in-order machine. In practice this fails because
(a) most languages don't provide enough information about the non-overlapping of storage for compilers to be able to disambiguate that the store from the last dependency chain doesn't overlap with the loads that starts the next dependency chain
(b) most of these dependency chains are separated by branches of some sort, and the compiler can't be sure which way the branches will go
(c) trying to optimally schedule hundreds of instructions in a compiler is a combinatorial explosion problem, one that takes a lot of time, usually too much to be practical.

Retire, Complete, and Writeback tend to be bundled together in modern cores, and (mostly) happen in order.
Retire is easy to understand. It's the final point at which instructions have done (almost) everything they are supposed to do, and are no longer speculative. Their resources can be released, and reused by new instructions.
Writeback used to mean that results that were held in physical registers were written back to "architected registers". This is a term you will see if you read really old papers (say pre-2000 or so.) The evolution of out of order technology went essentially something like

(0) an in-order machine, with let's say 32 architected integer registers x0..x31 has a register file of these 32 registers. Every instruction affects its destination architected register.

(1) very early out-of-order has each ROB entry providing a slot that acts as the destination register of each instruction. So Register allocation happened automatically as allocation of the ROB slot. Writeback meant copying from that ROB slot register to the architected register file.
This is clearly built on the in-order model, with the idea that when anything fails, you fall back to the architected register file as the true state. But it means the number of physical registers equals the number of ROB entries -- and what to do about FP/SIMD vs int registers? And you waste a register slot for instructions (like branches) that don’t use a register. And all that writeback copying uses power.

(2) the physical register file was detached from the ROB. This solves the FP/SIMD vs int problem, and allows each of int register file, fp register file, and ROB to be independently optimally sized. But you're still paying writeback costs

(3) various alternatives were adopted for how to recover from misprediction using maps that describe how physical registers map to logical registers. Done correctly, this avoids the need to have a separate architected register file along with writeback; the state of the machine exists only in the physical register file together with the map between the physical and architected registers.
I refer frequently to the paper (2004) http://pages.cs.wisc.edu/~rajwar/papers/taco04.pdf *An Analysis of a Resource Efficient Checkpoint Architecture* by Haithim Akkary. This was written just at the point of this transition, so it describes multiple options for how this can recovery be performed.

(4) the usual state of the art for industry today is much (not all) of what Akkary is proposing in that paper.
Apple is the only company I know of that's using the next step in the evolution which is to use a History File to record the changes made to the mapping tables as execution proceeds. This change decouples the ROB itself (a queue of instructions) from the History File (a record of changes to register mappings). This is as opposed to recording these mapping changes as part of the ROB.
The first advantage of this is that once more the two can be separately sized, rather than tying the number of instructions in the ROB to the number of changes made to registers.
The second is that the two are deallocated separately at Retire – the ROB can free up to 56 instruction in a single cycle; the History File, operating independently and with a more difficult task, can free up to 16 registers in a single cycle.

(5) The next step beyond 4 would be the use of Virtual Registers (to be discussed below). Apple doesn't do this yet, but probably will soon.

Finally the Complete stage is mostly meaningless (by the time an instruction is considered eligible to Retire it has executed and so completed its job)… except for the case of stores.
In the old days, at the the point of Retiring a store, the store would be copied from the Store Queue to the L1D cache. (This could involved pulling in a line that was the target for the store.) And it was only when that line had been returned from L2 or DRAM, and had the store written to it, that the store could be considered Complete.)
Nowadays, for a store:
- Execution consists of calculating the store address, and placing the store address and store data in a store queue.
- At a later point (when the store is non-speculative) the store will move from the store queue to a write buffer (sitting between the store queue and the L1D cache) at which point the entry in the store queue can be released.
- But the store is still, (in some sense) not complete! At some random later point the store will move

from this write buffer into the L1D. And at some random even later point, it will be made visible to other cores.
It's something of a question of exactly what you're trying to achieve as to when you define a store as Completing, so once again this has become a somewhat obsolete stage.

Essentially Retire now means Deallocate (resources) in the same way that Rename means Allocate (resources).

Here's another way to think about this, as described in (2009) https://www.disca.upv.es/spetit/publications/ieeetoc.pdf *A Complexity-Effective Out-of-Order Retirement Microarchitecture*.
The basic OoO CPU allocates, at the last stage of in order execution (ie at Rename) various resources that will be required by later execution, including LSQ entries, registers, and ROB slots. And it releases those, again in-order, by waiting until whatever instruction is at the head of the ROB has completed, and then running forward along the ROB as rapidly as possible, releasing resources until it hits another blocking instruction. The most common reason to block the head of the ROB waiting for an instruction to complete is waiting for a load that missed to DRAM, though we may wait a few cycles for shorter instructions like say a divide.
This basic design makes it very easy to recover from either misspeculation or exceptions. In either case we simply have to execute up until the problematic instruction (eg the load that generated an MMU exception, or the branch that was mispredicted) is at the head of the ROB, and then restart the machine. This works because Retire/Deallocate/Commit happens in order and won't allow anything to be stored to permanent machine state until we know that each instruction being committed should in fact have executed.

But this also ties together various different ideas (maintaining valid machine state vs deallocating resources that are no longer needed) and a pattern we've already seen repeatedly and will see more is you speed things up by breaking up a task that is done adequately by one structure into two tasks both done better by optimized structures.
In this case the big problem is that resources (eg LSQ entries and registers) are being held long after they were needed for execution, simply because deallocation only happens at this in-order final step. So, for example, if the ROB is blocked by a load to DRAM, then we will execute as much as we can but eventually we will run out of something (usually registers) and the machine will freeze until the load returns from DRAM. Can we avoid that? We know that nothing unexpected can happen in terms of the load (MMU exception tests have been cleared, LSQ overlap with a possible store has been cleared) so why do all the later instructions need to wait for this load before they release registers for reuse?

Thinking this way gives you various possibilities for *OoO Commit*. Like everything, people have imagined various versions of this, and as a very new concept (quite possibly not yet implemented by anyone) there is not yet any consensus on the best way to do things. But the paper I reference tackles one

of the less obvious aspects of the problem:
The nice thing about the ROB is that it's a large, but in-order structure, so it's not very area or power expensive, it maintains the program order you want "for free". Suppose we are blocked at the head by a load, but the following few instructions have nothing to do with that load and are just some simple arithmetic. If we commit them out of order to release some registers, how can we keep using the ROB as an in-order instrument (and we do need that order for cases where the head of the ROB is speculative, or generated an exception, and we need to flush everything after the head). Simply removing random elements from this in-order queue and moving everything else up is energy expensive. But leaving holes means a godawful mess in terms of tracking which instruction follows what.
The idea of the paper is to replace the ROB (handles *Validation* and *Deallocation*) with a shorter queue that only handles Validation. Instructions are placed into the Validation Buffer (VB) in order at, eg Decode time, just like the ROB, and move through the VB in order, just like the ROB. But they only remain in the VB as long as the head of the VB has the potential to fail. So, for example, now if a load to DRAM is at the head of the VB, it only remains at the head until we know that there's no possible exception or no possible load/store address aliasing. Likewise branches can exit the VB once they are resolved. A third sort of instruction is something like a divide, where (ideally) we establish in the first cycle of execution that division by zero is or is not possible (ie possible exception) and once that's settled, a non-exceptioning divide can also leave the VB.
The end result is that most instruction situations that would block the ROB, and eventually cause the machine to lock up, can in fact pass through the VB in a few cycles, without the VB ever filling up. Later those instructions can, out of order, release their resources as appropriate. For example, once again, the long-lived load to DRAM will have to hold onto its destination register for many cycles, until the data is returned from DRAM (and then even longer, as subsequent instructions wish to use that register). Presumably this "release of resources" will be something like decrementing the reference count for the source and destination registers of this load instruction.
But unrelated instructions after the load can now release their registers out of order, with a net effect that the pool of available registers is effectively (averaged across many programs, and very dependent on other machine details) effectively about 30% or so larger. Likewise the LSQ is effectively about 10% larger due to LSQ entries being released earlier.

This scheme is, like all schemes, not perfect. Consider, for example, the bete noire of the modern OoO CPU: a load that misses to DRAM followed by a branch that depends on the loaded data. This will allow the load to rapidly graduate out of the VB but the branch will now sit at the head of the VB, blocking progress until the load returns. But this is still better than the ROB case. The ROB prevented progress for every load that missed to DRAM, the VB only prevents progress in this (somewhat more rare) situation of a load followed by a branch.
If we are willing to be more ambitious, and allocate more resources, we can even handle this sort of situation somewhat. When we realize this case has occurred, we can generate a checkpoint, a snapshot of the machine state just before the branch, and then let the branch graduate and keep executing past it. If the branch did turn out to be invalid (should not have been graduated past the head of the VB) we can restore state from the checkpoint.

This scheme, while feasible, is of limited benefit beyond pausing when the VB fills up and we wait for the branch to be resolved; the reason is that a checkpoint can restore register state, so it can allow us to recover from registers that were reused as we executed the speculative (and ultimately incorrect) code. But a checkpoint cannot store memory state. So what do we do about stores that are also speculative? We can't graduate these from the VB, and so now LSQ slots will be the resource that ultimately fills up. (But what if we graduated stores that are "safe" in every respect except that they are branch speculative to a different type of structure that is more like an L0 cache and less like the LSQ...?)
The most extreme version of these ideas shade into things like transactional memory, where you now have "speculative" bits associated with data in the L1 cache, and you allow some speculative transactions to proceed all the way to the cache, with the "speculation boundary" now being that these speculatively written values in cache are not allowed to leave the cache (not to service snoops, not to write to DRAM or DMA or anywhere else) until the speculation episode is over and we go through to clear all the speculative data bits. Both Intel and IBM (in POWER8 and 9) have tried transactional memory and given it up as being too complicated to be worth the payoff (though I don't believe either fully exploited it as I've described here, they used it only for lock elision and transactions, not to increase how aggressively an OoO CPU can speculate). IBM still has transaction memory in the z/ Series so at least one team still has faith in the idea. ARM has the TME (Transactional Memory Extensions) instructions defined but so far no-one on Team ARM seems brave enough to try where Intel and IBM POWER have failed.

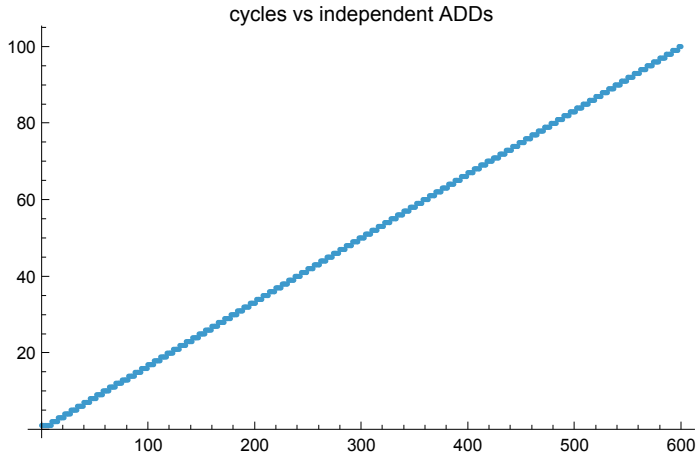
So, bottom line is that
- you can replace the ROB with a VB, at not too much design disruption, and the payoff is effectively a 30% or so larger register file and 10% or so larger LSQ, for no extra area or power. Very nice.
- if you want to be *much* more aggressive you also speculate your way past *in-order validation* of things like branches, but the cost to do so is a radically redesigned L1 cache *and* interaction with the rest of the SoC (cache can't release any speculated writes to cache, which affects snooping, DMA, etc).
Let's get step one, transitioning from ROB to a VB, implemented and optimized, before we start worrying about these crazier ideas!

Back to our stream of integer instructions, and to experiments.
Each of these allocates a physical register in Rename. That register can be freed once the instruction Retires (but not earlier). Which means that if the integer instructions are blocked by a delay block at the head of the ROB, eventually all physical registers will be allocated, the machine will stall, our timing will see a glitch. Let's try it!
Remember as we perform these experiments below, all the details I gave above (eg Apple's use of a history file decoupled from the size of both the ROB and the physical register files) was not told to us by Apple! It's all the result of experiment, and that's what this long run of experiments below is designed to figure out.

Integer Physical Register File Size (ADDs)

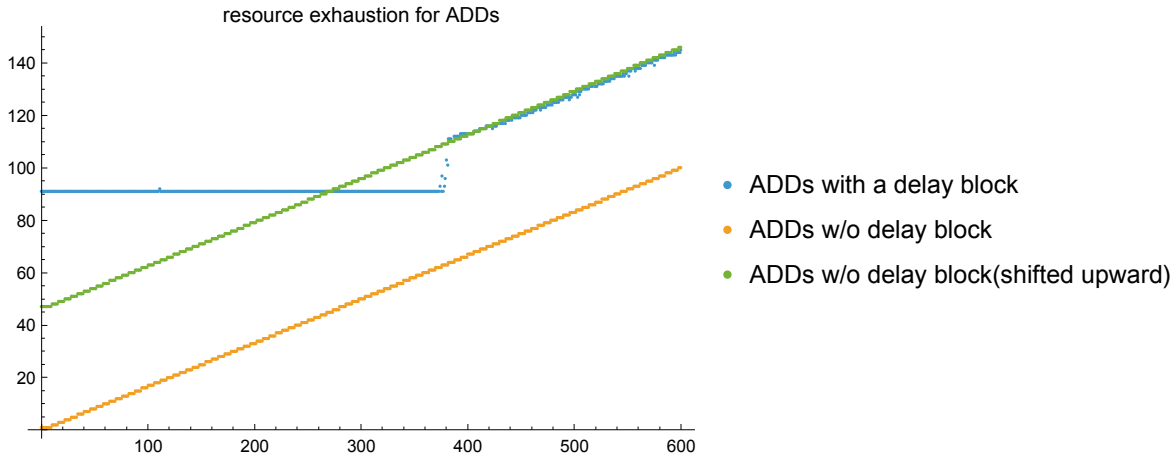
Before going further let's validate our foundational ideas about the physical register file.
We start with the simplest model, then see if it breaks as we add tweaks.
Start with a run of instructions (`ADD x0, x5, x5`) that each require a physical register to hold the generated result.

Out[462]=



No surprises, the primary notable fact is the slope (6 independent integer adds/cycle).
Now let's add in a delay.

Out[464]=




```
(* {377,91},{378,93},{379,96},{380,103},{381,101},{382,111} *)
```

Here are the notable points:
- Clearly at ~378 ADDs, there is a notable jump in the cycle time.
- The slope of the delayed ADDs is unchanged, just shifted up by 46 cycles.
- So at ~378(=63*6) physical registers, at 63+2 cycles in, ADD progress stops, ie at INT_STOP_-TIME=(num phys reg/6 + 2 [two cycles for branch and enqueing the sqrts])

For how long? Until the ROB clears, so
WAIT-TIME=DELAY TIME (13*numSqrt) - INT_STOP_TIME.

I *think* the transition occurs over ~5 cycles because of 8-wide Rename.

```
| Delay<---------------------------------->|                 | Delay<---------------------------------->|
|Ops<--------------------------xxxxxxxx-------------->||Ops<--------------------------xxxxxxxx-------------->|
```

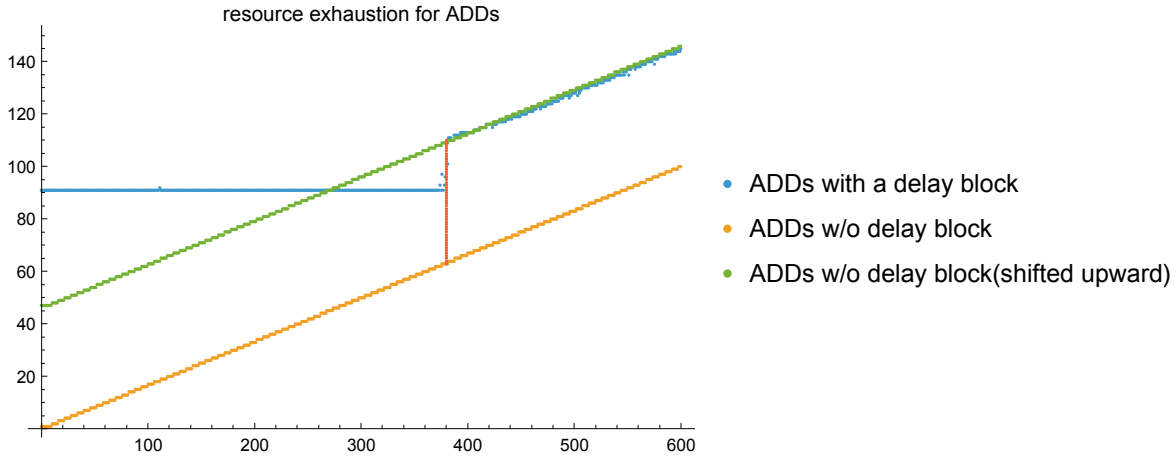
Let's assume there's a few cycles of jitter in xxxx (just assume that for now). And that the actual physical register count is say 380. This would mean that for N very near the resource limit (say 379 or 380) those N's will occasionally hit the resource limit (and execute the slow path, where xxx force a delay and everyone has to wait for ROB to clear) and will occasionally not hit (meaning the fast path, no XXX delay, no requirement for the next block of fsqrts to delay till the ROB clears before they can begin execution).
If such a jitter existed then we'd see something like maybe 1/6 of the time the 378 case is slow, 5/6 of the time it is fast, and the average cycle count is 5/6 to 1/6 weighted. Likewise the 379 case is 2:6 to 4:6 weighted and so on, and we'd get the sort of staircase we see. Depending on exactly how we aligned all the instructions, and the source of jitter, we might be able to get a perfect jitter free-case, to a perfect linear ramp.
I assume the jitter reflects differential widths in different parts of the machine (eg Decode can generate 8 instructions per cycle, but integer execution can use up only 6 per cycle) meaning that slightly different numbers of instructions could be enqueued (and blocking the transit of fsqrt from the in-order path to the OoO path) in different places on different loop iterations. But trying to pin that down further using this particular harness seems inefficient.
Going forward we should GENERALLY expect a slightly noisy transition region (over 6? 8?) cycles because of our jitter explanation, and not waste time obsessing over its existence.

So what is definitely established is that
• there's some sort of limitation at around 380 ADDs (seems like physical registers)
• assuming the above point, we've also learned that registers are deallocated at Retire (deallocation is delayed by the Head of ROB blocking) rather than some more ambitious schemes that de-allocate a register when all *users* of that register have completed. We'll discuss this later.

It's important to have a correct understanding of the analysis, which is somewhat more abstract (less mechanistic) than Henry Wong's method. In particular it's very tempting to view the x-axis (the N axis, where N is the number of filler instructions between delay block) as representing time, and doing that will drive you crazy!

To be sure you understand, consider the graph below.

Out[465]=



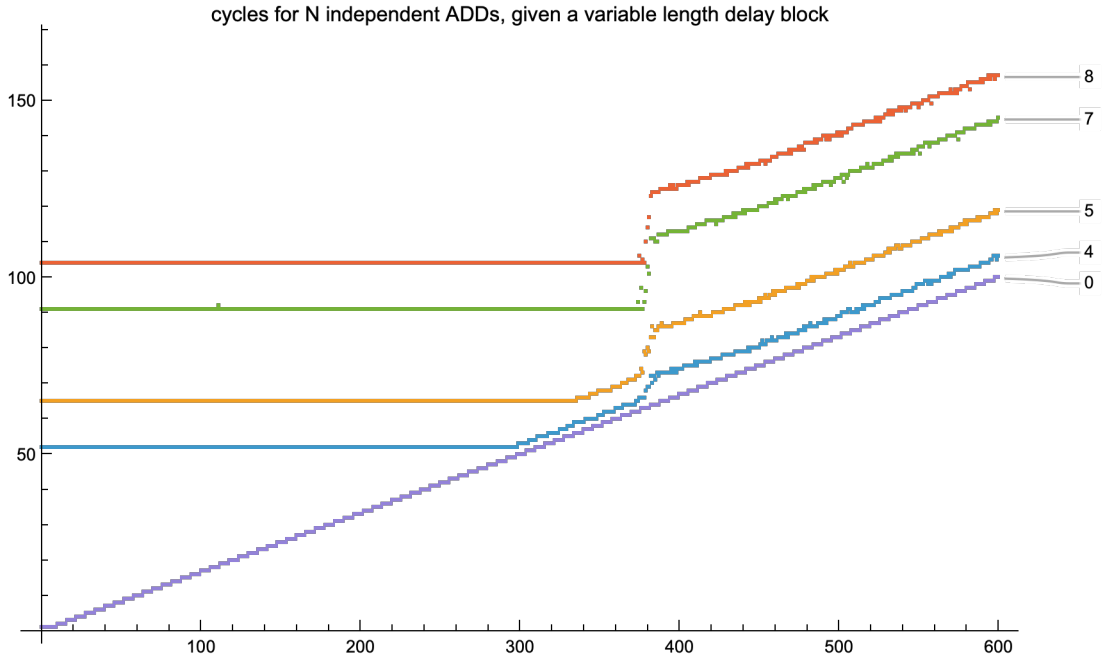
Essentially we have two independent blocks (call them chains) of sequential execution.
The first chain is the delay block, the block of 5 (or 10, or 20) chained `FSQRT`s. The amount of time this block would take to execute is independent of the number of filler instructions (ADDs), and can be viewed as the flat part of the blue curve, extended to arbitrarily high values of N.
The second chain is the `ADD`s. For small N, they take an amount of time linear in N, following the left hand side of the gold curve. For large enough N the execution of the ADDs is forced to block (the xxx time) while we wait for a resource to be freed.
Thus for N<~380 the time taken by the `ADD`s follows the gold line,
for N>~380 it follows the green line,
with the two lines linked by a jump occurring at N=~380.

SO: Loops with filler of less than ~380 execute on the fast path; loops with filler of more than ~380 execute on the slow path.
The curves show what happens for loops with differently sized filler, they *do not* show what happens in time as successive `ADD`s of the filler are executed!

The whole blue curve (the one we measure) is the time taken to execute (in parallel) the filler block (following the curve gold/red/green) and the delay block (always taking a constant 7*13=84 cycles). The time for a loop iteration is always the larger of these two possibilities.

To get a clean curve, it's ideal (though sometimes not possible) to have the delay block last substantially longer than the filler block takes to reach resource exhaustion.
Observe what happens when we don't do that.

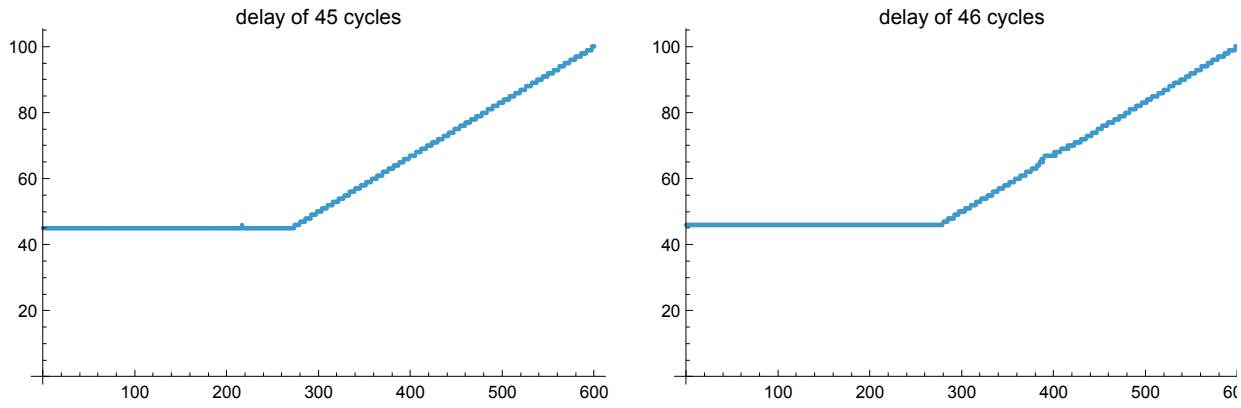
Out[473]=



We point out these salient facts. Note that 5*13 takes 65 cycles, which is time to perform 390 adds, so a delay of 5 is right on the cusp, able to show the phenomenon but not a clean jump.
• For a delay that is too short (4 FSQRT) we clearly don't cleanly hit the phenomenon of interest (ie ADDs delayed because of inability to allocate a resource, blocked behind the head of ROB).
• For a delay that is just on the money, (5 FSQRT) the crossover region (the jump at ~380 ADDs) is not smoothly isolated and it's easy to get distracted in the hopeless task of trying to figure out the exact structure of the crossover region.

an aside for experts

Out[476]=



It's interesting to consider why we still get a (small) bump for the case of delay of 4*13 (52) cycles.
The first parts of the curve, up to N=~380 are clear. What's unclear is why there should be some delay for, say, a filler of 390 ADDs.
After about 312 ADDs are executed, the head of ROB clears, the physical registers used by the first few ADDs behind the head of ROB will start to be released (in fact at a rate of 16/cycle as we will see) and there is no obvious reason for the ADD chain ever to be blocked or delayed, no reason for any xxx time.

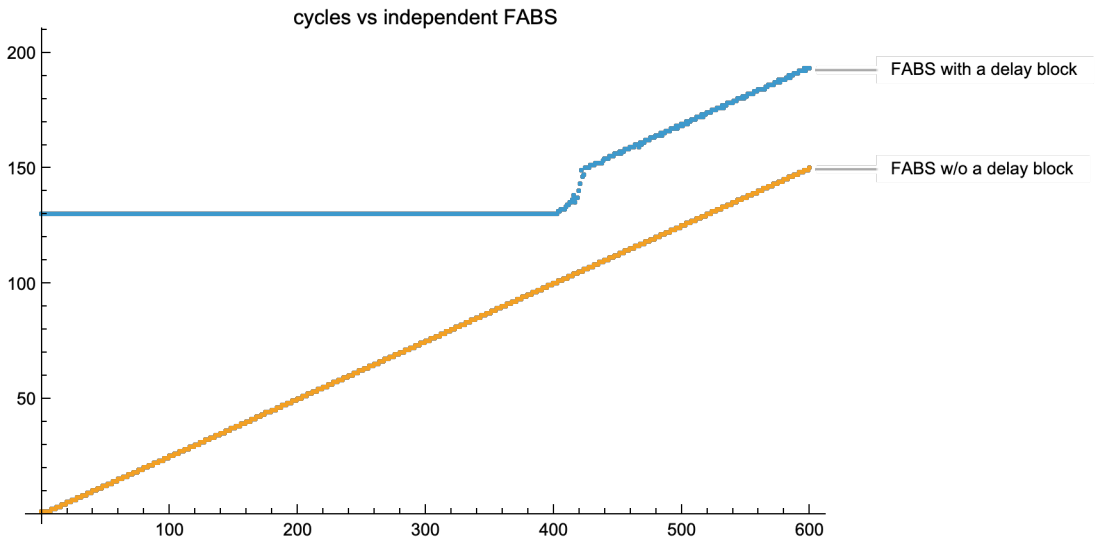
This understanding is (kinda sorta) confirmed by comparing a delay
of 45 cycles (4 `FSQRT s1, s1, s1 FADD s1, s1 FABS s1, s1`)
with 46 cycles (`4 FSQRT s1, s1, s1 FADD s1, s1 FADD s1, s1`).
45 cycles behaves as expected, 46 cycles shows a slight but definite bump at N=~380.

I believe this effect is a consequence of a low-level implementation detail in Apple's register file. I'll explain this below, in power aspects of the register file, once we have explained higher level aspects of the register file.

FP Physical Register File Size (FABS)

Let's try the same strategy for fp registers. The code is as before except we replace the `ADD` with `FABS d2, d0`

Out[479]=

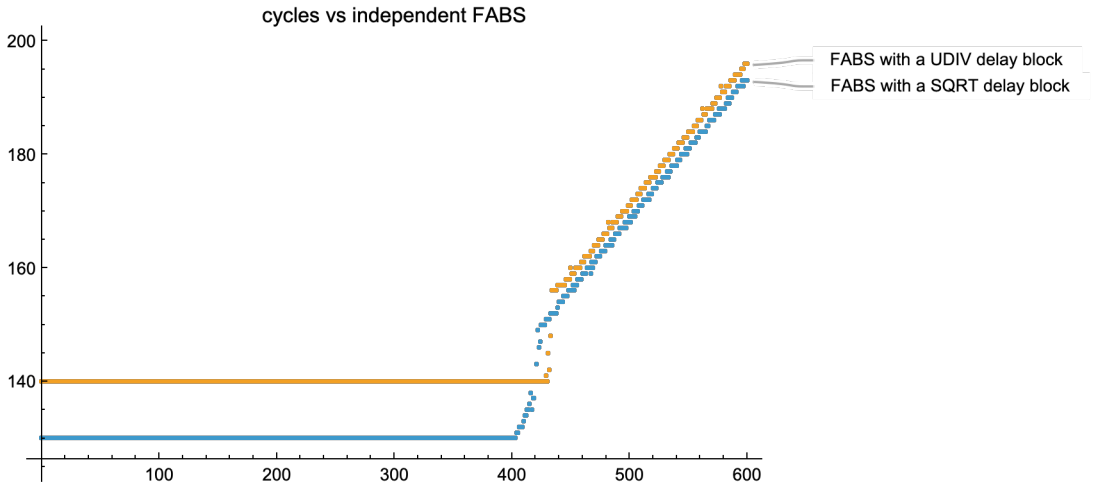


In[ ]:=
```
(* {403,130},{404,131},{405,131},{406,132},{407,132},{408,132},{409,132},{410,133},{411,134
{414,135},{415,136},{416,138},{417,135},{418,137},{419,137},{420,140},{421,143},{422,149},
```

So we see the structure we expect by now. The slope of both lines is 4 independent fp instructions/cycle, as expected.
The jump happens over the range 403..425, so over a range of about 25 instructions, centered at about 415 instructions.
The jitter range is a little messier than before, I assume at least in part because the delay block and the `FABS` are sharing registers and execution paths.

We can use an integer delay block (chained `UDIV`, 7 cycles/iteration) to get a slightly cleaner result

Out[481]=



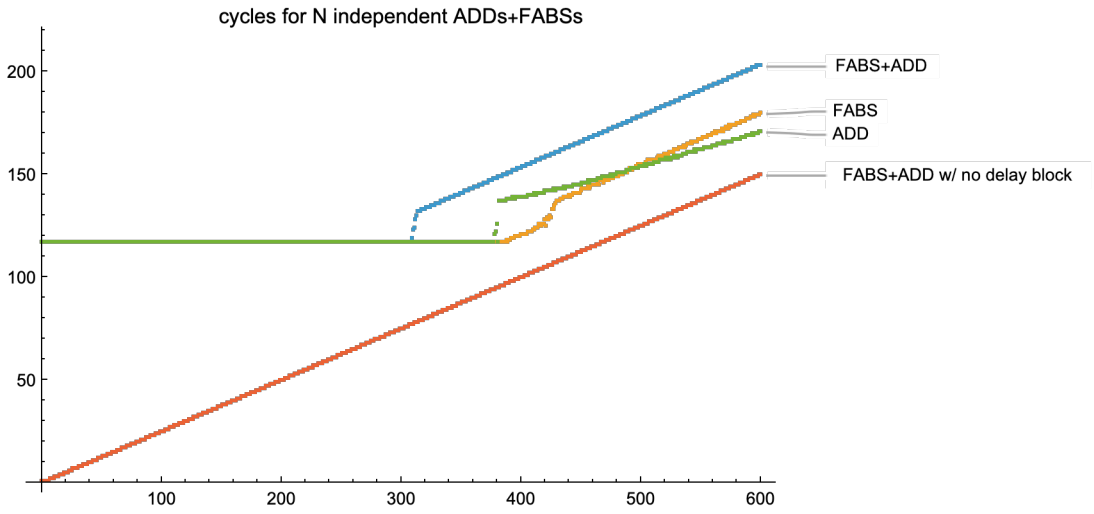

```
(* {428,140},{429,141},{430,140},{431,145},{432,142},{433,148},{434,156},{435,156} *)
```

We see a narrower transition region, and without the delay block using up some of the fp registers, we see that the fp register file is perhaps closer to 432 or so in size.

Register File Size (All Registers)

The apparent number of physical registers for int and fp are fairly similar.
One can entertain a number of possibilities (is there a common register pool?), so lets examine what happens when we try to allocate both integer and fp registers. Let's try a probe consisting of an `ADD` and a `FABS`.

Out[486]=



In[ ]:=

```
(* {308,117},{309,119},{310,123},{311,124},{312,128},{313,130},{314,132},{315,132} *)
```

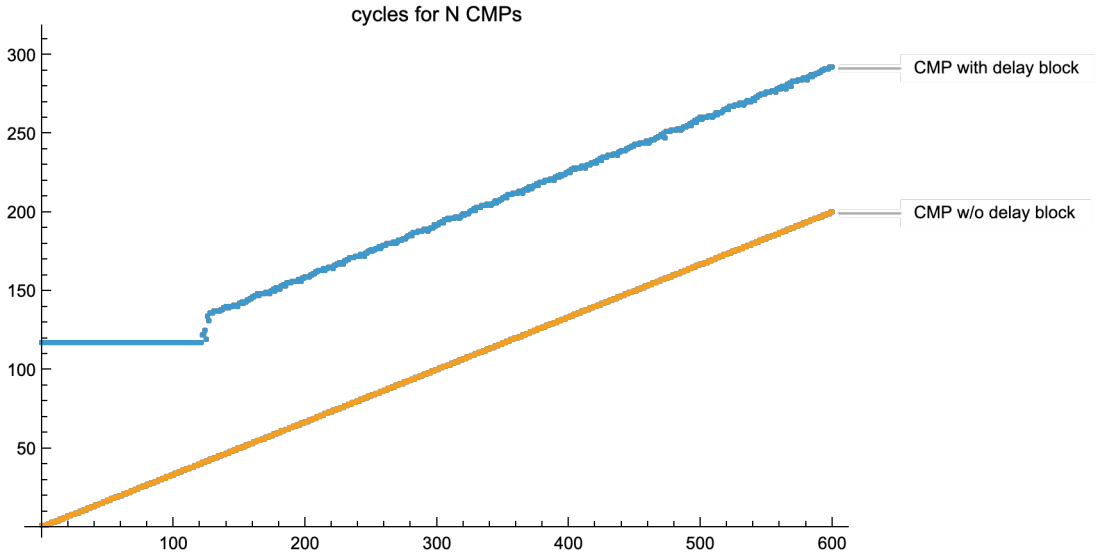
Now this is interesting!
The red curve omits the delay block, and shows a slope of 4 ops/cycle. We are throttled by the 4 fp

pipes.
The green curve and the gold curve are, respectively, the pure integer test and the pure fp test.
The blue curve is the case of interest, with a probe of (`ADD+FABS`).

We see clearly a standard jump, centered at about 312 or so. How can we interpret this?
Clearly the idea that registers are shared between int and fp fails. If that model held, we'd jump at something like 192 (half the ~384 registers allocated to fp, half to int, stall).
But something clearly is shared! There's an additional resource that constrains us, a pool of ~624 somethings that's shared by int and fp registers.

We can continue our investigation by testing the third available pool of registers, the physical flags register.
Let's start by testing `CMP x0, x0`

Out[489]=

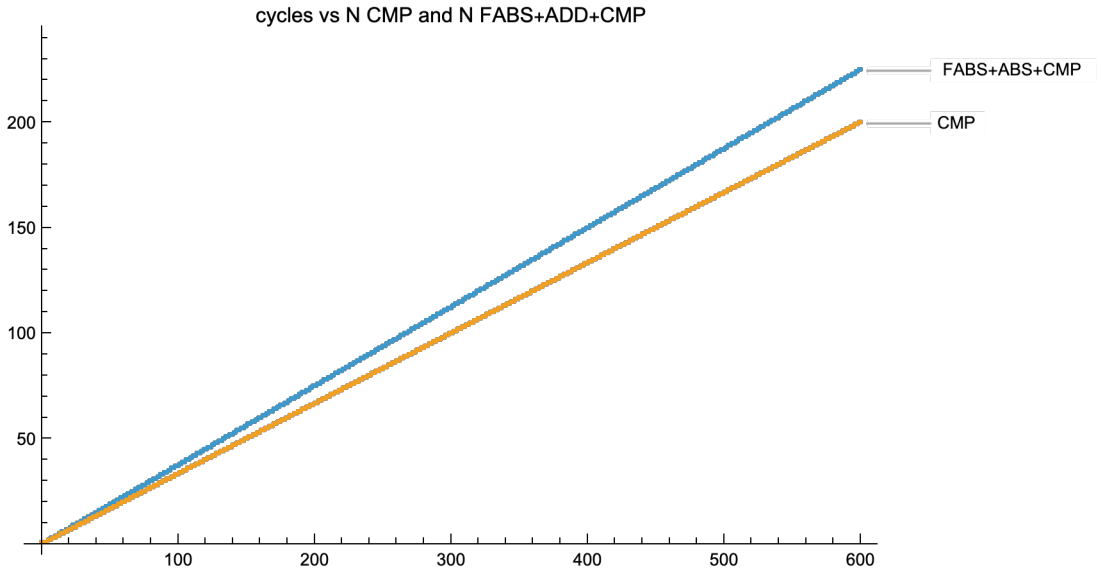


In[ ]:=
```
(* {121,117},{122,122},{123,122},{124,125},{125,119},{126,134},{127,131},{128,136} *)
```

So no real surprises. This time the slope of the lines is 3 instructions per cycle;.
However the jump is at ~124 (128?) physical flags registers. So not the same number as int or fp...

Now for the exciting part, run all three together! First let's investigate the simple case, with no delay block.

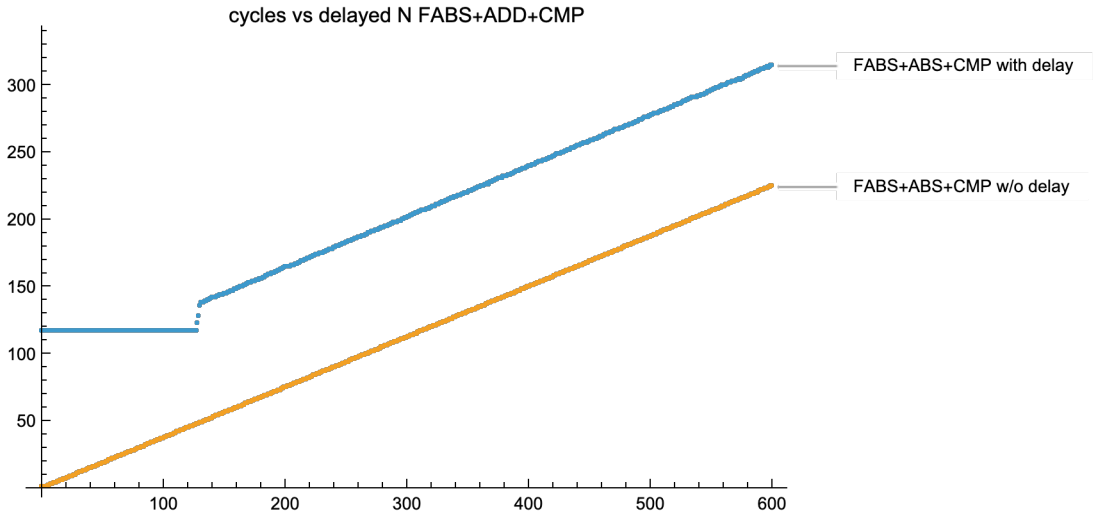
Out[491]=



Now our rate drops to 600 triplets in 225 cycles, 8/3=2.667 instructions/cycle.
The limiting point is the 8-wide front end (Decode, Map) of the machine. We have a total of 3*600 instruction, processed 8 /cycle, which takes 1800/8=225 cycles.

Now add the delay block.

Out[493]=

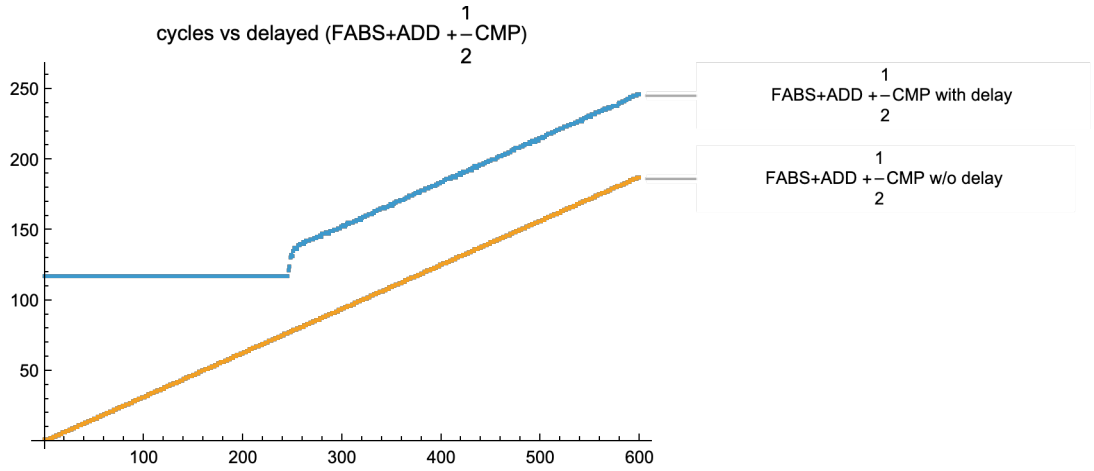


In[ ]:=
```
(* {127,117},{128,123},{129,128},{130,136},{131,138},{132,138},{133,138},{134,139},{135,13
```

We see that we're constrained by the number of compares; we can't get beyond that to test the region of interest. If there is a common pool of 624 somethings, cut 1/3 each way gives 208. Not reachable via a unit of (CMP+ADD+FABS).
What about 624/5=124.5? Right on the cusp! So let's try one CMP paired with two ADD and two FABS. We'll implement this as only allocating the CMP for every second (ADD/FABS) pair, so the unit is (ADD+–

FABS+.5 CMP).

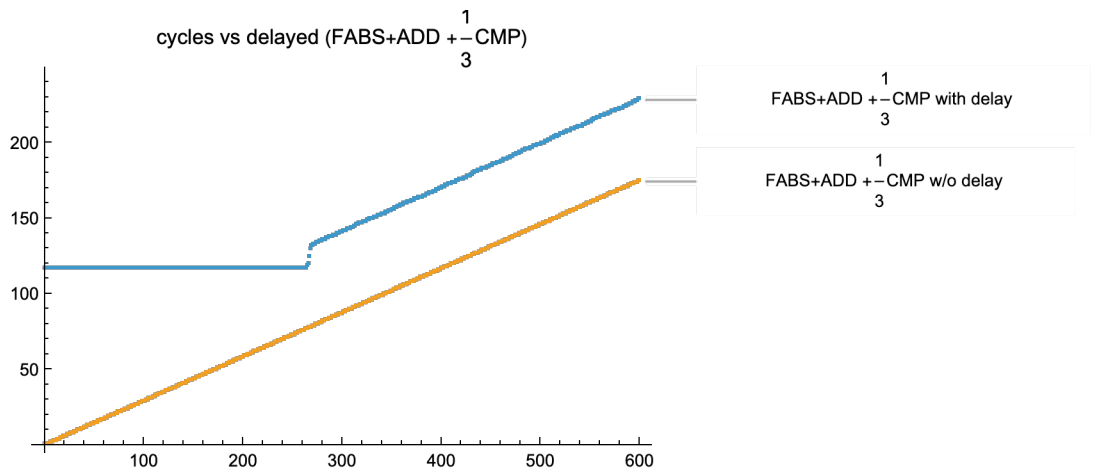
Out[496]=



In[ ]:=
```
(* {245,117},{246,121},{247,124},{248,130},{249,132},{250,131},{251,135},
{252,137},{253,137},{254,137},{255,136},{256,139},{257,139},{258,140}*)
```

Almost! But not quite?
The jump is at ~250. Is this what we expect?
250 units corresponds to 250*(ADD+FABS+ half a CMP), so we're possibly still being limited by the ~128 flags physical registers before we can get to the good stuff, since half of 250 would be 125.
But 250*2.5=625. It's plausible that we're seeing a signal. But let's confirm.
Drop to a third of CMP.

Out[499]=



In[ ]:=
```
(* {264,117},{265,119},{266,120},{267,125},{268,130} *)
```

OK, finally, the signal we're looking for.
The fundamental unit of allocation is $\left(\text{ADD} + \text{FABS} + \frac{1}{3}\text{CMP}\right)$ (implemented as adding a CMP for every third ADD+FABS).

We execute 266 of this unit) so 266*(2+1/3) =620 register allocations! And this time all three register pools (int, fp, even flags) are only partially exhausted, with 266 int registers, 266 fp registers, and 87 flags registers being used up at the point of the jump.

So there is some *communal* structure related to the allocation of registers, of size ~620 entries, which is used up *in addition to* the register files.

The History File (ROB structure tracking all changes to Register Mapping tables)

What are these unknown shared ~620 resources?
I made many different hypotheses, and experimented with many different things, but the real answer appears to be in this patent: (2019) https://patents.google.com/patent/US20210064376A1/ *Last physical register reference scheme*.

One way in which Apple optimizes the Retire tension I discussed is through a structure called a History File.
Conceptually the history file sits next to the ROB, and while the ROB holds a sequence of instructions (all instructions, in-order), the smaller HF holds the sequence of changes that were made to the three (int, fp, and flags) register mapping tables.
The HF has ~620 entries, and requires an entry for every instruction that will modify the mapping tables (so basically any instruction with a destination register, including a flags destination). The HF also includes a single bit flag that this is the last mapping referencing the physical register referenced by the mapping.

To understand this aspect of the ROB, perhaps start by reading (2001) https://courses.cs.washington.edu/courses/cse378/10au/lectures/Pentium4Arch.pdf *The Microarchitecture of the Pentium® 4 Processor* which describes the P6 and then the P4 schemes.

The P6 scheme was very simple (as these things go). The ROB and the PRF were the same structure; allocation of a ROB entry was the same thing as allocation of a destination register. Speculation was handled by having a separate PRF, named the RRF (Retirement Register File), that represented the "true state of the machine as of Retire" so that you could recover just by pointing all the Map tables to the RRF.
Problems with this scheme include
- that your ROB and PRF are linked in size, though really you want the ROB to be some scaling factor larger than the PRF; and
- the copying of a GPR to the RRF on every retire costs additional energy.

The P4 scheme has separate (and separately sized) ROB and PRF, and a separate Retirement RAT (Retirement Mapping Table) that's updated at retire.

This is progress! We have
- separate sizing of ROB and PRF; and
- the energy cost of updating an entry in the Retirement RAT is less than that for copying a register to the RRF.

It's hard to see what the problems with this scheme are, until you start being more ambitious.

Suppose we have a situation where there are many instructions in the ROB, and halfway down the ROB there's a branch that we know is mispredicted because we just executed the branch and learned that. So we need to engage in branch mispredict recovery.

The P4 recovery scheme is to mark in the ROB entry for the branch that it was mispredicted, and continue as usual *until the branch retires*. At that point
- the instructions before the branch have all retired (they were ahead of the branch so were valid);
- as they retired they updated the RRAT (so RRAT represents an accurate register mapping table at the point of the failed branch);
- and so handling the misprediction mainly means copying the RRAT into the frontend Mapping table.

But this also means that all that time from when we learned the branch was midpredicted until it retires is dead time! The machine kept chugging away at instructions after the incorrect branch instead of doing something useful.

What we would prefer, as near as feasible, is, rather than waiting till it retires, *as soon as* we know a branch is mispredicted
- we keep all the instructions *older* than the branch (and still executing) alive
- we kill all *newer* instructions
- we flush all *inappropriate enqueued* instructions and begin fetch from the new address
- the new instructions begin execution even while we're still retiring the old instructions in the ROB that were ahead of the the branch.

For this sort of scheme to work, obviously we need machinery to mark and flush instruction as appropriate; but most relevant right now
- we need to have the correct mapping table in place as soon as the newly-fetched instructions enter the machine; so
- we can't wait until the branch Retires and we can swap in the RRAT
- we need to construct the correct table, and install it at the correct point of execution (so that it's seen by the new stream of instructions as they hit Map)

Details of how you might do this are given in http://www.cs.wisc.edu/~rajwar/papers/taco04.pdf, which we've already referenced once. This paper discusses many things, but in section 3.2 it discusses two slight modifications to the P4 scheme to reconstruct the mapping table as fast as possible without waiting until the branch Retires. Both alternatives require each instruction as stored in the ROB to carry additional information that looks something like "this instruction swapped pRegA for pRegB as the mapping for lRegC"; and by running through these mapping statements in sequence you can recon-

struct the mapping table from a given starting point.

It's unclear what Intel does nowadays but they were aware of this limitation in P4, and addressed it in Nehalem.
(2011) https://www.intel.com/content/dam/www/public/us/en/documents/research/2010-vol14-iss-3-intel-technology-journal.pdf *Intel Tech Journal Nehalem Issue*
page 16 discusses the problem, but says nothing beyond "we have a fix".

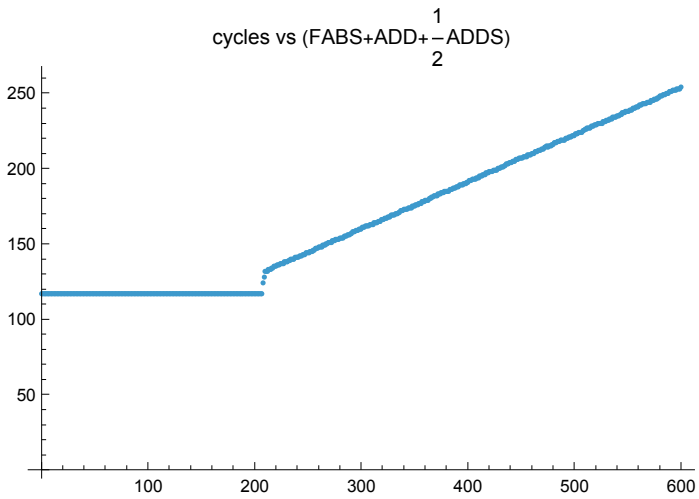
Akkary's more performant solution, the one called HBMAP+WALK is essentially what Apple uses. However rather than store the mapping updates associated with each instruction (updates that are not required by many instructions) Apple separates those changes from the ROB into a separately sized *History File*, with a pointer in each ROB entry pointing to a History File entry (or something like that).
The essence of the History File is that it doesn't track values or instructions, all it cares about is changes that were made to the register Mapping files, so that by rewinding the History File, you can rewind machine state to the last known good point that you wish to reach.
A secondary task of the History File is to note when physical registers can be freed for reuse. We'll see how that happens below in the Duplicating Registers section.

How Do “set flags” Instructions, Like ADDS, Modify the History File?

Buoyed by this understanding, let’s test `FABS+ADDS`. (Recall that an S suffix on an integer instruction means that instruction also sets the flags register in addition to the ADD or whatever.)
The question of interest is: how the flag register rename is handled by the history buffer?
The obvious expectation is that ADDS, generating both an add result and a flag result, will require two slots in the History File., since there are two remappings being performed, in two different (int and flags) mapping tables.

However we have to be careful in this test. We can’t exceed 128 ADDSs before we hit the limit of the number of physical flag registers! We’ve already seen this above when we were probing the History File.
Let’s try (`FABS+ ADD+ .5 ADDS`). This consists of 2.5 instructions, but might require either 2.5 HF slots (one for each instruction) or 3HF slots (2 slots for the `ADDS`, one for `ADD` destination, one for the flag destination). So we would expect this to max out at N units of either 206(=620/3) or 248(=620/2.5).

Out[501]=



In[ ]:= `(* {207,117},{208,124},{209,128},{210,132},{211,132},{212,132} *)`

Hah! Isn't it nice when our understanding advances to the point where was can start to make testable predictions?!

So that's pretty clear! Instructions that both perform some other calculation and set the flags register (ie have two destination registers) require two HF slots.

More precisely, given the way I described the History File (to unwind register mappings) one slot will be allocated for each register rename. Meaning that we should expect

• zero HF slots for stores

• zero HF slots for explicit prefetch, `PRFM`, instructions

• zero HF slots for writes to special registers (somewhat... some special registers are renamed)

• one HF slot for standard loads

• two HF slots slots for load pair (which takes two destination registers).

These are all confirmed. (Note our goal right now is to understand the HF, not to probe details of loads, stores or PRFM instructions!)

BTW the special registers case is surprisingly interesting. Reading (and especially writing) special registers has usually been a very slow path, because the easiest way to handle a special register write is to delay performing the operation (and all subsequent instructions) until the relevant instruction is at the head of the ROB, ie to "serialize" the instruction stream. The reason for this is that there are multiple different special registers that all do something different (change memory mappings, change how interrupts are handled, change floating point behavior, ...) and you can't make those changes while the instruction is speculative because they would be so difficult to unwind if the speculation proved in error.

But that doesn't stop Apple. Check this out: (2019) https://patents.google.com/patent/US10838723B1/ Speculative writes to special-purpose register.

Different registers are handled differently but the basic idea is that

- Some easy cases (most obviously moving data to and from the flags register) are treated via machinery close to the Rename Mapping machinery.

- For other cases, the write to the register is allowed to proceed out of order, but the new value is retained in specula-

tive storage. Reads of the new value (with an age stamp later than when the new value was stored) get told the new value; everything's consistent. Then at the point when the MSR instruction is ready to retire, the new value gets written to permanent special register storage and the machine state is changed.
So you don't have to pay serialization costs every time you change a minor register, or when you make a series of changes. Even when the system does require serialization, it can require this only after a sequence of changes has exhausted temporary storage.

There' s a second interesting issue here.
Any modern machine these days will crack more complex instructions into smaller instructions. For other CPUs this seems to be primarily about execution complexity, but Apple have generalized this into a powerful tool. The insight is that different stages of the pipeline may want to treat an instruction as a single unit or multiple units depending on exactly what the instruction is doing.
For example an ADDS or a LDP (load pair) want to be treated as two instructions for the purposes of register allocation (Rename) and deallocation (History File) but as a single instruction for the purposes of execute.
Conversely an instruction like ADD (shifted), which shifts one of its arguments before adding it, wants to be treated as a single instruction for the purposes of allocation and Retire, but to be split into two operations for execution. The patent is here: (2012) https://patents.google.com/patent/US9223577B2 *Processing multi-destination instruction in pipeline by splitting for single destination operations stage and merging for opcode execution operations stage*.
One interesting aspect of this that I slid past you is that most instructions that split into two executed parts, like ADD (shifted), don't have to assign a physical register for the intermediate result (in this case the result of the shift); the intermediate result is just picked by the ADD off the bypass bus and no register is wasted. However, strangely, the EXTR instruction, which looks to my eyes looks like it could use this same bypass mechanism, for whatever reason (real issue? or just no-one ever optimized it?) does allocate a genuine intermediate register, with the extra resource allocation waste that implies.

ROB Duplicate Registers (MOV xn, xm)

This is hardly the end of the story. We know from other investigations that Apple provides zero-cycle moves. The idea of zero-cycle `MOV` is obvious, in that you "execute" the move merely by updating the register Mapping tables, rather than by sending an instruction through an integer execution port. What makes this not exactly trivial is that you now need a variety of extra book-keeping to track when it is safe to release a physical register (which now may have multiple duplicate users).
It's also worth noting that Apple seems to prioritize the zero-cycle aspect of this technology (also used to load Immediates into registers); it's nice that the technology allows the machine to act as though it has a few additional physical registers (because a `MOV` "reuses" a physical register, rather than allocating a new one) but that's not the priority, the priority is to reduce the latency of `MOV` in a chain of operations.

Apple appears to have used three successive solutions to this, and it's worth understanding all three because even when a solution is abandoned, some ideas from it live on in other aspects of the CPU.

The central problem around renamed registers is when they can be reused.
The first half of renaming is obvious – you need a table that maps architectural registers to the current physical register (or something equivalent, like the ROB slot of the instruction that will eventually

produce the value of interest), and you need a pool of free registers from which you allocate a physical register for each successive destination register.
The difficult part is how you move registers into that free register pool – how can you, as cheaply as possible, and as early as possible, know when a physical register can be reused?
Conceptually there are three parts to when the physical register is no longer required:
- when the store to the register has been performed AND
- when all the readers of this physical register have executed AND
- when the logical register to which this physical register is mapped has been overwritten
Each of these three parts has multiple solutions, and it's a constant weighing of options as to which is best. For example for the second sub-problem, one can imagine options that include
- have a table that more or less notes the ROB slot for each reader of a physical register, and clears that entry as the appropriate ROB entry retires, until all entries are clear
- have a counter that goes up when a reader goes through Rename, and is decremented when a reader is Retired, ie a reference counter
- have a way to scan the entire table (make it a very wide matrix of bits) so that you can easily see if a column is bit-free vs has at least one bit set indicating a user

We see Apple working their way through variants of these ideas over time.
First off, an early patent is (2005) https://patents.google.com/patent/US20070050602A1 *Partially decoded register renamer*. Good luck understanding this patent first pass through! But it's actually interesting once you figure it out.
It refers to a core somewhat like A6, so out of order probably 3-wide or so, and primarily interested in the question of: how does the ROB communicate with the register allocation mechanism that some instructions have Retired? (Regardless of what solutions you adopt to the issues I raised earlier, this sort of communication is necessary.)
To understand the solution (which may be used, even today, in a fancier version) you need to know that the machine being described has a 64-entry ROB, treated as 16 "rows" which can each hold four instructions, and one row can Retire every cycle. So you want to communicate with the register allocation mechanism
- four instruction IDs that have retired
- look up those instruction IDs in a table (ie use a CAM structure) so you can set various flags for the appropriate registers
- but you don't want to pay the cost having four simultaneous CAMs
So the idea is, instead of indicating the Retiring instructions as four integers, you indicate it as a rowID (a 4bit value) and a mask (four bits) indicating which of the four instructions in that rowID are Retiring. Now you only have to use one CAM (for the rowID) and for each entry that matches the CAM you run a secondary test that it matches the appropriate bit in the bitMask.
It's a low-level patent, but shows the sort of tricks Apple is constantly playing, using whatever structure is present in a set of values (in this case, that Retiring instructionIDs are sequential) to reduce the workload.

Now let's look at what needs to be added if we want to use zero cycle moves (and thus allow duplicate logical registers associated with the same physical register).

The first Apple solution uses the RDA (register duplication array), a CAM that can describe ~8 registers that have been subject to duplication. If you create further duplicated registers beyond the CAM limits, the `MOV`'s execute like normal instructions.
Early Apple patents like 2012 https://patents.google.com/patent/US20130275720A1/ *Zero cycle move* discuss the RDA in various contexts.
(The RDA is still being referenced as of 2018 https://patents.google.com/patent/US10838729B1/, but that seems to be a lazy lawyer using obsolete boilerplater and diagrams!)

By 2014 we see https://patents.google.com/patent/US20160026463A1 *Zero cycle move using free list counts*, a scheme that (to my mind) makes a lot more sense, that trades the RDA (a CAM structure) for a few extra bits in each physical register tracking the number of users. So we have a reference-counting type system.

Finally we see in 2019 a third scheme https://patents.google.com/patent/US20210064376A1 *Last physical register reference scheme* which specifically states "It is noted that in the previously used register duplicate array (RDA) scheme, a new entry would have been created for PR6 with a reference count of 2. However, in the new physical register last reference scheme, keeping track of the total number of references to a physical register is no longer necessary. Rather, it is sufficient to track only the last reference to the physical register. This is a more elegant scheme that is easier to implement, uses less area, and has higher performance."
(This same patent shows how Apple uses a History File.)

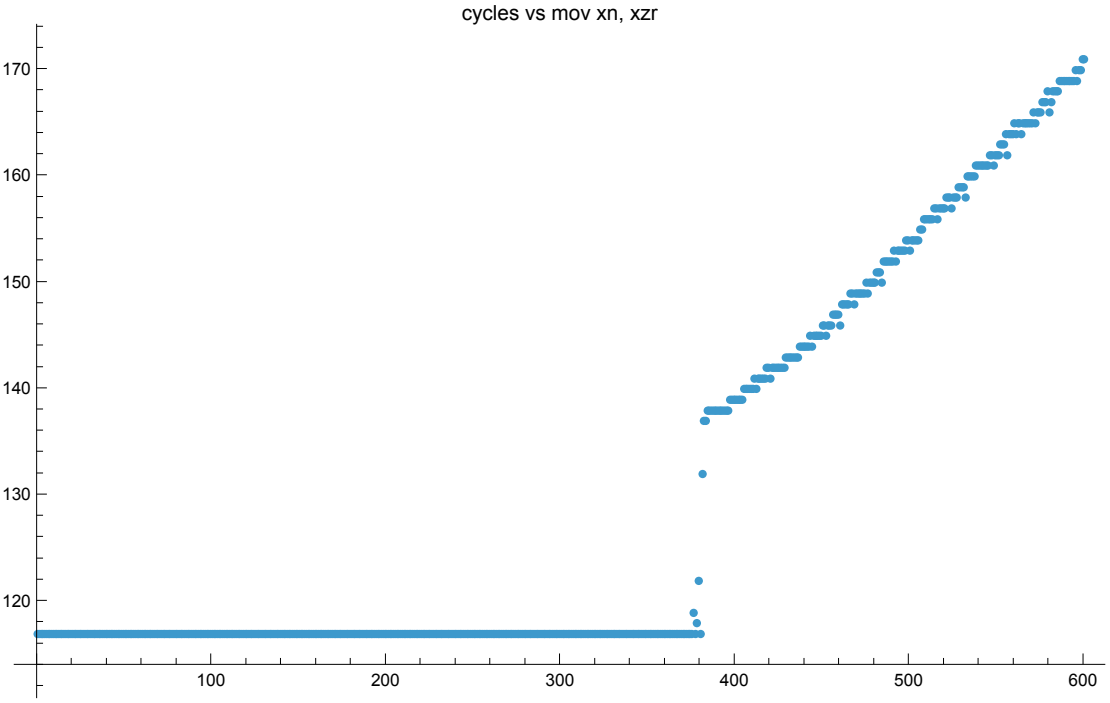
The M1 appears to implements this more elegant scheme, and appears to show none of the limits one might expect from the two prior schemes.
This third scheme imposes no limits on the number of duplications (`MOV`'s) that can occur, either on the number of different source registers that are duplicated, or the number of destinations to which a given source register can be duplicated. If you look at the patent, you may wonder how it is implemented -- it requires a frequent lookup against the Mapping table to see whether a physical register is the target of a mapping, and this looks like an expensive CAM lookup. My guess is that in fact the relevant structure is implemented as a bit matrix, a structure we will see again when we discuss Scheduling.

Let's see what we can learn experimentally about this duplication mechanism. Forget the zero-cycle aspects for now; what this means for resource allocation is that an operation copying one register to another (`MOV xn, xm`) does not allocate a new destination register as the physical register for `xn`; instead it simply updates the mapping of logical `xn` to point to `xm`'s current physical register. There are multiple ways this could be implemented (as the Apple patents already point out) so we want to probe possible ways in which this might fail, or behave suboptimally

xzr

First we'll try creating multiple references to `xzr`.

Out[503]=



In[ ]:=
```
(* {378,118},{379,122},{380,117},{381,132},{382,137} *)
```

In[ ]:=
$$\frac{600-382.}{171-130} = 5.317$$

So we see that about 380 `xzr` allocations can be performed.

Unexpectedly `MOV` from `xzr` is handled like a standard integer operation, executed in the int pipelines. We cannot see this from the graph, but the fact that the limit kicks in at ~380 (suggesting the allocation of a physical register) is indicative.
Looking at the performance counters tells us that `MOV` from `xzr` is, in fact, not eliminated.
We also see from the slop of the ramp that the rate appears to be about 6 `MOV`s/cycle, ie the standard integer execution rate.

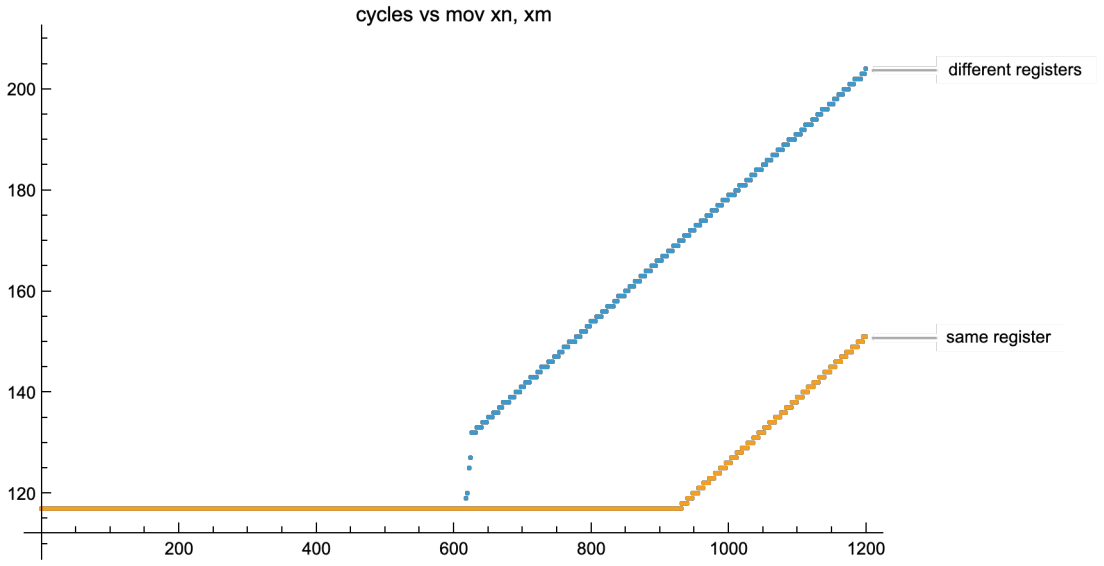
There may be something strange about the implementation here. But it may also be as simple as “don’t do that!”.
Why, in *real code*, as opposed to fake benchmark code, would you need this operation? If, in some instruction, you want a register with a value of zero, use `xzr`, it’s already there! Why waste a second register as a copy?
If for some strange reason you absolutely need a second copy of a zero’d register, Apple has provided a fast optimal path for zero’ing registers via `MOV xn, #0`. To also special-case `xzr` would require extra work in decode, and does it make sense to spend that area, power, and cycle time rather than just telling people not to do that?

As we shall see going forward, there's more apparent weirdness in how `xzr` is implemented. But always you have to remember – is this something it makes sense for me to even want to do in real code?
Rather than implementing `xzr` as just one more register which just happens to always hold zero, Apple appears to have implemented it as a *decoding pattern*, which does things like modify the instruction to which we decode, or set a flag on that instruction. And that's probably optimal in terms of performance and power, if you're willing to do that extra work…

xn

But that's not the end of the story! Let's try a non-special register, like `MOV x0, x2`.
This case reaches 620 with no glitches (as opposed to the `xzr` case above)
(Just for fun I also included the case `MOV x0, x0`. This should be treated as a `NOP`, and that's in fact what we see. There's no reason your code should ever include `MOV xn, xn` !; but if it does, they behave just like `NOP`s, using no execution resources except ROB slots, and the curve jumping at around ~2300 when the ROB slots are exhausted. In the curve below you only see the curve start rising at 880 because at that point 880 NOPs/8 per cycle exceeds the 110 cycle delay.)

Out[506]=



In[ ]:=
```
(* {616,117},{618,119},{620,120},{622,125},{624,127} *)
```

In[ ]:=
$$\frac{1200-624.}{204-127} = 7.4805$$

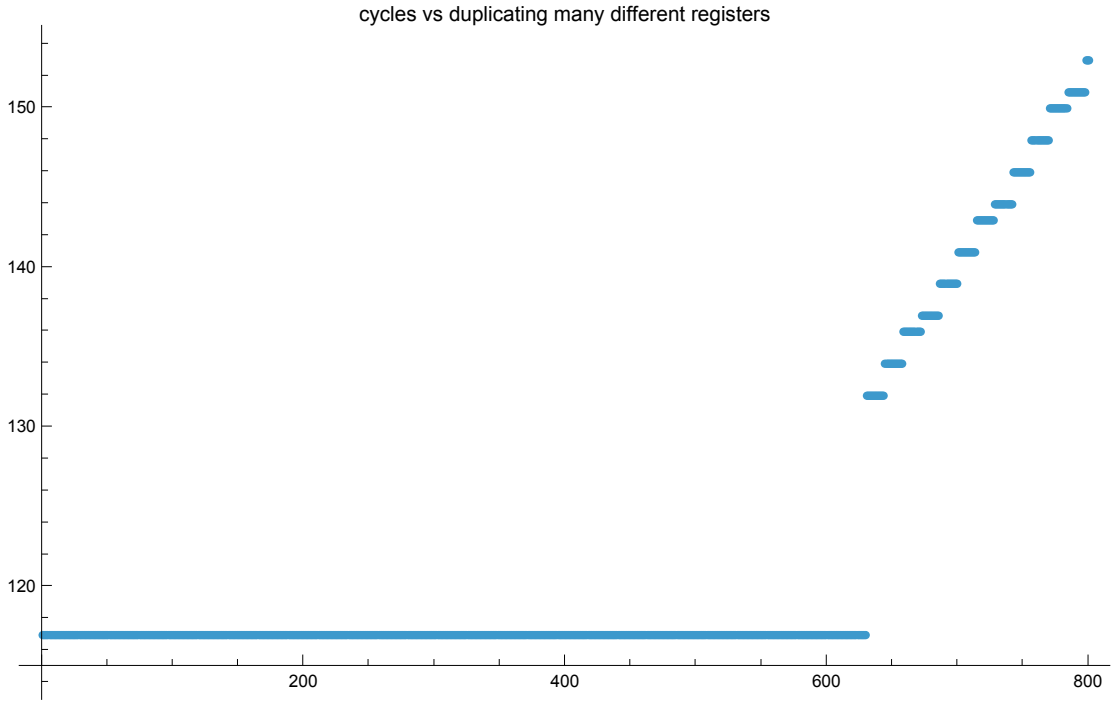
So we're seeing not a limit at the number of physical register (380, as in the previous graph) but at the number of HF slots.
Once again to see the most interesting aspect of this, you also need to look at the PMC to see that no instruction issue occurs: all eight `MOV`'s (different register case) execute purely at Rename.

We can get some confirmation of this from the graph, since the throughput of the MOV's is ~8/cycle, which is Mapper/Rename throughput, not the 6 we'd expect if the instruction had to go through the integer pipes.

The same zero-cycle (and 8-wide!) behavior holds for FP/SIMD registers (eg `MOV.16B v0, v1`).
We would also expect that, since it's the common pool of History File slots used for both integer `MOV` and FP/SIMD `MOV.16B`, when we pair the `MOV` with a `MOV.16B`, the jump will be at ~620/2=~310, and that's exactly what we see.

What further aspects of duplication can we test?
The RDA (remember, the subject of the 2012 Apple patent) can only hold a limited number of entries, each describing the number of duplicated references to a register. We can use this to test if Apple is, in fact, still using an RDA.
If we create as many duplicates as possible (ie duplicate multiple different registers) presumably we will flood it, at which point duplications will have to happen via standard execution, not by register rename.
We create a probe that consists of fifteen instructions
(`mov` x_i, x_{i-1} for i..1..29)
And replicate this (N/15) times.
(We can't do much with `x31` which is xzr/sp and special, so we are limited to 15 pairings).

Out[509]=



In[•]:=
```
(* {628,117},{629,117},{630,132},{631,132} *)
```

The result shows the expected jump (extremely sharp!) when the HF is maxed out at 630.
More important, however, is that the throughput never deviates from 8 instructions/cycle, so the

Rename magic is able to continue working with so many different duplicates established; if there is a limit to the number of duplicates, it's more than fifteen.
(Remember, what we are testing for is the presence of an RDA, a Register Duplicate Array, that can hold only a limited number of Register Duplicates, ie source physical registers that have been mapped to more than one logical register.)

There are other ways to run the test, like performing fourteen one time duplications of the form mov xi, x(i-1), to exhaust the RDA; then following with N mov x29, x28. Same results.
We can also add 15 floating point duplications, for a total pool of 30 duplicated source registers, no difference.
So, yeah, seems like no RDA is being used.

subregisters

What if we try subregisters, eg `MOV wn, wm`, or `MOV.8B vn, vm`, or `MOV dn, dm`?
On the M1 we don't get any of these fancy tricks when copying subregisters . Such copying is slightly trickier for various reasons, the most obvious of which is rules for how the high bits have to be handled; but it seems like it should be doable, (eg via an additional "zero the high bits" flag in the logical->physical map, or that same bit set in the physicalRegisterID [this concept will make more sense when we discuss immediates].)
Maybe it's not worth doing? (But it seems like it should be, especially for code that's using lot's of floats or doubles but is not vectorizable). So maybe it's on the drawing board for a later CPU?

There is something of a historical precedent:
Suppose you have a machine (like the A7) for which you expect both 64-bit and 32-bit wide registers to be common. One way you might arrange things is that your register storage behaves as two side by side banks of 32-bit wide registers, with the ability to allocate a row of two registers as a single unit (perhaps by a high bit set in the registerID). This would give you both some number of 64-bit registers while also giving you more 32-bit registers during the transition period while 32-bit software is common.
We see a (kinda sorta) version of it here (2010) https://patents.google.com/patent/US20120110305A1 *Register Renamer that Handles Multiple Register Sizes Aliased to the Same Storage Locations*, dealing with the old ARMv7 way of aliasing floats, doubles and neon registers. Clearly this is obsolete; but it deals with a slightly more complex version of the problem.
Likewise (2012) https://patents.google.com/patent/US9317285B2 *Instruction set architecture mode dependent sub-size access of register with associated status indication* discusses how power can be saved when reading or writing a 32b (W) register, and the use of an extra bit in the logical to physical mapping table to note that the logical user is utilizing the 32b subset of the referenced 64b physical register.

Generically, it seems like there is scope here for potentially massive register allocation improvement.
Imagine instead of two register files for int (each 64b wide) and SIMD (each 128b wide) we have a single

register pool consisting of rows of four 32b units. Register allocation would consist of providing (appropriately aligned) a full row (SIMD register), a half row (double or x register) or quarter row (single or w register). Conceptually as raw storage this is feasible, one just has to be a little careful about how the free lists are treated; and a few high bits associated with each physical registerID clarifying the width of the item that is being extracted.
Where things become slightly tricky is in register aliasing (eg you write a signed value to a w register, then read that as an x register).

In the integer case this is fairly easy to handle; you just need to give the register file some logic so that when values flow in or out they are appropriately shifted and sign-extended, and you need an extra bit associated with each 32-bits to clarify how the sign extension should be performed on the outflow.
I haven't bothered to look at the precise neon SIMD/DOUBLE/FLOAT aliasing rules to know if there's a difficult problem on that side. But the vision is tantalizing! A unified register pool means that instead of being limited by just the 400 or so integer registers, code that used no fp registers would have many more int registers available, and code that used many w registers would have a pool of even more registers available.
(Of course to get full value out of this, the size of the history file needs to be increased, but that doesn't seem like a problematic structure to grow.)

It's looks like this idea of allowing a wide register to act as multiple shorter registers has been reused, yet again, in Graviton 3, as discussed in https://chipsandcheese.com/2022/05/29/graviton-3-first-impressions/ (look for the Out of Order Structure Sizes section)...

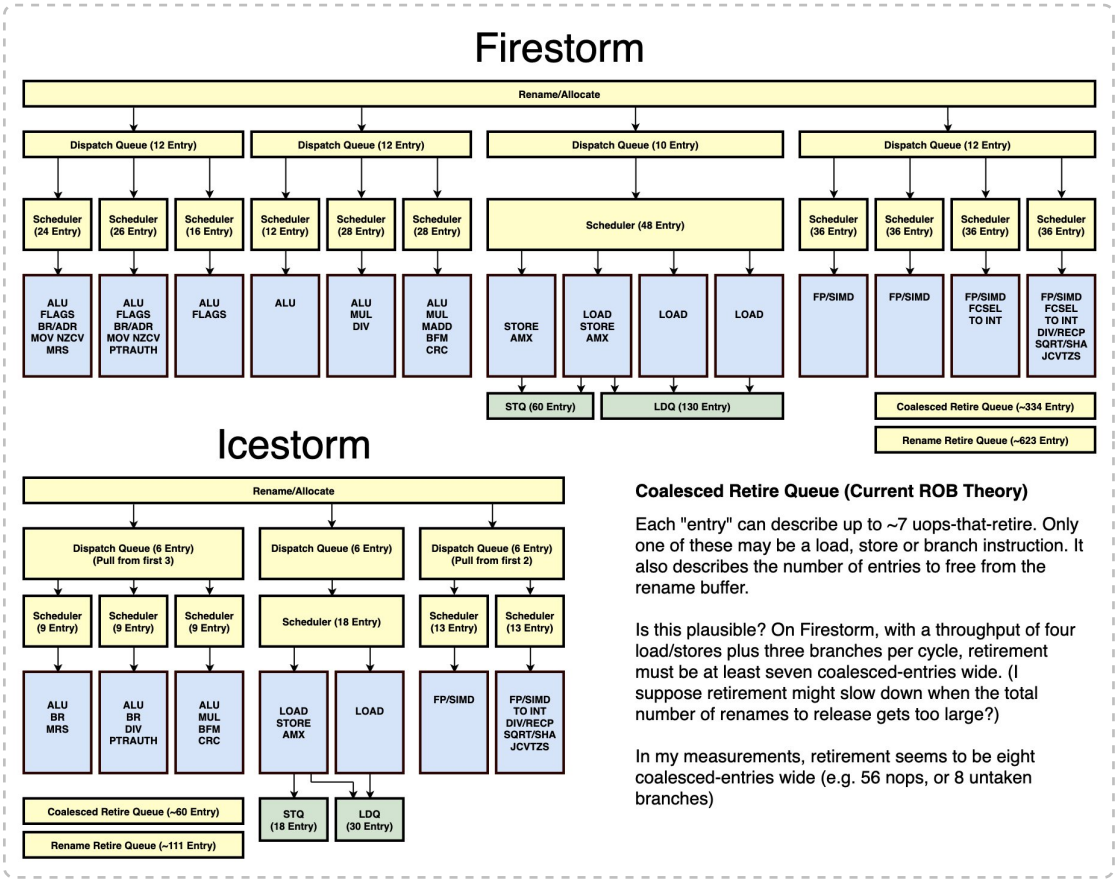
Instruction Scheduling

Going forward, you may want to keep in mind this diagram by Dougall. The ROB stuff is provisional (and I think better explained below) but the basic ideas and numbers match my investigations.
And what he calls Dispatch Queues should, I suspect, be thought of as Buffers. Buffers are cheaper than Queues because they don't make the same ordering promises. I suspect the Apple Dispatch Buffers can lose ordering (in cases where it doesn't matter much, namely when the Scheduler Queues are so filled up that the Dispatch Buffers are more than just pass-through).

Note that this diagram should be treated as provisional. In particular (eventually both of these will be explained/justified below)
- the monolithic 48 entry Load/Store Scheduling Queue is in fact 4 separate 12 entry queues, just like the other cases
- the other entries apart from load/store are probably about 2x too large (that is, the size of say the two MUL queues is probably ~13 each), likewise the FP queues probably 18 each. Essentially (as will be explained below) while each queue feeds a primary execution unit, the queues are paired, and if one

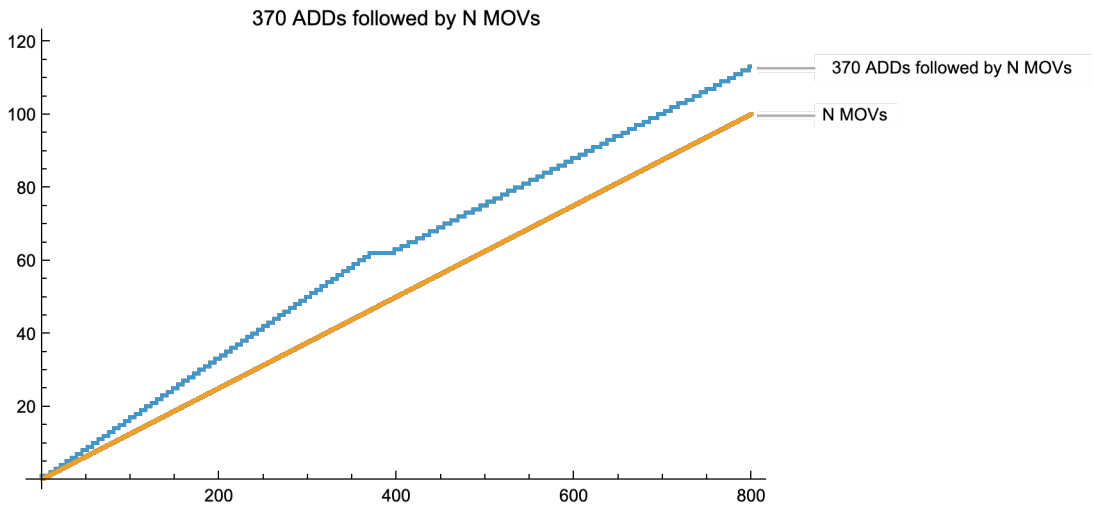
queue does not have a runnable instruction, it can accept the “second choice” runnable instruction from the companion queue; thus for many testing purposes it will look like the queue feeding a particular execution unit is twice as large as it is, because the “amount of scheduling space” is the sum of the two queue sizes.
xxx I have not yet had time to probe the full details of this pairing beyond having seen it in action in the case of load store.
Given the numbers, an obvious guess is that on the integer side
- the two MUL queues are paired (each 13 in size)
- two two BR queues are paired (maybe one of size 12, one of size 14, or both of size 13)
- the ALU/FLAGS queue and the ALU queue stand alone and are not paired.
On the load/store side, the store queue is paired with the ambidextrous (store+load) queue.
I would guess that the two FPCSEL queues are paired, but have not tested that.

It makes sense to pair together queues as similar as possible (except where prevented by the fact of the two different integer Dispatch Buffers) because that gives you the maximum amount of sharing flexibility; whichever queue you dump an instruction in, if the other queue can’t find a runnable instruction, two instructions can still be scheduled. This doesn’t work as well when one type of instruction can only go into one execution unit – which is the case for the store vs ambidextrous unit, which is how I discovered the phenomenon.



Consider now the following experiment. Suppose we use up almost all the int physical registers (via ADDs) then try to perform some MOV's. What we are interested in is the transition between these two types of instruction, one of which is executed in the OoO pipeline, one of which is executed in Rename.

Out[511]=



The low-N and high-N parts of this curve are easy enough to understand. We have no delay block. The first segment of the graph corresponds to the 370 ADDs, processed at a slope of 6/cycle; these are followed by the MOVs, processed at 8/cycle, and the difference in slope is visible, compared to the gold curve which is just MOVs without the initial ADDs.

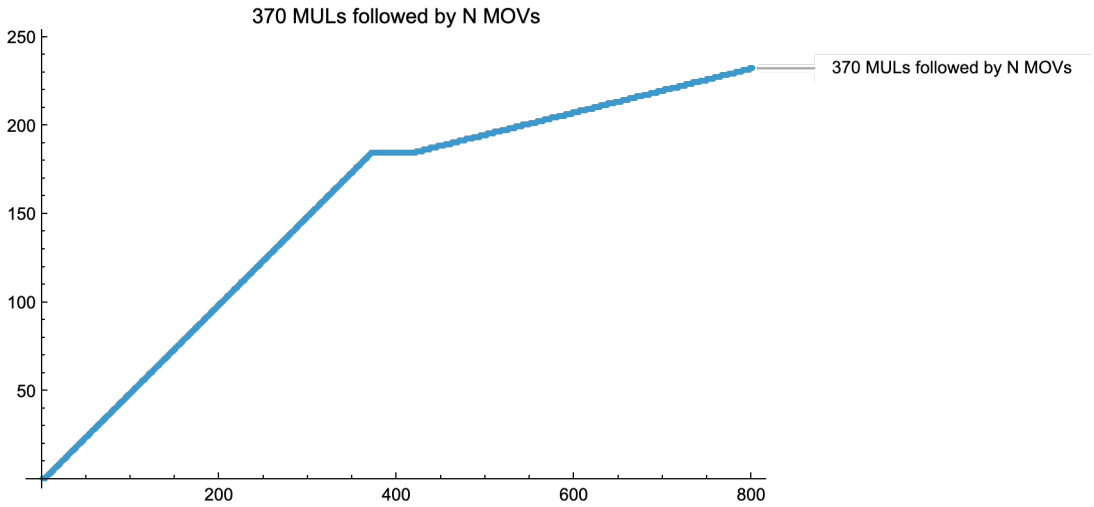
More significant for us is the flat region from 370 to about 397
What's happening there? This tells us something about the int instruction scheduling.
To simplify (this will all make sense after you read below then come back here)
- Rename can generate 8 instructions (8 ADD's) per cycle.
- These can be moved (8 per cycle) into a Dispatch Buffer
- But only 6 instructions per cycle can leave the (integer) Dispatch Buffer
- And those 6 instructions can immediately execute, so they do not fill up the Scheduling Queues
- Hence every cycle the Dispatch Buffer gains two instructions, until it is full
- The integer Dispatch Buffer has a capacity of ~24 instructions, so it soon fills up.
- Once we add a few MOV's to the end of the instructions to be executed, at first those MOV's take zero cycles to execute
+ more precisely they execute in Rename at the same time as the ADD's that were in the Dispatch Buffer are drained
+ so we get 4 free cycles (24 buffer size/6 ADDs per cycle) of NOPs that will not add to our cycle count
+ and 4 cycles of NOPs is 32 free NOPs before the additional NOPs start to increase the cycles/loop iteration

This is the basic idea with the one final tweak being that integer code actually uses two buffers, each of size 12.

(BTW there's no significance to the use of 370 ADDs, you just need "enough" where enough is more than 12 cycles*8/cycle=96)

If you understand that, then you should understand this:

Out[513]=



This is very much the same idea, only using `MUL x0, x5, x5` as the probe. What should we expect?
We get an initial slope of two instructions per cycle (only two execution units support multiply, but they are fully pipelined).
The flat region runs for {370,185} to {420,185}, so we get about 50 free `NOP`s. Does this make sense? Yes!
The Dispatch Buffer that feeds the two `MUL` units has a capacity of 12, so will drain in 6 cycles. 6 cycles allows 48 `MOV`s to execute in Rename.
So we validate both our understanding and that there are two Dispatch Buffers feeding the six integer Scheduling Queues, as in the diagram.

M1's Implementation of instruction scheduling

The machine is 8-wide all the way though to Rename; and throws out 8 ADD instructions per cycle into the Scheduling Pool. By Scheduling Pool we're being deliberately vague, but mean the idea of some pool of instructions between Rename and Execute from which instructions are Scheduled for Execution in an out-of-order manner, as their dependencies are resolved.
One's natural assumption, the usual model, is a single level scheduling pool (called something like a Scheduling Queue) but it's time to revise that.

Recall that the scheduling queue holds instructions that can't yet execute because at least one of their dependencies is not yet available; for example the instruction may be waiting on a load, or it may be one of the final instructions in a chain of ten or fifteen sequentially dependent instructions.
We want the scheduling queue to be as large as possible, as this allows the CPU every cycle to look over more instructions to find something to execute. Or to put it differently, the larger the scheduling queue the more independent chains of sequentially dependent instructions you can run in parallel, so the

more ILP you can extract.

separate scheduling queues

However scheduling queues are phenomenally expensive in power! (Every cycle you need to scan the entire collection of instructions to check which have become runnable, then of those instructions choose the oldest for execution.)
The most obvious solution, at least as a first step, is to at least split the single scheduling queue into multiple queues. This means that you sometimes waste some queue space (if you're doing only integer computation, all that space devoted to the fp scheduling queue is wasted) but it allows the sum of all the queues to be larger for the same, or lower, power.

Apple takes this idea to an extreme by adopting a queue in front of each execution unit, rather than, perhaps, a common queue for integer, a separate common queue for fp, and a third separate common scheduling queue for load/store. But a problem with these multiple queues now is that you want to ensure that they are load-balanced. Obviously some integer instructions can only go to one or two pipelines (there are only two branch execution units, only one integer divide unit), while other integer instructions like ADD can go to any of the six integer execution unit queues. You can't do anything about the specific pipeline(s) that some instruction are forced to go down. But you can balance other instructions as much as possible around those restrictions.

dispatch buffers

Apple's solution to this (which has accumulated other benefits over time) is to install a Dispatch Buffer, a secondary instruction storage pool, between Rename and the Scheduling Queues.
The initial goal of this Buffer was load balancing -- after instructions with strict requirements are sent to the appropriate queues, the remaining instructions are sent out according to which queues are least full, so that overall queue fullness remains even across all queues.

Of course the the Dispatch buffer also acts as a secondary storage for instructions, a way to put them somewhere and allow Decode and Rename to keep going, even if all the integer queues are full and can accept no more instructions. A Dispatch buffer is cheap because it doesn't have the ordering requirements of a Scheduling queue (which instructions are oldest?) and doesn't have to be scanned every cycle to figure out which instructions have become runnable. AMD uses a Dispatch Buffer for this reason, for FP/SIMD instructions. You can see the AMD case here: https://en.wikichip.org/wiki/amd/microarchitectures/zen_2 (Look for the big diagram, then in the FP section for the Non-Scheduling Queue)

However a Dispatch Buffer is also imperfect for these same reasons -- if Scheduling Queues all fill up, and then the Dispatch Buffer, so that this Buffer contains more than just a few instructions, then, once execution starts flowing again, it's somewhat random which instructions will be the first to be moved from the Dispatch Buffer to the Scheduling Queues and on to execution; and choosing the wrong instructions from the Dispatch Buffer might delay the execution of critical instructions for a few cycles.

So a Dispatch Buffer is not absolutely free storage that should grow as large as we like; we want it to be large, but we don't want to have to pay the side effects (poor scheduling) of it being large! It needs to be kept to a limited size so that, even if there is some degree of sub-optimal issuing of instructions to the Scheduling queues, there's also limited delay that can occur.

There's one other, less obvious aspect to the Dispatch Buffers: each one can accept up to 8 instructions from Rename. Assume you have a long run of integer instructions. Each of the six integer queues can only accept one instruction per cycle, so in the absence of a buffer, Rename would only be able to clear 6 instructions per cycle. The machine wouldn't halt, but the front-end would be forced to run 6-wide rather than 8-wide. It's even worse if there's a long run of FP or load/store instructions where only 4 instructions could be enqueued per cycle.
Each Dispatch Buffer can accept a full eight instructions per cycle, thus it can clear Rename for the next eight instructions, and this can continue for at least a few cycles, though obviously not indefinitely. As I said about designing not just for the mean of a program, but for the standard deviation...

evolution of Apple's scheduling queue

The Scheduling Queue is another case where we can see the evolution of Apple's thinking over time. (Recall what I said, that Apple use the term Reservation Station for what I'm calling Scheduling Queues.)

2013 (two level scheduling – dispatch + scheduling queues)

We start with (2013) https://patents.google.com/patent/US9336003B2 *Multi-level dispatch for a super-scalar processor*, which describes a basic two level scheme, where the main point is load-balancing. BUT with the secondary point that the buffers are capable of accepting up to eight instructions (whereas queues only need to accept one instruction per cycle). These details matter! Most hardware structures grow quadratically in area as you increase their inflow or outflow. So there's a real win if you can dump all this quadratic complexity (eight-wide inflow) onto a very simple structure, a buffer with no ordering or other properties, while keeping the queues as 1-in, 1-out per cycle.

2015 (non-shifting queue, based on relative age field)

This is somewhat augmented with (2015) https://patents.google.com/patent/US20170024205A1 *Non-shifting reservation station*, a non-shifting Scheduling Queue which I will describe soon, and the technical companion patent (2016) https://patents.google.com/patent/US10120690B1 *Reservation station early age indicator generation*.
To understand this 2015 patent, recall that the usual way a Scheduling Queue is implemented is exactly as a physical queue. Using a physical queue means that temporal ordering is trivially preserved -- it's easy to see which instructions are oldest (at the head of the queue) vs youngest (at the tail of the queue). But a queue uses a lot of power because it has to be "collapsing"; that is, every time an instruction is scheduled from the middle of the queue, you have to move all the other values along the queue to remove that vacancy and open up space at the end of the queue.
Now you can imagine various possible ways to reduces the cost (leave the holes as long as possible, ie

only collapse when you have to; maybe use indirection, like a linked list rather than a linear sequence of storage; ...). The Apple solution is to use an age matrix, which is, just like aspects of the 2019 History File patent, a structure that is large (by the standards of tech many years ago -- now transistors are cheap!) but low power and has low complexity wiring. The age matrix tracks the temporal ordering of the instructions while not requiring them to continually be moved from one slot to another.
I urge you to look at this one; it's fairly easy to understand but so neat!
Suppose a particular scheduling queue has 30 entries. Then associated with each entry is a 30 bit vector, each bit of which represents one of the queue entries. If a bit is set to 1, that means this queue entry is older than the entry represented by the bit. It's a fairly simple simple algorithm when to flip bits to one then back to zero, and the flipping doesn't have to happen very often (ie low energy).
(A similar solution is used for the Load/Store Queue as we'll discuss when we get to Load/Store.)

2016 (earlier testing of relatives ages, allowing larger queues)

Now we have this 30 bit vector indicating age of each slot relative to others lots. OK, *in principle* we understand how it encodes the age info, but what do you actually do with that info? That's what the 2016 patent covers, somewhat.
One can imagine at least two ways of using these age vectors. One idea is you perform a pop count on each vector, then find the slot with the highest popcount. Conceptually easy, but it assumes that we have fast cheap circuits that perform popcount, and perform "find largest in a set of 30 integers"...
Alternatively we can use good old divide-and-conquer. Compare adjacent pairs of entries to find which is older (which is a simple test of the bits at the two appropriate positions in the two bitvectors), and replicate that 5 times (binary log of 2). The patent points out that you can do at least the first round of this in an earlier cycle (relative age won't change, but you may have to test all slots, you can't mask out slots that aren't executable); and by doing one or more of these relative age comparisons in an earlier cycle you have to do fewer (easier to meet cycle time) in the next cycle.
One interesting aspect of this recursive structure is that really what you are trying to calculate is not actually "oldest runnable instruction" it is "best choice for runnable instruction". You could replace/augment the age bitvector with something encoding criticality information while still using this same tournament "compare successive pairs" structure to get a criticality-based scheduler; perhaps via something as simple (at least at first) as adding a single extra "critical" bit to the relative age bitvector that overrides age if it's set for only one slot, otherwise it's ignored and age wins. That allows us to start with a basic criticality predictor that generates just a single yes/no prediction.

2017 (pairing scheduling queues and issuing from either)

Next we get (2017) https://patents.google.com/patent/US10983799B1 *Selection of instructions to issue in a processor*.

Consider on the one hand Intel's scheduling solution which is (more or less) to use a single large age-ordered queue.
In theory this gives you total information – you can see all the instructions, all their ages, and can

schedule the oldest ready instructions. That may sound ideal but
- you pay a substantial cost for that in energy, in inability to scale, and in lack of time to implement various smarts
- you're optimizing for something (oldest ready instructions) that is not even *really* what you want (see my mention a few pages below of *criticality*).

The Apple solution runs in the opposite direction, starting with the simplest solution that will work (multiple queues, each independently scheduling only their contents). Clearly such a solution can result in multiple types of imbalance, and so we see every year simple tweaks that substantially improves the balancing while not costing much in area or energy. So
the Dispatch Buffer tries to keep a collection of queues reasonably balanced in their fullness, and
- the two patents described above track relative age (so that, approximately, albeit not perfectly, oldest ready instructions schedule first).

However we could still have an imbalance where queue A has two instructions ready to execute while queue B has no instructions ready to execute. That's not ideal! Can we do anything easy to fix this?
Consider how the age mechanism I described above works. Essentially we have multiple rounds of "find the better instruction [by runnability and age] from comparing these two instructions". Each such round take in two candidates and outputs one candidate.
So suppose that at the very last round, we also offer the second choice candidate from queue A to queue B (and vice versa).
Queue B will accept the second choice candidate as better than what it has (which is nothing!) and will schedule the second choice candidate from queue A to execution unit B!
Obviously one needs a few interlocks and tests and so on to ensure that you don't land up scheduling a Divide instruction to a queue without a divider; but it's one more easy'ish tweak to the pre-existing system to improve load balancing and throughput a little more.

2017 (splitting scheduling queue into a fast front half and a slower back half; not implemented? or only for FP?)

Finally we get (2017) https://patents.google.com/patent/US10452434B1 *Hierarchical reservation station*.
Honestly I don't get how this one fits into the general pattern of Scheduling Queues, or any of our test results. Maybe it represents a set of ideas that were ultimately discarded? Or that are only used for FP?

The problem this claims to be solving is that age/readiness comparing the full set of instructions in the Scheduling Queue may not fit within cycle time. The *early age* (2016) scheme was one of dealing with that. But this suggests a different scheme.
The idea is that we split the Scheduling Queue into two parts, primary and secondary. We run the scheduling machinery on both halves.
- The results for the primary queue are some number of instructions to be issued (oldest ready instruc-

tions);
- the results for the secondary queue are both
+ some number of instructions that could be executed (oldest ready instructions) and
+ a second array, of *oldest instructions*, to be moved to primary on the issue of the newly scheduled instructions from primary.

Of course I don't know, but the logic looks to me like the queue they had in mind
- was split into primary and secondary,
- initially all secondary did was figure out which instruction each cycle to move to primary;
- then the inventors realized that by adding a little logic, they could also issue instructions from secondary if primary could not supply enough instructions.

But there's a lot I don't get about how this fits into the 2015/2016 scheme! We don't want to be moving instructions around (that's the point of the aging bits) but this scheme seems to require moving instructions from primary to secondary.
Could the primary and secondary queues be interleaved, or side by side, so that the distance moved (from one slot to the next) is not very large?

Perhaps this scheme is specific to FP?
Look at the sizes of the different Scheduling Queues. My rough heuristic for the size of a Scheduling Queue is that it wants to be able to hold enough instructions so that something can always be found to Issue from that Queue. If the Queue is too small, then an instruction that could be Issued is unable to Issue because it's hiding up in Dispatch or even in Rename, invisible to the Scheduler. But the Queue costs power, and if it too large, those extra instructions are always just sitting in the second half of the Queue are doing nothing for performance because an instruction ready for Issue can always be found in the first half.
This heuristic suggests that the optimal length of a queue will vary depending on both
- how much (on average) instructions tend to depend on the previous instruction (chained dependencies mean the later instruction has to wait in the queue) and
- on the average latency of these instructions being waited upon.

This seems to match the pattern we see for Apple.
In particular for FP/SIMD both long dependency chains and long instruction latencies are common, hence we see each Scheduler Queue as a hefty 18 entries in size. So maybe this patent is more or less specific to FP, splitting these 18 entry queues into two 9-entry halves, or a 12 entry front-section and 6 entry back section, that are separately scheduled as per the patent?

Another way to try to make sense of this patent is to skip over the details it gives (which are very clear that it is the scheduling *queue* that is split in two) and assume the patent operates at a more abstract level.
The basic model is that a Scheduling Queue performs two tasks
- figure out which instructions have become runnable (all their dependencies are satisfied)

- figure out the "best" of these instructions to execute. (Ideally this would be the most critical instruction; right now it tends to be the oldest instruction).
If a given Queue cannot find an instruction that's runnable then, as described, we try to grab one from the paired queue. Suppose that also fails, what else can we do?
Suppose we add logic to the Dispatch Buffer to figure out which instructions have become runnable. (We could even make this logic somewhat cheap in that it only works for truly runnable instructions, with no attempt at speculative scheduling based on "it looks like this instruction will become runnable next cycle because its dependency takes two cycles and started one cycle ago".) We could also abandon any attempt at finding the "best" of the runnable instructions in the Dispatch Buffer, we just pick a random one.
This, at the cost of a fairly cheap (I think) addition to the Dispatch Buffer logic gives us a third possible source for an instruction if neither the primary queue nor the paired queue has a runnable instruction available. Even if the instruction that executes is not optimal or critical, using any particular execution slot going vacant is better than not using it!
This model is somewhat different from the exact words of the patent, but matches the spirit of the patent. So???

dependency tracking

register-based dependencies

The other part of handling Scheduling is tracking instruction dependencies. The traditional way to do this is through physical registers: `ADD x0, x1, x2` depends on x1 and x2. We map those to p11 and p12, and route the instruction to scheduling as `ADD p10, p11, p12.` Then Scheduling won't consider the instruction for Issue until P11 and p12 both become valid.

This physical register based scheme is an obvious grandchild (after various improvements and tweaks along the way) of the original Tomasulo scheme from the 1960s. You may enjoy reading (1987) https://www.cs.auckland.ac.nz/compsci703s1c/resources/Sohi.pdf *Instruction Issue Logic for High Performance Interruptable Pipelined Processors*; an *old* paper which is interesting precisely because it's written at such an early time, talking about the precise mechanics of (one early version of) how you do this, before the details became routinized as just the words Rename and Scheduling.

A similarly old paper (2000) which at least introduces many of the ideas I've been discussing about how to handle register renaming is
(2000) https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.3852&rep=rep1&type=pdf *The Design Space of Register Renaming Techniques*.
These historical papers are helpful in two ways
• they explain the problem that needs to be solved and
• reading them you helps you understand the terminology and mindset of comments you find on the internet.

However it's important to realize just how long ago those papers were, in a Moore's Law world. What was optimal and state of the art in 2000 may make zero sense in 2020.

This register-based scheme works, but isn't the only way of doing things. The biggest problem is that, for today's version of Tomasulo-type schemes to work, registers have to be *early-allocated*, ie have to be allocated at an in-order stage of the pipepline (traditionally in the Rename stage). And the problem with early allocation is that a very expensive and limited resource (a physical register) is now locked up from Rename until the instruction Retires. But if you think about it, the register is only required from the point at which the instruction completes Execution (to hold the instruction result); all the time before Execution (while the instruction is waiting to be scheduled) that physical register is achieving nothing useful; its only purpose is for its registerID to establish instruction dependencies.

So: can we *late-allocate* a register, at the point of instruction execution (in fact completion, when we want to write out the result!) rather than at Rename?
Yes – if we have some alterative way of tracking instruction dependencies...
One version of this, an extension of Tomasulo, uses *virtual registers* (1999) https://upcommons.upc.edu/bitstream/handle/2117/101362/00809456.pdf *Delaying Physical Register Allocation Through Virtual-Physical Registers*.

The flip side of late register allocation is *early register release*, ie releasing a physical register as soon as it's no longer required (because all potential users have executed) rather than waiting until Retire. The details of how this could be done (not optimally, but picking up some of the value) appear in the Haithim Akkary paper referred to earlier, describing Checkpoints and other instruction for very large OoO processors.

As far as I know no commercial CPU (even Apple) currently uses register late allocation or aggressive early release. However IBM (in recent POWER) does use late-allocation of load/store queue slots.
Apple definitely uses at least one of late-allocate or early-release for load/store queue slots, probably early release. (A 2019 patent described below suggests a scheme with traditional allocation of LSQ slots, but early release.)

instruction-based dependencies

An alternative is, rather than saying that `ADD x0, x1, x2` depends on (the physical registers mapping to) x1 and x2; to say that the `ADD` depends on the two previous instructions, call them I27 and I35, that will produce x1 and x2 respectively.
This could be implemented by, eg, having I27 and I35 in fact refer to the ROB slots of the relevant instructions, and essentially instead of marking a physical register as valid at Execution completion, the ROB slot is marked as valid. Let's call these numbers associated with previous instructions SCH#'s. Thus associated with each instruction in scheduling is a Dependency Vector, which describes the instructions that have to complete before this instruction can execute. (This may sound more sophisticated than virtual registers, but in fact it harkens back to the mid-1990s when, as I've already described, something like the Pentium III attached a physical register to each ROB slot, and so marking

a physical register as valid was the same thing as marking a ROB slot as valid.)

The Apple Scheduler patent is (2016) https://patents.google.com/patent/US11036514B1 *Scheduler entries storing dependency index(es) for index-based wakeup*.
To understand it, we need some history.

The original 1960s Tomasulo renaming stuff, already mentioned, is from an utterly different world in terms of transistor budgets and the relative speed of different components. (You will see, in the history, an explanation of some of the SCH# naming conventions in Apple patents, and will see why they refer in the patent to possible ways of doing things that clearly seem dumb!) For our purposes the aspects of Tomasulo that matter are
- Registers were renamed via a mapping table
- A destination register was NOT allocated right away.
+ the scarce resource was a slot in the Reservation Station and physical registers.
- When a slot in a reservation station became available, a waiting op was moved to that slot. Call the address of that slot a "tag"
+ it should be clear that this tag conceptually functions as an ID for an operand for all subsequent instructions. It acts *like* a physical register ID.
- When an instruction in the reservation station is able to execute (dependencies satisfied) a destination register is allocated.
+ that register is marked with the tag we mentioned earlier.
+ so at this point we have a destination register marked with the tag of an instruction, and possible dependent instructions marked with such a tag
- Once the instruction has executed, it broadcasts its result on the bypass bus along with the tag.
+ dependent instructions in the reservation station look for the matching tag and grab the result (their operand value) from the bypass bus.
+ likewise the register file looks for the matching tag, and stores the result in that physical register.

Compare all this to the model you should have of renaming today. We
- Provide the destination register at Rename, not much later at Issue. This means the destination register is idle for those cycles between Rename and Issue (as we have already discussed) but it means much of this tagging nonsense goes away.
- Instead of a tag (allocated when the instruction moves into the Scheduling Queue) dependency can be built upon the destination register's physical registerID.
- An instruction has as unresolved dependencies, not a tag (the ID of an instruction that has yet to execute, attached to the result of the execution on the bus) but a physical registerID on an invalid register. When an instruction executes, it dumps the result (plus the destination physical registerID) on the bypass bus. The destinationID tells the bus where to place the value in the register file (so no tag comparison needed there), but *all* waiting instructions in *all* scheduling queues still have to look at *all* the physical registerIDs flowing over the bypass bus to see if one of those registerIDs matches a dependency so that they can now be marked ready to execute.

You can see that the two procedures are very similar, just with the role of the tag being replaced by the destination physical registerID. But you can also see that we have a huge scaling problem here, trying to compare all the possible physicalRegisterIDs generated in a cycle (there are 14 units that could all generate a result in one cycle on the M1, and some of them [think load pair, or an instruction like ADDS that also sets flags, could generate two results) with the physical registerIDs that are the dependencies for every instruction in every queue. We need to rethink the problem based on today's concerns, not the concerns of the late 60's.

For now assume that the earlier instruction on which we're dependent is identified by ROB slot. Given ~2300 ROB slots, that would mean ~2300 instructions on which we could depend. But think about what we are trying to do.
Suppose instruction i0 depends on architectural register x1, which is mapped to physical register p2.
At the point where i0 enters the scheduling queue, p2 can either be valid or invalid.
If p2 is valid, then i0 has a dependency on p2, but it's not an "op-dependency". As far as *scheduling* is concerned, i0 can execute right away, because p2 is known.
On the other hand, if p2 is invalid, then there is an op-dependency on p2, and i0 cannot execute until the producer of p2 has executed. Thus for *scheduling purposes* we want to know what *pending* instructions i0 is dependent on.
This simplifies the problem; we don't actually care about every instruction in the ROB, we only care about instructions that are in the scheduling queues and haven't yet executed; an instruction that executed before i0 even entered the scheduling queue has stored its result in p2 and is not relevant for scheduling.
So the number of instructions we need to track for scheduling is really the number of instructions that are waiting to execute ahead of this one, which is a maximum of ~400 (worst case total occupancy of all the scheduling queues and dispatch buffers).

How are SCH#'s implemented? I think the original implementation (when the SCH# terminology was born) had no Dispatch Buffers, and had the Mapper allocate an instruction slot in a particular scheduling queue. Thus the SCH# could literally refer to the slot of the queue in which the instruction was placed.
But the current implementation is, I would guess, much like register allocation. Imagine a pool of 400 numbers, 0..399. Each is marked busy or not. Map allocates each instruction a free index from this list, marking the index as busy; at instruction is removed from the scheduling queue, that same index reverts to free, and that index is used to broadcast "instruction with this index has generated a result". This gives every instruction a unique persistent ID from the beginning to the end of its scheduling career, from a set that's as large as required but no larger.

bitvector-based dependencies

Now how is dependency implemented? You might imagine something like an array for each instruction that holds a maximum of maybe three SCH#s. That seems obvious, but If we trust the patents, the

obvious answer is wrong.
It appears to be that each instruction has an associated dependency bitvector, with a bit set for each dependent SCH#. This means a dependency vector is ~200 bits long, though only a few (perhaps two or three) of those bits are set.
There are a *lot* of patents that suggest this bitvector implementation, eg (2006) https://patents.google.-com/patent/US7647518B2 *Replay reduction for power saving*, which says "The dependency vector may comprise a bit for each entry in the buffer 42" (buffer 42 is essentially the scheduling queues).
The most detailed explanation seems to be in this patent (which otherwise covers very different material, but has a full explanation in the section covering Figure 2): (2014) https://patents.google.com/-patent/US20150207496A1 *Latch circuit with dual-ended write*.

Why run your dependency based on instruction number rather than the Tomasulo physical register-based scheme?
One possible answer is that it allows you to have dependencies beyond just register-based.
We will discuss more details around load/store issues and, in particular, load-store dependency, below. But the issue that matters right now is easily understood:
suppose you have a load that depends on an earlier store (ie the load load's from an address to which a store stored in the recent past). Clearly, for correctness, the load must execute after the store has executed, otherwise the load will load stale data from the cache.
There are a variety of aspects to exactly how this is done, but summarizing it all
- there is a predictor table (the LSDP -- Load Store Dependency Predictor) that records these dependency pairs as they are encountered.
- interestingly, this table is associated not with the Load/Store unit but with the Mapper unit (the unit that figures out inter-instruction dependencies in a group, and the physical registers upon which an instruction [encoded as logical registers] depends. Why is this?
- because the LSDP implementation is handled as just one more dependency of the load! Just like the load might depend on instruction M to produce one of the registers that goes into the address calculation, it can also depend on instruction N (which will perform the store) as an instruction that has to be completed before the load can execute.
(I have to say I find this a genuinely cool idea, one that I have not seen before and did not think of for myself.)
Details of this (along with many other details of how the LSDP detects loads that depend on stores, and how those load/store pairs are aged out of the LSDP table) can be found in (2012) https://patents.-google.com/patent/US20130298127A1 *Load-store dependency predictor content management*.

This requires being able to add an additional dependency to the Load – but that is not a problem given the dependency bitvector, it's just one more bit flip!
This idea can be (and is) generalized by Apple in *many* different directions.
- For example another aspect of Loads (to be explained later, for now we just give the idea) is Replay. A Load may fail temporarily because a resources it requires is unavailable (eg a TLB lookup or an L1D cache lookup). This can be resolved simply by having the load retry every few cycles, but that wastes

energy and resources. For a range of Apple devices (from the A6 to the A10; as of probably the A11 an even more sophisticated scheme was implemented) Replay was handled by adding to the bitvector a synthetic dependency associated with the event being waited upon.
- Another version of these synthetic dependencies is used by a scheme that performs thread context switching in hardware and in the background as other instructions execute.
- Yet another version is used in Apple's value prediction scheme.

The dependency bitvector makes it easy
- to add new dependencies to any instruction.
- to define new types of dependencies. These merely have to grab a free SCH# from the pool (which we may want to expand to 450 or so) and broadcast that SCH# at the point when the synthetic dependency (eg service this particular TLB miss) has been completed.
None of this flexibility would be possible with a fixed sized dependency array of say a maximum of three dependencies per instruction.

implementing the scheduler

Finally how do we use all this? Remember we started with (2016) https://patents.google.com/patent/US11036514B1 *Scheduler entries storing dependency index(es) for index-based wakeup.*

Step one is we realize that the problem splits into three conceptually distinct tables.
Consider a Scheduling queue with 30 entries. Treat each entry as actually composed of three fields that are very different, so it's like we have three tables each thirty entries in size.
First table holds the operation, its physical destination registerID, and its source operands. These could be immediates, they could be physical registerIDs. Point is, this first table tells you everything you need to execute the instruction, but nothing about how to schedule it. It does not tell you when the instruction can/should run, just what you need once you've decide to run the instruction.

Second table holds the dependency information.
So this second table is going to be a huge bit matrix. 30 rows high, around 400 columns wide. We've already discussed the dependency bitvector. So for any given row (a pending instruction) there will be just a few bits set. Maybe bit 7 of row 4 is set , meaning
- the instruction in Scheduling Queue slot 4 is not yet ready to execute. It depends on some physical register that hasn't yet been filled, and the instruction that will fill that register is the instruction that has been given SCH# index 7.

Third table ties these two together. It holds the SCH# of each instruction, and the relative age data (discussed in the *Non-shifting reservation station* patent above).
So now think what happens conceptually during execution. The bypass bus still exists, still carrying all the generated results along with their destination registerIDs.
But there is a parallel bus that exists about 400 bits wide. Each time an instruction issues, from all the

different execution units, each unit sets one of those bits active depending on the SCH# of the instruction that they are issuing. So that bus is carrying say 10 or so hot bits representing all the instructions that will soon be generating a result. (It doesn't matter whether they generate two results or one; all that matters is that a younger instruction depends on this instruction to have been completed before it can execute.)
That 400 wide bitvector can be grabbed by every scheduling queue, copied to the top of the scheduling bit matrix, and the hot bits allowed to "flow" down their columns. Every 1 that they encounter in the column gets flipped to a 0. (The 1 meant this row's instruction was waiting for this column's instruction to have been completed.) Then every row can be or'd to see if it's all zero; if so it's marked as ready and can be scheduled as soon as other details (like age relative to other instructions) allow.

There are plenty of details I'm omitting here.
One obvious detail is, as we have mentioned, there are additional dependencies that can be added to the bitvector, but those trivially fit into the model.
Trickier, and I ignored it to get the idea across, is what to do about instructions that take longer than one cycle to execute – I was sloppy about the timing details, but conceptually, of course, what you want is for each execution unit to place its "this instruction is about to complete" bit on the 400-wide common bus in the cycle just before the instruction will complete.

In one sense this is no different from the earlier tag and physical registerID schemes
- 14 units are each broadcasting that they have completed an instruction, and
- the identifier for that instruction has to be tested against every waiting instruction.
But we have dramatically regularized the problem! We have a single wide bus carrying the info from all those execution units. We have a single bit matrix (per Scheduling Queue) that is tested against that wide matrix. Our scheduling problem now boils down to figuring out nice circuits to perform the two tasks of interest (flip all 1s in a hot column to 0s; then find all rows with only 0s) and because these are clean bit-matrix problems we can solve them however is most convenient, even by analog techniques.

Note also that we simplify things by dealing with three different structures:
- a scheduling matrix that tells us which instructions are able to execute (all dependencies fulfilled)
- an age matrix which tells us which of the ready instructions are oldest
- an instruction list holding what needs to be passed on to the execution unit

This idea is called matrix scheduling. It may not sound like the patent if you read the patent blind, but I think you will agree that it's the only reasonable interpretation of how things have to work (given how wide the M1 is, and given everything else we know about the machine, like distributed scheduling queues and dependency bitvectors).
I *think* the specific point of the 2016 patent is the exact timing in this scheme. If you followed the tagID/destination physical registerID scheme, you'd broadcast the SCH# of each instruction in the cycle that it completes. This is easy, matches history, and the signal provided clearly means that the instruction has completed.
But it makes the Scheduler timing very tight; and it’s not necessary! With the matrix scheme I have described, you know, from the moment the decision is made as to which instructions will issue, what the SCH#'s of interest will be, so you could broadcast them in the issue cycle.

That's fine for most (one-cycle-latency) instructions, but you need something extra to handle the multi-cycle latency cases, hence the patent. I think, for example, that the bit-shifting scheme used by load-dependent instructions to maintain them in the scheduling queue until the load is proved valid (or needs to Replay) could be reused as a timing mechanism for this purpose.
(But honestly this is one of the uglier Apple patents I have encountered – determined to claim everything, equally determined to reveal nothing. If Apple gets this rejected by a court at some point, they'll have only their lawyer to blame – part of the patent deal is that you explain how you do something, you don't just claim that something can be done. And this patent elides the explanation of the only part that's interesting, namely the multi-cycle timing.)
The one other thing the patent covers is a few technical details that reduce energy consumption by allowing rows that are no longer of interest (have all their scheduling requirements already matched) to opt out of the matching procedure.

criticality based scheduling (future, not yet implemented by anyone)

BTW instruction scheduling, though it's been thought about for so many years, is hardly a solved problem! In particular, scheduling based on the oldest runnable instruction is merely a heuristic it's not optimal. Better would be to schedule instructions based on *criticality*, which is a measure of how much subsequent instructions will be sped up by executing this particular instruction right now. There are way to measure (approximately, of course, we can't look into the future!) criticality fairly easily, and there are criticality predictors that do a good enough job. The concept is explained very nicely in Pierre Salverda (2008) https://www.ideals.illinois.edu/items/11472/bitstreams/41916/object, *Principles of Instruction-Level Distributed Processing*.
But as far as I know no-one is yet using criticality to inform their instruction scheduling.

split scheduling queue (obsolete, only of historical interest)

For completeness, there's an additional early Apple Scheduler patent here (2008) https://patents.google.com/patent/US20100162262A1 *Split Scheduler*, but I think it's only of historical interest. The problem that patent wants to solve is to be able to schedule dependent instructions back-to-back, and the solution adopted is to try to segregate "immediately dependent instructions" from "other" instructions, so that the immediately dependent instructions form a very small pool that can easily be scheduled.
However the problem to be solved is better solved via speculative scheduling, so this is all moot.

There are more modern interesting ideas in the space of splitting a scheduling queue; one of my favorites is (2015) https://hal.inria.fr/hal-0122519/document *Long Term Parking (LTP): Criticality-aware Resource Allocation in OOO Processors*. But so far no-one appears to have adopted any such ideas.

microcode

Even on ARM occasional microcode may be necessary, for the occasional complicated system control instructions (like the instructions to invalidate particular sets of items in the TLB). (2011) https://-

patents.google.com/patent/US9280352B2 *Lookahead scanning and cracking of microcode instructions in a dispatch queue* dates from the Swift (A6) days but may well be (to some extent) the way the issue is still handled.
What's interesting about this is how it compares with Intel. Intel and AMD insert microcode at Decode. This complicates Decode, but means the microcode instructions have full access to Rename/Allocation facilities. If we take the patent literally, Apple does the equivalent after Rename, detecting "complex" instructions as they pass through Dispatch, on their way to be inserted in various either Dispatch Buffers or Scheduling Queues. When the complex instructions are detected, they are replaced with some number of microcode instructions which are fed into the Dispatch process until they're done. This is presumably slightly more energy efficient and more resource efficient (eg it looks like these distinct microcode instructions do not take up ROB slots); but it also constrains how wild they can be if they can not, for example, branch, or allocate generic registers. However, if the ISA does not require such shenanigans, why not take advantage of inserting microcode at this latest possible opportunity?

(Simple instruction cracking, at least for instructions that touch multiple registers, is handled by the (2012) https://patents.google.com/patent/US9223577B2 *Processing multi-destination instruction in pipeline by splitting for single destination operations stage and merging for opcode execution operations stage* scheme, as already described which, in Decode, generates multiple instructions [which can each be handled appropriately in Rename/Map/Allocate] then in some subsequent stage [maybe using the same sort of pattern detection as the "complex instruction" detection, in Dispatch] merges them back to a single instruction.)

optimizing the issue queue

We don't often think of it that way, but instruction queue slots are a resource, like ROB slots or physical registers. And like those, we should be interested in any scheme that can give us effectively more instruction queue slots.
Traditionally the instruction queue is one of, perhaps *the*, hottest parts of the chip because every cycle the entire queue has to be probed to figure out what instructions can be woken up (based on values that became available in the previous cycle) and the "queue" has to be repacked, or something equivalent, to both maintain age order and handle the holes from random instructions in the middle of the queue being scheduled.
We've seen that Apple tries to deal with these issues in a variety of ways
- multiple per-execution unit queues rather than one large queue
- clever dispatching of instructions to balance across queues (reducing the downsides of multiples imbalanced queues vs one large queue)
- using a tournament to find the one best (oldest, but ready) instruction for scheduling in each queue, rather than exact age maintenance
- using queue pairs to handle imbalance, where one queue might have two ready instructions and its paired queue has no ready instructions

- using a non-scheduling buffer to hold instructions even when the instruction queue itself is full.
Ultimately these are all an attempt to provide something close to issuing ready instructions by age, but without promising perfect scheduling by age.

The point of all this is that the Issue Queue is what allows us to look ahead in the instruction stream for independent instructions to execute. The large our look-ahead window, the more likely we can see past the next thirty (or fifty, or one hundred) instructions all dependant on what we are doing right now to find non-dependant instructions that we can execute simultaneously. This is different from the function of the ROB, which is essentially to maintain speculative state in such a way that we can recover if the speculation proves to be a mistake. So the ROB is (as of M1) effectively around 1000 instruction or so of speculative state, while the Issue Queue(s taken together) as of the M1 provide a look-ahead window of about 280 instructions.

Can we do even better? In fact we can. Consider (2020) https://www.diva-portal.org/smash/get/diva2:1390628/FULLTEXT01.pdf *Rethinking Dynamic Instruction Scheduling and Retirement for Efficient Microarchitectures*. This thesis points out that about 30% of instructions at Decode or Rename time already have their input values available, they have no need for the area and power complexity of the Issue Queue (which is all about holding instructions waiting for input values, so that as soon as a new input value arrives the oldest such waiting instructions can we woken up and issued); this 30% can issue right away. So why not bypass them into a much cheaper structure?
You can do even better than this if, in addition to detecting *readiness* at Rename time, you also detect *criticality*. Now you split your instructions into three streams, essentially *ready* (so don't need fancy scheduling), *non-critical* (so also don't need fancy scheduling, they can wait around in a simple structure till ready, and then just be executed whenever there are free execution units) and about 40% or so of instructions that are both not ready and critical, and so do need the full machinery of the Issue Queue.
(The second half of this thesis discusses out of order retirement, something we already discussed earlier in this document. The thesis approaches the problem of out of order retirement from a different angle, IMHO less practical than the idea of the Validation Buffer, but it does include essentially limit studies that suggest that practical "safe" out of order retirement can get you about 50% better IPC, while "unsafe" retirement can get you around 100% better IPC.)
The above ideas tackle one side of the issue queue, namely trying to bypass it for items that are already ready.
The other side of the problem is avoiding stuffing the issue queue with instructions that will not be ready for a long time (and thus are just wasting space and burning energy every cycle, to no useful end). We've mentioned one solution to this is Long Term Parking; a different paper giving much the same ultimate solution is (2004) https://d1wqtxts1xzle7.cloudfront.net/93424809/ICS-04-IQ-libre.pdf?1667281228=&response-content-disposition=inline%3B+filename%3DScaling_the_issue_window_with_look_ahead.pdf&Expires=1732473961&Signature=Tj5CYWqB2gKAm-UaGjZXYLCGIRIX7g-COBqY4iX2liRu6tsIkpjarewsIb954~OS6FtICS-aZ-Kehzru3qF94ZEK~lyy7PfSqMIGuMuQe~j2YAYFRG-NRhqNlJ4yXPfmW6WHViQcLpRO27IRWQUiD-9BjceiWdBURINeJSlRuyM2Mob6wsIXowub1eDXcBXJbGapjbvRmBAb3RGYiloX~R29y0rt9lkNO2h8RvjrRrL1UTC84WMVLFm8I8sGB6l7ZgRIDJwJ7G7T6Bc4m-

LJB5q8BLuHMw8YvXpX8QVZo9YNd90owYffmJWt15h2jnvB3yIoG5PuDYl9DGr8vN0P2Ag__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA *Scaling the Issue Window with Look-Ahead Latency Prediction*.
In both cases we begin with a latency predictor that attempts to predict instructions that will take a long latency (generally loads that will miss in L1), then steers the dependents of those loads into a light-weight buffer where they can sit comfortably until they become runnable. For most CPU designs you also need to estimate when the load will hit the cache, or accept some lost cycles while the instructions were waiting in the buffer while runnable. For Apple's design this is probably less of a concern, given that Apple already appears to have very fine-grained notification from the L1 cache up into the core that any particular load has just hit the cache.

This idea of long term parking, or shunting predicted long-wait dependents to a separate buffer sounds good – but there is a problem you have to be honest about. How useful, in the real world, is your predictor?
We are interested in loads that are predicted to miss in L1. But most (not all, but most) of such loads can also have their addresses fairly well predicted, in which case we can prefetch their data. So this machinery only work ideally for a fairly narrow case of loads whose address cannot easily be predicted (eg walking some large complicated data structure) or cannot be predicted early enough for prefetch to be useful. A lot hinges on the quality of the predictor, and it's unclear in many papers that the predictor has been paired with a realistic CPU design, specifically a realistic set of prefetchers.

Experiments on instruction scheduling

Explaining the integer Dispatch Buffer

So back to the issue that got us interested in scheduling in the first place -- in the previous graph, what's up with the pronounced flat region in the middle?
Here's my best explanation:
Assume a two level Scheduling Pool: an initial buffer (the Dispatch Buffer) followed by a genuine Scheduling Queue. We know that Rename can dump instructions into the integer Dispatch Buffer at 8/cycle, but maybe instructions can only leave that buffer at 6/cycle, so one per scheduling queue? (Faster might be desirable under rare conditions, but it's usually unnecessary, and it's power/area expensive.)
This would mean that the Dispatch Buffer is going to fill up at a rate of 2/cycle (8 in, 6 out). Or, more precisely, perhaps each of the two Dispatch Buffers fills at 1/cycle (4 in, 3 out)?
After 96 ADDs, a Dispatch Buffer that can hold 24 (2/cycle*12 cycles) will be saturated. In fact Dougall's experiments (probing for exactly this size, not detecting it as a side effect like we are doing) has the 1st level int Dispatch Buffer as 24 elements (split into 12+12).
So effectively the machine runs like

First 96 ADD's:	throughput 8-wide, fills up first level Dispatch buffer with ~24 instructions
274 ADDs from 80 to 370:	throughput 6-wide, since only 6 instructions can pass from Rename to

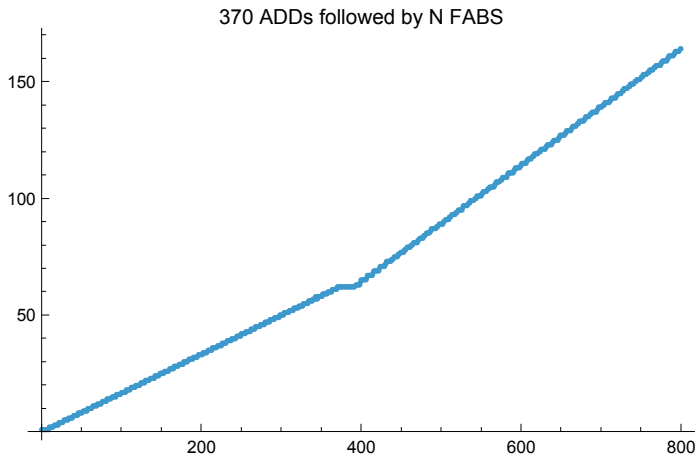
Dispatch
excess cycles in this phase is (290/6-290/8=11.4. We have permanently lost ~12 cycles in our 8- vs 6-wide comparison.
MOV's start: for about 4 cycles (24 ADDs/6) we get to process ~32 MOVs (or NOPs) for free in Rename, in parallel with int Execution as the excess ADDs stored in the 24-slot Dispatch Buffer are drained.

In other words the flat region (when it's apparently costing us no extra cycles to process MOVs (or NOPs, or anything else executed in Rename) are when we're overlapping execution in Rename with instructions that were present in the Dispatch Buffer and are now making their way to Execution. For this to happen we need the flow rate into Dispatch Buffer to be higher than the flow rate out from Dispatch Buffer, so that a backlog of instructions builds up in Dispatch Buffer that can then drain in parallel with the subsequent instructions executing in Rename.

testing the fp Dispatch Buffer

We can check this understanding by testing a few variants.
If this analysis is correct, we should see the same phenomenon if we follow the ADDs with something like FABS which executes not in Rename, but in a different pipeline, so can again run in parallel with the int Dispatch Buffer draining.

Out[515]=



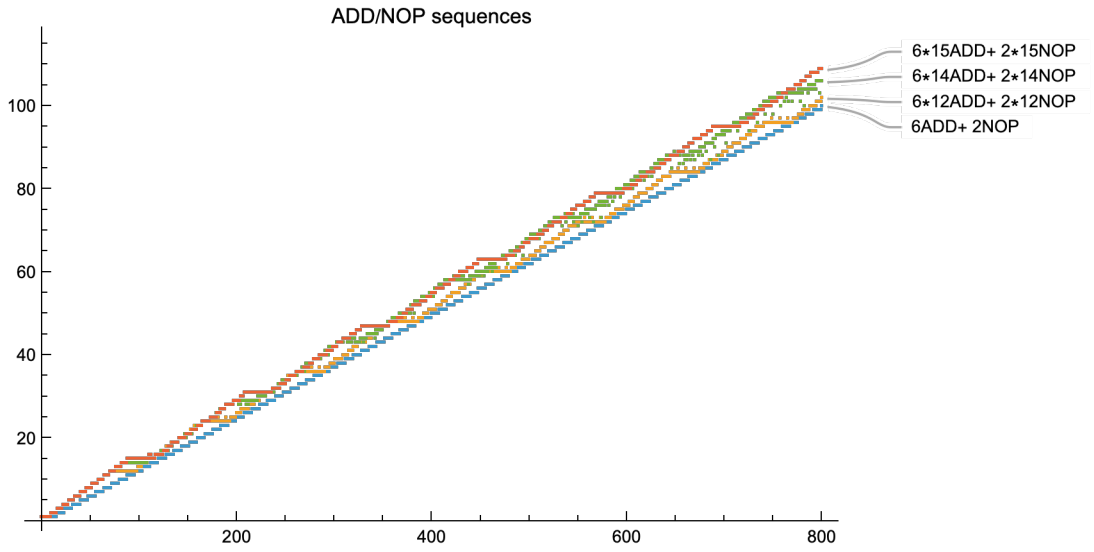
Well, that's nice! As we predicted :-) And in fact the signal is cleaner (the flat run is 23 instructions long)

Interpretation as smoothing mechanism

Now let's try something a little different.
Suppose we interleave (`6ADD+ 2NOP`). We know that can run at 8 wide indefinitely, it's perfectly balanced with, every cycle, 6 instructions going into the integer pipeline and 2 NOP instruction being executed at Rename.
Now suppose we change this to (`12ADD+ 4NOP`). Now it's not *perfectly* balanced: The first cycle we

dump 8 ADDs into the Dispatch Buffer, but only withdraw 6. The next cycle 4 ADDs into the Dispatch Buffer -- along with 4 NOPs executed at Rename, and then 8 NOPs executed at Rename. But still balanced overall – as long as we have a buffer that can hold 2 ADDs.
We can continue like this through (k*6 ADDs + k*2 NOPs), until the point where the Dispatch Buffer is not longer quite large enough to hold all the excess ADDs. Let's see what I mean:

Out[520]=



Consider the case of 6*15 successive ADDs. This is 90 ADDs, so takes just over 11 cycles (8*11=88) in the pipeline up to Dispatch. Each of these 11 cycles, 2 excess ADDs remain in the Dispatch Buffer, so at the 15*6 case we have almost maxed out that buffer. Throw in imperfect alignment and imperfect balancing of instruction and we can say the buffer reaches its capacity. And we see that it is indeed noticeably slower than the predecessor cases.

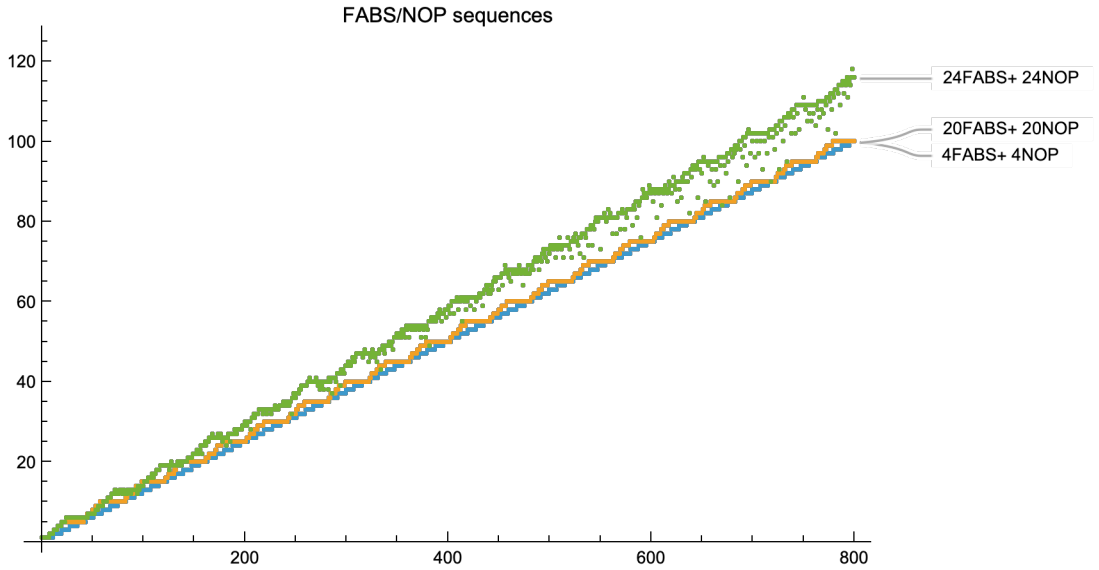
Essentially the case 6*1 through 6*13 always take the fast path (102 cycles per loop iteration), anything above 6*15 takes 108 cycles, and 14 is a slightly noisy version in between.
To put it even more simply, the Dispatch Buffer acts to smooth local variations in the types of code, so that a dense run of ADDs can still run 8 wide, as long as the dense run is not *too* long. We see the transition here between the case of *not too long* and *too long* here at between 6*13 and 6*15 successive ADDs. Once you are unbalanced for too long buffering cannot save you and your maximum throughput is reduced (ie slope of the line moves upward).

The same sort of buffering over regions of dense code (for example a stretch of FP or load/store dense code should likewise help us to still maintain an IPC of 8, as long as
- overall the balance of instructions is not too biased to only one type of instruction, and
- the runs of a single type of instruction are not too long.

acceptance width of fp buffer

Now the next obvious question is what are the exact properties of these Dispatch Buffers. For example consider the floating point dispatch buffer. Presumably this releases instructions 4/cycle, but does it accept instructions 8-wide or something narrower? Let's see.

Out[524]=

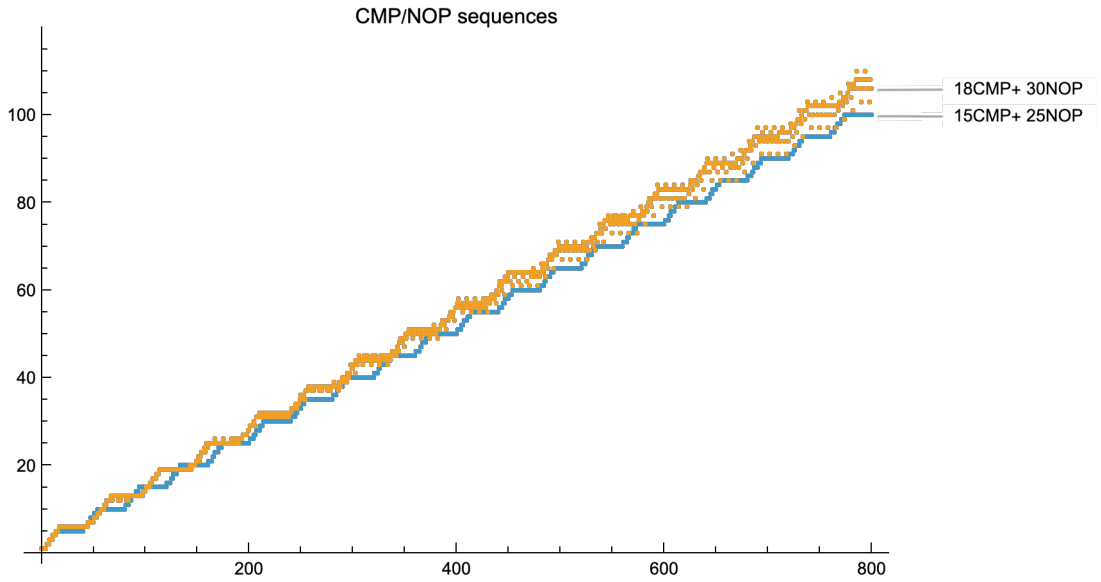


So let's assume that the green line were exactly matching the lower two lines. That would mean that three successive (8x `FABS`) instruction probes could be processed through Rename (and into fp Dispatch), followed by three successive (8x `NOP`) instructions without a hiccup. Which would mean that the Dispatch Buffer must be able to accept 8 instructions per cycle, and be able to hold at least 12 instructions (since each of those three cycles, its net occupancy will grow by 4 instructions.
We see that we can't quite achieve that with 6*(4x `FABS`+4x `NOP`) but we can with 5*(4x `FABS`+4x `NOP`). Suggesting that the Dispatch Buffer is ~12 in size. (The methodology is somewhat sloppy because it will be sensitive to exactly how the FP instructions are aligned in the run of eight instructions that's passing through Decode to Rename – sometimes there will be perfect alignment, sometimes there will be a straddling of some FP instructions at the head and at the tail of a line.)

probing that the integer buffer is split in two

Now what if we try this same methodology with CMP? We believe that CMP can execute of three of the six integer pipelines, and Dougall's diagram suggests that one of the two integer Dispatch Buffers feeds these three pipelines, meaning we should expect to only be able to buffer 12 of these instructions. Do we see that?

Out[527]=



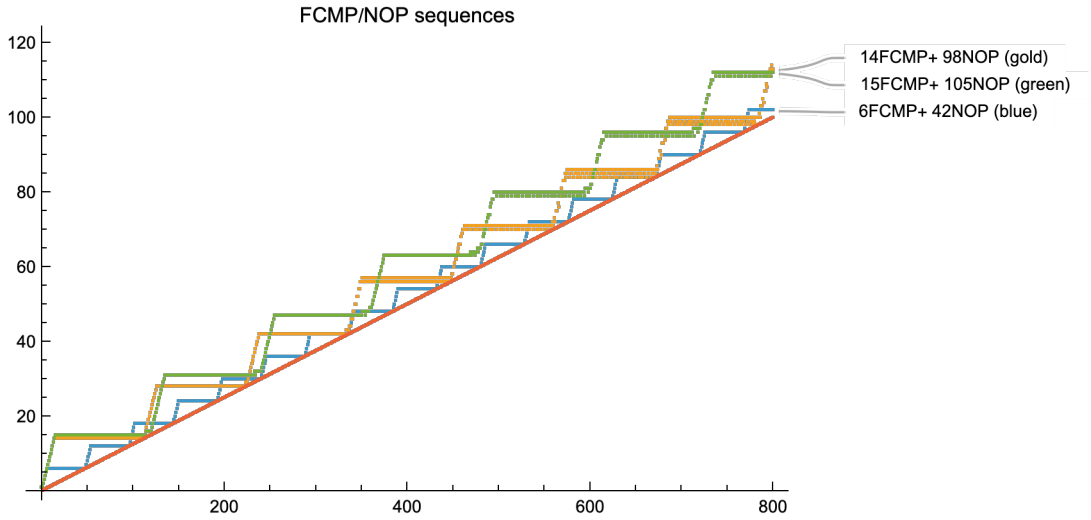
So let's think about this. When we run 6* (3x CMP+5x NOP) we have a sequence of 18 successive CMPs running through Rename. That's a little over 2 (round it up to three!) cycles.
If we take this as meaning that we can transfer all those CMPs into a (12-sized) buffer while, during two cycles, extracting 2*3=6 CMPs, the numbers kinda work out – certainly better than assuming a 24-sized buffer. It also seems that these two integer Dispatch Buffers can accept 8 instructions per cycle.

You will recall when we started down this path we also ran the experiment for MUL, another instruction specialized to only some integer execution units, and again we saw the same sort behavior, a buffer of ~12 in size.

probing if the fp buffer is split in two

What about the fp Dispatch Buffer -- is it possible that there are two of these, each feeding two of the four fp pipelines?
We can try the same technique with FCMP (which can only execute down one of the four fp pipelines). The results are

Out[531]=



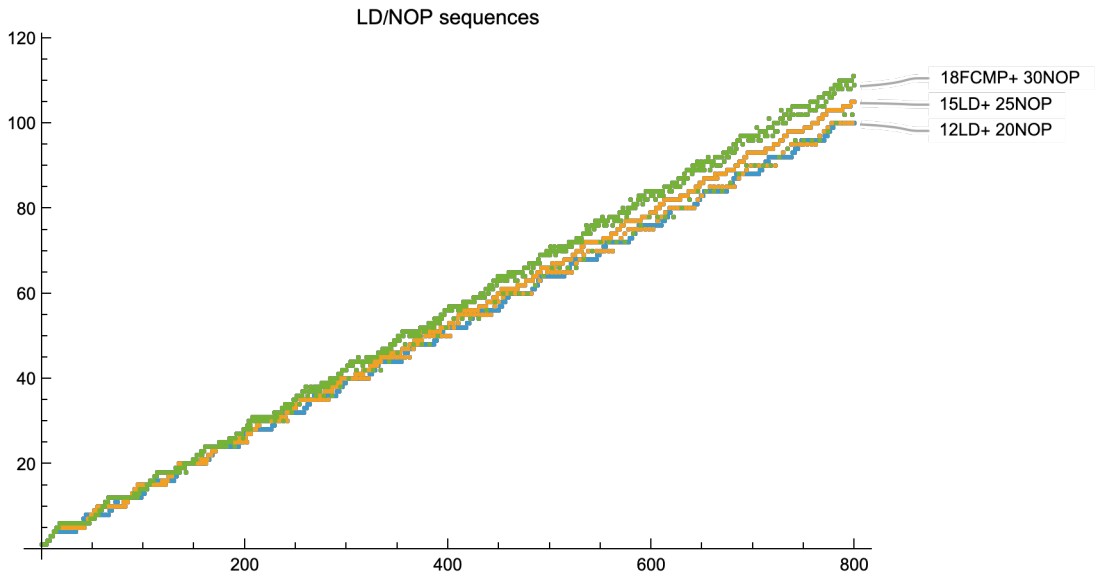
Now this one is actually surprising! The jump (deviation from a long term rate of 8 operations/cycle) occurs at 15* (1x `FCMP`+ 7x `NOP`). It's hard to square that with the precise numbers on Dougall's graphic.
If we can sustain 14 `FCMP` without reducing the overall throughput (ie overall the gold curve stays at the same slope as the blue curve [6 FCMP] and the red curve [8 ops/cycle perfection]), that implies we must be able to in a first cycles move 8 `FCMP` from Rename to FP Dispatch, of which one moves on to Execute. Then in the next cycle we move over 6 FCMPs, one goes to Execute, leaving us with a net of 12 in Dispatch. This is so perfectly matched it suggests that perhaps the size of the FP Dispatch Buffer is slightly larger, perhaps 14 rather than 12?

Either way, the point we wanted to establish is established – we have a single FP Dispatch Buffer, rather than a split buffer like integer.

probing the ld/st Dispatch Buffer

Since we have the machinery in place, we might as well also test load/store!

Out[535]=



If we compare this to the CMP results (`CMP`, like `LD`, is 3-wide) we can see that we overflow the LD/ST Dispatch Buffer slightly sooner than for `CMP`, which agrees with Dougall's result of a size of 10 for the LD/ST Dispatch Buffer.

Handling Immediates (MOV #)

Overview

So far we've seen that we have what feels like ~380 int physical registers, but ~624 entries in the History File. If we're doing integer-only work, we'd like to be able to make use of all those HF entries without first being limited by the physical registers. How can we do that? Well

• some of the HF entries will probably be used by instructions that set flags.

• another aspect of bridging the gap is register duplication (zero-cycle MOV), which requires an HF slot, but not an additional physical register.

• a final aspect, which we'll examine now, is the special treatment of immediates in the context of register initialization.

There are a few different types of immediate, and, unfortunately for us, they all seem to have somewhat different paths, and different characteristics!

zeroing

Most common is the need to zero a register, and one can imagine multiple ways to do that, eg

`MOVZ x0, #0` (aliases to MOV x0, #0) runs 8 wide

`MOVI x0, #0` (normally aliases to ORR, not allowed for #0 so we will try `AND x0, xzr, #1`) runs 6 wide

`MOVN x0, #0` (not allowed for #0 or #-1)

`MOV x0, xzr` runs 6 wide
`EOR x0, x0, x0` runs 1 wide! Not only not an idiom, but forces the chain dependency!

So Apple clearly want us to use `MOV #` (whatever the preferred machine code might be), and aren't going to make an attempt to catch other cases. (It is remarkable that `MOV x0, xzr` is not special-cased as a Rename-time remapping...)

non-zero immediates

How about constructing non-zero constants?
`MOV x0, #1`
`MOV x0, #-1`
`MOV x0, #0x1234`
`MOV x0, #0x12340000` (coded as `MOVZ x0, #1234, lsl 16`)
`MOV x0, #0x1234000000000000` (coded as `MOVZ x0, #1234, lsl 48`)
`MOV x0, #0808080808080808` (one of the "boolean" coded logical immediates)
ALL of these (!) are recognized and run 8-wide.

Now let's not get carried away with the 8-wide goodness! Apart from an immediate of zero, these are implemented as 2-wide in rename plus 6-wide in execution, so not quite as free as register duplication, but often free.
The common case of an isolated initialization or two will be free; along with 6 other integer ops, and even the common case of many back-to-back initializations will run 8-wide, albeit often using integer slots.

dependencies of immediates

How about dependencies?
`MOV x0, #0; MOV x1, x0` seems to run fine (800 ops in 100 cycles), though I'm not doing true latency tests.
In other words both these zero cycle operations either stack together in the same Rename cycle, or appropriately pipeline in successive Rename cycles as you would hope.

Likewise for `MOV x0, #0`; `MOV x1, x0`; `MOV x2, x0`; ...`MOV x7, x0`.
M1 even appears to be able to note and appropriately handle (via zero-cycle rename) all the dependencies inherent in the chain
`MOV x2, x1`; `MOV x3, x2`; `MOV x4, x3`; ..`MOV x0, x8` !
Apple appears to feel (I assume this is what LLVM will generate) that if you want to set many registers to the same value, the way to do that is via
`MOV x0, #`; `MOV x1, x0`; `MOV x2, x0`; `MOV x3, x0`; ...

Out of interest, we can also test
`MOV x0, #1` + `MOVK x0, #0x1234, lsl 16`

to create a 32-bit wide constant.
This runs at 4 (of these pairs) per cycle, suggesting that 2 of the ops (probably always the baseline MOV#s) run at rename, while the other six (two of the baseline MOV#s, four of the "with keep" bit-insertion `MOVK`'s) run at execute. This matches Dougall's throughput for `MOVK`. And also tells us that `MOV+MOVK` is not fused.

floating point immediates (no interesting support)

ARM supports a few floating point immediates, but the M1 does not handle these in any special way at Rename.
Likewise the way to zero an FP register is via `FMOV dn, xzr`, but (like everything xzr related...) this is not special-cased.
So there are various dimensions along with the current scheme could do better, especially for FP.

implementation

How is this immediate-handling implemented?

2012 scheme (special register names)

The patent 2012 https://patents.google.com/patent/US9430243B2/ *Optimizing register initialization operations* discusses the original implementation, suggesting the goal of zero-cycle initialization is to perform initialization one cycle earlier, not either to offload work from the integer pipelines, or to give us some free registers; those are nice side effects but not the primary goal.
The implementation has changed since 2012, but the *primary* purpose (lower latency initialization) appears to be unchanged.

The 2012 patent suggests that unused physical registerIDs can be used to encode a few small constants (think eg 0, +1, -1). The idea is that if you have, say, 192 physical registers, you can use registerIDs 192..255 to encode special immediates. Beyond that the details are vague (apart from a suggestion that registerID 255 is hardwired to 0).
However it is important to note that (like many good ideas from the 2012 A7 era of Apple cores) this idea seems now to have been superseded by a new scheme, that's not yet in the patent literature.

I can imagine an extension of the 2012 idea that uses wide'ish registerIDs, about 22 bits or so. The first bit or two would indicate an immediate (of four or so types) vs a physical register. If a physical register, subsequent high bits could indicate the register file (int vs fp vs flags), and even special purpose registers and (indirect, ie delayed allocation) ROB IDs.
This would allow immediates to be "executed" 8-wide at Rename time, and would allow fp immediates also to be executed at Rename time. When a subsequent use case wished to read the register, it would submit the registerID, as usual, to the register file, which, rather than performing a lookup, would have

logic to decode the tag and generate the appropriate immediate onto the bus delivering the register value. (That part is described in the 2012 patent.)

This is not as outrageous as it might appear. Something like this already has to be done in part: One of the "registerID" slots of instructions transported down the integer pipe for int execution today has to be wide enough to hold all the various forms of immediate as part of the instruction definition. So what I am suggesting is not that absurd a modification, it's more simply allowing that same wider registerID across *all* the registerID slots.
But that's not what we have today... Oh well, there will be more chips! And Apple has changed the scheme at least once, they can improve it some more!

So that's one path to handling immediates, via encoding them in a wider "register dependency" field.

current scheme (separate register pool)

Another path (which appears to be what Apple is doing right now) is using a separate register pool. Imagine that at Decode you interpret that an instruction is a `MOVI xn, #123.` What can you do with this information? One possibility is to find a free physical register and store the immediate value in that free register; then the next cycle you can map that physical register to xn.
This sounds good in theory, but runs into a wall as soon as you think about implementation details. The basic problem is that the primary integer register file has a certain maximum number of write ports and the storing of the immediate in that register file adds to the pressure on those write ports. One could just accept that cost, that you can't always get what you want including as many writes to the register file this cycle as you want; but another alternative is to create a separate "immediates-only" register file which can only be written to by this Decode path. We'll see below that when using immediates we appear to have access to about 38 more integer registers than usual! Which strongly suggests that there is indeed such a separate pool.

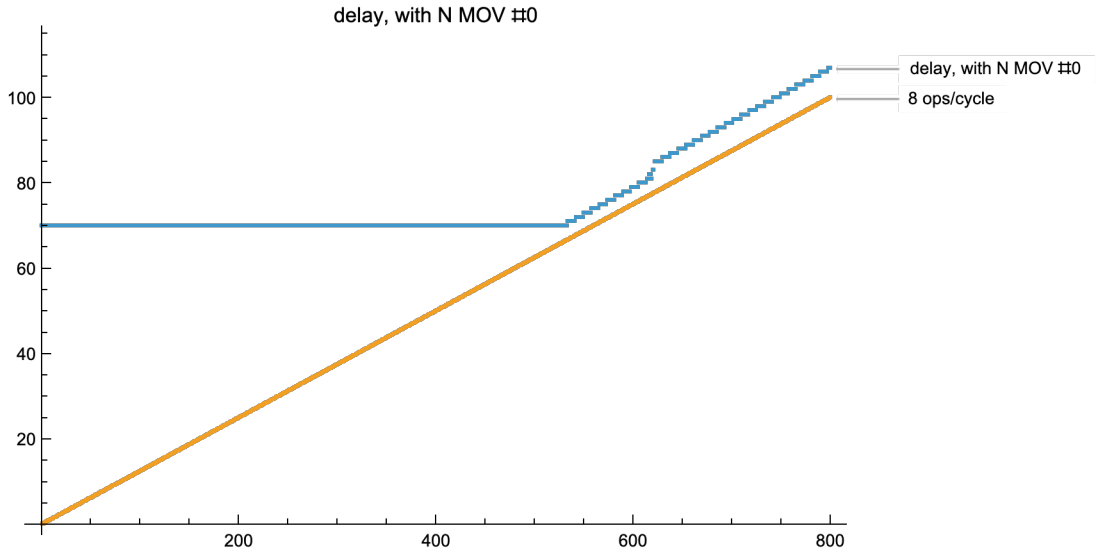
I can't find any academic discussion of how common immediate constants are in code, and so whether they are common enough to justify all this effort.
However eventually we will discuss Value Prediction, which also strongly benefits from having a separate register pool (again for reasons of write port pressure), and it's possible that this pool of additional registers could one day service value prediction in a future core.

Probing immediates experimentally

mov #0

Let's test a delay block along with N `MOV #0`'s.

Out[537]=

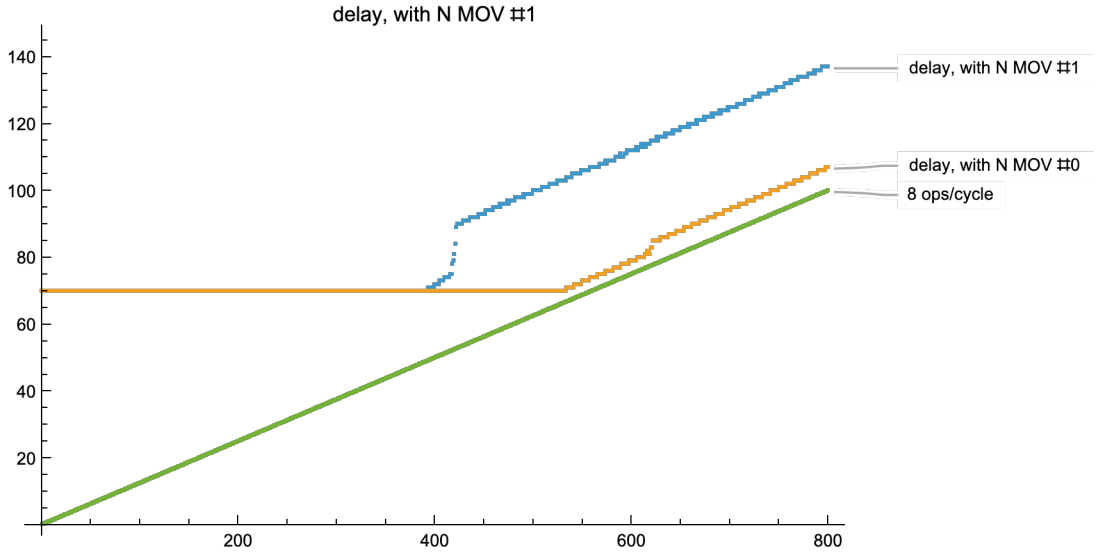


No real surprises. The slope is 8/cycle. There's a brief stutter around N=620, presumably something to do with the History File, but let's not get distracted.

mov #1

Compare this to the exact same structure but using `MOV #1`.

Out[539]=



Rather different!

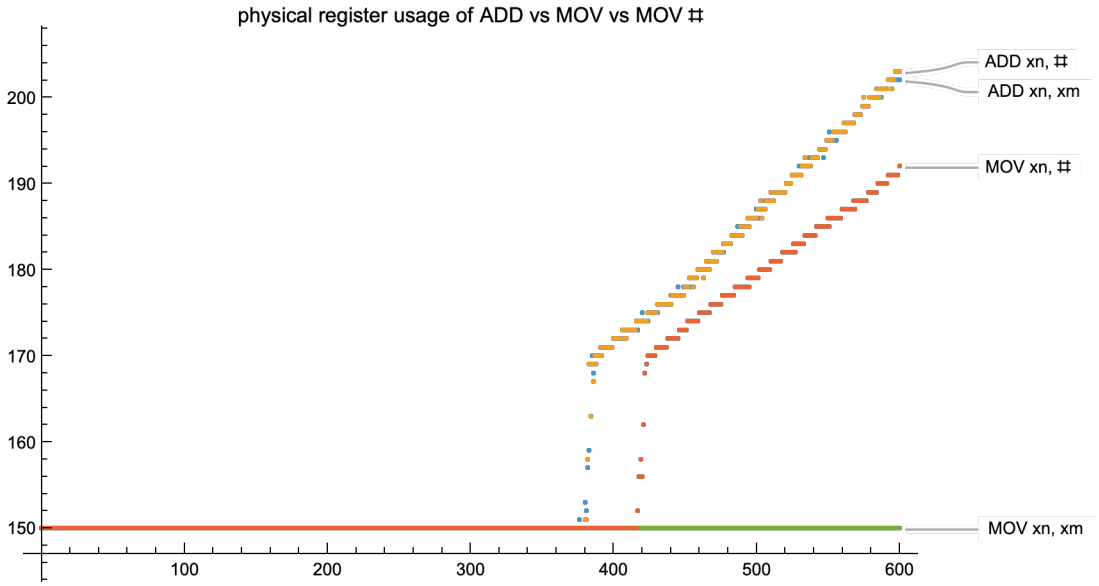
It's appears that `MOV xn, #0` is special-cased and is handled like `MOV` or `NOP`, fully resolved at Rename.
However any other immediate, eg `MOV xn, #1,` is "semi"-special-cased.

existence of additional, special, immediate registers

We already know that we get two free MOV# Rename's per cycle; any more than that have to route through Execute.
But compare the the MOV #s to standard register usage, in terms of the use of physical registers.
(For reasons that will in time become apparent, we'll also switch to a much longer delay time.)
So see what I mean, compare

```
ADD x0, x5, x5
ADD x0, x5, #1,
MOV x0, x5
MOV x0, #1
```

Out[544]=



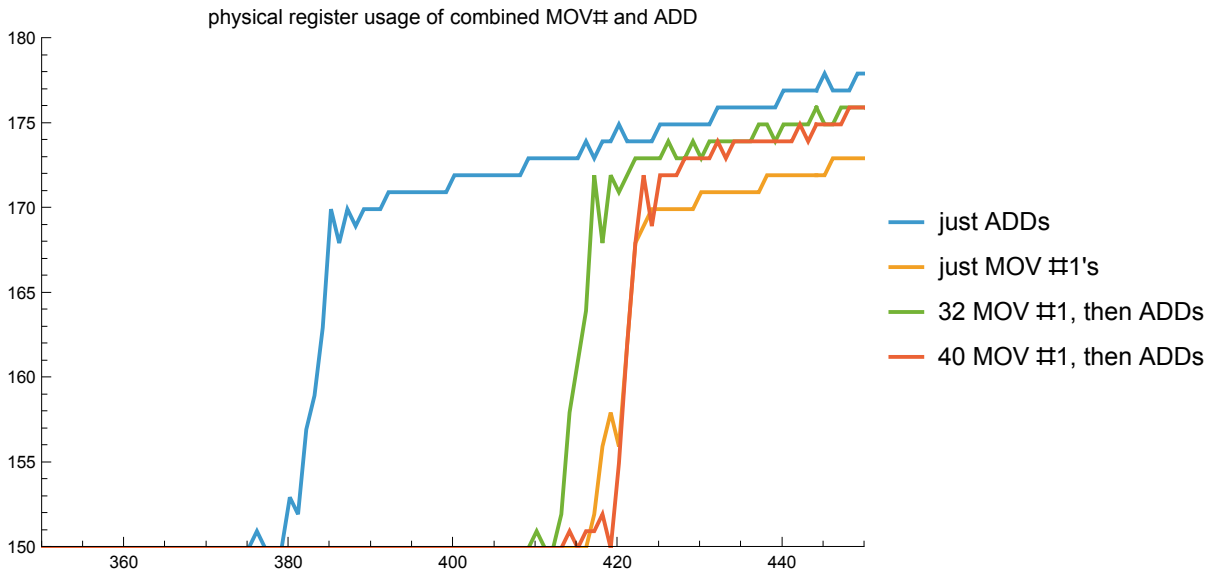
So
• the ADDs (two registers, or register and immediate) max out at around 380 physical registers.
• the MOV x0, x5 does not use physical registers at all, and maxes out at around 620 History File entries.
• MOV x0, #0 is like MOV x0, x5; fully executed at Rename, doesn't use a physical register.
(No surprise since it's doubtless a rename to a hardwired zero – but not xzr.)
• The unexpected case is MOV x0, #1, which maxes out at around 418 physical registers.
It's like it has access to about 38 more registers than a standard integer instruction!

Let's examine this more carefully.
Our hypothesis is that there's *some* extra storage available for holding immediates; ie a few special registers (call them I registers or iRegs) that can only be written to at Rename, and that aren't part of the main int physical register set.
But we need to be careful in testing this because, remember that we can only execute two MOV#'s in Rename per cycle. So we need to pad those MOV#'s with NOPs to ensure that nothing goes down the

standard integer pipeline (where it presumably fills a standard integer register).

We start by considering a probe that consists of 32x (`MOV x0, #1; NOP NOP NOP`) followed by ADDs. The idea is to see whether the `MOV x0, #1` took storage away from the ADDs.

Out[547]=



(Normally I haven't added "joining" lines between actual data points, but in this case the lines are helpful to guide the eye, even though they exaggerate the noise in the image.)

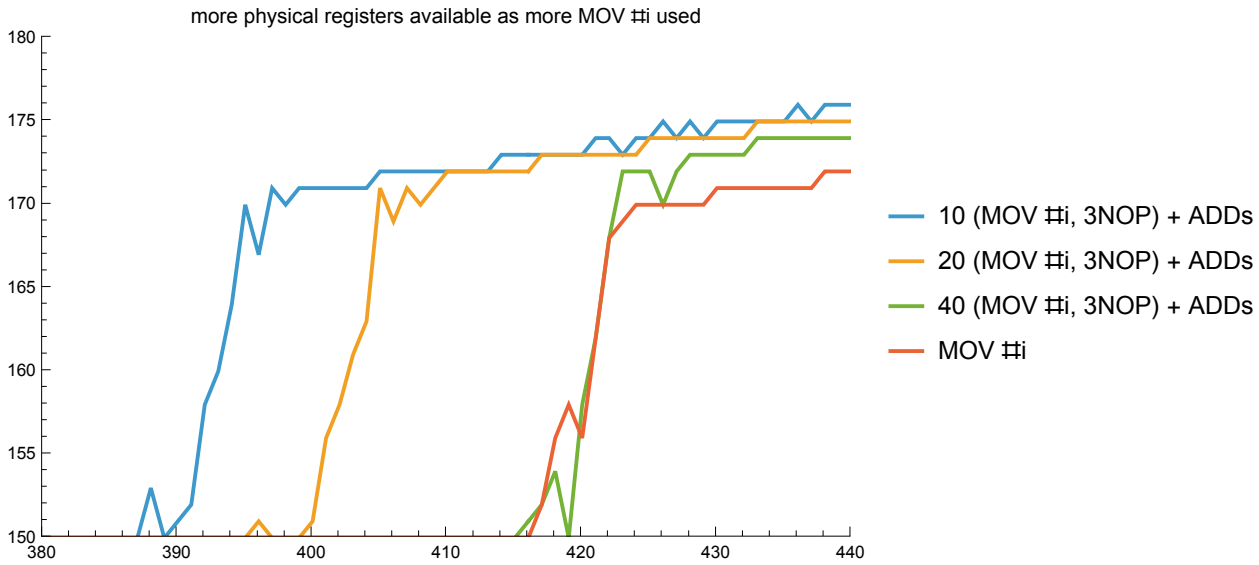
Hypothesis confirmed!
Look at how the green curve jumps at close to the same place as the gold curve; ie the MOV# register usage has not removed registers from those available to the ADD.
The red curve (40 rather than 32 `MOV #1`'s) suggests that there are more than 32 I registers available, perhaps as many as 40.

how many immediate registers? how many values can they hold?

So the next question is how many of these registers are there really, and how they are "organized".
For example one possibility is that there is only a single immediate register, which can hold a single non-zero immediate value (but with multiple references).
Another possibility is a set of 40 registers, each of which can hold a different immediate.
Or anything in between, like a CAM that can hold up to 8 distinct values?

First let's try 40 `MOV x0, #i` before the ADDs. (ie the immediate value ramps from 1 through 2, 3, ... 40)

Out[552]=



So we see that if we use 10 `MOV#i's`, we get essentially 10 additional registers (we max out at 390 rather than 380). Same for 20.
For 40 we don't get quite all the way to 420, so the number of additional iRegs appears to be around 36.
Also we are using a different immediate value for every `MOV`, so we must have at least 36 distinct "storage objects" available.

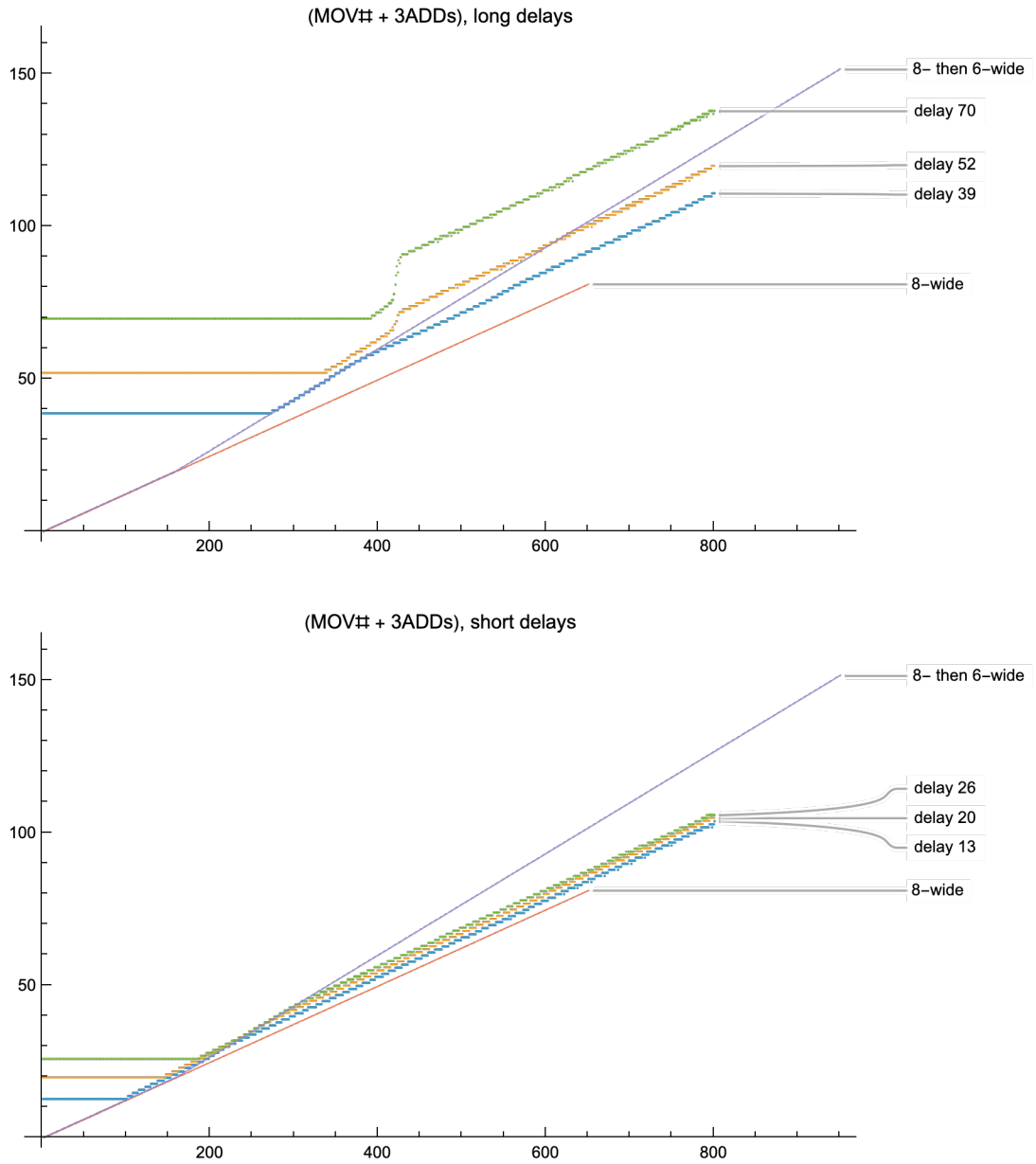
what happens when we use up all the immediate registers?

One final test . We know that the I-registers can be written to from Rename.
Consider the following probe N times (`MOV x0, #i  ADD x0,x5,x5  ADD x0,x5,x5  ADD x0,x5,x5`).
This produces a curve that jumps at around 416 or so, no surprise, and that has a slope of 8 instructions/cycle, again no surprise.
But think what the jump at 416 means.

Presumably there are no more than about 36..40 iRegs. And 416=52*(2+6).
So we know that the first 40 or so MOV#'s are handled at Rename, apparently via being written to special iRegs. What about the (2*52-40=64 `MOV #i`'s that execute after those first 40, once the 40 iRegs have been allocated?)
One can imagine two possibilities.
• The first is that there are generic write paths from Rename to all registers in the int register file.
If this is the case, then we should be able to process the 416 or so operations in 52 cycles.
• Alternatively, if the write paths from Rename to the int register file are only present for the iRegs, then we'd expect the 416 operations to be processed using 20 cycles for the first 160 operations, and then (416-160)/6=43 cycles for the remainder, taking a total of 63 cycles.

Can we see this difference?
It turns out we don't actually need to work so hard!

Out[561]=



To make things simpler, I split this into two plots, one showing small delays, one showing longer delays.
The longer delay plot (look at eg delay 39) answers the question that interested us, namely we see a stretch of 6-wide execution before reverting to 8-wide.
In other words, it appears that only iRegs can be written to from Rename, and when an iReg is not available, MOV# has to run 6-wide.

But beyond this, one might ask about the other details in the plots.

• What I think is happening is that for short delay, the pattern for every loop iteration is first to use up the iRegs, which are then rapidly released when the delay completes; thus there is a very limited to no window during which no iRegs are available.

• On the other hand assume a longer delay, say 39 cycles. One's first quick thought is that the iRegs should always be used at the start of a loop, then released as soon as head of ROB clears, so should be available again and we should immediately revert to 8-wide. That is what happens on the first iteration through, but it's not what happens long term.

Long term I think what happens if, say, you're running with N=300 and delay of 39 is that iRegs are used when they can be, released at some later time, and land up being spread throughout the ROB rather than being all bunched at the beginning of the ROB.
And so even when head of ROB clears, that doesn't free all the iRegs right away. In fact it's only if you have N>~420 or so that you provide a long enough period of time after head of ROB clears for *all* the registers to be freed and the a reversion to 8-wide.
Later we will see that freeing registers (when it is possible, ie if there's not a delay instruction blocking the head of the ROB) occurs at 16 registers/cycle, which is fast but not infinitely fast.

summary of experimental findings

So I think the model is that
- there is a special hardwired zero physical register (ZPR) (which, strangely, appears not to be xzr!) and a `MOV #0` is treated as a Rename duplication to that ZPR
- there is a augmentation of the physical register file that can hold perhaps 36 distinct values, and can be treated as part of the register file (can be looked up via RegisterID)
- there is a special write path from Rename to the iRegs, but not generic pRegs, to allow up to two writes from Rename per cycle
- but these two Rename writes are only possible if free iRegs are available

- thus if Rename sees that an instruction involves the write of a known value AND a free iRegister is available, then Rename will simply write the value and mark the `MOV#` as complete, so that it doesn't need to pass on to Scheduling and Execute.
But if any of the conditions fail (not a known value, no free iReg, already both ports to the iRegs are in use) the `MOV #` is not marked as complete, and it passes on to scheduling for normal execution.

- `MOV #` is primarily a latency reduction tool, but it does also give us the effect of a few additional physical registers.

In this context, this LLVM check-in, https://github.com/llvm/llvm-project/commit/d5f1131c812d-f57560c7563475cb0d674a101636, from April 2021 is especially interesting, suggesting that pretty much the entire ARM community has concluded that MOVI works better than using xzr. (The context is zero'ing FP registers, but honestly, if anything moving data across to FP should be easier for a dedi-

cated register than handling generic immediates.)
It’s not clear to me quite why handling a dedicated zero register should be so problematic, but the lesson of ARMv8 generally seems to be that xzr was probably a reasonable idea in the context of the ISA encoding (ie as a way to encode two or three functionalities in a single instruction) but not a good idea as a way to actually use zero as a value (eg in a MOV).

Some Register File Implementation Details

You may recall that when we first started looking at exhausting the physical register file we saw some unexpected blips. Now is the time to consider what they tell us.
Suppose you have to create a register allocator – what are you being asked to do? You need machinery that performs the following tasks
• you need a pool of registers
• you need a way to know which registers are in-use versus which are free
• given the pool of free registers, you need to provide Rename with up to 8 physical registerIDs every cycle.
Each of these tasks has many further details!

Structure of the register pool

Consider the pool of registers. We talk about a "register file" but there are many possible implementation details here.

For example: in Apple's first 64-bit cores, the integer file was in fact split about 40% 32-bit registers and 60% 64-bit registers! This allowed saving some area, and the free register allocator would preferentially provide 32-bit wide registers for instructions taking a w- register rather than an x-register.
Details here: 2013 https://patents.google.com/patent/US9639369B2 *Split register file for operands of different sizes.*

Apple no longer does that (at least I can find no evidence for it), but what they do do is split the register file into four banks which can each be individually powered on or off. To put this in context however, we first need to look at how registers are marked free.

Apple’s implementation of a “free register list”

Obviously one part of freeing registers is having the ROB tell you when a physical register is no longer in use; we'll assume that's a solved problem and ignore it. More interesting is how do you preserve this information of what registers are free?
One option is to maintain a queue- or list-like structure (so basically an array of slots, newly freed registers go in one end, registers to be allocated come out the other end). This works and is easy to run wide (just pull 4 or 8 or however many you need values from one end), and has been the standard

solution for years, hence the usual term free register list. But this solution limits your control over the process.

Apple have (once again) gone through at least three refinements.

bitvector (2012)

The first method, (2010) https://patents.google.com/patent/US20120143874A1 *Mechanism to Find First Two Values for microprocessors* is based on bitvectors. Instead of a queue, imagine that when a physical register is free we flip a bit associated with the register, so that a bitvector consisting of *NumPhysicalRegisters* bits tells us which registers can be reused. This uses little storage but does require, every cycle, a search along the vector for bits that are set to 1.
in this earliest version the bitvector is split into two halves, and each half is examined to find two free registers (so that we can find up to four free registers, which is good enough for an A6 class CPU). As an aside, the suggested physical register size is 56 registers! Those were the days.
The real focus of the patent is the circuit for finding two free registers (ie the first two bits set to 1) within a bit vector; done in a recursive fashion, based on groups of three successive bits.

Next is (2012) https://patents.google.com/patent/US20140013085A1 *Low power and high performance physical register free list implementation for microprocessors*. This is the same bitvector as above but with multiple ways to make this scheme work better (and to scale to finding more free registers). The important ideas are
• we actually use multiple bitvectors. Remember the older A7 design was 6-wide, so the maximum number of register allocation per cycle is 6. We have three bitvectors which, between them, cover the entire set of free registers.
So 1/3rd of the free bits live in the first vector, 1/3 in the second, 1/3 in the last.
• we have three pairs of scanners that run over these three bitvectors, one scanning forwards, one scanning backwards; so each scanner has to find one free bit, and between the 3*2 of them you can find 6 free registers.

• but for this to work well, the free registers have to be evenly distributed over all three bitvectors, and most of the patent is about how that is done.
• some ideas for how to do this are obvious (like, to the maximum extent you can, if you only need to deliver say 4 registers, deliver them from the two bitvectors that are most full, and don't scan the one that is emptiest).
• but one particular part of the solution is of interest going forward, namely that Apple draws a distinction between two classes of free registers.

There are, what you might call, “long-term free” registers which have been marked via the bitvector as free.
There are also what you might call “short-term free” registers (Apple calls these “returning physical registers”) which the ROB has just told the Register Allocator, are free. These are treated differently

with the register allocator (as far as feasible) trying to *immediately* recycle the "short-term free" registers so that, ideally, they do not have to have their setting in the bitvector toggled on, then immediately off again, and to avoid the more power-expensive operation of scanning the bitvectors.

multiple bitvector banks (2016)

The second stage of evolution we see in (2016) https://patents.google.com/patent/US10372500B1 *Register allocation system*. This uses a system that's clearly built on the ideas of 2012, but with additional techniques for saving power.

The idea is that, as I mentioned, we split our register file into multiple banks (the patent suggests 4, and I suspect that's still the case; it matches the results I have seen). The most important goal of this banking is to save power by trying to limit (as much as possible) activity to just some of the banks. You only need many physical registers under a few circumstances; much of the time you can do with far fewer. So we want to arrange things to, as far as possible, stick to using registers in as few active banks as possible, and only activate a sleeping bank when really necessary. The patent is about how we do that.

The details are very low-level, but the overall idea is: as far as possible, ensure that new register allocations (while building upon a scheme much like the earlier 2012 bitvector scheme) are bunched towards a single register bank. Sounds good, but how to do this?
• One part is fairly obvious, namely the use of a "short-term free" register queue, presumably on the theory that such registers come from register banks that are already powered up. (The usual situation, by far, is that the ROB is not especially occupied, and registers are rapidly used then freed, they normally don't hang around in the ROB for hundreds of cycles!)
• The second part is much more ingenious! It still uses the multiple bitvectors of the 2012 patent, but rearranges the mapping of these bitvectors into the register banks so that all the (now eight rather than six) scanners are likely to find free registers in the same single bank. You need to look at the patent to see what's done, but it is very elegant.

elide register allocation where possible (2017, not yet implemented)

The third stage of evolution would be some sort of resource amplification, one or more of
• late register allocation (virtual registers)
• early register release (as soon as there are no users, no need to wait till Retire)
• no register allocation (read the register value directly off the bypass bus). This can be done when architectural register values are immediately overwritten.

We know that the third option is in limited use for some cracked instructions, and is covered by the patent 2017 https://patents.google.com/patent/US10691457B1 *Register allocation using physical register file bypass,* which is discussed in more detail below in Register bypassing and early release. Today the only case where these are implemented is a few (not all) cracked instruction cases, but this patent suggests we may see these ideas in a future design.

xxx need to do some tests of cracked instructions here, plus analysis of numbers from dougall's throughput+counter pages

Power aspects of the register file

I'm always reluctant to blame an anomalous result on "power saving" because that's too easy, and can be used anywhere! At the very least before making such a claim, I want to have a valid model in mind of how the power saving might work.
However I think the glitch we saw here is indeed the result of power saving, in the form of the details described above.

Specifically what I am assuming is something like
- we have four register banks
- under the particular delay conditions that lead to the glitch, while the delay is active 3/4 of the registers (assuming 380 registers, that would be 285 registers) land up active and waiting in the ROB. Pretty much the cycle that the ROB clears is the cycle that allocation moves into the fourth and final bank. All allocation then occurs from this bank while the other three banks run in low power mode.

- but the timing is such that: remember I said that the times just balanced so that all the free registers were used up in exactly the same number of cycles that the registers were freed from the ROB... So I think what happens is simultaneously
+ the ADDs are allocating free registers from the one bank that is powered up
+ the ROB is freeing registers that are allocated to the three powered down banks
+ the registers that are freed by the ADDs (which complete very soon) are nonetheless still in the ROB (and not yet marked free) behind all the registers that were allocated during delay

so we get to a cycle where simultaneously
+ the last register is allocated from the powered up bank
+ no registers have yet been released to the powered up bank (though that's about to happen)
+ plenty of registers have been released, but they are all allocated to the three powered down banks

Presumably the system copes by powering up one of the powered down banks, and that takes two or three cycles, generating the glitch we see. In theory this might not actually be necessary -- waiting just one cycle would result in the first set of registered from that fourth, powered up bank, being released, but the machine can't look into the future!

There are still aspects to this that are unclear. Does this bank power-up delay always suspend register allocation for a few cycles like we are seeing here? Or under normal circumstances does the machine have heuristics that indicate it makes sense to begin power up before everything runs absolutely dry, and that power-up can happen "in the background" without requiring all register allocation to halt?

I’m open to alternative explanations for this glitch, but so far I see nothing I consider reasonable except the above.

Context switches

A single core will occasionally engage in context switching, sometimes within the same thread (servicing an interrupt), sometimes between threads in the same process, sometimes between processes. Is there any way we can reduce the cost of these?

dirty flags

Let's begin with the paradigm case, switching between threads in an app. The essentials of how this is done include
- creation of a new thread includes creation of a Thread Control Block, which includes a region of memory that will hold the registers of a thread when it is not running
- switching threads by the OS includes storing all the registers of the old thread to that thread's TCB, then loading all the registers of the new thread from its TCB

One way to reduce this cost is to have some sort of "dirty" flag that is cleared when a new thread is swapped in, and set as soon as the thread uses a register (or more generally, one from a set of registers). It was not uncommon, for example, to do this for either the FP or SIMD registers on the assumptions that many processes did not use these and so time spent context switching them could be avoided. The PowerPC Altivec VRSAVE register is an example.

hardware based register save/restore

Next up in sophistication is to delegate saving register context switches to the hardware. In principle this would involve steps something like
- the OS queries the hardware as to the size of a thread storage block (so it knows how large to make a TCB)
- the threadID/some thread-associated register is the address of the TCB
- a context switch would consist of something like four instructions
move specialPurposeRegister1 oldThreadID
move specialPurposeRegister2 newThreadID
switchOut specialPurposeRegister1
switchIn specialPurposeRegister2

Done correctly, this in theory allows the CPU to add additional state without requiring an immediate OS update, and allows the CPU to engage in the previous "dirty" flag optimizations without bothering the OS. It also allows optimizations like storing the registers directly to, say, L2 instead of wasting L1 cache space on data that will likely not be reused for many cycles (in principle an OS context switch could do the same thing via non-temporal stores, I don't know if any do).

In practice the one ISA that does this, x86, in the usual x86 fashion screwed up every possible angle of the implementation so no-one actually uses it.

apple's 2015 patent (not yet implemented?)

What Apple does is to unite and improve both of these ideas. (2015) https://patents.google.com/patent/US9817664B2 *Register caching techniques for thread switches.*
The basic ideas are
- there is some way (unstated, but presumably based on a special purpose register) to indicate the current threadID, and that a context switch has occurred (which could be as simple as changing the value in that SPR)
- after that point, under "ideal" circumstances the hardware will automatically save out old registers and load in new registers
- but it will do it on demand, not as one large block of time that stores all then loads all.

The idea is to add a new field to the mapping table that maps each logical register to a physical register so that the mapping says
x0 is mapped to physical register p3 with a tag of threadID.
Now imagine what happens right after a context switch. The first new instruction wants to read register x0, so it consults the mapping table. The mapping table tag does not match the current threadID so the following happens:
- we write out p3 as the value of x0, using the threadID tag and some offset algorithm to know where to write x0 in the TCB
- we load in x0 using that same offset, but based on the the new threadID, and we place that new threadID in the mapping table tag for x0
This is the basic idea, and it has as a consequence that writing out and loading in the new register values will be spread out over time, hopefully mostly interleaved with real computation.

There are a number of details to make this more performant.
- First is that these tags may be associated with more than one register. It would make sense, for example, to have a single tag for two integer registers since loading or storing two is no more expensive than loading or storing one. Presumably you pair these based on statistics about what registers tend to be used together.
- Secondly the stores are straight to L2, without wasting space in L1. It's unclear if this is done at cache line granularity (ie, as an extension of the first point above, aggregate under one tag enough registers to pack a cache line, and store these out sequentially filling a single cache line transfer buffer); or perhaps at some smaller granularity that's an optimal match to the bus width between L1 and L2 (maybe 32B quarter of a cache line?)
- Third the threadID tags are in physical address space, not virtual address space. This I assume makes life easier for the OS, and allows a minor advantage of not having to perform TLB lookups (and perhaps some other slight streamlining relative to a normal load/store). It does make me wonder the nature of the connection between L1 and L2 and how partial lines are handled. This use of a physical address

also mean the OS has to be careful about these TCB pages; they can't be swapped out or compressed or similar shenanigans!
- Finally there's an additional flag that is set when a register is modified (ie a dirty flag) to be used in the obvious way, to avoid having to save an unmodified register.

Note also that this means the CPU is capable of doing something very strange, namely creating synthetic instructions, with no warning (and allocating whatever resources they might need) in the middle of the pipeline! This is something I've never heard even remotely considered in any other CPU, and Apple has provided zero patents on it, at least that I've found. These synthetic loads and stores (required to write out the old thread's version of a register and load in the new thread's version) are close enough to normal instructions that they fit nicely into the operand dependency model we're about to discuss. The machine does not stop while the registers are brought up to date, instead the instructions that depend on say x0 are given additional dependencies that they cannot proceed until these synthetic register load/store instructions have completed, but other OoO behavior can as usual route around them. Much later we will see another version of this creation of synthetic instructions when we look at Apple's (currently very limited) use of value speculation.

Note how nicely general this scheme is. It can (not necessarily unlikely, for the SIMD registers) allow a SIMD register to persist, unreferenced, through multiple context switches and still be there when the initial threadID runs again. It can also easily be adapted (with an appropriate interrupt "threadID", perhaps NULL) to interrupt handling, allowing handlers to use whatever registers they feel like, and have these transparently saved as appropriate then reverted on return from interrupt.
The one place you have to be careful is: what if a thread moves to a different core? The patent suggests having (Apple custom) instructions that do things like force flush all the modified registers to the TCB.

The scheme as I have described it is general and could be used for all the registers that are stored during a context switch, which extends to the NZCV flags register, the FPSR floating point status register and possibly various SPR's. As a practical matter, the SIMD registers (and the AMX registers?) are the most likely immediate candidates. My intuition says that even the integer registers would benefit, especially, as I suggested, for interrupts, but who knows *exactly* what Apple has implemented

It should be pointed out that I have engaged in some correspondence with an author of the 2015 *Register caching techniques for thread switches* patent. His belief (though he cannot be sure because he left Apple a few months before the patent was filed, let alone any hardware released) is that while the idea is valid and will work, it has not yet been implemented in an Apple core.

It’s interesting to compare this scheme (perhaps not yet implemented) with the register security tag scheme I described at the very start of this document., and which does appear to be implemented. The security tag is a much simpler mechanism than context-switching, but is a very nice half-measure. It requires a similar tag associated with each entry of the architectural to logical mapping, a validation of that tag on every access, and machinery to update the tag under certain conditions. But it does not require the subsequent steps of automatically writing registers values out to DRAM, or reading them in

from DRAM.
Hence it would be not crazy to imagine that the context switch mechanism might eventually be implemented, by expanding the existing security mechanism.

apple's 2019 patent (reduced context)

Paired with this we have the related but different (2018) https://patents.google.com/patent/US20190220417A1 *Context Switch Optimization*.
This is a very strange patent, meaning I have to be rather speculative in trying to understand it. The basic idea is very easy: suppose that Apple defines, for some processes, an alternative slightly simplified version of ARMv8 that uses a subset of the registers, for example it uses half the integer and one quarter of the SIMD registers. Defining this is not too hard; it's mainly small modifications to the compiler. But if we do this (for whatever reason) and rely on it, then we want the CPU to flag when we don't follow the rules. Thus we define a reduced processor context (the set of allowed registers), a register (swapped on context switches) that states whether the processor is operating with access to all registers or (possibly more than one alternative) reduced processor context, and some minor machinery to generate an exception when an inappropriate register, for the reduced context mode, is touched.

Now why bother to do this? Hypotheses:
• the 2015 patent sounds good, but if it iss not yet implemented, then the actual cost of context switching remains a notable block of time one might want to reduce. If the OS can define many of the ongoing background processes as reduced context, it can save context switching time.

• even if the previous patent is in place, and even if you modify the compiler to use fewer registers, without some sort of explicit contract, the OS has to allocate a full-sized TCB to hold all the ARMv8 ISA registers. If we want to save some space (again for these many background processes and OS threads) because we know they have minimal register requirements, it's not enough to just compile them with fewer registers, we need to also have the CPU's automatic register saving mechanism incapable of overwriting the bounds of a reduced context TCB.

relevant for chinook?

• (most speculative) this has nothing to do with either Firestorm or Icestorm! Few people know that, along with Apple's large and small cores, there is at least one other core they have designed, a tiny core, which is the controller for things like the GPU, NPU, ISP and suchlike. We know that the codename for this core in the A12 was Chinook, and that it is an AArch64 core (apparently with NEON) at the same architecture level as the A12 (eg it provides Pointer Authentication). It would be reasonable to assume, given how Apple has designed the small vs large cores, that this is mostly a "parameterized" version of the basic large core design (ie same overall OoO structure, branch prediction, scheduler, etc) just with even fewer of everything than the small core.
With that in mind, consider that Icestorm has ~80 physical integer registers and ~88 physical SIMD registers. Maybe Chinook has the bare minimum the design can support without locking up (?34 of each or whatever), and is most performant when actually locked into a subset of these, using 16 integer

and 8 SIMD or whatever?
The justification for this wild speculation is

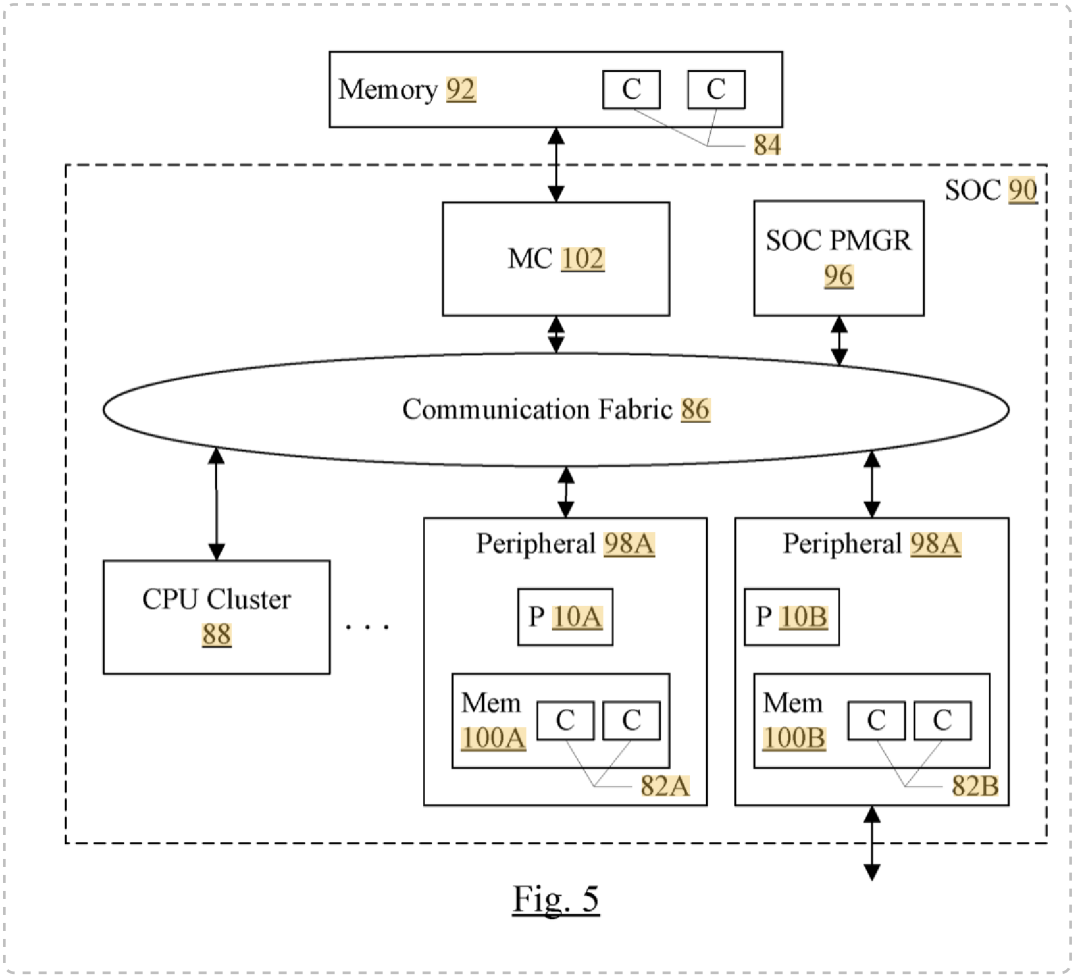


Fig. 5

with accompanying text (removing some irrelevant parts)

- The workload of the processors 10A-10B may be characterized as having more frequent context switches than the workload of the CPU processors in the cluster 88. In some cases, the context switches may be much more frequent (e.g. one or more orders of magnitude more frequent). Additionally, the workload of processors 10A-10B may also be characterized by infrequent, but non-zero, use of one or more data types specified in the ISA. For example, in an embodiment, the workload may include infrequent, but non-zero use of vector registers. Accordingly, reducing the context saved and restored in the processors 10A-10B may be significant in terms of improved performance, reduced power consumption, and memory footprint...

The size of the local memories 100A-100B may be limited, e.g. compared to the memory 92, and storage in the local memories 100A-100B may be used for other data besides the contexts 82A-82B, so reducing the context memory footprint may improve performance as well since more local memory space may be available for process data other than context save data.

- The peripherals 98A-98B may be any set of additional hardware functionality included in the SOC 90. For example, the peripherals 98A-98B may include video peripherals such as an image signal

processor configured to process image capture data from a camera or other image sensor, display controllers configured to display video data on one or more display devices, graphics processing units (GPUs), video encoder/decoders, scalers, rotators, blenders, etc. The peripherals may include audio peripherals such as microphones, speakers, interfaces to microphones and speakers, audio processors, digital signal processors, mixers, etc.

In other words, although this may seem initially like crazy speculation, I think it's reasonable to interpret the patent as essentially confirming
• the existence of Chinook
• that it's used as a general purpose controller all over the SoC
• that it's essentially a further scaled down Apple small core, not a separately designed bespoke core
• and that Apple makes it "effectively" smaller by things like careful ABI.

It could be argued that any sort of AArch64 OoO design is far more than what's needed for many of these tasks, but one has to wonder if that's 1990's thinking. The Apple solution may use a little extra area (cheap) but in return delivers
• security (this isn't a second class core; it gets the full security treatments of all the other Apple cores including things like TLB protections and PAC)
• easy integration with existing developer tools, compiler, and the OS
• enough performance that engineers can spend their time worrying about how to make the entire device better, rather than worrying about how to make an ARM M4 do whatever needs to be done.

Register bypassing and early release

We've described multiple ways that Apple can make the physical register file appear larger. But they haven't taken the next step in doing so, namely using virtual registers or something similar (for late allocation) and allowing for early register de-allocation.
They clearly have thought about the issue, as in this patent: (2017) https://patents.google.com/patent/US10691457B1 *Register allocation using physical register file bypass*.

Consider an instruction sequence like `REV x0, x0; CLZ x0, x0; ADD x0, x0, x1`
Note two things.

- First the data that is transferred from the `REV` result into the `CLZ` input does not need to ever be stored anywhere – there is no way to access it after the `CLZ` executes. So why not omit allocating a register, and just have the `REV` read the value from the bypass bus?

- Second the `ADD` overwrites the value of the x0 output from the `CLZ`, so even if we did allocate a physical register to the result of the `CLZ`, why not just deallocate it right away because, once again, no code can access it after the `ADD` has overwritten logical x0?
This gives us two different techniques for requiring fewer physical registers, one avoiding the allocation of a register by use of a tag, the other allowing for early deallocation of a physical register.

These are both nice amplification technique -- not incredibly powerful, but also not especially difficult.

However a test on the M1 suggests that, in spite of the patent, neither appears to be implemented, at least not as of the M1.
The test sequence I used above is drawn from the patent (so you'd hope it would demonstrate the effect, if anything!) However, to simplify the problem I also tried the easier test sequence (no complications as to which execution units different instructions can execute in)

```
ADD x0, x5, x5; ADD x0, x0, x0; ADD x6, x7, x7; ADD x6, x6, x6;
ADD x8, x9, x9; ADD x8, x8, x8; ADD x10, x11, x11; ADD x10, x10, x10
```

As expected this will run at 6 instructions per cycle, but more interesting is, if we add in a delay, how many instructions will execute before the jump (ie how many registers will be allocated). Note that these pairs all satisfy the conditions of the patent – the instruction result is generated to a register which is immediately overwritten by the next instruction. Even so, we see the jump at the usual ~380 instructions, no difference in the apparent register file size.

One might ask why this is not implemented, especially given that, as we will see, an equivalent is implemented for load/store?

xxx test/explain the cracked case as seen by the performance counter numbers

Before answering this, note is that there *are* apparently cases where cracked instructions [like `ADD (shifted)`] are implemented as
- two instructions where
- the second instruction reads the intermediate value straight off the bypass bus without allocating a register, as we discussed earlier.

So why not implement the same mechanism more generally?

I suspect the issue with using tags as the patent describes is that you *also* need a way to force the two instructions to schedule together – it doesn't do you any good to have the second instruction plan to read the value off the bypass bus if it isn't scheduled *immediately* after the first instruction. Presumably this tying mechanism exists for the case of cracked instructions, but Apple hasn't yet implemented something allowing generic tying of instructions? Ideally such a solution might be part of a more generic fused instruction mechanism, which is one reason it might be delayed.

Similarly, consider the failure cases for virtual registers, eg what if, at the point of instruction execution, you discover there is no free physical register available? It's not an impossible problem, but once again it requires some support machinery that won't exist until you add it.

What about implementing the alternative of early register release?
In that case, my guess is the fly in the ointment is the last physical register scheme, a patent I have already referenced, (2019) https://patents.google.com/patent/US20210064376A1.
Getting that scheme to work along with early deallocation is not impossible but, once again, it's one more thing that needs to be figured out. It's certainly possible, perhaps even likely, that the initial machinery for these various ideas is in the M1/A14 hidden behind chicken bits, but ready for later

release when they're considered 100% working.

How rapidly can the ROB retire instructions?

We still haven't nailed down everything register related!
We know that registers are released at Retire (no early release) but how rapidly are they released at Retire? The minimal rate must be 8/cycle (to maintain 8-wide throughput) but could it be larger?

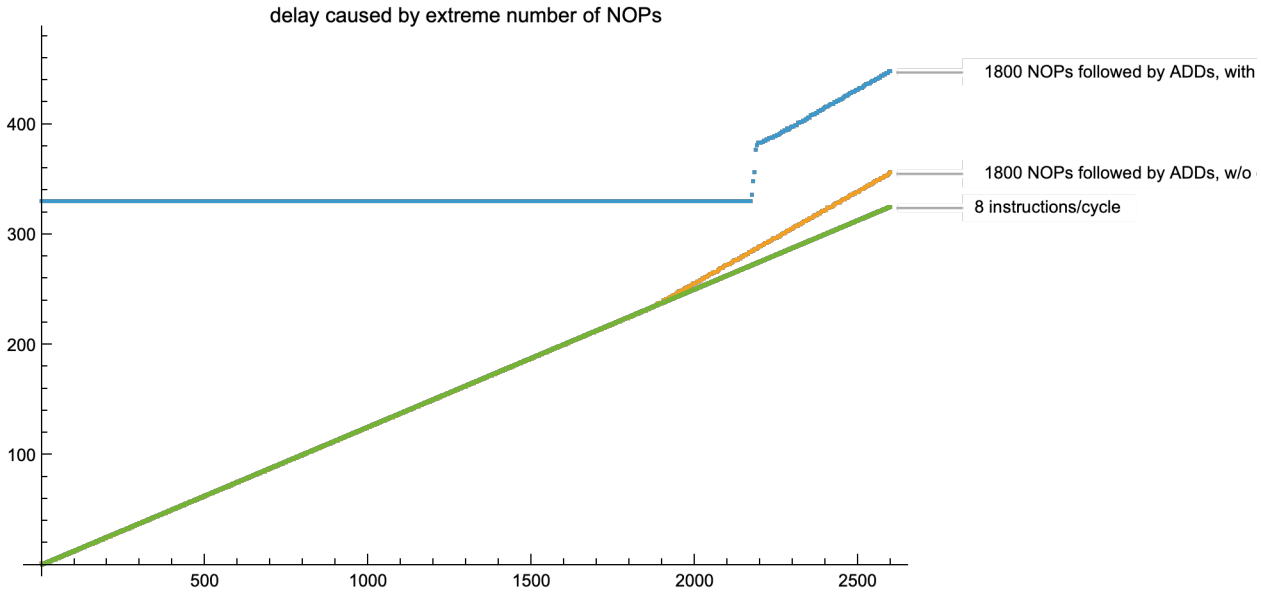
Why do we care about this? If we're waiting for an integer register to be released when head of ROB clears, what's the problem if 8 integer registers are released per cycle, given that integer execution can only use 8 integer registers/cycle? The issue is that after a long delaying head of ROB (like a miss to DRAM), there are many instructions, all blocked on different things.
Yes, there are some integer instructions blocked on an integer register. But there may also be some fp instructions blocked on fp registers, likewise for flag registers. And there may be load or store instructions blocked waiting for load or store queue slots. So ideally when the head of ROB clears, we'd like to run through the ROB as fast as possible, freeing resources as rapidly as possible, so that as much stalled activity as possible can resume as soon as possible!

retiring NOPs (56/cycle)

Given this insight, let's start with an easier case.
We begin with a delay block consisting of our usual `FSQRT`, followed by a large number (~1800) of `NOP`s, followed by instructions that use up all the int registers then wait on them.
The question of interest is how long it takes to clear the `NOP`s (ie clear out the head of the ROB) before the `ADD`s can start work.
So the structure is
- delay block of 33 `FSQRT`s (33*10 =330 cycles; needs to be long enough to be sure everything is packed into the ROB and scheduling queues!)
- delay block of 1800 `NOP`s (should take 1800/8=225 cycles, to pack the ROB behind the `FSQRT`s)
- resource depletion block of 370 `ADD x0, x5, x5` that should use up all the physical registers
- probe block of variable number of `ADD x0, x5, x5`

The probe block should not be able to run until more physical registers become available, which cannot happen until both the delay block ends *and* the NOPs are cleared.

Out[564]=

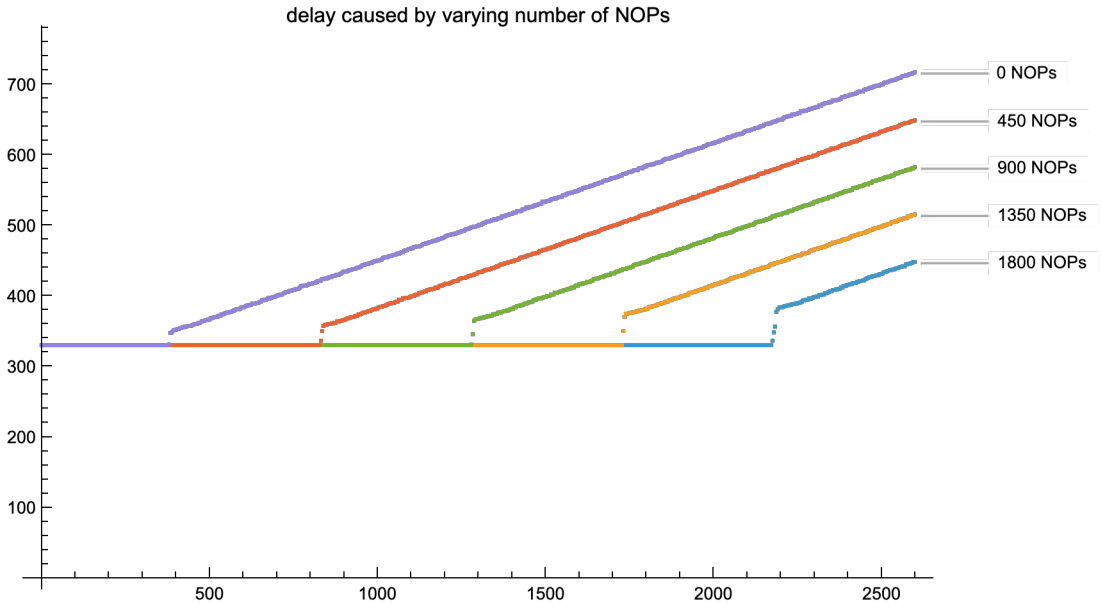


In[•]:=
```
(* {2172,330},{2176,336},{2180,348},{2184,356},{2188,377},{2192,381} *)
```

Well that's nice and clean -- once you figure out the correct probe!
The gold curve is, of course, the non-delayed version. Easy to see the break in the slope as we switch from 8/cycle `NOP`s to 6/cycle `ADD`s.
The obvious fact about the blue curve is that the jump is at ~2180=1800+380, exactly where we'd expect (stall once the `ADD`s run out of physical registers).
But more significant is that the ramp jump is about 53 cycles. Of course much of that could be because of xxx time, ie the time spent waiting for the `FSQRT`s to complete, which is not interesting. What we care about is the time spent clearing the ROB after the last `FSQRT` completes.

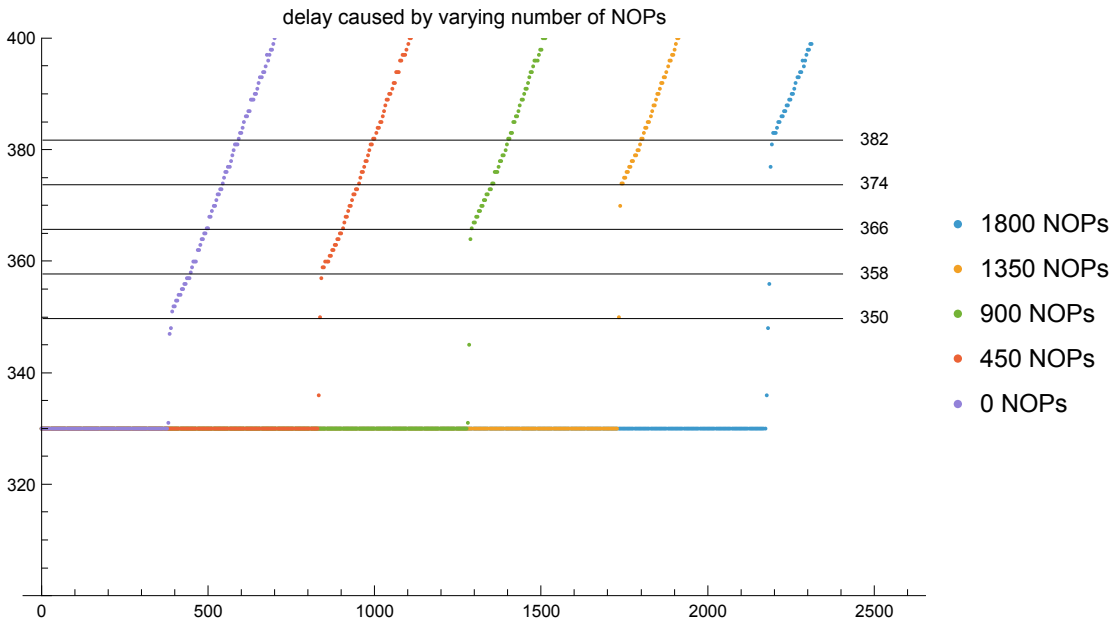
We can get better insight into that by varying the number of NOPs.
Consider the image below, where we use the same structure, but vary the number of NOPs from 0 through 450, 900, 1350, to 1800.

Out[569]=



There's clearly a variable delay (look at the size of the jump) from when the machine stalls until when ADDs start being processed again, and that stall grows as the number of NOPs grows.
What's the exact relationship? Let's zoom into the relevant part of the image and add some guide lines.

Out[572]=



This image looks messy but really all we have done is zoom in on the interesting part of the original image (the jumps) and overlay some lines.
There's room for disagreement, but to my eye the lines are plausible cycle values at which the ramp of each curve starts.
The interesting point is that the lines tell us the amount of delay (the time it takes to again start process-

ing ADDs from when the machine stalled.
There is a baseline delay of 20 cycles (even with no NOPs). More interesting is that the additional delay increases by 8 cycles for every 450 NOPs.

So the time it takes the ROB to clear 450 NOPs is 8 cycles, ie the machine clears 450/8=56 NOPs/cycle!

structure of the ROB

We will see evidence for this later as we cover load/store and branches; but in fact the ROB is best thought of as consisting of ~330 “rows” where each row can hold 7 instructions. Most instructions can go in any slot, but “failable” instructions must go in the last slot (which provides extra storage). Failable instructions are those that might require the CPU to flush and restart. These can be branches (mispredicted...) or loads/stores (possibly some exception conditions, but the main case I am thinking of is a load/store dependency misprediction when a load occurs out of order relative to a store and so reads possibly invalid data [all to be explained later].

The usual structure of the ROB, when not running weird test code, will look something like maybe two or three instructions in a row, then a load at the end of the row, then maybe five instructions, then a branch terminating that row, and so on. The ROB slots are cheap, so Apple is happy to give us thousands of them! The real constraints in real code will be either load/store/branches filling up the ~330 failable ROB slots, or int/fp code using up all the HF slots.

So a better way to think of Retire is that Retire can clear 8 ROB rows per cycle. This can ultimately mean clearing as few as 8 instructions (if we had eg a sequence of eight successive loads, or some combination of successive loads, stores, and branches but nothing else). Or it could mean as many as 8*7=56 instructions if every one of the eight rows was fully populated.

A slightly more sophisticated way to think of this is that the ROB is required to track one part of recovery from speculation failure (branch misprediction or load/store dependency misprediction) while the History File (and checkpoints) track a different part of this recovery. In other words, as I keep stressing, only the failable slot of a 7-entry row of the ROB actually matters; the other 6 slots just hold “inert” entries from instructions giving their ordering, but they are never used.

Hence an obvious question is: why even have the other 6 instruction slots? We can treat the ROB as ~330 entries which hold a pair (failable instruction, 3-bit counter) and the 3-bit counter describes how many instructions would have been in the other 6 slots.
(Why not allow then allow 1+7 entries instead of 1+6? Presumably the case of the counter as all 1’s or all 0’s is treated specially in some way, eg as a synchronization of some sort? or that case means the entry is invalid?)
This scheme is not visible to an outside tester, but seems a reasonable way to save a few transistors and a little energy.
This refinement was suggested to me by Arthur V.

freeing up registers and HF slots (16/cycle)

That's fairly impressive, but clearing `NOP`s is easy. What about a more difficult task like restoring registers?

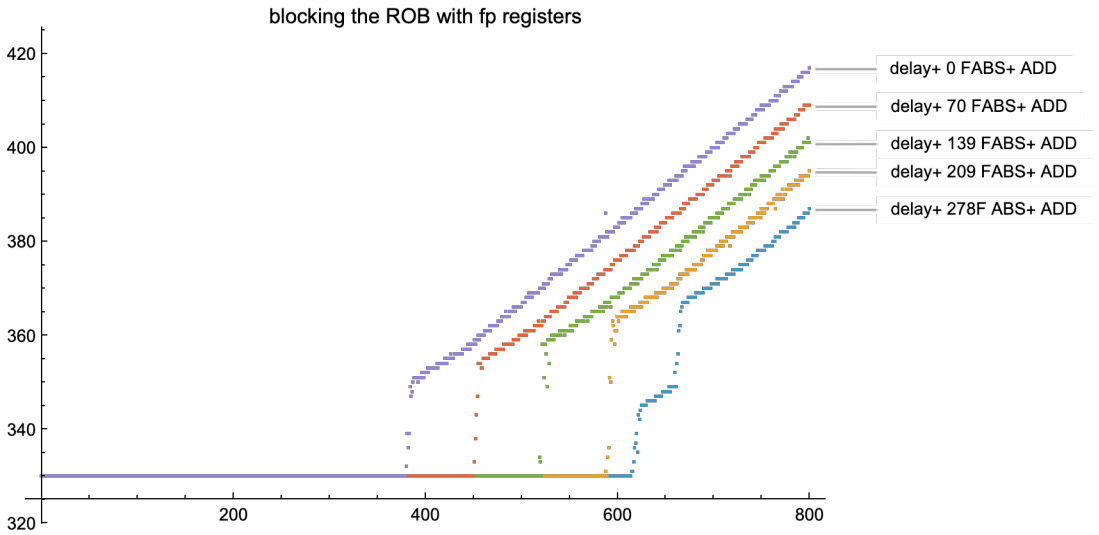
The plan now is to replace the `NOP`s with `FABS`, which will use up slots in the History File. How many registers can be freed per cycle?

We want to use enough FABS to make the phenomenon visible, but not so many that we flood the History File with floating point register manipulations and don't leave space for all the ADDs. Using a maximum of 278 FABs seemed to do OK.

If those can be cleared at 56/cycle, clearing the whole 278 will take 5 cycles.

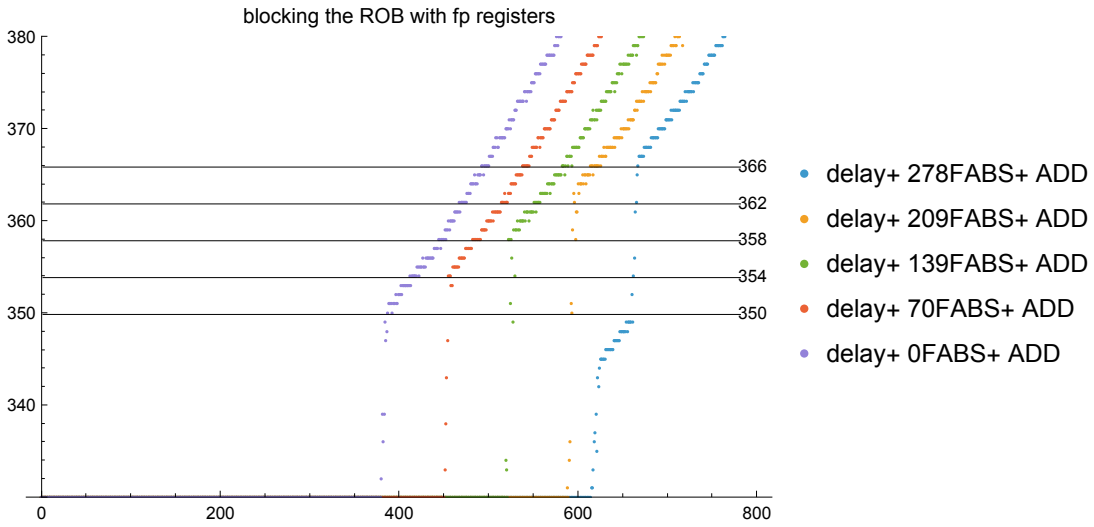
Hopefully we can see that against the 20 cycle baseline (and the noise that will arise from using FP for both this timing and the delay.

Out[581]=



Well clearly the delay is a lot longer than 5 cycles! So let's use the same "step through quartiles" trick we used for NOPs, and zoom in.

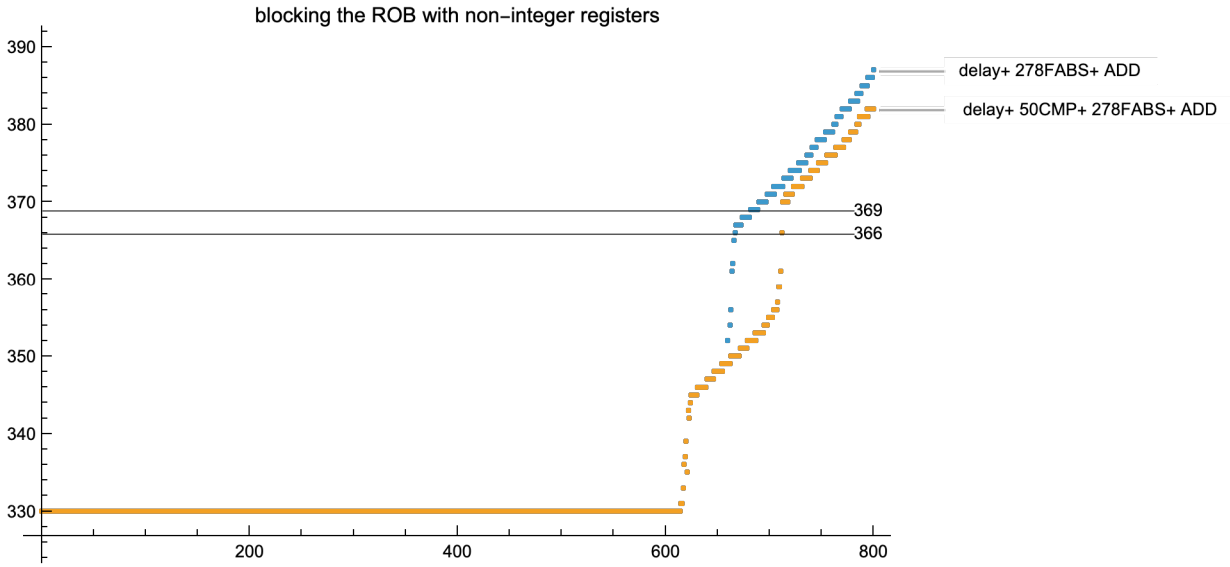
Out[584]=



I think it's legitimate to see clearing 70 `FABS` from the ROB as taking 4 cycles, so 17.5 clearances/cycle. Probably best to read that as 16 per cycle.

We can attempt to validate the above hypotheses by prepending, before the 278 `FABS` that use up physical FP registers, 50 `CMP`s that use up physical flags registers. If our hypothesis is correct, this should result in an additional delay of another three cycles (50/16=3).

Out[587]=



We see the point is validated; an additional delay by what can plausibly be viewed as an additional three cycles.

Obviously the methodology I am employing is not great for incontrovertible, exact measurements! But I'm interested here in large-scale exploration to figure out the overall design of the system; I'll leave it to others coming later to perform the high precision experiments.

Experts might want to know why the curves show two jumps. Consider the blue curve, and compare with the previous graph.

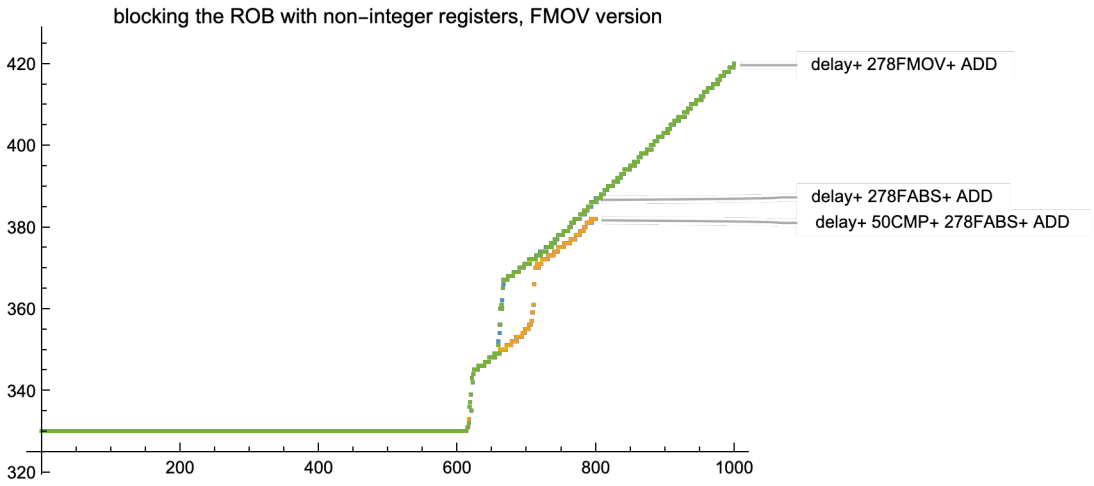
The ADDs face two possible constraints: physical registers and HF slots.
When there are not too many FABS enqueued (the cases of 70, 139, 209 FABS above), then the ADDs run out of physical registers first and block.
But when there are enough FABS enqueued (the case of 278 FABS) then the first constraint that is hit is running out of HF slots.
After the FSQRTs clear the FABS start to clear, and HF slots are released. This allows ADDs to restart for a few cycles – until they now run out of physical registers. The FABS HF slots were releasing FP, not integer physical registers.
The CMP case is an even stronger version of this. Think about it. In the CMP (gold curve) case at any given N value, 50 fewer ADDs have been enqueued than for the blue curve case. We still run out of HF slots at the same point because CMP, FABS, and ADD all use up slots from the same HF pool; but once HF slots start to be freed, there are 50 more integer physical registers available than in the blue curve case, hence the intermediate run before the second delay (running out of integer physical registers) can proceed until N is 50 operations higher.

freeing just HF slots, not registers (also 16/cycle)

There's one more thing we need to test.
To clear entries from the ROB to eventually free up those integer registers, we have to
- clear entries in the HF, and
- release registers to the free list.
It's possible that these two tasks take a different amount of time (for example releasing HF slots happens rapidly, but then moving registers onto a free list so that they are available for reuse happens at a slower rate.

So let's try a variant that doesn't involve freeing registers, just HF slots.
Replace the FABS with `FMOV d31, 30.` The idea is that while clearing a FABS entry from the HF involves freeing an actual physical register, almost every FP HF entry only records a change in the mapping tables; it doesn't actually change a register to be free to move to the free list. The idea is to test if manipulating physical registers in Retire is slower than manipulating HF entries.

Out[589]=



As you can see there is no difference!
It appears that HF slots and registers are both freed at 16/cycle, there's no faster path for HF slots that are not freeing a physical register.

so how close is the M1 to a KIP (kilo-instruction processor)?

Just as a fun exercise, how close is Apple to a KIP (kilo-instruction processor)?
A KIP has been the CPU designer's dream since ~2000CE, ie a design that can maintain "in progress" one thousand instructions, the idea being (at the time the phrase was coined) that this would be enough usually to keep a CPU from stalling even if a load missed all the way to DRAM. Unfortunately as CPUs have become higher GHz and wider, 1000 instructions is no longer enough (if a miss costs you, say, 330 cycles, and you're 8-wide, you'd like to be able to power through 8*330 instructions before stalling), but the number is still a good milestone.

Obviously Apple is way beyond this 1000 limit in trivial fashion (sequence of NOPs) but what about real code?

As explained, the ROB size itself is not a problem, it's various other structures that are more difficult to scale up (eg LSQ, physical register files, history file) and we have seen how Apple has continually disaggregated or rethought these into different structures that scale better (perhaps parallel structures like the banked physical register file, perhaps move the most difficult to scale part out of a structure as we will see with the splitting of the traditional Load Queue into the LEQ and the LRQ).
Given all this, if we want to pack as many "real" instructions into the ROB as possible, what are the limits?

The first is that anything that changes a register will use a History File slot, and we have ~620 of those. But we also have some instructions that don't change the register file, most notably stores and branches. Unfortunately stores and branches (and loads) use the same "failable" slot of the ROB, even the simplest non-problematic and non-conditional branches like "B .+4", so we might as well just use stores (since branches are also limited by other data structures beyond the number of failables).

If we use the probe (`ADD x0, x5, x5; FABS d0, d0; STR x5 [x2]`) we can in fact reach about 310 iterations of this before he see a jump!
So ~930 realistic'ish instructions! Not bad! It might be possible to go slightly beyond this with a few other instruction classes; I didn't push the issue.

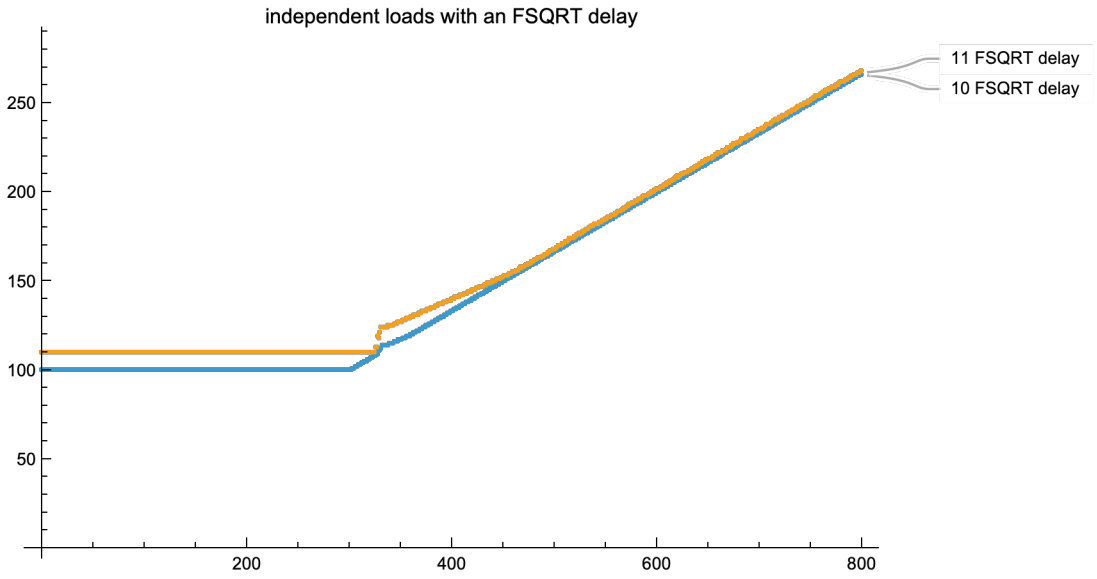
Loads and Stores

Experiments to test LSQ sizes

first attempt at testing queue sizes (FSQRT based)

We' ve so far explored some aspects of the ROB, and the size of the physical register files. Now let's explore the size of the load/store queue.

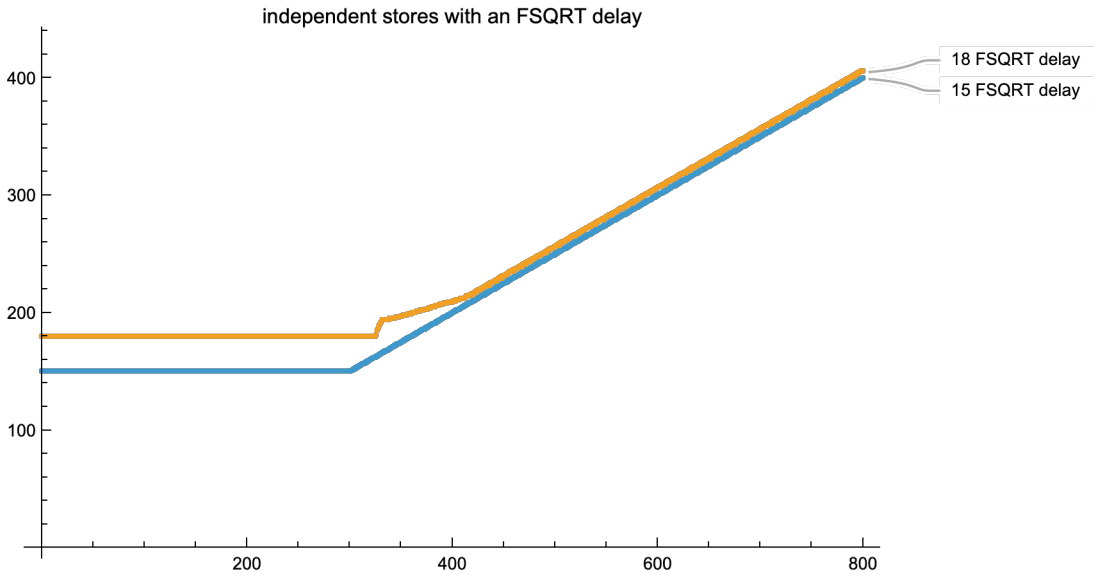
Out[592]=



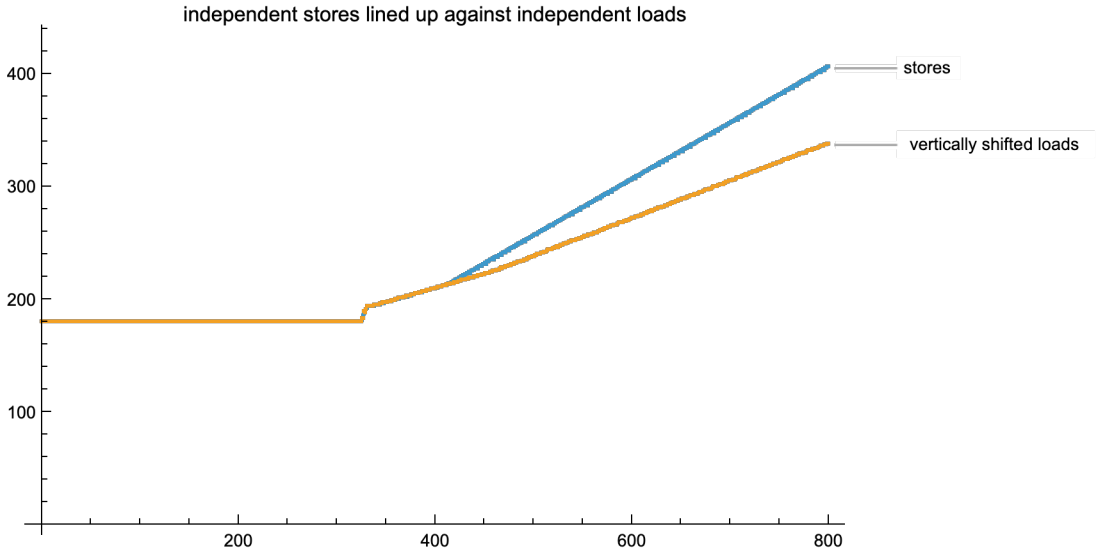
The gold curve gives us the immediate info of interest, a jump at ~330, which would suggests that the LSQ can hold 328 loads. But there are unexpected issues here!
- First is that the limit seems suspiciously close to the size of what I have called the "failables" part of the ROB; ie the limit looks like how many loads we can hold *in the ROB*, not in the LSQ.
- Second is that we don't seem to actually lose any cycles! Yes we have the jump at ~330, but that's followed by a regime where we appear to be executing loads at 4/cycle rather than 3/cycle, till we get back on trend!
So there's something weird going on.

Do we get the same behavior for stores?

Out[595]=



Out[597]=

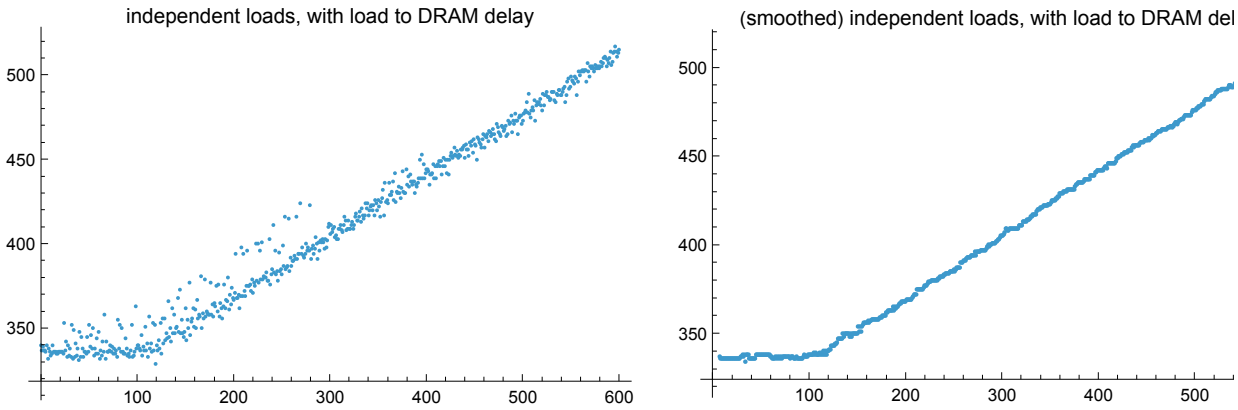


We see exactly the same pattern for loads and store (ie
- same jump at ~330,
- same running at about 4-wide till N=~420).
The difference in slopes after that arises from M1 having
- 2 load units
- 1 store unit
- 1 ambidextrous unit
so that it can run either 3 loads/cycle or 2 stores/cycle.
These results seem ...unlikely... They indicate a massively large load-store queue, same sized for loads and stores.

second attempt at testing queue sizes (miss-to-DRAM based)

Let's try a different tactic. Instead of using our trusty FSQRT as a delay block, we'll use a delay block based on loads that continually miss to DRAM.

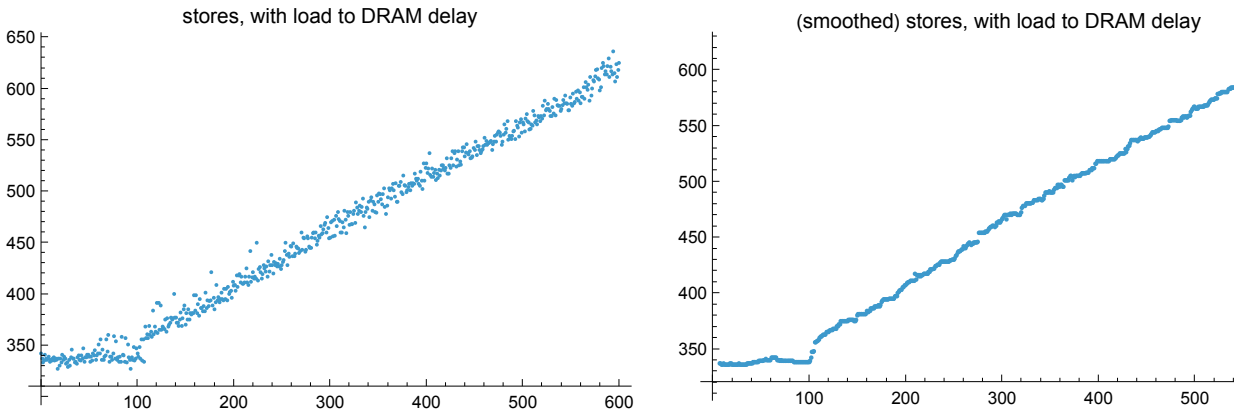
Out[601]=



As you can see, the problem with using a load delay is that there's a lot of noise in the signal, and while we can reduce this (at the cost of much longer runtimes) we'll be cheap and just use a smoothing filter (moving median) to make it easier to see the point. And the point is that, with this delay, as opposed to with an FSQRT delay, there's a discontinuity at N=~125!

We see the same pattern with stores, in this case with a discontinuity at around 100 stores:

Out[605]=



So we find ourselves with the following conclusions:

- there appears to be load queue, of size ~125 loads
- there appears to be store queue, of size ~100 stores
- but these do not behave like our previous constrained resource tests!

Clearly we need to understand exactly what the load/store queue(s) do, how they work, and how they

can be optimized.

Theory of the load store queue

The load/store queue is a large, complicated subject. Even more so than the rest of the CPU, there are technicalities in this area that are specific to x86, or specific to Intel, but which are not universal constraints on how things have to be done. Be open-minded!

why do we need a store queue?

First the store side: Why do we need a Store Queue at all?
Once we have a speculative CPU, this means that we are going to be executing *speculative* stores. But we cannot allow such stores into the cache for two reasons, one obvious, one slightly less so.
- The obvious reason is that if you overwrite a value in the cache, then discover that the path of execution was misspeculated, what can you do? You can't undo the write and recover the previous value that you overwrote!
- Less obvious is that once you write to the cache then, (more or less), that value, although still speculative, becomes visible to other CPUs via snooping, and once again, you can't undo whatever those CPUs do in response to that snooping, even if you want to undo the write.

So we need somewhere to hold each pending store until that store becomes non-speculative.
It is traditional to make this storage a queue, with the oldest stores at the end and newest at the beginning. The reason for this temporal order is that the code may very well generate multiple stores to the same address, so you need to be careful about the ordering of stores (and also, as we will see, of loads).
Remember that our CPU is out of order...
So imagine code that looks like this
store xM into xA
...
store xN into xB+xC
...
It is possible that xA=xB+xC. It's also possible that one of these stores may *execute* in reversed order, eg maybe xA is generated by a slow load, whereas xB and xC are already available in their registers. If we don't get ordering correct, we'll store xN, then overwrite it with xM, ie the reverse of correct program order...

So the basic idea becomes clear. To ensure correct store behavior given both speculation and out-of-order execution, we want something like
- a time ordered queue to hold stores, for which
- each slot looks something like (store address, store data, various flags [valid, free, etc])
- queue slots *allocated at Rename* (ie at a point where the instructions are still in order)

And the most basic flow looks something like

1) allocate a store queue slot at Rename
2) store sits in the issue queue for the LS unit until *both* its (store address and store data) dependencies are satisfied
3) (store address, store data) are written to the queue slot
4) ...at some later point... the store address has to be tested against the TLB to make sure there are no permission issues, no VM fault, etc
5) the store instruction reaches the head of the ROB. Assuming the TLB issues didn't generate a fault, the store can complete.
6) ...at some even later point... the store data is written to cache, and the store queue entry is freed.

Pretty much every step here has interesting non-obvious wrinkles, and we'll return to them, but for now accept this framework.

why do we need a load queue?

Now think about loads. Why do we need a load queue? Well think about this:
We have a bunch of pending data in the store queue, and loads whose addresses match those earlier stores have to get their data from the store queue, not the stale data in the cache.
And they have to get the *correct* version of the data, the *latest version that was stored just before the load*, from the store queue.

It is for this reason that we generally talk about the LSQ as a single queue that holds loads and stores. This becomes clear when you think of loads, and remember that loads can also happen out of order relative to each other, and relative to stores.
The idea, then, in this most basic model, is that a single LSQ has slots that hold both loads and stores, allocated in program order at Rename. Then,
- when a load executes, it scans the LSQ backwards in time from its position, looking for the *nearest* store slot with a matching address, and reads that data.
- when a store executes, it scans forward in time, looking for *every* load slot (there maybe more than one) with a matching address, and feeds that slots its store data.

This model has to deal with various dependencies you may not have thought of that have to resolve before load or store execution can proceed.
For example:
Suppose that (once again, out of order!) you're a load looking along the queue backwards in time to find a matching address. Maybe some of those stores
- have not executed yet,
- so there are slots that you know are store slots,
- but the address has not yet been recorded.
What to do? In the model we have described so far, we have to delay execution of the load until we can be sure that *every* earlier storage slot has its address attached, so that we can be sure we read the

correct data from the store slot if there is a matching earlier address.

So, then, the basic load instruction cycle is something like
11) allocate a load slot, in order, at Rename, from the common LSQ
12) load sits in the issue queue for the LS unit until its (address) dependencies are satisfied
13) load scans backwards in time along the LSQ looking for matching addresses.
14) if there are any holes in that scan (ie stores with unresolved addresses), wait till they are filled
15) if there are matching addresses, grab the data from the nearest match, otherwise from the cache
16) ... at some point in the previous 3 steps, validate TLB/faulting/permission issues...
17) load instruction reaches head of the ROB, and its LSQ slot can be freed

Note also that we now realize we needed to include a (3a) step for stores, involving scanning the LSQ forward for matching load addresses.
And we're just getting started!

speeding up the basic design

So with this pool of ideas in place, let's now consider various issues.
Of all the complications, the biggest is the problem of loads and stores possibly having the same address. Forcing loads to delay for every store loses us a hefty fraction of our OoO performance. And, of course, most of the time loads and stores are to different addresses! We are making the machine run a lot slower for a case that's very rare – but has to be handled correctly when it happens.
Specifically:
• if we know a load doesn't match an earlier store (and the same in reverse, a store doesn't match a later load) then we don't have to worry about this business of making sure data goes to the correct additional slots, we can just do the obvious thing – for a store, hold onto the store data in the store slot; for a load, read the data from the L1 cache.
So we'd like to know about a load/store address collision ASAP.

split store address from store data

So this leads to the first, easiest part of a a better solution: realize that a store may be able to resolve its *address* long before its store data is known.

If you can attach an address to a store slot, OK, any load that reads from that address will have to delay until the data is available. But all the other loads (and this is most of them) with different addresses can go ahead, all they care about is that *an earlier slot is not relevant to them*, and won't acquire an address *that becomes relevant to them*.
You are probably well aware that, for example, x86 splits stores into two operations, one corresponding to the store address, the other to the store data, and each is separately scheduled. Every performance CPU now does the same thing conceptually, though they may implement it rather differently.

predict load/store dependency

Alternatively, or in addition to, separating the store address from the store data, you can build a **predictor** that predicts whether a load is likely to depend on a recent earlier store. Then what you can do is run step 13 as before, but if step 14 shows missing store addresses, consult the predictor and either wait or just assume there won't be a relevant store that later occurs, and go ahead.
This is, of course, a new form of speculation ("load/store" or "address alias" speculation) but fortunately it can be handled by our existing machinery like the ROB. What's necessary is to validate that when every relevant earlier store slot has its address eventually filled in, that address did not collide with the load that we (LS-speculatively) executed and whose data we (LS-speculatively) pulled in from the L1 cache.

If there *is* an address match, we
- mark the load in the ROB as misspeculated,
- update the LS predictor,
- flush the bad load and every subsequent instruction,
- restore state, and start again; much like a branch misprediction.

(Note that I was a little quick above – if we're clever, we can actually compare the later stored data to the data used by the load from the cache. If they are the same [and you'd be amazed how often they are, many many stores are the same thing repeatedly stored to the same address] then no harm no foul! No need to create drama and go through a recovery process...
This is possible in theory. I'm unaware of whether any cores actually implement this optimization.)

It is a fortunate fact that LS dependence prediction is surprisingly easy, and predictors are remarkably accurate, way above 99% for most code most of the time; it's a much easier problem than branch prediction.

Let's now consider a basic LSDP (load store dependence predictor) in more detail; how it might work to see how it might be improved.
The basic flow is as we have described
- we have an early store A, and a later load B
- store A does not execute for a while because its store address is not available
- load B executes optimistically
- when store A actually executes, it notes the overlap of its address with load B and raises an alert
So what the predictor should warn us about is that load B depends on store A. But how exactly will it do this?
The prediction cannot be based on the store and load addresses because we don't know those! Specifically, if store address A had not been delayed (ie unknown), we would not have had to speculate on whether its address did or did not match load B!
How about we identify store A and load B by their PCs, PCa and PCb? Obviously this may not be perfect, but speculation is about what usually works, not what's perfect, and using PCs to track probably

colliding load/store pairs works very well.
So we imagine the LSDP as essentially a table of (storePCa collides with loadPCb) pairs.

Now how would we use such a table?
The simplest idea is at some point in the pipeline we check the PC of each load against the table. In event of a match we mark this load as “pessimistic” and do not allow it to execute until every earlier store address is known.
That will work, and is already a good start, but now let’s consider various sub-optimal aspects of this implementation.

- as described this mechanism does not allow for multiple loads that depend on the same store. This could be dumb code (multiple successive loads from the same address) or reasonable code (store a 4-element SIMD vector, then successively load each of the four elements as scalars for subsequent processing). This is an easy fix that you can imagine for yourself.
- there’s also the possible reverse problem, where a load depends on multiple stores (so store four successive scalar values, then load them as a SIMD vector)
- we are being too pessimistic, waiting for every earlier store to execute, not just the store(s) that affect this particular load
- as described, once a load/store pair is in the predictor, it’s there forever! There’s no mechanism for removing entries from the predictor once they become irrelevant.
- a silly (but necessary) concern is that essentially this mechanism as described above, specifically the recording of load/store PC pairs, is the subject of a patent by the very litigious WARF (University of Wisconsin).

splitting the single load-store queue into separate load and store queues

We now know why
- an LSQ exists,
- why it’s (conceptually treated as) a single queue,
- the issues around why you want time ordering of the (load and store) addresses, and
- what to do about missing addresses.

So consider resource issues:
If we want to hold up to, say, ~600 pending speculative instructions (size of History File), and we expect ~half of them to be loads or stores, we need ~300 LSQ slots. When we run out of LSQ slots, like all resource allocation under the traditional model, Rename stalls until the rest of the pipeline makes some progress, so eventually a load or store reaches the head of the ROB, is retired, and its slot is freed.

But a large LSQ is not cheap!The problem is not so much the area of the queue, to hold all the addresses and store data; the problem is that
- every time a load or store executes, it has to compare its address against so many other pending

addresses, and the logic structure that does that (called a CAM) is power hungry, growing worse as there are more and more addresses to compare.
- and with a single queue, we have to waste power checking every address against every other, even though stores only want to compare against load addresses, and vice versa.

What can we do to improve life?
The original model I described, before load/store dependence speculation, is essentially what we saw in something like the MIPS10000 in late 1990s.
The x86 equivalent at the time only required a store queue because all loads and stores were executed *in order* relative to each other. (Meaning loads occasionally pulled their data out of the store queue, but there was never a situation where store queue entries had not yet been filled in, or a case where a store could generate data that should have been read by an earlier load).

IBM POWER4 (2002) takes us a long way to the full model I have described here. They have out of order load and store execution, and they have prediction of whether loads will alias with earlier stores. What IBM do at this point, and what seems to be the standard going forward, is to split the model I have described into two parts.
Imagine separate load and store queues (now possibly of different lengths) for which each queue entry has a few bits that we can call an "age".
Rename allocates ages as a single (continually increasing) stream that's common to loads and stores, but stores get store buffers from the SQ, loads get load buffers from the LQ.
Each buffer, at allocation time, has its age stored in the age slot. We still require that loads scan the store queue looking for earlier stores with a matching address, but they now find where to start the scan by finding an appropriate age marker; likewise for stores looking in the load queue.
You've
- given up a small amount of the ordering info in the single LSQ,
- but you have less work
- also each queue (and so each pool of addresses) is smaller, even though the total number of Load and Store Queue slots together can be larger.
You can now also start to optimize each queue separately for its particular tasks, giving it a slightly different logic structure.

Non-standard ideas for improving the LSQ

use the store queue as a L0 data cache

At this point you can make the following observation:
- on every load you have to pay the price for accessing the SQ (to match the load address with possible store addresses).
- And if you hit a load in the LSQ you don't have to pay the price of accessing L1 (because the store data

is in the Store Queue entry)
- Therefore you might as well try to store as much in the LSQ as possible!
This leads you to try to structure things so that after stores retire, and even after the data is written out to cache, you try to hold onto every store queue entry (marked as valid, but free) for as long as possible, so that you can maximize the number of load hits to the storage queue.

This idea (and the mechanics of how to implement it) are discussed here: (2019) https://www.diva-portal.org/smash/get/diva2:1316126/FULLTEXT02 *Filter Caching for Free: The Untapped Potential of the Store-Buffer*.
(This works surprisingly well. On Skylake sized machines, you can get 30 to 50% hit rate, saving around 15% of the energy used by the load/store/L1D subsystem!)
Is Apple doing this? As a pure energy savings measure, with no performance impact, it's hard to test. I have seen nothing in the patent record, but if there's nothing to patent beyond the basic idea...

two level LSQ

This is nice, but we still want larger load/store queues, and they are still expensive!
A very nice alternative is described in (2005) http://webdiis.unizar.es/~ktm/papers/ISCA_05_v4_final.pdf *Store Buffer Design in First-Level Multibanked Data Caches*.
This paper points out a version of a design principle that's, as I have tried to stress, very dear to Apple: don't use one hardware structure to do two jobs!
The Store Queue does three different things:
- it enforces correctness by holding stores until they are not speculative
- it enforces correctness by ensuring correct load/store ordering
- it provides performance by forwarding pending stores to loads accessing those store addresses.

But an interesting fact is that most load/store forwarding happens with loads that are very close to their stores. This means you can split these structures into two:
- a store-forwarding buffer that is optimized for speed. It only holds about a fifth of the store queue entries , it's a simple FIFO (so no complex logic to figure out what is or is not held). It supplies the first matching store to a load, which in theory may not be correct but is usually good enough. And it's multi-ported enough to be probed and read-out by three simultaneous loads per cycle
- a "traditional" store queue which is concerned with correctness, but doesn't have to be fast or multi-ported enough to handle three loads every cycle.
The idea is that most loads that are going to match to a store are (correctly) handled by the small fast buffer, and anything that slips through the cracks is handled by the large buffer, as a Replay or even a Flush, but this should rarely be necessary.

The paper itself also talks about many other things; some are design options irrelevant to our interests, but it's nice in being above average in how well it explains some of the background to Store Queue issues and options in resolving them.

(Haithim Akkary earlier proposed a two level STQ in the grand KIP paper to which we've referred many times, but does a terrible job, IMHO, in explaining why it's interesting or how it should be structured.)

virtual LSQ

More interesting is a step IBM took a few year after POWER4, with POWER7 in 2011, which is to "virtualize" much of the load/store queue.

Just as we discussed with registers, the standard LSQ model uses early allocation (at Rename) and late release (at Retire) of LSQ entries.
This means that much of the time the LSQ entry is marked as not free, but is not being used productively;
- much of the time it's empty waiting until the load or store executes, or
- it's long after everything relevant to the load or store executed, but the instruction is being blocked by something at the head of ROB and until then the LSQ entry cannot be released.
And as before, this is because in the most obvious LSQ model, or even, the fancier POWER4 split LQ and SQ model, you need allocation *at Rename* to attach *ordering requirements* to the LSQ entries.

So how to work around this?
What I am calling age tags are still allocated at Rename, in program order, across loads and stores in increasing order.
But the actual load and store buffers are not allocated until the instructions are ready to execute as they leave the Scheduling Queue.
This means that you maintain can ordering across a large pool of loads and stores (use more bits for the age tag) while using a smaller pool of Load and Store Queue entries (and thus addresses you have to check). Most of the entries in the virtual Load Store Queue (ie load/store instructions ordered by age tag) will spend much of their time in the Scheduling Queue, not in the Load and Store Queues.

As with virtual registers, if you temporarily run out of physical backing store (load or store queue slots) the addresses will pile up in the Load Store Scheduler, not in Rename, so more of the machine can make progress (even while load and/or store are temporarily blocked, other instructions can still move on to the integer and FP pipelines).

You can see some of the issues involved in a design like this (along with a more careful explanation of the various scans of the load store queue for various purposes) here: (2007) http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.3533&rep=rep1&type=pdf *Late-Binding: Enabling Unordered Load-Store Queues*.

Apple definitely seem to be using a virtual LSQ (evidence in the experiments section below). Different patents suggest that what Apple variously calls the GNUM (retirement group number), or the RNUM (reorder buffer number), or the LNUM (load queue? number) are used as the ordering property for

load/stores, fulfilling the role that I called "age bits" when describing POWER4.

Late resource allocation is a nice resource amplifier, with the flip side of the idea being early resource release. For example if there is no older store ahead of a load with an unknown address, then there is no possibility of the load having been mispredicted as not matching a store, and so having loaded stale data from the cache. In other words, under these circumstances, there is no need to continue to hold onto the Load Queue entry.
Again the experiments (with some patent validation) show that Apple is using early release of both Load and Store Queue entries.

Apple's implementation of load store dependency

2009 (load referenced by instruction count relative to store)

We start with a very basic 2009 https://patents.google.com/patent/US20100205384A1 *Store hit load predictor*. (If you look at the inventor name on that patent and think *Apocalypse Now* or, even better, *Zeroville*, you are clearly way too much of a movie geek!)

Like most of the early (pre-A7_ patents there are interesting ideas here, but ultimately things changed a lot.
In this early version, noteworthy points are that
- stores are tracked by PC.
- problematic loads are tracked by *an offset* (number of instruction in the instruction steam) relative to the problematic store.
This is clearly an end-run around the patent, but it works!
- the scheme can track a load dependent on multiple stores, but not a store that affects multiple loads.

The essential idea is
- when a *store* (not a load) passes through Mapping, its address is compared with the store addresses in the predictor.
- if there is a match, the associated count in the predictor's entry for that store is read.
- we now start counting instructions and if we see a load <count> instructions after the store, then that's a load that needs to be handled carefully.
- we know the SCH# of the earlier store, and we add that SCH# to the dependency list of the load.
Isn't that cool? Another lovely case enabled by the dependency bitvector. So now the load delays execution not just until its address register(s) are available, but also until the store has executed.

So this handles the legal issue, and the matter of waiting on the exact problematic store.
How does this handle the other issues we raised?

- the LSDP table is a CAM, so it's fully associative, and it's allowed to hold multiple entries with the same store PC but different load offset counters.

This is how multiple loads dependent on the same store are handled.
When the PC enters the CAM, multiple entries are triggered, and multiple counters started with the appropriate load offset counts.

- Alternatively the table can be populated by multiple store PCs each with a different count that ultimately references the same load.
In other words, the table can store a load that depends on two (or more) stores. And the bitvector mechanism can capture those multiple stores as multiple dependencies.

This use of the bitvector mechanism is no small thing. Simply having a predictor that tells us “this load probably depends on a recent store, be careful” is better than nothing, but how do you act on that information?
The bitvector gives a very precise answer – create a dependency on the SCH# of the earlier store, so that the normal scheduling of instructions prevents earlier execution.
Without this mechanism you’re forced to resort to messier, less precise, ad hoc solutions, for example *Store Sets* (1998) http://people.csail.mit.edu/emer/papers/1998.06.isca.storesets.pdf *Memory Dependence Prediction using Store Sets.*

- The patent points out this possibility of a load depending on multiple prior stores; but a practical matter will be how many of these "instruction N from now" counters is the Mapping stage tracking at any given time? Presumably simulations will show a number of dependencies that make sense, and for more complicated situations (16 byte loads followed by a single SIMD load?) we revert to marking a load as pessimistic and just waiting for all earlier stores to execute.
So one could imagine something like the described mechanism handling anything up to 4 simultaneous "pending problematic loads" (ie four counters) all of which could eventually resolve to the exact same load (with four store dependencies) and any load that somehow fails to be captured under these conditions (either too many dependencies; or just too many simultaneous load/store dependencies happening in this narrow stretch of code) being relegated to a separate table of "pessimistic loads", based purely on load PC with no test of store PC. At least that's how I would do it.

- One sub-optimal aspect of this scheme is updating the table. The CAM is treated as a FIFO, so that entries are aged out purely based on new entries entering, with no tracking as to whether any particular entry is still useful or anything like that.
This is a reasonable first start and I'm guessing that CAM is small, so adequate as a first attempt.

- A second sub-optimal aspect (not essential, but part of Apple’s 2009 implementation) is that the store PC held in the CAM is a simple hash (Apple suggests about the ten low order address bits) of the PC. This means that, for example, there will be occasional non-problematic stores that match in the LSDP predictor and start a counter. If that counter expires matching a load, then that load will be delayed. Presumably the combination of both these events (store matching low address, then load matching the count) is not too common, but it’s not ideal that it can happen.

I raise this issue because keep it in mind when we later look at branch predictors.
The LSDP (as of 2009) is a CAM, so it's a fully-associative cache, but the cache entries are based on a hash of the store PC.
Various branch predictor structures are N-way set associative (rather than fully associative) and indexed by a hash of the PC, but the entries in the indexed set are compared with a tag comprising the full PC. In other words the branch scheme is
- less flexible (it cannot cope with more than N entries that want to match a single index)
- slower (two stages of first tag comparison, then data lookup)
- lower power
- will not alias (match in the address hash of two unrelated cases).

2012 (load and store both referenced by hash of PC and instruction registers)

This is updated in 2012 https://patents.google.com/patent/US20130326198A1 *Load-store dependency predictor pc hashing*. What changes?

We learn that (for this implementation) the size of the LSDP table is 256 entries.
It's still a fully associative CAM, but operated rather differently.
The unit of interest remains a load PC/store PC pair where the load at the one PC has been found to be dependent on the store at the second PC.
Once again we want to search the table every time the store PC execute, to find the associated load PC and mark it as a dependency.
However the details all change.
Firstly the business of instruction offset counts was clearly an awful hack that we want to get rid of.
How can we do so while remaining legal?
The answer is the second change.
ARMv7 had load and store multiple instructions which, as the name says, could store or load a succession of registers. Even ARMv8 has load or store pair.
Now you can have a situation where a store single register is followed by say a load pair, where the first register of the pair matches the store, but the second register of the pair is independent (or variations, like a store pair followed by a single load). Which means that while saying that a particular load instruction depends on a particular store instruction is true, it's not the detailed truth. More detailed truth would be that a PC+register identifier (target register for the load) depends on a PC+register identifier (source register for the store). And that's our solution! We can make the hash ID into our table of load/store pairs based on both the store PC and the store register, and now have a lookup that both solves our technical problem and our legal concern. As an aside (though less important) the patent states that hashes of the simplest form (just using the minimal lowest order bits of the PC, as was probably the case with the 2009 patent) lead to an undesirable degree of aliasing, and so the PC address part of the hash uses (to simplify) the 24 lowest bits of the PC, first 12 xor'ed with second 12. (I'm constantly on the lookout for details of Apple hashes because I think in many places in the design Apple has replaced a

traditional few lowest bits as an index, with a more sophisticated hash.)
Likewise the matching load is described by a hash of the load PC with the target register.

Replacing the count-based mechanism of the 2009 patent, the new mechanism is that
- every store, as it flows through Rename constructs the PC+register id hash,
- probes the LSDP CAM, and
- sets an “armed” bit on every match.
- This armed bit is later cleared when that store is executed.
Meanwhile every load, in the same way, at Rename
- probes its hash against the CAM and
- in the event of a match *that is armed* (ie the store is present, but has not yet executed)
- marks the load as picking up an additional dependency for every store that is matched.
- the load now cannot execute until the dependency is resolved, which happens when the store executes.
(Actually in this original design, the dependency is actually resolved quite a few cycles after the store executes, which is sub-optimal. With multiple intermediate steps, the details depending on things like when store data is available relative to a store address, this seems to have reached the optimal end point with (2020) https://patents.google.com/patent/US11175917B1, discussed much later, finally ensuring that the load is allowed to execute essentially at the earliest possible point after the store, so that it will arrive at the LSQ just in time to accept the store data.
I *think*, in this 2012 design, stores are still a single operation, not yet split into separate store address and store data operations. Once that split happens, some of the details of how the load is woken up for issue have to change.)

An additional improvement is that a confidence field is now associated with each entry. This is used in two ways
- under low confidence conditions, the entry is retained in the table but stores do not trigger an arming (ie the entry is essentially ignored)
- when a new entry is added, it will be a low confidence entry that is purged to make space.

The patent again raises the issue of a load that may depend on multiple stores but is unclear as to the preferred solution suggesting either
- multiple entries match the load (ie the load picks up multiple dependencies) OR
- an entry is marked as "multimatch" and if that bit is set, behave as I earlier suggested, waiting for all older stores to issue before the load.
Possibly both mechanisms are used, the multiple dependency case for the most common situations (maybe up to two or three dependencies?) and the multimatch case for more than that.

The mechanism avoids the 2009 counters (which are conceptually cool, but one more complication). I don't know if the energy cost of the first scheme (having to decrement a bunch of counters every cycle) is worse than this second scheme (having to perform an additional CAM lookup on every load).

The second scheme is probably more accurate – neither patent exactly mentions it, but they both seem aware that a problem with the first (instruction count) based scheme is if you have conditional branches between the load and the store, which will vary the instruction stream count between instructions...

One issue worth noting is that both stores and loads probe the table as a CAM, meaning that it's not trivial to convert from a CAM to a lower energy structure like a set associative table. More difficult, and requiring a multi-step lookup, but not impossible (basically one table for stores and ARMing, a second table for loads, and cross-pointers between the two). By comparing this patent with the next one (both issued on the same day!) it looks like the lookup table was indeed split from a design that can CAM on both load and store hashes to two separate tables linked by a common “entry number”.

(2012) confidence tracking

The companion patent 2012 https://patents.google.com/patent/US9128725B2 *Load-store dependency predictor content management* tells us how the confidence field is maintained. One could imagine a few different mechanisms (basically you want to increment confidence every time this entry correctly caused a load to wait, and decrease confidence every time it caused an unnecessary wait).

The precise way Apple do this is to check whence a load that matches an *armed* store ultimately acquires its data.
If it acquires the data from the store queue, then we increment confidence (we probably need to wait on the store since it was fairly recent).
But if the load acquires the data from the L1D, then we decrement confidence (maybe there was a recent store, but regardless, it is separated enough in time from the load that it's fully processed and present in the cache by the time the load executes).

This is both a better and worse scheme that might first appear, depending on details the patent does not describe.

- It is true that just finding the data in the store queue is not a very reliable indicator that the store was recent enough to matter (there are good reasons to hold store data in the store queue for as long as possible, even after the store is safely executed and even retired).
But the fact that only loads that matched *armed* entries in the LSDP means that the arming is restricting interest to stores that happened recently relative to the load. So better than you might at first expect.

- On the negative side (as described in the patent) this means that loads that match an unarmed entry (maybe just by coincidence in the hash bits; maybe en entry that used to be valid but now the store always happens too early to be relevant) are not aged out.
So the scheme needs to be augmented by some sort of aging scheme to remove obsolete entries from

the table. Something like:
- every 10,000 cycles you subtract 1 from every entry's confidence level.
The patent mentions that you need such a scheme, but gives no details.

2016 (LSDP optimized for Replay)

The next evolution is 2016, https://patents.google.com/patent/US10437595B1/ *Load/store dependency predictor optimization for replayed loads*.
This patent addresses the issue of splitting a store instruction into store address and store data.

Assume we have a store then at some later time a load that matches the store address, and the LSDP has not been trained on this pair.
What can happen?
- nothing at all. The load is much later than the store, the CPU has no reason to link the two together.
- the load is forced to Replay because it sees the store address in the store queue, but there is not yet any associated store data.
- the load sees no matching address (the store address has not been provided yet), the load goes ahead reading from cache, and later we Flush.
Clearly
- the first case is easy, and clearly
- the third case is used to add an entry to the LSDP.

But what about the second case?

You could argue that Replays are not that expensive (true) so just ignore it, and don't use up an LSDP entry.
But Apple's choice, as of this patent, is something subtler. The thinking is "we got lucky with this Replay, but clearly we have a situation where there is a store happening close in time to a load from the same address, and we should keep an eye on this".

How should you handle this? Imagine we created a separate Replay Predictor for this case.
As before,
- we store a hash of the store PC,
- a hash of the load PC,
- we arm an entry when the store occurs, and
- if a load PC matches an armed entry, the load PC sets up a dependency.
After all, a Replay is cheap, but it's not free; it would still be better if we didn't issue the load until the store it depends on has executed, rather than have to issue the load twice (once fails because no data, then Replays).

Now if you think about it, everything about this new predictor is basically the same as the existing

LSDP! So why not treat them as a single table?
That's basically what Apple does, with only two tweaks.

First
- an entry is marked with a bit that says if it was generated by a Replay vs a Flush.

Secondly
- If the entry is marked as initiated by a Flush, then we want to be really careful about delaying the load. A load delay might cost a cycle or two, but that's much better than a Flush.
So even a very slight belief (weak confidence) that this entry is legitimate will delay the matching load.

- On the other hand, for a Replay the balance of cost and benefit is much closer.
What's wrong with always forcing a dependency on the store? The load can't execute anyway until the store is done!
The problem is that load takes multiple cycles. Once the load begins execution it has to
- calculate the address (add two registers),
- perform a TLB lookup, then
- scan the store queue.
The earliest it can get the value from the store queue is within three cycles, just maybe two cycles.
The world we *wan*t is that the load picks up the value from the store queue the cycle after it is deposited there;
the world we *get,* if we force dependence, is that the load picks up the value with an additional delay of one, maybe two cycles.

And we can't improve this because we don't have earlier indication of when the store data might be ready than the fact that the store has completed!
In essence, the best we can do is speculate that this load will find its data ready in time. And that's exactly the tradeoff we have here; that is the predictor we have built!

In conclusion:
- The Replay cost is not that high; the cost of forcing a dependency on the store is an additional cycle or two that might have been avoided.
- If we're not confident that the delay is really required (maybe the timing between the store and the load usually works out OK, just occasionally the store data is slightly delayed) then why force a *guaranteed* delay?
- So for Replay entries, we require a rather higher confidence value before we force the load to depend on the store.

Apple give a difference argument for the same end point. The reasoning they give in the patent is that
- Store queue Replays happen occasionally for random reasons that do not repeat.
- If you immediately started acting on a Replay entry stored in the LSDP, then you would delay a lot of

loads until the entry ages out, based on these random events.
- By forcing a higher confidence threshold before the Replay entry is acted upon, essentially you require the load to have to replay *twice in succession* before it's considered a real problem case that needs to be respected.
I think both explanations are probably reasonable ways to view why this modification is worth making.

BTW, an additional way to use this Flush vs Replay indicator bit is when you have to replace entries in the LSDP: preferentially replace Replay entries rather than Flush entries. The patent does not mention this, but it's an obvious extension

dealing with non-aligned/overlapping loads and stores

Having a load/store dependency predictor, and the speedup it gives you, is nice, but at some point you still have to actually compare the load and store addresses in the load or store queues, to make sure there is not an overlap. This is not as easy as it might at first seem, when you remember that we could be dealing with a misaligned load that just overlaps in one or two bytes with a different misaligned store...

One way to deal with this is to treat LSQ entries at the granularity of a cache line. A load or store is recorded in the LSU by the cache line it would occupy, and with a bitmask indicating the bits that the load or store cares about. This uses a little extra storage but is easy enough to implement in terms of simply
- matching cache line addresses,
- and'ing bitmasks from matching cache lines, and
- seeing if there's a 1 in the result.
(What about loads or stores that cross a cache line? Easiest is to convert them into two entries, though other options are possible. That's a waste, but if such cases are common in your code, well, dammit, write better code!)

Apple's implementation of Replay

what is Replay?

A second advantage of the bitvector dependency scheme is that it allows for Replay dependencies. I've alluded to these many times, but now let's consider them in some detail.

Replays are situations where an instruction could not complete because some detail wasn't ready in time. The standard case is that a load begins execution because its address registers are ready, but the calculated address misses in the TLB. Or it hits in the TLB but misses in the L1 cache. Or it matches an address entry in the Store Queue, but the associated Store Data isn't yet present.
Every one of these cases has the form that something isn't quite ready, but will be soon, so ideally we'd

just hold onto the load and try it again in a few cycles.

The problem also extends beyond loads because of the concept of Speculative Scheduling. This is discussed in a paper I mentioned at the start of this document, (2015) https://hal.inria.fr/hal-01193233/-file/ISCA%2715 _Scheduling.pdf *Cost-Effective Speculative Scheduling in High Performance Processors*. The issue is that in high frequency pipelined CPUs, you have to schedule the next cycle of instructions based on a hope that the current round of instructions will complete correctly; you don't have time to wait until the instructions have confirmed successful execution before scheduling the next set of instructions. This means that you might schedule multiple dependent instructions that assume that, at the time they start executing, the data they are expecting from a previous load will be present at some expected location (like the bypass bus). If the load fails, those instructions will still read whatever random data is on the bypass bus, and that's not great!

Replay recovery (early 2016)

Our first concern, once we realize a Replay is necessary (eg the TLB, the cache, the store queue, have indicated that the data expected by the load is not present) is recovering from the invalid data. The easy answer is simply to Flush everything after the relevant Load, wait till the Load data has arrived, then start again, but obviously that's awful! Slightly better is just flushing the instructions that are *in execution* after the Load miss is discovered, but that's still not great. The classic paper (2004) https:://pharm.ece.wisc.edu/papers/hpca2004ikim.pdf *Understanding Scheduling Replay Schemes* discusses the known ideas, some of which inform Apple's scheme.

The patent is (2016) https://patents.google.com/patent/US10514925B1 *Load speculation recovery*. Essential ideas are

- when an instruction dependent on a load begins execution, it is not removed from the scheduling queue. Instead a "timer" attached to the instruction begins a countdown. These instructions are said to be in the "shadow kill window", or sometimes the "load recovery window"
- if the load behaves as expected, the timer is "cancelled" and the instruction removed from the scheduling queue; otherwise
- the instruction remains in the scheduling queue, to be rescheduled when the load replays.

The nice thing about this scheme is that
- it scales to multiple dependencies. If three instructions are waiting on the same load, they will all be cleared from the scheduling queues, or left to Replay, when that load indicates its success or failure
- it is recursive in the sense that an instruction dependent on an instruction dependent on a load inherits the "timer" and will likewise be cleared or left to replay.

The exact details differ from what I've said; specifically the "timer" is a bit that is shifted each cycle, but the idea is as I said.

You should look at the patent details, but it's really very clever, while also being rather simple! In particular it builds on the existing dependency bitvector scheme, and it handles all the complications you can imagine, like an instruction that depends directly on load A and indirectly on instruction B that depends on load C that started a cycle after load A.
It also has the marvelous advantage that it is trivially adapted to value speculation! I have found no patent proving that Apple uses it in this way, but it's such an obvious next step.

Replaying instructions

Given the above design, we see that after a Load fails to acquire the data it expected, it (and all the instructions that depend on it) will still be in the scheduling queue. That's good, it means the Load is ready to immediately re-execute (after which the dependent instructions will be scheduled).
But when should the Load re-execute?
The traditional answer is simply to retry the Load every *n* cycles, which is easy to do but wasteful of both power and performance (every cycle you retry, some other instruction is prevented from using that execution unit).

2006 (extra dependency bits)

(2006) https://patents.google.com/patent/US20080086622A1, *Replay reduction for power saving* first lists a number of different circumstances that might require Replay, then describes adding a few extra dependency bits to the dependency vector for all these different various circumstances.
The relevant bits are set true the first time the instruction fails, so that the instruction will not be re-scheduled [because a "dependency" bit is not resolved] , until the problem clears, at which point the fake dependency is marked as resolved and the instruction is scheduled. (As always, details in the patent).

This is a good start because the instruction does not waste resources retrying until retry makes sense. But it's not perfect because the Scheduling Queue is a small structure, and while a Load sits there waiting on eg a TLB or L1 cache miss, other loads cannot use that Scheduling Queue slot.
An alternative would be to move the Load to some sort of separate queue while it waits (and even, if possible, return or some how reduce the resources it is holding while it waits).

mid 2016 (move Replays into the Load Queue)

Apple has moved partially to this with the second generation (A7..A10) of cores. For example 2016 https://patents.google.com/patent/US10133571B1 *Load-store unit with banked queue* states (as an aside, apart from the main patent idea) that the load queue and store now hold loads/stores that are waiting on either TLB translation or a cache miss.
This means that these loads (and stores) that are waiting on a Replay are cleared out of the LS Scheduling Queue (which is progress!) and now occupy the rather larger Load or Store queues.

But it still leaves instructions dependent on those waiting loads clogging up the other scheduling queues :-(

Note that this also now moves some Scheduling functionality,
- out from the Load Store Scheduling Queue (where the Scheduler waits for dependent physical register values to be available, as required to calculate an address)
- down to the Load Queue (where loads wait on things like TLB values or data to be available in cache, and they are waiting on Replay events).
This is a repeat of what we have seen so often – split a generic structure (in this case the generic scheduling queue) into specialized versions. So now the generic queue depends on other instructions and physical registers, the specialized Load Queue now has as dependencies changes in Replay sources, like TLB entries or cache data becoming available.

The primary concept of the patent is to split the Load Queue (which now supports Replay) and Store Queue into multiple banks.

For the Store Queue, this split is probably a power-saving measure. We start by filling up the first bank in order, then when it is full, switch to the second bank while Retires drain from the first bank, then we switch back. I expect the hope is that most of the time the Store Queue is not very full, and so only one of these banks needs to be powered on, apart from a small transition time when a few older values in one bank are waiting to retire even as the newer values are being placed in the other bank.

For the Load Queue, the split is driven by the need for Replay. The details are not given (we will see them in the next patent) but essentially each bank is associated with a Load Pipeline, and loads are allocated sequentially across banks so that banks are populated at an equal rate. This means all banks are powered up, and means some degree of co-ordination between them is required to maintain ordering; but it also means they can behave like Scheduling Queues with a fairly easy “choose one item that’s ready and, ideally, the oldest” every cycle for Replay.

2019 (split Load Queue into a Replay optimized queue and an Address validation queue)

The 2016 scheme means the Load Queue (which was supposed to exist and to be optimized for comparing against earlier Store addresses) has taken on an additional Scheduling task. Having one structure perform two tasks is never ideal so, once again, apply the standard design idea: hence 2019 https://patents.google.com/patent/US20200394039A1 *Processor with Multiple Load Queues*. We split the Load Queue into two parts, one that’s a continuation of Scheduling, one that’s the traditional “validate against Store addresses” queue.

- The first queue (the LEQ or the "fun" queue) is devoted to performing whatever still needs to be done to perform a load, and as fast as possible. This queue will hold load instructions in various stages of execution, from the initial address calculation, to TLB lookup, to cache access, to data value returned. Instructions in this queue can be Replayed if necessary, and can engage in various speculative shenani-

gans for the sake of speed.

This queue inherits the structure described above, one bank per Load Pipe. Each LEQ bank can be thought of as an extension of the associated Scheduling Queue. Both attempt to schedule an instruction each cycle; of course usually the Scheduling Queue will win because Replays should be rare, but when a Replay is ready to execute it is the preferred instruction.

Replay begins, in terms of timing anyway, at the beginning of the Load Pipe with AGU then TLB lookup. This is clearly not ideal, redoing work that has already been performed. It's unclear if the work is re-performed or if the instruction just runs through those steps to match timing, but does not actually waste energy on a second Address Generation and (especially) TLB lookup. Matching the timing is more or less essential to ensure that Replay does not collide with normally scheduled Loads – if the two timings were not matched then you could have things like
+ in cycle N the Scheduler fires off a Load, which spends a cycle in AGU and TLB
+ in cycle N+1 Replay fires off a load that bypasses AGU and TLB, and so
+ both loads arrive at the next stage (simultaneous lookup of matches in the Store Queue and the L1D) and so collide.
One could imagine a few ways to handle this, but Replays should be rare, so it may not be worth it. On the other hand, a simple bypass that prevents the Address Generation and TLB lookup (while still enforcing the timing) should be feasible.

Interestingly in this new scheme (and probably also the 2016 scheme) we no longer use the dependency bitvector for this Replay Scheduling. Replay is a more specialized Scheduling because each Load only has one dependency (a specific TLB entry, or cache line or whatever) so we can switch to a simpler Scheduler. Each Load has associated with it one ID that describes the event of interest, and that event is broadcast over some bus when it occurs, and checked against every LEQ entry, with matches marked as Ready. This should remind you of a reversion to the original tag-based Tomasulo algorithm!

- The second, larger queue (the LRQ or "boring responsible parent" queue) is devoted to ensuring correct behavior. This holds onto loads even after a data value has been returned, on the off-chance that a Load Store Dependency has occurred, and recovery will be needed. It doesn't have to be as low latency as the LEQ, so it can be substantially larger, or even use low power, slower technology.

The LRQ is, I assume, structured like the Store Queue described above, so it's split into some number of banks (perhaps two or four), that are filled sequentially, with the hope that under most conditions only one or perhaps two of the banks need to be powered up.

Note that Stores can also have Replay conditions (for example the Store TLB lookup can miss).
Store Replays are not performance critical, so perhaps Apple continues to use a single Store Queue structure for both Stores that are waiting for Replay and Stores that are waiting to Retire? (That's certainly what both the 2016 and this 2019 patent suggest).

An alternative could be to create something like the LEQ, holding Stores waiting to Replay. Such an alternative, not being performance critical, could be simpler, for example having only one instantiation, so that only one such Store could be Replayed per cycle.

(And an interesting aside also suggested by the patent is that it describes a little of how Apple allows for early release of the slots for either of these Load queues.)

2019 (convert Flushes to Replays)

Obviously we want to avoid Flushes (Load assumed that it would not match a Store, and acquired stale data from the cache) as much as possible, and a good LSDP is the first line of defence.
But in addition we have (2019) https://patents.google.com/patent/US10983801B2 *Load/store ordering violation management*.

The idea is (at an abstract level, stripped of implementation details) something like
- the somewhat obvious way of checking for load/store dependency is *in the cycle after a load address is calculated*,
BUT
- suppose we move the test instead to the cycle at which the data would be stored into a register/-placed on the bypass bus?

This delay (of maybe 3 cycles under normal conditions, more if there is a cache miss) gives a few more cycles for unresolved store addresses to possibly become resolved, meaning that there'll be a few more cases we catch where we can recover simply by preventing the load result from propagating to the target register and just forcing the load to Replay, rather than Flush!

(Another issue this resolves is that the usual load/store queues explanation seems to operate in virtual address space.
It's rare, but you could have collisions that only appear in physical address space as a result of the load and the store targeting the same physical address via different virtual addresses. This will also be caught by the mechanism described, since this later point at which the load address is checked against the store queue is now after TLB lookup.)
The 2019 patent has some nice timing diagrams that show how this works, if you are interested in the details and still somewhat confused; and it explains exactly how it is done rather than my grossly simplified explanation. (Rather than moving the test to the end of load execution, there's a primary test at the start of load execution, plus an additional test at the end to catch all changes that might have occurred during the intervening few cycles.)

The above scheme sounds great, but still imposes some latency! Consider for example this situation:
- The load begins.
- The store begins. The address is available, the store data not yet.
- The dependency is detected

- The store data becomes available.
- The store data is “processed” meaning that it is inserted into the store queue.
- After this processing, the store is “connected” to the LEQ (the load execution queue, the queue specialized to deal with replay issues)
- At which point the load can begin replay to, eventually, pick up the data out of the store queue.

All this imposes a substantial additional latency, around 7 cycles or so. Not a catastrophe compared to other options like flushing, but not great.

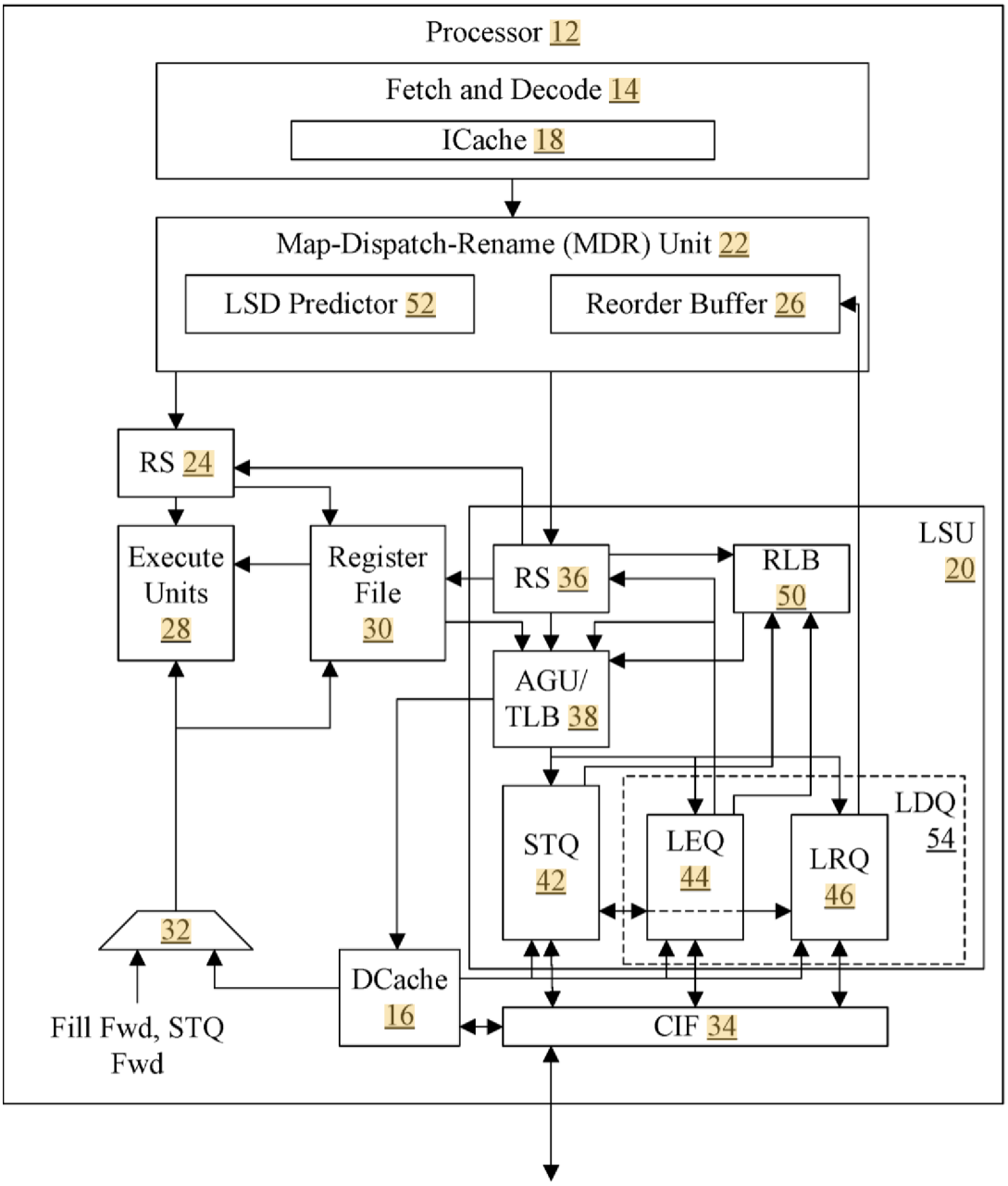
(2020) faster store forwarding

The basic problem is that the execution flow only gets round to informing the load to begin replay after the store is essentially completed. But one doesn't want to modify that flow too much for the usual power reasons, and because the flow is optimized for the common case of no load/store dependency. What to do?

(2020) https://patents.google.com/patent/US11175917B1 *Buffer for replayed loads in parallel with reservation station for rapid rescheduling*.

The solution is Apple's standard solution to this sort of problem: stop trying to make one structure serve two purposes. So we add a new (small, specialized) structure in parallel with the load/store scheduler queue, called the Replay Load Buffer.

You should be able to identify all the various parts in the diagram below, but the parts of interest to us right now are RS 36 (Reservation Station – the scheduling queue for load/stores, and the RLB 50 right next to it. The RLB is a specialization of the scheduling queue that is small and takes priority over RS36 (in the event that both have ready instructions).



The idea is that the RLB knows the store address instruction that caused the replay *and* the ID of the store data instruction that's associated with that store address. Thus now rather than waiting for knowledge to flow from RS through AGU through STQ to LEQ and finally back to Replaying the load, the replayed load can issue "in sync" (actually two cycles behind) the store, and grab the data essentially at the point that it's available in the Store Queue. The two timelines below show the old timeline (before an RLB). Note how it's seven cycle from the IS(issue) of the StoreData till the IS of the dependent Load. With the new timeline we are able to ISsue the load only two cycles later.

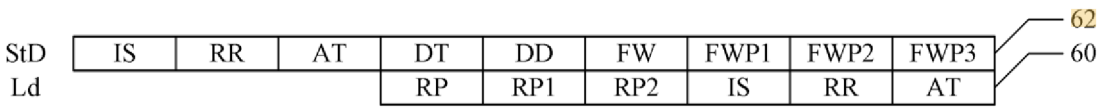
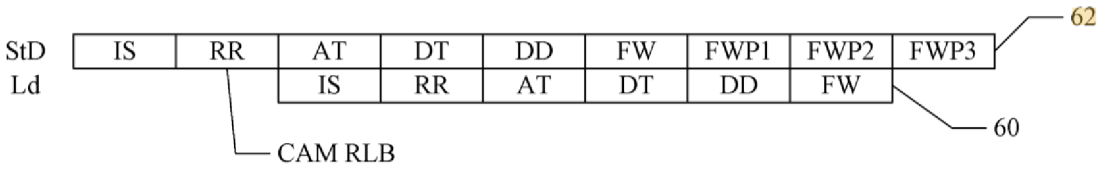


FIG. 3



(RR=Register Read
AT=Address Generation/Translation
DT=Data Tag Lookup
DD=Data Read [not sure what these two refer to in the context of a write! maybe it's just to unify the timing between load and store?]
FW, FWP1, FWP2, FWP3 are forwarding, plus one/two/three [ie handling the data after it's been acquired]
More important are the RP, RP1, RP2 stages (Replay stages) that correspond to communication overhead in the previous scheme, compared with the new scheme which is able to have the RLB detect that the relevant Data Store instruction is about to Issue, and immediately restart the load.)

A secondary win (which appears to be new, when I compare this patent with the previous equivalent) is that the load is removed from the scheduling queue when first issued. In earlier designs as far as I can tell the load remains in the issue queue, but frozen until it is given the go-ahead to replay. So this means that the issue queue becomes effectively a little bit larger since it's not wasting slots on these loads waiting to Replay; probably a small effect, but every bit helps!
Unfortunately (at least to my eyes) the Replayed load still passes through AGU and TLB (the AT stage). This still seems sub-optimal (both energy and latency), and feasible to bypass?

The various patents I've listed suggest that Apple is well aware of how Replays are a lot cheaper than Flushes, and strives both to limit Flushes and to make Replay efficient. This is no small matter. Replay is traditionally considered to result in a *substantial* performance reduction; known to be so for both Alpha and Pentium 4, and believed so for later designs.
This paper, for example, (2019) http://uu.diva-portal.org/smash/get/diva2:1316465/FULLTEXT01.pdf *Minimizing Replay under Way-Prediction* ascribes a 7% slowdown just to Replay caused by Way Misprediction, even ignoring other causes of Replay like cache bank collisions, TLB misses, or cache misses.

But Apple seem to be in a position
- to minimize Replay (no Way Prediction? clever bank handling),
- to perform Minimal Selective Replay (only the precise instructions dependent on the Replaying load have to be replayed, and they will already be present in the Scheduling Queue, not have to be moved back there), and
- to perform it at exactly the right time (by means of the tag-based dependency added to the LEQ entry of the Load that failed)

Ultimately Apple has a way to implement Speculative Scheduling efficiently because the following elements can be used together
- instructions waiting in a Scheduling Queue can read a result off the bypass bus before it is written to a register
- instructions are held in the Scheduling Queue (and results are held from being written to a register) until an instruction has completed
- instructions are scheduled dependent on a previous instruction (a SCH#) rather than dependent on the register that instruction produces.

That seems like a technicality but what it means is suppose I have a load that feeds a result to add which feed the result to a mul.
I can speculatively issue the add to pick up its input from the bypass bus, from the load; then the mul to do the same from the add. This chaining works even when I don't write back the intermediate values to a register.
If the dependencies were based on registers, I would have to be flipping back and forth between marking a register as valid (so that an instruction dependent on it can schedule) then marking it invalid (because its data is incorrect because of Replay). Once again carefully considering the different aspects of an issue pays off:
+ for SCHEDULING I want to depend on prior instructions. If a prior instruction is marked as "executed" that means I can now Schedule (picking up the result of that instruction off the bypass bus) but this is idempotent. If I learn the data I pulled off the bypass bus was invalid (Replay) I
= don't mark my result register as valid, not yet
= I don't mark my instruction slot in the Scheduling Queue as completed, not yet
= meaning I can execute again, just as soon as the SCH# I am waiting on (eg the previous load) once again marks that it has issued.
+ for DATA PERSISTENCE I want to mark physical registers as having valid contents, but I can't undo this if I realized I made a mistake.
= So I can't mark physical registers as valid until I know their data is trustworthy.
= So I can't use flipping a physical register to valid as a Scheduling signal, unless I give up Speculative Scheduling. (Or use a much more heavyweight Replay which involves restoring a lot more state at Replay.)

Because
- all the instructions stay in their Scheduling Queues AND
- no results are written to a register (more precisely, random junk may be written, but the register will not be marked as valid)

until the load is valid, Replay is not that expensive. It requires redoing every instruction along a dependency chain a second time, but only one more time, but no Flushing.

(2020) store forwarding and atomic operations

There are always some weird case you didn't think of, as in (2020) https://patents.google.com/patent/US20220091846A1 *Atomic Operation Predictor*.
We won't discuss the details till Volume 3, but you are surely aware that, for the purposes of synchronization between CPUs, certain types of operations have the form
- load a value (and mark the cache line in which it lives as "locked")
- modify the value
- conditionally store the modified value (depending on whether or not some other CPU touched the "locked" line)
An example of this sort of pattern might be an atomic increment. The point is that you do not want two CPUs to both read the existing base value, both increment it, and both store it (in which case only one store will go through, and the value will be incremented only by one, not by two...)

Now consider a situation where the CPU engages in some version of this conditional modification of a value. The final store is conditional (on the state of the cache line) but there may be a subsequent load from that store address. And that subsequent load could just have the store value forwarded to it from the Store Queue, using all the machinery described above. Is that good or not?
Well it's good if the store conditional is likely to succeed, because it means we saved some latency.
But it's bad if the store conditional is likely to fail, because in that case the load (and all subsequent instructions) have incorrect data and we need to Flush.

This is a speculation problem and the history has followed the same pattern as usual.
We began with the CPU not speculating (a load from the address of a Conditional Store will wait until the Store Completes);
then we switch to the CPU speculating that we will have the common case (assume the Conditional Store succeeds, with the ability to detect a misspeculation and Flush);
and finally we create a dedicated predictor.
So that's what we get in this patent, a simple predictor attached to the Store Queue holding a few of the most recent Store Conditional addresses, and a counter of success vs failure. The rest is obvious: if the predictor suggests that the Conditional Store will fail then it will use the Replay machinery to hold back the dependent load until the Conditional Store completes.

Can you make such a predictor smarter? The patent suggest at least two options. The first is in many common cases a value is used as a flag or counter/semaphore, and the common cases of a value of 0 or 1 tend to occur when the value is uncontested, it's only when values of more than 1 appear that once can start to expect contesting. So looking at the value to be stored can bias the initial value of the predictor. Secondly different exact patterns of atomic operations are used for different purposes, and again some of these purposes have different statistics (for example one might use a different type of

code structure for a lock that's expected to be mostly uncontested vs one that's expected to be heavily contested).

Moreover, once you have such a predictor in place, can you use it for more things? Apple give a few examples of this (in a kind of matter of fact way that to me suggests they are already doing these, not just listing future patentable ideas).
First is that these conditional atomic patterns usually exist in a loop, something like ("try to load and increment until the store does not fail"). The branch prediction system may have assumed the test in this loop does not fail, and thus continues with code after the loop. There is nothing we can do about that code that's already in the machine, it will just have to flush; but we can try to avoid wasting any more energy by throttling the front end until we resolve either whether a flush is necessary or we can keep going. We may even be able to use the value from the store conditional predictor to update the branch predictor, though I suspect that's still in the vague "possibly for the future" stage.
Second is that we can also use the prediction (we'll probably succeed in the vs we'll probably fail) to choose an optimal cache line state for when we make the coherency transaction attempting the conditional write to the cache line. If we expect to gain the line, we'll ask for the line as Exclusive (since we want to write to it); if we don't expect to gain the line, we'll ask for it as Shared (so that we can read from it without drama for the next time through the loop). This will force a second coherency transaction (to gain Exclusive) if we do unexpectedly gain the line, but, well, prediction is always a game of hopefully you win more often than you lose!

Back to LSQ measurements

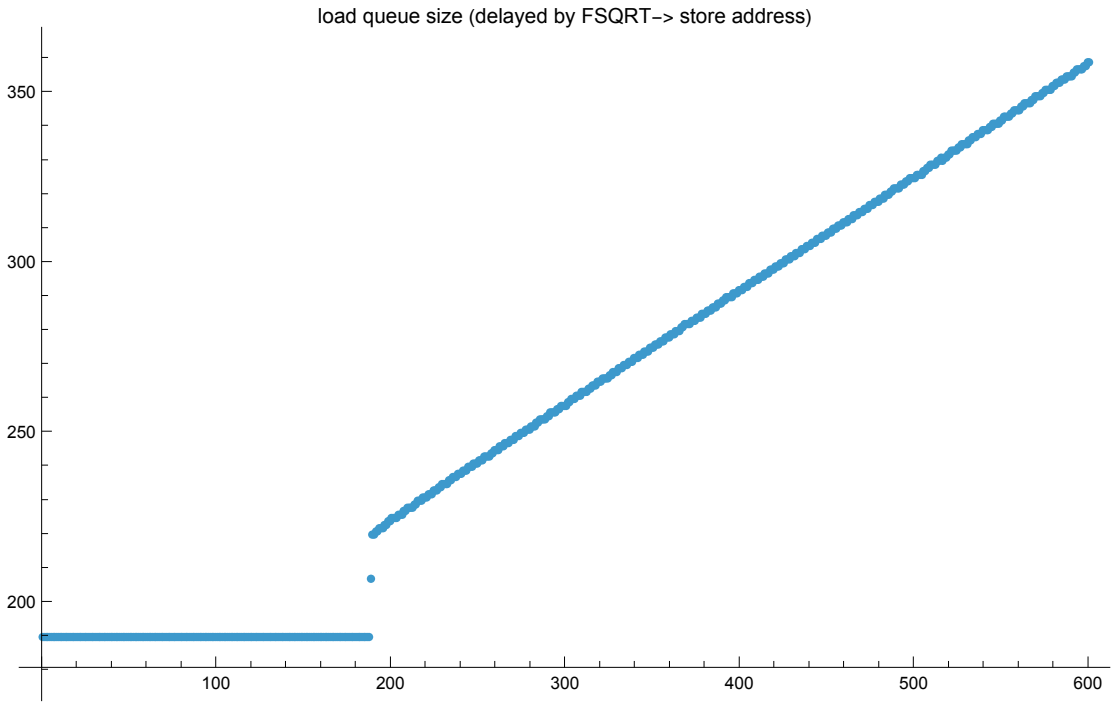
So recall where we started. We hoped to discover the size of the load and store queues, and instead we found what looked like sizes that varied substantially given different test probes.
So, given our new understanding, can we create a cleaner example of this issue, that we see what looks like a ~125/100 sized LSQ under one test but not another?

Yet a third different value for the “size” of the LSQ

Loads have to remain in the LSQ as long as they are in some way speculative. So imagine the following construct:
• long chain of `FSQRT`s to act as a delay
• convert the result of the last `FSQRT` to an integer using `FCVTAS x0, d1`
• store a result using that integer as an index, ie `STR x5, [x3, x0]`
• lots of loads

Under these conditions the loads will be scheduled and will, in fact, even execute speculatively (under the assumption that their load addresses do not collide with the *unknown* store address) but they cannot complete until the store address is in fact known. This gives us a much cleaner result:

Out[607]=



We see a clear jump at N=188 (load queue can hold 188 entries) and this matches Dougall's results achieved using somewhat the same idea (but with speculation from a branch dependent on the FSQRT chain, rather than on memory-aliasing dependent speculation).

What about stores?

Out[609]=

Again we see a nice clean jump, giving us ~118 store queue slots, again matching Dougall's result.
(You might wonder why a store would force subsequent stores to remain speculative. Surely even if the second store address matches the first, the second store will simply overwrite the first, so who cares?
In the generic case there could still be a problem because of alignment – if one of the two stores is not aligned, then it would be possible for a matching-width store to overwrite some bytes but not others...

If we force the alignments to match, I think that in principle the ARM model allows this to become non-speculative, but Apple doesn't catch that and special case it. This probably makes sense; they already have most of the win available from a good LSDP, and trying to optimize the very special case of this test is probably of little real world value.)

Explanations

So: We have seen three different results for different attempts to measure the size of the load and store queues:
- the most naive test (uses `FSQRT` delay) gives an apparent size of ~330 for both
- using a load miss to DRAM delay gives apparent sizes of ~125 (L) and ~100 (S)
- using an `FSQRT` delay generating a delayed store address gives apparent sizes of ~188 (L) and ~118 (S)

Why these various different results?
What appears to be happening is that Apple is using a virtual load-store queue. In other words
- while instructions are still in-order at the Map/Rename stage, load/stores are given a sequential ID that describes their relative ordering, but they are not allocated a load/store queue slot.

- at issue time (ie at the start of execution, when the address is about to be calculated) the load/store queue slot(s) (two in the case of loads, for the LEQ and the LRQ) are allocated.

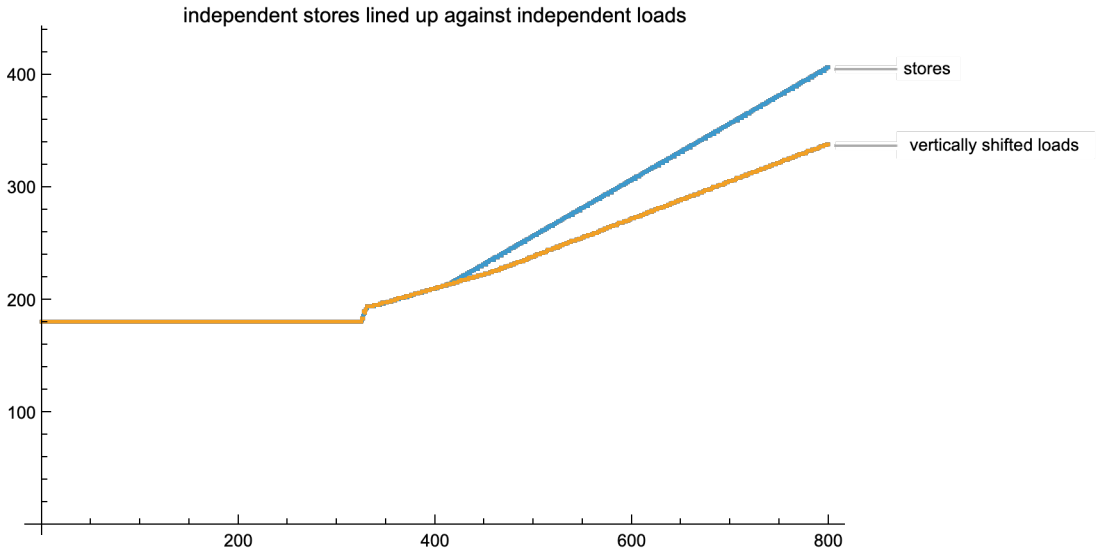
As counter evidence I will mention (2019) https://patents.google.com/patent/US20200394039A1 *Processor with Multiple Load Queues* which is very clear that
- entries in the LEQ are allocated at issue
- entries in the LRQ and STQ *could be* allocated at issue
- LNUMs (that track relative ordering) are allocated at Rename
The most reasonable analysis is that as of the 2019 patent Apple was allocating LRQ and STQ entries at Rename, but was primed to switch this at any time, and it seems that by the A14/M1 that time had come.

We will first discuss the load cases, then see if there is anything different in the store case.

early load queue deallocation (simple FSQRT delay)

The easiest case is the first case, a simple `FSQRT` delay

Out[610]=



The salient issue in that case is that there is no question of load/store aliasing because there is nothing in the store queue against which an alias could occur!
Thus, while the loads occupy resources in the ROB, they don't have to hold onto their load queue slots (because the only reason for those slots was to ensure that older stores that have not yet completed do the right thing relative to younger loads, and there are no older stores).
More generally, in principle, once there are no older store that could, in any way, affect a load that load could release its load queue slot.

We see that Apple appears to be doing this, and as confirmation there is, once again, a patent: (2013) https://patents.google.com/patent/US9535695B2/ *Completing load and store instructions in a weakly-*

ordered memory model.
In other words, what we are seeing is proof of early resource deallocation for the case where the resource is Load Queue (ie LRQ) slots.

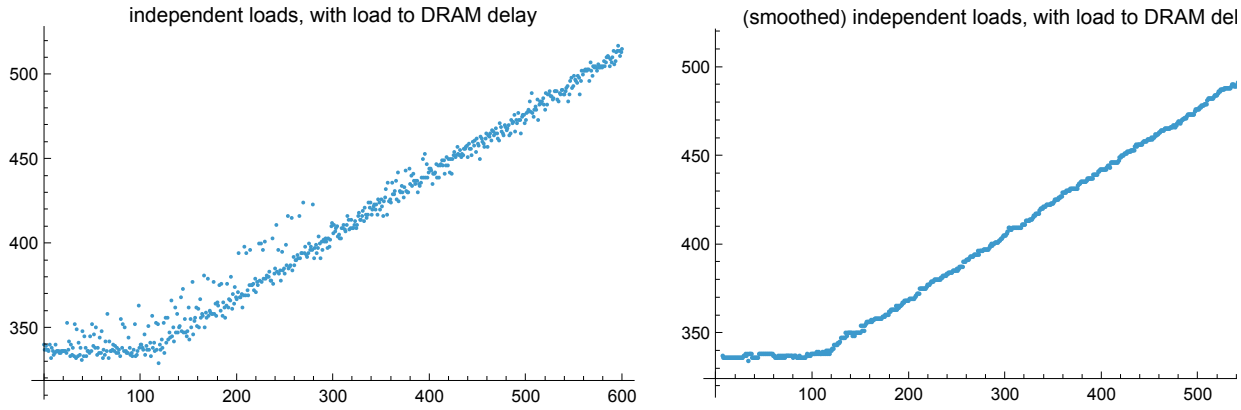
The resource limitation that causes the jump at around ~330 is something we have already discussed, that the ROB consists of ~330 rows, each row can hold 7 instructions, but only one "failable" instruction, and loads/stores (and branches) count as failable instructions.

We'll discuss the other interesting feature of the curve later, once the "sizes" of the Load and Store Queues is settled.

the size of the physical load queue (load to DRAM delay)

Now what about the case of the load-miss-to-DRAM based delay? ie this diagram:

Out[611]=



The most important thing to realize in this case is that the metronome that is determining the base ~330 cycle delay is a series of chained loads.
load A → load B → load C. If each load (to some random location that's hopefully never in cache) can execute immediately then (give or take a whole lot of noise in the response time of DRAM and the NoC) the time per loop iteration will be ~330 cycles.
But if something delays when load B can start executing, then the time per loop iteration will be more than ~330 cycles.
So what we see is that if we have load A (load to DRAM) followed by N local loads (loads to cache) there is no delay as long as N<~125. Once N>~125 we see a delay. So these intermediate 125 Loads are using up some resources that has to be freed before load B can begin execution.
What is that resource? It's not ROB related (that's N=~330, or HF related, ~630, or physical register file related, ~380). It is in fact "genuine", not virtual, Load Queue slots. load B cannot start executing until it has a Load Queue slot.

late load queue allocation (FSQRT delay with ambiguous store address)

Compare this with with our final case, the `FSQRT` delay generating an unknown address for a store, as in the nice clean plots that begin this section.
- while the head of ROB is blocked by multiple `FSQRT`s,
- loads (or stores) are fetched, renamed (ie given *virtual* LSQ resources) and send down the load store pipeline.
- They issue, at which point they are allocated a *real* load (or store) queue slot.
- Eventually those slots run out and the loads (or stores) pile up, first in the 48 LS Scheduling Queue entries, then in the 10 LS Dispatch Buffer entries.
- When those 58 entries are full (along with the genuine ~130 load queue entries, then Rename stalls and we get the classic jump, at N=58+130=188 for loads, and at N=58+60=118 for stores.

If you haven't spent much time with more traditional CPUs, this may seem obvious. But it's not!
A traditional CPU would stall loads and stores at Rename when it was not possible to allocate a Load or Store Queue slot. Hence it would stall at ~130 loads, or ~60 stores.
A CPU with a virtualized LSQ at Rename only has to allocate an age tag. The loads/stores can then proceed into the Dispatch Pool and Scheduling Queue, and can keep doing so even when all the LSQ slots are full, up to the point where the Dispatch Pool and Scheduling Queue slots are also full.
Hence late allocation allows us to enqueue 58 more loads or stores than traditional allocation before we have to stall. We get about 50% larger effective load queue size, about 100% larger effective store queue size, at the cost of a slightly more complex algorithm.

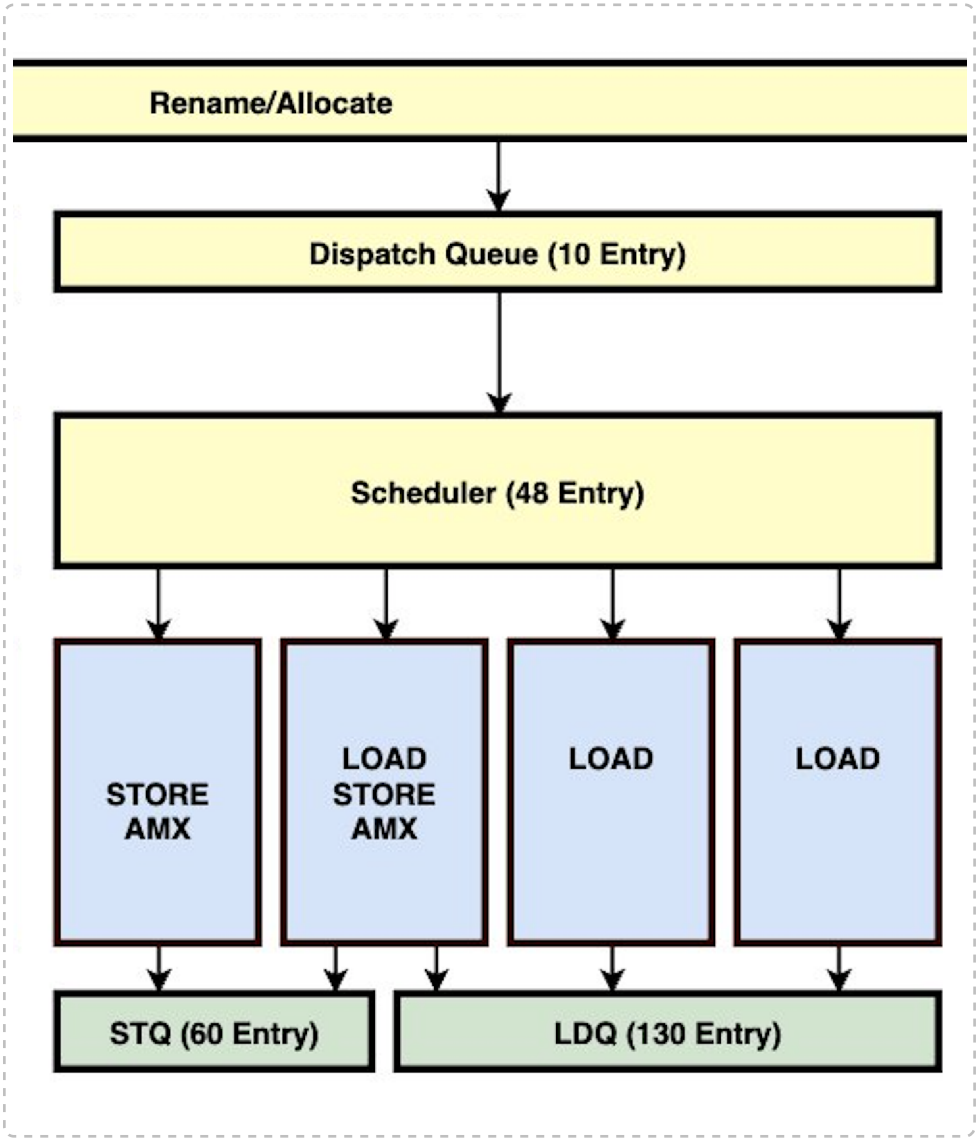
The difference between this case and the DRAM case is that
- for the DRAM second loop iteration to begin, the loop delay (the load B) has to proceed past the earlier stages of instructions storage (in the Scheduling Queues) to begin execution with possession of a genuine Load Queue entry
- for the `FSQRT`+Store second loop iteration to begin, the FSQRTs need to get past Rename to the Floating Point Scheduler, but they don't care if 58 extra Loads have been dumped into the Load Store Dispatch and Scheduling queue and are blocked there.

So what we see is that the M1 has a pool of ~130 genuine Load Queue entries (ie LRQ entries), but for most code patterns this pool is amplified by both
- early deallocation of Load Queue entries (if there are no ambiguous store addresses remaining ahead of a Load, that Load no longer needs to hang around in the LRQ)
- late allocation of Load Queue entries (so that excess loads, that might not be able to execute because all the physical Load Queue entries are busy, can still be shifted out of Rename into storage in the Load Store Dispatch Buffer and Scheduling Queues. Doing this means that even though they can't execute until a Load Queue entry becomes available, they will not block instructions behind of a different class, like the `FSQRT` delay instructions.

structure and evolution of the load-store Scheduling Queues (bonus discovery!)

What about the regions between about 330 and about 410 where we get what looks like 4 loads/cycle, not 3?
That's a reflection of the Dispatch buffer, and we saw a similar situation when we first discussed Dispatch Buffers. Before proceeding, you should also refamiliarize yourself with paired scheduling queues.

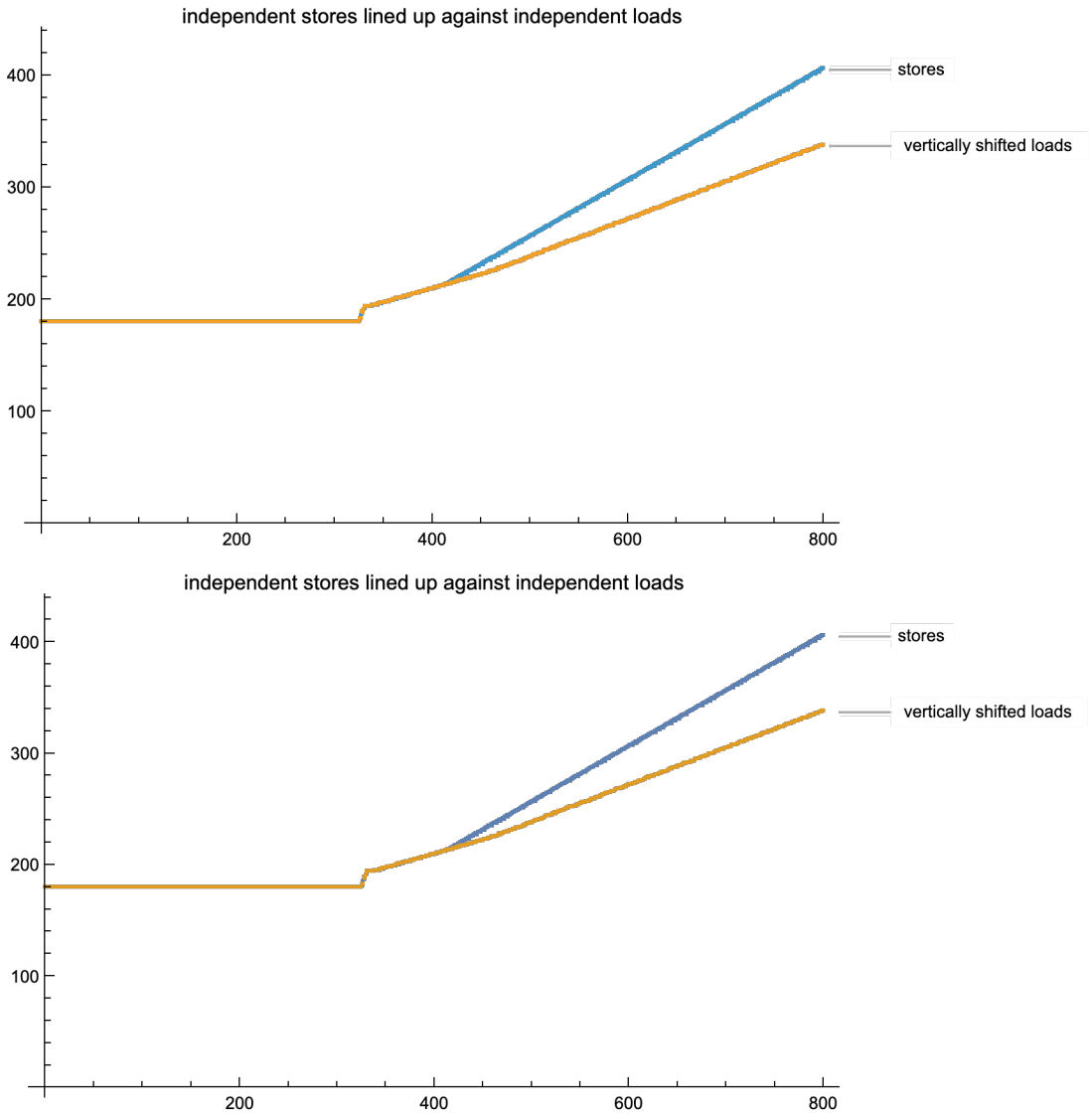
So recall the relevant part of Dougall's diagram:



This shows a single Dispatch Buffer holding 10 entries (probably correct) feeding what looks like a single, surprising large, Scheduling Queue (incorrect).
I suggest that the Scheduling Queue is split into 4 queue of 12 entries, and so somewhat like the integer and FP setups.
Below is the justification for the argument (which I don't claim to be definitive, but it's the best explana-

tion I can give for the phenomenon we see of the “4 ops/cycle for load or store, in the range of N from about 330 to 420” in the graph we have already seen:

Out[612]=



Let’s start by assuming the diagram is correct.
- Load/Store has a Dispatch Buffer of size about 10 instructions.
- This feeds into a single Scheduling Queue that feeds four Execution Units.
- The Dispatch Buffer, as usual, can accept 8 instructions per cycle, but can only output four instructions per cycle into the Scheduling Queue.
So suppose N=410. Then at the end of one cycle we have
- 330 loads have completed, and are sitting in the ROB
- 80 loads could not move pass Rename (because no ROB entries were available)

The last `FSQRT` completes, the loads in the ROB Retire, the new cycle starts.

- First thing to happen is that the 80 loads piled up behind Rename have to be handled. At first these are moved into the Dispatch Buffer at 8 per cycle, and moved out of the Dispatch Buffer into the Scheduler Queue at 4 per cycle. But that means a net of 4 in the DIspatch Queue, so that can only be sustained for about two cycles before we drop to a throughput of clearing 4 loads per cycle.
To clear out 80 loads will thus take ~20 cycles.

- As far as the Scheduling Queue is concerned, we are piling up an excess of one load per cycle, (4 in per cycle, from Dispatch; 3 out per cycle from 3 Load units) so we cannot sustain this indefinitely, but we can sustain it for quite a while, certainly for 20 cycles (at which point we have
+ an excess of 10 loads still in the Dispatch Buffer,
+ an excess of ~10 loads in the Scheduling Queue,
+ and have executed 60 loads).
- At that point the 80 Loads in excess of 330 are done, the `FSQRT`s are in Rename, and can propagate to the FPU, start to execute, and start the next delay cycle.

So we see that for this case of 410 loads, the time taken is the initial `FSQRT` delay, plus the time it takes to clear the 80 loads being able to do so at 4/cycle.
So this describes the initial flat region (the `FSQRT` delay), the jump (run out of ~330 ROB failable slots), and the high speed region from 330 to 410 (Scheduling Queue can take in 4/cycle, emit 3/cycle, and has space to sustain this for quite some time as the Scheduling Queue slowly fills up at one excess instruction per cycle).

That all seems plausible, but we are still left with the question of why we revert to 3 loads/cycle at N=~420? If the above explanation is the full story then at that point we should have the Dispatch Buffer full, maybe 12 instructions enqueued in the Scheduler, and space for an additional excess of 36 or so. We should be able to sustain 4-wide till a much higher N!

My guess is that this reflects some structure within what Dougall draws as a monolithic load/store Scheduler Queue of 48 entries.
XXX I don't have time to draw any diagrams, so you will have to make do with text diagrams for now but what I imagine is the following
- this apparently monolithic queue is actually four queues each 12 entries in size
- the Dispatch Buffer is able to feed one instruction per cycle to each queue (just like integer and FP)
- this works out OK (ie it's allowable to enqueue loads into a queue that looks like its attached to the STORE/AMX execution unit) because
- of the patent we described that pairs queues together and allows an instruction in one queue of a pair to execute if the other queue of the pair cannot find a runnable instruction.

If we accept this chain of logic, then what we see matches what we might expect – we can run 4-wide until one of these queue (the one attached to STORE/AMX) is full (because every cycle it is never chose,

the other queue always has a runnable load). And then we run three wide.

A reasonable objection is that this will lead to a drastic reordering of the loads – the loads that were dispatched to the STORE/AMX queue will (if we accept the way the patent described paired queues) always be considered second class, and will be delayed as long as their are loads in the other load queue, meaning a possible indefinitely long delay. Perhaps so, and perhaps Apple are aware of that, but figured it was acceptable? The A12 has two load pipes and 2 store pipes, I can't find data for the A13.

Here's what I imagine as the evolution of the design. (Timing details for what core got when may be slightly incorrect; this is a conceptual model!)
For the A11 we have a fully symmetric design. So we have the Dispatch Buffer feeding 4 identical queue, each 12 in size, arranged like
S0 S1 L2 L3

Now for the A12 we decide we will add paired scheduling queues. How should we pair?
The obvious choice is to pair the two L's together, and the two S's together; it's simple, and instructions from one queue can easily be shifted to the paired pipeline without even having to test the instruction type. The downside is that the total pool of Scheduling storage for Loads remains at only 24 instructions, likewise for Stores. If we have an imbalance of lots of Loads relative to Stores, we can't make use of the Store Scheduling Queue space.

What about if we pair (S0 L2) and (S1 L3)? The win is that now all our scheduling queue space is available to hold either an temporary excess of Loads, or (less likely, but happens sometimes, eg if you're filling a large structure with zeros, an excess of Stores). The downside is that (without substantially rethinking the details of how we choose the the oldest ready-to-execute instruction; and the mechanism of second choice from one queue feeding the other queue) we can get cases of imbalance. If the S queue is all filled with loads, and the L queue always finds a runnable load, then the S queue will stay blocked that way until something changes (eg a dependency eventually means the paired L queue cannot find a runnable Load); this is the imbalance we saw above.

So which of these two options is overall better is a question for the simulator, but I would not be surprised to see that the LS pairing is better. For normal code (with a mix of loads and stores, especially if Dispatch tries to place stores in an S queue, and only places a load if no store is available in the Dispatch Buffer) then it's probably rarely an issue. And for code that consists of a long stream of loads or stores and nothing else, it's not much of an issue the precise order they get executed, so if some loads get stranded in an S scheduling queue for many cycles, well, so what.

So let's assume that's the A12. With A13 we get AMX, and AMX instructions are executed (ie transported to the AMX unit) via the Store pipeline. So now we have (SA0 L2) and (SA1 L3). This works in the same

way as above, is still essentially balanced, and still gives us the ability to absorb long streams of pure loads, pure stores, and even pure AMX instructions, in a way that can devote all 4x12 Scheduling Queues to that single instruction type if that's all that is seen.

Finally with the A14 some bright engineer notices the following:
Suppose we define a third load pipeline. This means essentially we
- need an AGU (but can reuse a Store AGU)
- need a port into the TLB (details dependent on exactly how the TLB is structured, but reuse of the store port may be feasible)
- need to create a third path to the L1D (details dependent on exactly how the L1D is structured, but reuse of the store path may be feasible
- need to create a third bank for the LEQ
- need to create a third port to test load addresses against address in the Store Queue
In other words a lot of reuse of existing Store machinery is possible.
And because of the pre-existing paired Scheduling Queues, we don't have to do much real work at the Scheduling/Dispatch level.
So we land up, finally, with a structure that looks like
(SA0 L2) (SAL1 L3)
Loads that go into the second pair (either scheduling queue) can be issued as loads, and that part of of mismatch from the A13 has gone away. The only problem is if the SA0 queue is filled with a long run of loads, which become blocked because loads are also being Dispatched into L2, and those L2 loads are always preferentially issued.

At each stage we have given our Load Store unit a reasonable boost in capabilities while also, at each stage, never paying much of a hardware price. The cost is that at each stage we introduced some asymmetry so that while performance is usually boosted quite a bit, there are weird corner cases that will do less well. If I'm correct that every four years or so the design gets a complete clean start redesign, then we may soon revert to a symmetric design with better performance than what we have today, and without the asymmetries.
For example what we also converted SA0 to a load supporting SAL0? Well, do we want to do the full work required to support a fourth load pipeline? Maybe that's not a good use of resources.
Alternatively we could have that SAL0 and SAL2 take turns to feed into a single third Load pipeline (basically if only one of them has a load, they get the pipeline; if both have a load, one gets it and flips a bit so that next time it's the other one's turn). That sees like the sort of quick elegant hack that Apple uses so often, providing most of the benefit of giving a full SAL0 (in particular better load balancing now) at very low additional hardware complexity.

Alternatively, we could change the design so that the Scheduling Queues become virtual! I think this is eminently practical.
Consider how the Scheduling Queues are used. Their job is to hold instructions, test that an instruction

has become runnable, and check ages.
These are the same job regardless of whether the instructions are loads or stores. The details of load or store only matter at insertion (Dispatch preferentially inserts stores into an S Queue, loads into an L queue) and extraction (Issue from an S queue will only extract Stores or AMX instructions). Note also that both Dispatch and Issue are already cross-wired to both of a pair of S and L Scheduling Queues.
So:
- differentiate between physical queues (call them 0 and 1) and logical queues (call them S and L)
- Dispatch and Issue operate as above on the logical queues
- but there's a degree of indirection between the SL and 01 queues.
- So we start off with S tied to 0 and L tied to 1
- Whenever one of the queues becomes full, we swap the indirection at both ends. (A few addition tests are necessary to ensure that, under certain circumstances like once both buffers become full, we don't just keep swapping every cycle!)
- Meaning that, if eg, we were servicing a long run of loads, the 0 queue will become filled with loads that are never extracted, until the swap, at which point the the 1 queue will become filled with loads.

This mechanism allows two nice improvements
- ALL the LS Scheduling Queues can act as buffer during a long run of either pure loads or pure stores; we get a much larger effective buffer before we have to drop to 3-wide Loads or 2-wide Stores as far as the rest of the machine is concerned. Right now the effective excess buffer is 10 (Dispatch)+ 12 (one L queue) or 24 (two S queues). But what I'm suggesting would allow that to expand up to 10+48 for both L and S (and A).
- We do a better job (not perfect, but a lot better) of not holding onto very old instructions without Issuing them until the machine has been forced to stall.

However as it is, we see the design we we see. Rename can feed Dispatch can feed Scheduling 4-wide up to about 90 instructions (N=~420) before queue SA0 becomes full of Loads and never able to get a chance at Issuing a Load.
Note this also explains the Store behavior of 4 wide until N reaches ~420. The logic goes as before. but now every cycle both queue L2 and queue L3 are being filled with Stores (Dispatch of 4 instructions per cycle), while only SA0 and SAL2 are being drained of Stores (2 Stores Issue per cycle). Again at around N=~420 we have both L2 and L3 filled each with 12 stores which have never had a chance to issue, and throughput drops to 2 Stores per cycle.

what about the store queue?

What about the Store versions of these tests? Do they teach us anything additional?

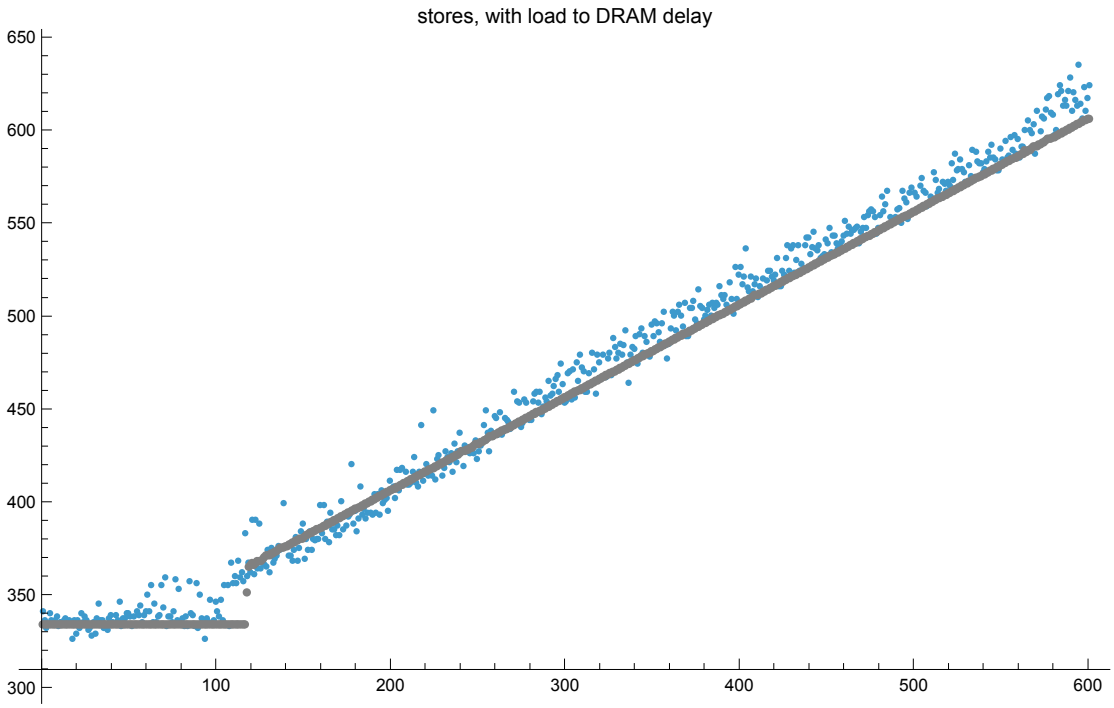
late allocation (miss to DRAM and FSQRT with ambiguous address delays)

Consider first the load-miss-to-DRAM case. That shows a jump at what looks like N=~100. Is that reasonable?

In this case we do not see a delay in how long it takes to start Load B until not only all 60 store queue slots are in use, but also the 58 Dispatch Buffer+Scheduling Queue slots.
Why the difference?
For the load case, the delay loads, load A, then load B, then load C, are fighting for the same Load Queue slots as the local filler loads. It does not help that the delay loads can move past Rename to pile up in the Dispatch Buffer and Scheduling Queue; that doesn't change the fact that the delay loop can't begin the next iteration until load B receives a physical Load Queue slot. ie the load case stalls at *Execute*.
In the store case, the delay loads have no interest in the Store Queue slots, so stall does not happen at Execute; rather now the common resource that both delay and filler store care about is the Dispatch Buffer+Scheduling Queue slots, and it is only when those are filled up that stall occurs, this time at *Rename*. Before that point, as long as a load can get past Rename, there should be a Load Queue slot available, and being Dispatched into a Load/Store Scheduling Queue should allow the delay load to get to it and start execution.

But I slipped a fast one past you in the above! Go through the argument again. The common resource that matters is in fact just the Scheduling Queue slots, not the Dispatch Buffer. As long as there is one space free in one of the four scheduling queues, then load B can be Dispatched into that Scheduling Queue, and can be Issued for execution in the next cycle; no delay. But if load B is placed in the Dispatch Buffer, then it cannot begin execution until a space opens up in a Scheduling Queue. Moving it to the Dispatch Buffer clears Rename and allows other types of instructions (int or FP) to make progress, but it doesn’t mean that the Load can make progress, not until a Scheduling Queue slot opens up.
This, and the fact that movement from the Dispatch Buffer to the Scheduling Queue is somewhat stochastic once the Scheduling Queues and Dispatch Buffer fill up (since the Dispatch Buffer makes no serious attempt to preserve ordering and Dispatch the oldest instruction) explains some of the noise we see in the DRAM Delay case. And it means that the N, number of stores, at which we should expect a jump is in fact 48+60=108, omitting the 10 storage units of the Dispatch Buffer. That looks (by eye, given the noisiness) more or less what we see.

Out[614]=



Certainly it seems reasonable to assume that, like Loads, we see delayed allocation of Store Queue slots, since both probes shows that many stores can progress past Rename even when the Store Queue (of size ~60 entries) is full.

early deallocation (simple FSQRT delay)

Consider this apparently simple case of a delay block of some `FSQRT`s, followed by a number of local stores (to the same simple in-cache address), which only jumps at N=~330? Here we have to tread more carefully.
It's undeniable that
- execution does not seem to be hindered by any resource limitation before the 330 entry ROB limitation
- there are not enough ways (either actual store queue slots, or "virtual slots" by holding store instructions in the LS Dispatch Buffer+Scheduling Queue) to get us close to 330 storage elements

- BUT! we said that stores have to be retained until they are non-speculative; we cannot allow speculative stores past the LSQ storage to make it out to the L1$!

If we equate “non-speculative” with “Retired”, then we have a problem because these stores have not Retired – that’s the whole issue of them still present in the ROB and using up the ~330 available ROB slots they can occupy.

Conceptually one might imagine two ways out of this problem.

One possibility is that we are, in fact, overwriting the same address every cycle. So *in theory* there's no need (once you establish that there are no intervening loads or anything else of interest between two

stores) to care about the older store value, all that matters is the newer value.
Under the right conditions Apple might cull older stores as being idempotent (ie they don't change the state of the machine).
This is easily understood, but I'm unaware of any core that attempts exploit this fact (which, ideally shouldn't happen in most code!)

Can we test this?
I tried changing the storage address so that every store stored to a successive location, and saw no difference.

But that's still not *absolutely* definitive, because how large is the amount of storage attached to each Storage Queue slot? It could be as large as a cache line! Maybe successive stores are aggregated somehow as a single unit in a single Storage Queue slot? Ambitious! But not impossible.

So I separated successive stores by 64 bytes and again, no change, the jump in the curve is at ~330 stores.
The issue is not related to idempotent stores.

The second possibility is that remember the whole point (the only point) of the store queue is to hold onto stores as long as they are speculative.
Not until they retire, not to force some sort of program order, *only* so that they don't escape out to cache until they are non-speculative.
Which means that as soon as we know they are non-speculative, the Store Queue can be released, and the store allowed to proceed to cache!

And this is, in fact, what happens. Essentially for every instruction in the ROB, Apple is tracking two things, both
- when the instruction Completes and
- when the instruction becomes Non-Speculative (ie every earlier instruction that was speculative or could cause a fault of some sort has successfully completed).
At the point that stores become both Completed (ie their TLB lookup has indicated that they are not problematic) and Non-Speculative, the store data can be sent off to L1$, and the Store Queue entry freed for re-use, regardless of how far from retirement the store is.

As usual, there's a patent confirming this hypothesis: (2015) https://patents.google.com/patent/US10228951B1 *Out of order store commit*.

2006 (historical interest, early release of store queue data for a strong memory model)

To understand the full flow of thought, we begin with (2006) https://patents.google.com/patent/US20080086623A1 *Strongly-ordered processor with early store retirement*, a patent from the PA Semi

days, and not especially relevant (because it's about a strongly ordered memory model -- perhaps PA Semi thought they might be forced to make x86 designs?). The patent says, essentially, that it can't think of anything to do about early release of load queue entries, but that there are circumstances under which store queue entries might be early-released. (Well, not exactly. What they seem to have in mind is almost the reverse of our concern. Assume a Store that is Non-Speculative and ready to Retire. At this point it tries to store its data, but the Store misses in L1. The idea seems to be that we will perform the Retire anyway, we will store the Store Data in some Cache buffer (to be merged with the cache line when it is delivered to the L1D), and we will free the LSQ entry.

2013 (early release of non-speculative loads)

This is followed by the more optimistic (2013) https://patents.google.com/patent/US9535695B2 *Completing load and store instructions in a weakly-ordered memory model* which we've already described (early release of loads once they become non-speculative).
The early release of loads when non-speculative is, perhaps, obvious (at least for a weakly ordered memory model); the patent is mainly about how you can implement separate load and store queues that maintain ordering relative to each other, while allowing both loads and stores to be (in-order) removed from one end of the queue while new instructions are added to the other end. It's worth looking at to see how these sorts of queues are implemented without having to move data between slots, and how wraparound is handled – but it will make your head hurt!

2015 (early release of non-speculative stores)

Finally we get the above-mentioned (2015) https://patents.google.com/patent/US10228951B1 *Out of order store commit* which gives us more aggressive store commit.
The difference here compared to the 2006 case is that 2006
- allowed stores
+ to release resources, and
+ commit to cache,
+ before retire,
- but this had to happen in ROB order.

2015
- allows the resource releasing out of ROB order.
Specifically, if the oldest store to be committed is a cache miss, then, while that cache miss is being served, we allow younger [but no longer speculative] stores to be written to cache.

It's interesting that they don't tell us how you can operate a low-power queue if you are allowed to remove items from the middle of the queue – obviously the circular queue techniques of the 2014 patent won't work!
Presumably they use ideas like those of the non-shifting reservation station, like we saw in (2015) https://patents.google.com/patent/US20170024205A1 *Non-shifting reservation station* when discussing

Scheduling .
(And it gives one confidence in Apple's design process if a good idea invented for one part of the CPU is immediately adapted to solve a similar problem in a different part of the CPU.)

2018 (how the machine tests that a load or store has become non-speculative)

This series of patents ends, for now, with (2018) https://patents.google.com/patent/US10628164B1 *Branch resolve pointer optimization*. In all the above work we refer to stores (or loads) as becoming non-speculative, and we know conceptually what that means, but not exactly how the machine implements it. The patent describes a way of implementing this. The problem is not quite as simple as you might think, because any given store might depend on multiple branches ahead of it, and those branches can execute out of order.
One way you might solve this is through a variant of our good old dependency bit vector scheme, with something like a bitvector that holds all the predecessor branches that have not yet executed. But there's an easier solution, the subject of the patent.

Simplifying to convey the point, imagine the stream of execution as it passes through Decode, where every instruction gets a sequential instruction number.
- Then any given store can know the number of the last branch before it in program order.
- If the Decode unit also propagates that number down to the Load Store Unit which propagates it to the Store Queue, then every Store in the Store Queue also knows the number of the branch after it.
- Finally as every branch is executed (possibly out of order) the number of the *oldest* branch that's still in the Scheduling Queue is noted. (The patent is somewhat vague about this, but it's the only interpretation that makes sense.)
- This oldest not-yet-scheduled branch is sent to the Store Queue.
Every store now knows that it lives between the branch ahead of it with sequentialID M and the branch behind it with sequentialID N.
When the oldest not-yet-scheduled branch sent to the store queue matches sequentialID N, then every branch earlier (ie branch M and forward) has been executed, and so the store (or load) is no longer speculative.

This sounds reasonable, but I don't understand why the behind branch sequentialID is required; why not just compare the branchID that's propagated to the Load Store Unit with the sequentialID of the store itself? I suspect there are a lot of details (how much can you trust that the Scheduling Queues and Dispatch buffer can precisely identify the oldest branch that has not yet been scheduled?) that ultimately rely in some way on this second behind branch sequentialID, but honestly I cannot understand the patent as it is presented, even when I try to fix up the parts that seem clearly wrong.

To some extent these out of order commit concepts seem "obvious". However historically (and still!) out of order commit has been seen as too complicated to be worth implementing. (2019) http://uu.diva-portal.org/smash/get/diva2:1263287/FULLTEXT01.pdf Maximizing Limited Resources: a Limit-Based Study and Taxonomy of Out-of-Order Commit is one of my less favorite papers (grindingly repetitive, and with singularly unhelpful graphs), but it does explain some of the issues in out of order commit. By their terminology it appears that Apple is using a somewhat (though not fully) aggressive version of

what they call safe out of order commit.

2011 (early version of the store queue storage structure)

(2011) https://patents.google.com/patent/US9131899B2 *Efficient handling of misaligned loads and stores* is about something different, but it describes the store queue structure at the time.
The store queue consists of two parallel storage arrays; one holds addresses, the other holds the stored data.
The address array is split into two banks, an even bank and an odd bank, based on bit 7 of the storage address. We'll see how this matches the design of the L1D, but essentially what it means is that bits 0..6 of an address give an offset within a 128B line, while bits 7..13 describe which line within a 16kB page (and thus describe the sets of the cache). Thus the store queue, like the cache, is split into even and odd halves based on the lineID.
Each storage slot of the odd half and the even half share a page address.

I think the way the design is supposed to work is that one entry refers to one executed storage instruction.
- In the normal case (a store that does not cross a cache line) the page entry and one of the even or odd line halves is valid; the valid half gives the address of the store.
- In the line crossing case the page entry is valid, both line halves are valid, and they refer to a line and its successor; by the pattern of the two addresses, you can tell how to split the two parts of the store data across a line.

Thus we have a structure that
- records the store data (regardless of alignment) separately simply as a blob of 8..128 bits
- records the store address in a way that's split by even/odd lineID and so makes it fairly easy to detect and deal with stores that cross a cache line (and means that, for most loads and stores, only the even or odd half of the store queue has to be probed, perhaps saving a little energy)

You might ask what about stores that straddle not just a cache line, but a page boundary? They could be treated as two separate stores (use two entries) but the patent suggests using a third small structure (not described) to handle those (presumably rare) cases. And additional complication you probably didn't think of in that case is that (one way or another) the store has to be sent through the TLB twice to translate both of the pages that are touched.

This structure (and its matching equivalent in the L1D) mean, among other things, that the address parts of a load matching this earlier store that crosses a line can be handled simultaneously, and the store data delivered as a single package to the load, so a line-crossing load is no more expensive than an aligned load. The same essentially holds true for line-crossing loads whose two parts hit in the cache.
My guess is that even if the different parts of a store are written to their separate cache lines in separate

cycles, the entire entry will not be reused until both parts of the store are written – meaning that the store queue can always service the entire load.

The nastiest case is a store that provides part of a load (think a double byte store that crosses a cache line, followed by a double word load that crosses a cache line and covers that store). Presumably in this case data has to be provided from both the store queue and two cache lines (and at least one of those cache lines could miss in cache), and it all has to be stitched together! The LSU provides auxiliary storage, called the Sidecar, to hold the various temporary pieces during this operation.

(2020) write combining buffer

Recall that the primary goal of the Store Queue is to hold *speculative* stores. We've suggested ways to extract more value from the Store Queue by treating it as an L0 Cache and encouraging it to hold more data. One problem with that line of thought is that, in line with the Store Queue's primary goal of handling speculative stores we have to think very carefully about how aggressively to aggregate successive stores (especially small stores like bytes) into a single Store Queue entry. The task does not seem hopeless (perhaps use a color/generation that changes every time we cross a speculated branch, and allow successive stores with the same color to aggregate, and then complete together?) but there are limits to how aggressively this can aggregate, especially when you consider eg the structure of a loop of stores.

Another alternative is to give up on using the Store Queue per se in this way (perhaps beyond some simple color-based aggregation to make it effectively larger?) and add a different type of structure, namely a set of Write Combining Buffers. These differ from the Store Queue in that they are populated by by stores that are no longer speculative, and so aggregating is no longer a problem. The advantage of this aggressive aggregation is that we save power and bandwidth. If we can aggregate multiple stores in a Write Combining Buffer, when that buffer is full we only need to take up a single L1 cache cycle to move the aggregate to the L1, and if we aggregate enough to fill an entire cache line, we can write the entire cache line to L2, avoiding
- L1 bandwidth
- allocating an L1 line that (more likely than not) will never be referenced before it ages out to L2 anyway
- reading the line from L2 (or even SLC or DRAM), and then fully overwriting it just a few cycles later.

So there's clear value in having at least one or two Write Combining Buffers sitting between the Store Queue and the L1 cache. But once you have such structures you have the same problem as with any structure, namely the governing algorithms. In particular: on the one hand, we want to keep each buffer "open" as long as possible, on the off chance that eventually a new value will come in to aggregate with the existing values (and of course the larger the aggregation the better, up the best case of all where an entire cache line is covered); but on the other hand we always want to be able to cover a short burst of stores without slowing down the rest of the machine (which will happen if the Store

Queue fills up because it can't retire stores to the Write Combining Buffer, because no Buffers are free and we're waiting to write some out to L1 or L2). So how do we balance these?
The 2020 answer is https://patents.google.com/patent/US11256622B2 *Dynamic adaptive drain for write combining buffer*.
Honestly, as bottlenecks go, this one is pretty unimportant, and so not much effort has been put into making the solution especially sophisticated.

The simplest type of solution is something like: assume we have 16 Buffers, and, on average, 4 free Buffers is enough to absorb most bursts of stores (given that some of the stores will hit in early buffers, and many of the stores will aggregate in the 4 free buffers). Then we establish a highwatermark of 12, and once we have more than 12 buffers active we start draining them (ie transferring them to L1), while continuing to accept non-speculative stores from the Store Queue, and ideally we never have to pause for running out of Buffers.
The problem with this is that while most code mostly has stores close together (eg filling up a cache line) some code may write to widely different addresses. One type of example is changing just one field in each entry of an array large structs (which might translate into, eg, changing two bytes in each cache line). Another type of example might be a stream of unpredictable, almost random, writes, as in writing to a hash table. For such code, a highwatermark of 12 may not work very well because the writes do not aggregate well in either previous or newly allocated buffers, each buffer only captures one or two writes, and so it makes sense to switch to a substantially lower highwatermark, perhaps 8 or even 4.
And that's essentially what the patent is about – track the number of non-merging stores (ie stores that require a newly allocated line) vs the number of merging stores and if it grows too high, respond appropriately, either by switching to draining immediately, or by modifying the highwatermark.
There's a slight tweak to this in that we may hold off draining if the L1 cache port is in use (which is another way of saying that stores always have lower priority access to the L1 cache than load [and probably prefetches?]) but nothing more. One could imagine some more sophisticated tweaks. For example which lines are chosen to be drained? It seems to be either random, or a roundrobin over the store structure. A fancier scheme might track how "efficiently" a Buffer is being filled and drop Buffer with lower efficiency, or try to hold for longer onto Buffer that are close to full. But that would take area and energy, and is perhaps just not worth it; the common cases that result in efficiently filled line will be handled more or less automatically anyway.

Testing the LSDP

Let's see what we can learn about the LSDP. The idea is to create varying types of store/load pairs and see how performance varies.
We build up complexity gradually.

independent addresses

To calibrate, start with a basic load and store to two different, unchanging, addresses, using two different registers:

```
STR x10, [x2]; LDR x11, [x3]        (x2!=x3)
```

There should be absolutely no interference here between the store and the load, and that's what we see.
We see nothing unexpected, and we see a throughput of 800 store+load pairs takes essentially 400 cycles, so two pairs (four memory ops) per cycle.

Now a slight variation:

```
STR x10, [x2]; LDR x10, [x3]        (x2!=x3)
```

What's different here? Note that the load now feeds into the store.

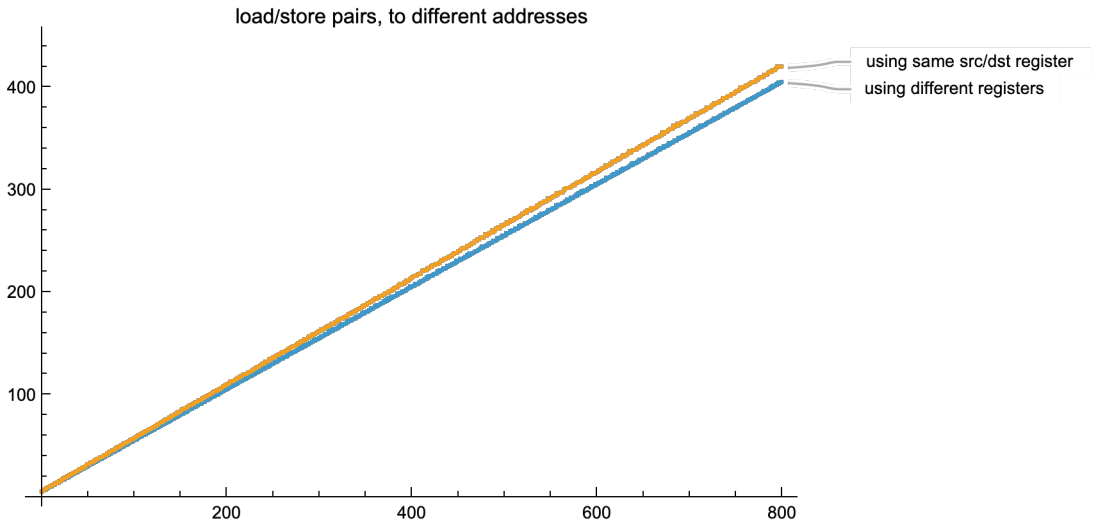
Your first thought might be that this has to run sequentially, each instruction waiting for the next. But be careful.
Write the code as

```
STR1 x10

LDR1 x10
STR2 x10

LDR2 x10
STR2 x10
```

etc

`LDR1` and `LDR2` are independent (they will load to different physical, renamed, registers).
What a common register enforces is that the store (one store) has to follow the load. That's all.
Hence there is nothing preventing each pair (as drawn) from executing in parallel, two per cycle (two loads, two store per cycle).
And once again that is what we see.

Out[617]=



The case of a chained dependency through the same register (gold line) is very slightly slower than the case of no chained register (blue line) but essentially the same speed of two pairs per cycle!

Given this understanding, we can now see why slight variants on this, using two different register widths, behave the same way.

`STR x10, [x2]; LDR w10, [x3]` (x2!=x3)

and

`STR w10, [x2]; LDR x10, [x3]` (x2!=x3)

also run at full (two pairs per cycle) speed.

same address (no prediction required)

Now we do the same thing, different registers but load/store to the same address.

`STR x10, [x2]; LDR x11, [x2]`

Once again your intuition is that we have forced a dependency, but again let's be careful. So write the code as

```
STR1 [x2]
LDR1 [x2]

STR2 [x2]
LDR2 [x2]

STR3 [x2]
LDR3 [x2]
```

Now think about how this code has to execute (for correctness) and how it will execute.

For correctness we require that

- every load (appear to) execute between the two store and
- every store (appear to) happen sequentially
But our OoO machinery is very flexible in how this is actually implemented.

The stores can occur out of order, just as long as the (address, data) pairs are placed into the Store Queue in the correct order (which is achieved by giving each store a Store Queue location at Mapping/Rename (while the stores are still in order).

The loads can also occur out of order, the only constraint being that each load of a pair must occur after its matched store.

So, conceptually, what now happens is that a whole lot of these loads and stores are thrown into the Dispatch Buffer and the Load/Store Scheduling Queues. Ideally their age is preserved in this process, and they are scheduled in order, but the Dispatch Buffer doesn't promise to preserve order, and the ambidextrous Load/Store unit may occasionally have a load placed in it the Dispatch Buffer rather than the perfect balancing of always stores. So ordering will occasionally be imperfect.

Assume perfect ordering. What should we expect? Some of this I have to guess, because no-one gives full details, but we have something like:
STR1 has been split into two instruction, StoreAddress1 and StoreData1, both targeting a particular slot in the Store Queue (slot allocated already).
STR1 (both parts) and LDR1 issue together.
Store Data places its data in the Store Queue slot.
Store Address and Load calculate their address, and perform TLB lookup in sync.
At that point, the next cycle or so, we have something like
- the store will place its address value in the Store Queue slot
- the load will send its address value to the L1D, and to the Store Queue
- if the timing is set up correctly, we can have something like the store places its value in the Store Queue slot in the first half of a cycle; the load performs a comparison against all the Store Queue slot addresses in the second half.
- so the load sees a match, the store data is already present, and the load completes (acquiring the data from the Store Queue rather than the cache).

If we have imperfect ordering (the load occurs one cycle or more earlier than the paired store) then what will happen is
- the load looks in the Store Queue, finds no matching entry, and gets the data from the L1D cache

For most cores that's the end of the story, at some later point the Store will execute, will compare its address to the addresses in the Load Queue, will see that its paired load has already executed (and acquired data from the L1D) and that load will be marked as having to be Flushed (or some similar recovery procedure).

For Apple (the 2019 *Load/Store Ordering Violation Management* patent we have already discussed) what will happen is that every store *in progress* will also compare its address against every load *in progress*. This means that, for example, right after address generation (even before TLB lookup) the store will have some sort of indicator of where its data is going. This can be compared with the in-progress load's virtual address and if they match, then that's simply an early version of the load's eventually matching the address in the Store Queue! (Of course this mechanism is not perfect; it won't catch weird cases like the store written to a physical address that's the same as the load, but via a virtual address that is different. And it may not catch weird partial overlap/alignment cases. But it will catch *most cases early*, with all cases being caught by the later tests.)

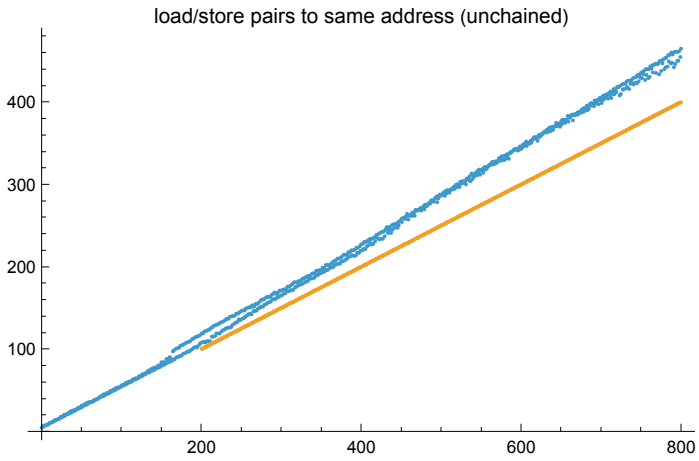
This means that even if the store started two or three cycles after the load, the there may be enough of an overlap in time for the load to detect that it has a matching store in progress and should not get its data from the cache. In theory the load might be able to get the data from the Store Queue (the store that matched its virtual address with the load knows its Store Queue slot, and the Store Data instruction may already have dumped its data there). I don't know if Apple does that. Even if they don't, the load can be marked as a Replay, and so it re-executes a cycle or three later, at which point the Store has fully completed, and the load can match its address against the Store Queue and pick up the data from there.

This is what we see in the curve. There's clearly some messiness (the occasional Replay) but overall our performance has not slowed down too much.
The gold curve (I only drew half of it to leave the lower part of the curve easier to see) shows what would be perfection (2 load/store pairs per cycle, no cycles lost to Replay or Flush) and we run about 15% slower than perfection. Not ideal, but we're not losing too many cycles, even in this worst possible scenario, where basically every load and store are setup so that it's easy for them to cause a Flush.

Note that what we have seen so far can be summarized as:
- if a register is shared, store has to happen after load
- if an address is shared, load has to happen after store
but in both cases, with only one of these two (register and address) shared there are no *serial* dependencies. There is only the dependency of each individual load/store pair, and the pairs can all execute independently. Hence we run at essentially two pairs per cycle in all these cases.

- no LSDP is necessary because the addresses (of the load and store) are known early enough. Ideally the loads execute at the same time as the stores (or a cycle later) so when the load looks up an address in the Store Queue, it finds its matching address.
Even in cases of a few cycles of slip between the load and the store we are still OK without an LSDP because our ability to compare an in-progress store address against every in-progress load address catches these cases and handles them without much overhead.

Out[619]=



same address but delayed data (prediction required to avoid replay)

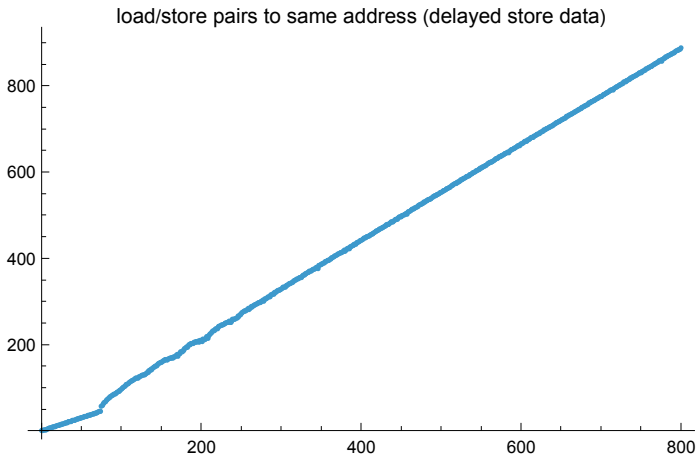
Now suppose that the data of the store is only available very late, but the address of the store is easily available. We would expect now that, just as above, mostly the loads and stores execute in sync, and the load hits a point where it has matched in the Store Queue, but the store data is not available. So it will have to Replay once the Store Data part of its associated store executes.

We can force this be using a slow instruction like FVCTAS to generate the store data, so our probe looks like

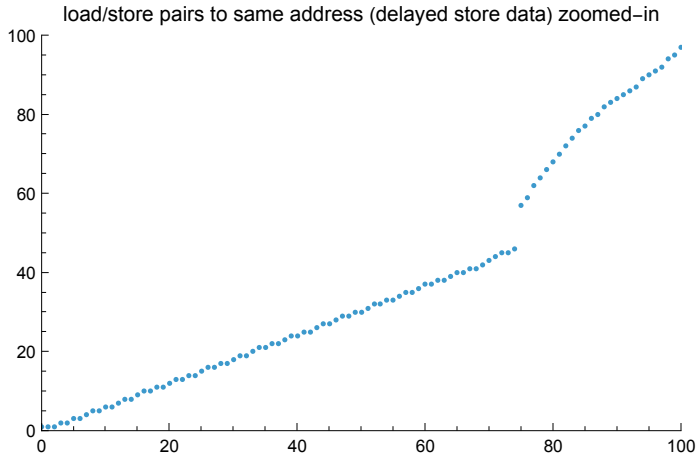
`FCVTAS x10, d1` (with d1=0.0) this instruction has a latency of ~13 cycles and converts d1(=0.0) to an integer

```
STR x10, [x2]; LDR x11, [x2]
```

Out[621]=



Out[622]=



Honestly, this is better than I expected!
There are clearly two regimes.
Up to 74 pairs, the machine processes about 1.6 pairs per cycle. Not quite two pairs, but close'ish.
After 74 pairs, the machine processes about .8 pairs per cycle.

So let's think about this.
Ideal execution would be that each load is delayed long enough that it only issues just before the x10 value is ready to be stored.
The machine is not psychic enough to do that!
But what it can do is record each time a load issued too early compared to its store, and record that in the LSDP.
Then the next time the load is placed in the Scheduling Queue, it will wait around with the store as a dependency, and will not execute until the store has executed (address and data).
If the machine could do that perfectly, then once the predictor is trained we shouldn't have to lose any cycles, we'd just have every load waiting around (for a long time) in its scheduling queue until its store

was ready. In the real world the delay between the load and the generation of the store data is long enough that scheduling queues will fill up, instructions will slip out of order, cycles will be lost while no appropriate instruction can be found in a particular queue, etc.

What we see is that we get close-ish to that ideal case for up to 74 load/store pairs. This suggests
- that the LSDP can hold ~74 entries
- when everything is working, load is delayed by the LSDP to the correct time required by the store data, so the load does not have to Replay.

Once we get past the capacity of the LSDP we switch to essentially one pair per cycle. I interpret this as meaning that the execution of every paid proceeds like
Store Address + Load (execute same cycle)
Load - sees no valid store data, sets a Replay dependent on the Store Data arriving for that particular Store Queue slot
Load - Replays successfully.

Now that each load of the pair has to execute twice how long would we expect a pair to take?
If 100 pairs take 50 cycles (no Replay), then naively 100 pairs take 100 cycles (Replay, each load executes twice).
This isn't quite correct because in principle three loads can execute per cycle if there are no stores (but can this happen for Replay conditions?), but let's accept it. Then we'd expect ideal throughput to drop to 1 (or a little over 1) pair per cycle.
We have all the imperfections of before, plus new collisions that can occur in the LEQ, so let's just round up the .8 per cycle we see to close enough to 1.

I think it's reasonable to conclude that
- we have seen the LSDP make its appearance
- it can hold about 74 entries
- when working correctly, we don't lose much load/store performance to the severe out-of-order execution generated by the slow production of the store data
- even when the LSDP can't help us, this particular case (address known, just data unknown) is handled adequately by the Replay mechanism. We lose half our throughput (since every load has to execute twice) but no more than that.
And as always what we are testing here is extremely unbalanced code! Normal code may well have store data that is very delayed, but it is unlikely to have a load of that store data immediately after the store, or to have such a high density of loads that the cost of the extra Replay loads is a noticeable additional burden.

same address, same register

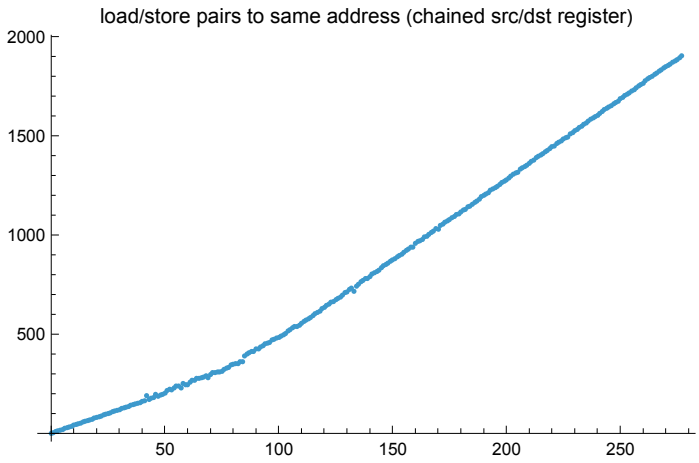
Now use a chained register:

```
STR x10, [x2]; LDR x10, [x2]
```

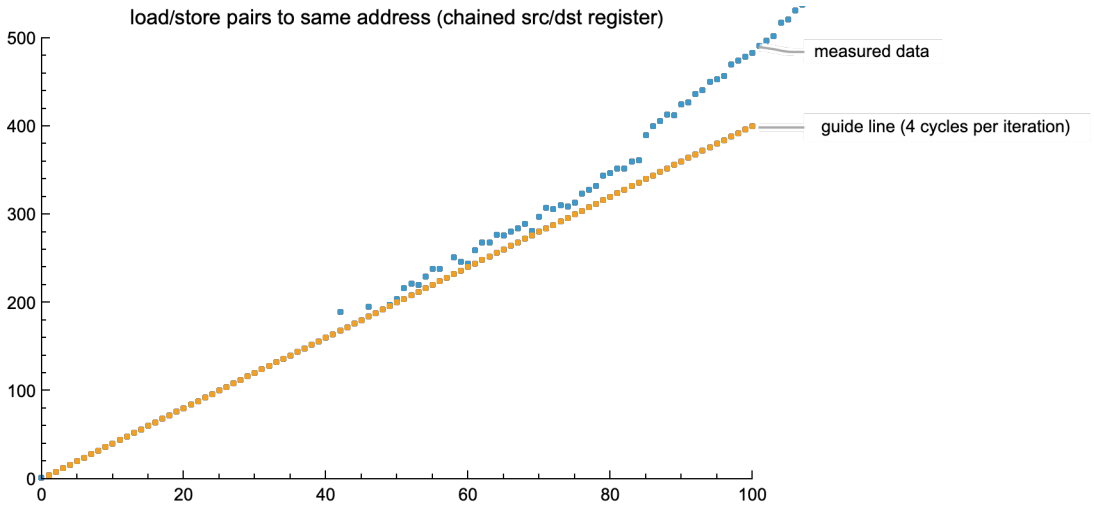
Unrolled this looks like:
STR1 x10, [x2]
LDR1 x10, [x2]
    STR2 x10, [x2]
    LDR2 x10, [x2]
        STR3 x10, [x2]
        LDR3 x10, [x2]

Earlier we stated that a common register requires the store to follow the load. And a common address requires the load to follow a store.
So we're basically stuck.
Unlike the previous cases, we can no longer separate the chain of instructions into independent load/store pairs that are run in parallel.
The best we can hope for is to run STR1 and LDR1 in parallel, meeting after three cycles at the STQ entry, another cycle to get that data value up to the bypass bus (and stored in the physical register corresponding to LDR1's x10); then we start the next pair of STR2+LDR2.
So we expect best case throughput of one pair per four cycles.

Out[624]=



Out[625]=



Clearly there are two regimes.
The initial regime can support about 70..80 load/store pairs. While in that regime the cost of a single load/store pair is about 4 cycles (slope of the initial section).
Once we transition to the slow regime, the cost of a single load/store pair jumps to about 8 cycles.

So this matches our analysis.
One thing it confirms is that loads and stores to the same address (ie the pair `STR [x2] LDR [x2]`) do indeed behave optimally. They can issue together, and the store address can be placed in the Store Queue early enough for the load to detect that address when it probes the Store Queue.
For the load to acquires its data from the Store Queue rather than the Cache is called *Store Forwarding*.
(In fact there are multiple ways to perform Store Forwarding. The way Apple actually implements this is probably via the Register File. This will be explained soon, in the Load Accelerators section, with a brief discussion below of Zero Cycle Loads.)

Secondly it confirms that the size of the LSDP seems to indeed be around 74 elements.

It also suggests something about the replacement policy for the LSDP.
My assumption is that the replacement policy for the LSDP is not especially sophisticated. Suppose you want to add an entry to the LSDP? What slot do you use?
If any slot is marked invalid, that's an obvious choice.
If there are no invalid entries, then low confidence entries are the next obvious choice.
Then entries that are used to avoid a Replay rather than to avoid a Flush (since Replay's are much cheaper than Flushes).
But when all entries are essentially the same in these respects, what do you do?
- Random is one choice. This requires remembering no state.
- Another is LRU, which, in exact form is difficult to implement.
- Another is MRU, also difficult to implement and probably optimal for streaming situations.
- A fourth option is to track the entry number you replaced last time and just increment that with

wraparound.
For "normal" code with a mixture of memory references of all sorts, this will behave like Random, skewed towards LRU (good).
For streaming code, it will behave like LRU, which is pessimal.

This fourth option seems to me to be what we are seeing.
In the earlier case where we had delayed store data, and a clear break in the curve, that suggests LRU (behaving pessimally, with essentially zero reuse).
This current curve has a much less pronounced break at the transition point, so it's behaving more like Random replacement of the LSDP, I'm guessing that's because entries are being added to the LSDP in a somewhat more random manner as execution proceeds. (Before the LSDP is populate, the stores will all be held back relative to the loads, because the x10 dependence is immediately visible to the scheduler; which means multiple loads will initially start too early compared to their subsequent stores. Throw in the ambidextrous load/store unit, and pretty soon you have a major mess in terms of the order in which the load store dependencies are detected and then recorded in the LSDP.)
Under Random (or Random-equivalent) replacement we'll see some entries being removed that are still being used, even as other entries would have been a much better choice for removal, so we should start to expect that even at only N=50 or so, we start to see some occasional slowdown because some loads that should have been held back by the LSDP have had their entries replaced prematurely and sub-optimally.

For most code the choice of LSDP replacement policy probably doesn't matter much. If you are writing code that is streaming (so it has a controllable structure and it's constantly generating loads that match the address of recent stores, you're probably doing something terribly wrong!) For normal code, Random is a reasonable choice, and if Apple do what it looks like to me, they get something probably a little better than Random because it skews to LRU.

The second regime is as we discussed before. In the second regime We are still trying to execute like
STR1 x10, [x2]
LDR1 x10, [x2]
    STR2 x10, [x2]
    LDR2 x10, [x2]
        STR3 x10, [x2]
        LDR3 x10, [x2]
but we do not have the LSDP to enforce the precise synchronized scheduling of each store with its matching load. So most loads execute early relative to their stores, don’t see the data available in the Store Queue (but are caught as Replays) and are forced to Replay. So our minimal cycle becomes
STR1 x10, [x2]
LDR1 x10, [x2] (a cycle or two off earlier than the store)
LDR1 x10, [x2] (replay)

STR2 x10, [x2]
LDR2 x10, [x2] (a cycle or two off earlier than the store)
LDR2 x10, [x2] (replay)
STR3 x10, [x2]
LDR3 x10, [x2] (a cycle or two off earlier than the store)
LDR3 x10, [x2] (replay)

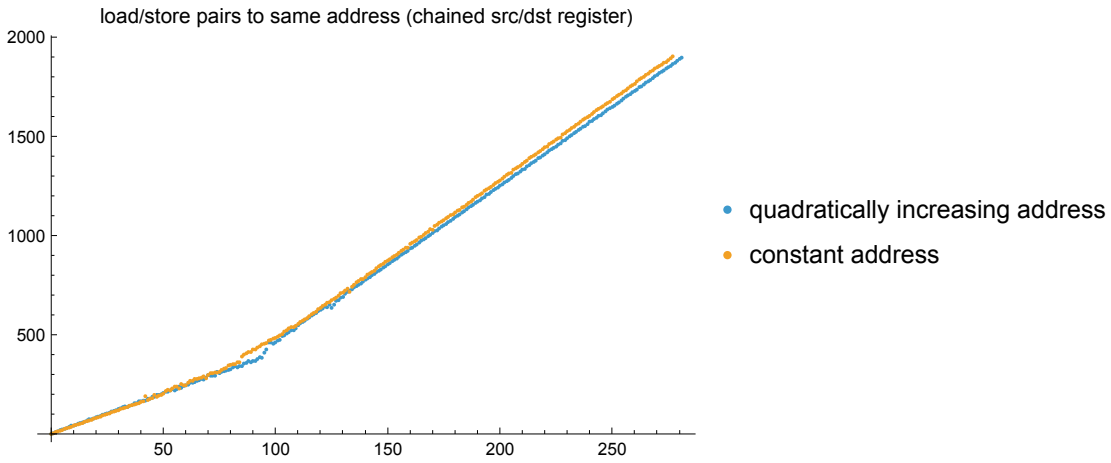
The minimal timing of one loop becomes not the latency of one load but the latency of a load+replay.

Experts might ask about Zero Cycle Loads in this context, and if that changes anything. Zero Cycle Loads will be explained towards the end of this load/store section.
The short answer is no, because the ZCL still has to be validated, and that takes the full timing cycle described above.
But to check this, I ran the code three times.
The second time used LDR x10, [x2, x0] (with x0 set to 0). An indexed address of this form is not susceptible to the basic ZCL but is still susceptible to the strided address ZCL.
Any simple attempts to bypass the strided address ZCL by varying the common address used by the load and the store is strided, so will still be captured by the ZCL! So let's use a quadratically increasing address stream!
initialization
MOV x0, #0; MOV x20, #1
loop
STR1 x10, [x2, x0]
LDR1 x10, [x2, x0]
ADD x0, x0, x20
ADD x20, x20, #1

but even this quadratically varying index stream (no ZCL!) has essentially identical results.

Remember that the LSDP matches a store PC with a load PC. The whole point is that it does not know that actual load and store addresses (if we knew those, we would not need a predictor!) So apart from the ZCL issue, we should expect (and indeed we see) no change even when the addresses being shared by the loads and stores change.

Out[627]=



force address prediction (delayed address)

Now we' re going to repeat these tests, but forcing the store address to be unknown at the time of the load. The plan is the same as how we generated delayed store data:
• generate an integer (slowly!) using
`FCVTAS x0, d1` (with d1=0.0) this instruction has a latency of ~13 cycles and converts d1(=0.0) to an integer
follow this with
STR x10, [x2, x0]; LDR x11, [x3] (x2!=x3)

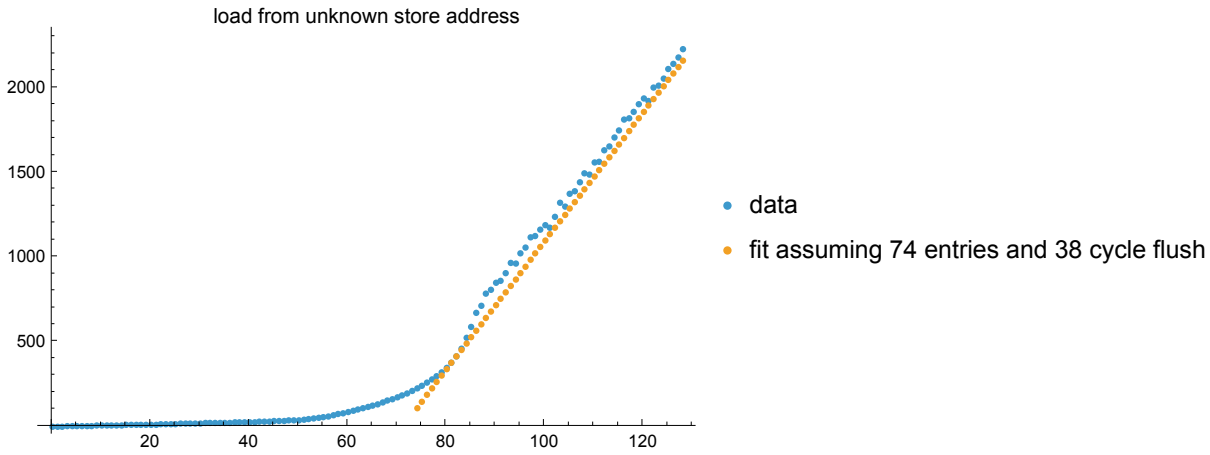
So the probe looks like repeated (`FCVTAS STR LDR`).
Once again the load and store are completely independent.
If the LSDP is working correctly, they should not interfere with each other in any way, and we should see no slowdown.
Even when we do overflow the LSDP, as long as loads are predicted by default never to match unknown store addresses we should still see no interference

And we see nothing unexpected, nice smooth two pairs processed per cycle.

load from (unknown) store address

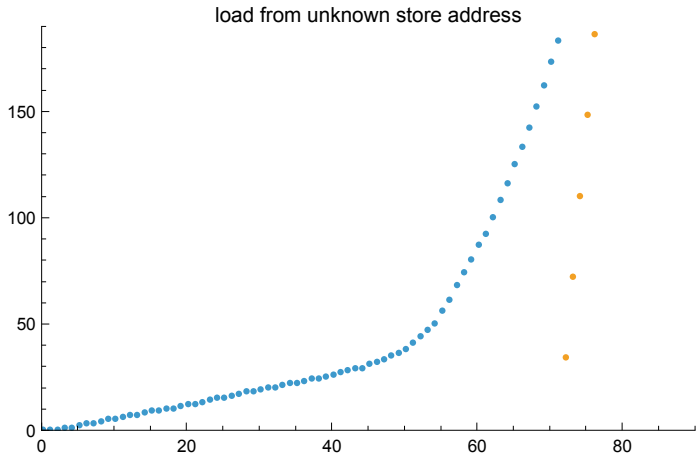
But now one small change, force the load and store to share an address:
`STR x10, [x2, x0]; LDR x11, [x2]`
Note that these correspond to the same address, but we are not forcing the load to delay on the x0, so it should be able to execute *well in advance of the store*.

Out[629]=



Yikes! Let's zoom in:

Out[630]=



So up till about 50 pairs we get adequate throughput. Not the 2 pairs per cycle I would expect, but something like 1.4 pairs per cycle. Honestly I can't see why the gap between what we expect and what we get is quite as low as it is. Each load will have to delay until its store executes, but in theory these should all be able to just wait in the various Scheduling Queues until the stores are ready, then immediately execute, with few lost cycles.

However we are more concerned with what happens once we exceed the capacity of the LSDP. The slope of the second half of the curve is about 38 cycles/pair!
I think this splits into something like 25 cycles as the cost of a flush caused by a load/store aliasing., and 13 cycles as the cost of performing the `FCVTAS` after the flush, either way the cost of a Flush is clearly phenomenal compared to either correct execution or a Replay.

The execution pattern is:
- the load executes (far in advance of the store, which is waiting for the `FCVTAS` result).

- It isn't stopped by the LSDP (which has overflowed and so has no matching PC) and so
- it executes assuming it matches no address in the Store Queue.
- Later the store address becomes available and is compared to the address in the Load Queue,
- we see that the Load picked up stale data (either from the L1D or an earlier entry in the Store Queue),
- and so the machine forces a Flush after that load.

(In principle doesn't have to be this way! Suppose the value that was loaded were stored in the Load Queue next to the Load Address. This could be compared with the Store Data and if they match, why Flush? No-one does this, but given that repeatedly storing and loading the same data is not that uncommon – sometimes for good reasons, sometimes not – it seems worth at least running a simulation to see what sort of a win this could buy you.)

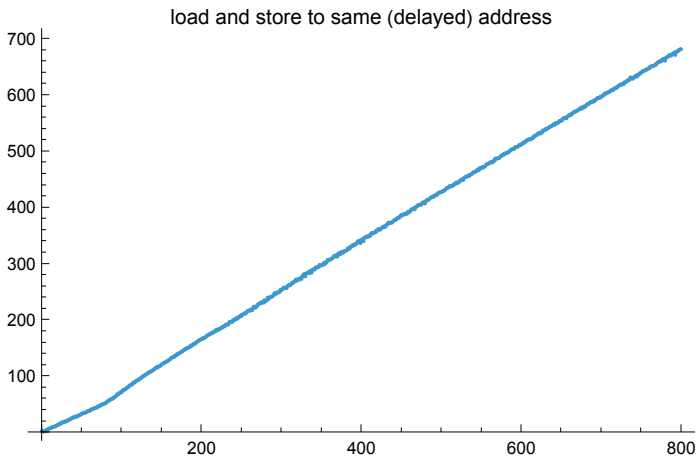
The curve suggests the LSDP holds only about 50 elements. But that's an illusion caused by random replacement and the massive increase in cycles when even just a small fraction of the load instances cause a flush. I added the gold line to both curves to show how we're still really dealing with ~74 entries in the LSDP.

force the load to be delayed like the store

What if we also force the load to delay, by making it also depend on the x0?

```
FCVTAS x0, d1; STR x10, [x2, x0]; LDR x11, [x2, x0]
```

Out[632]=



Now we see something like the first stage of the LSDP curve.
We see an initial slightly faster regime, about 1.5 pairs per cycle, for N<~70..80.
Then about 1.2 pairs per cycle.

So performance is adequate. Not great, when we compare with the 2 pairs per cycle when the load and store addresses are both known, but clearly we're just getting some sub-optimal scheduling (at N<74) and the occasional Replay (at N>74), not many Flushes.

As I've said earlier, I wonder if at least some of this glitchiness is the result of the ambidextrous load+store Execution unit?

Loads and Stores will be placed across the load and store Scheduling Queues as the Dispatch Buffer sees best, but because that one unit will occasionally switch from serving loads to servicing stores, the various queues

will slip out of perfect synchrony...

shift the timing of the load relative to the store

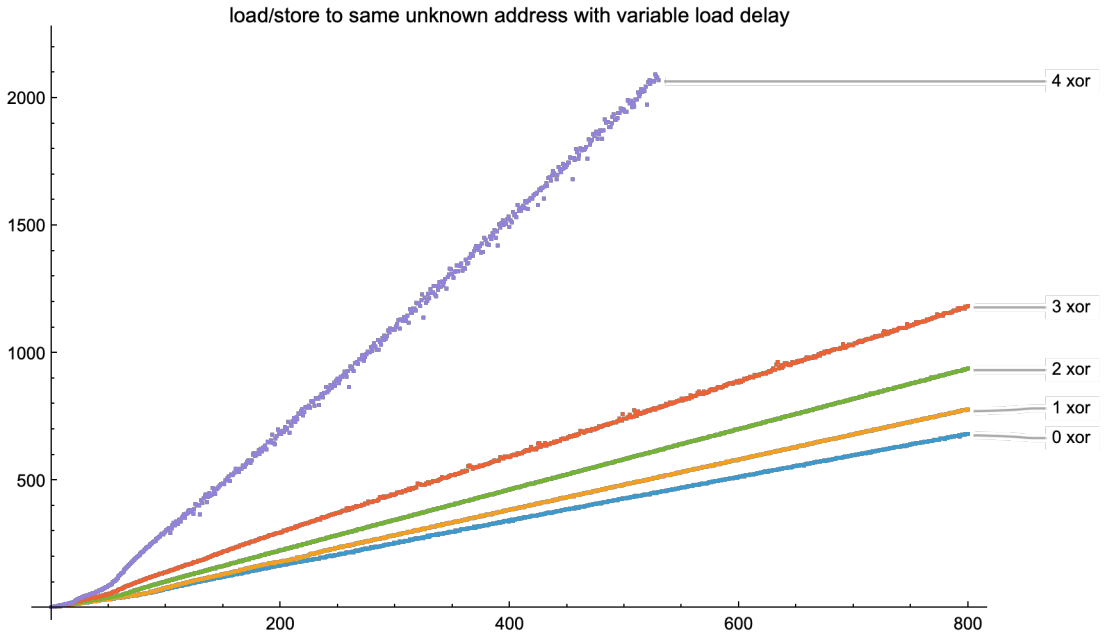
Now let's tie these ideas together. Suppose we have a probe that looks like

```
FCVTAS x20, d1
EOR x0, x20, x20; (followed by some number of EOR x0, x0, x0);
STR x10, [x2, x0];
LDR x11, [x2, x20]
```

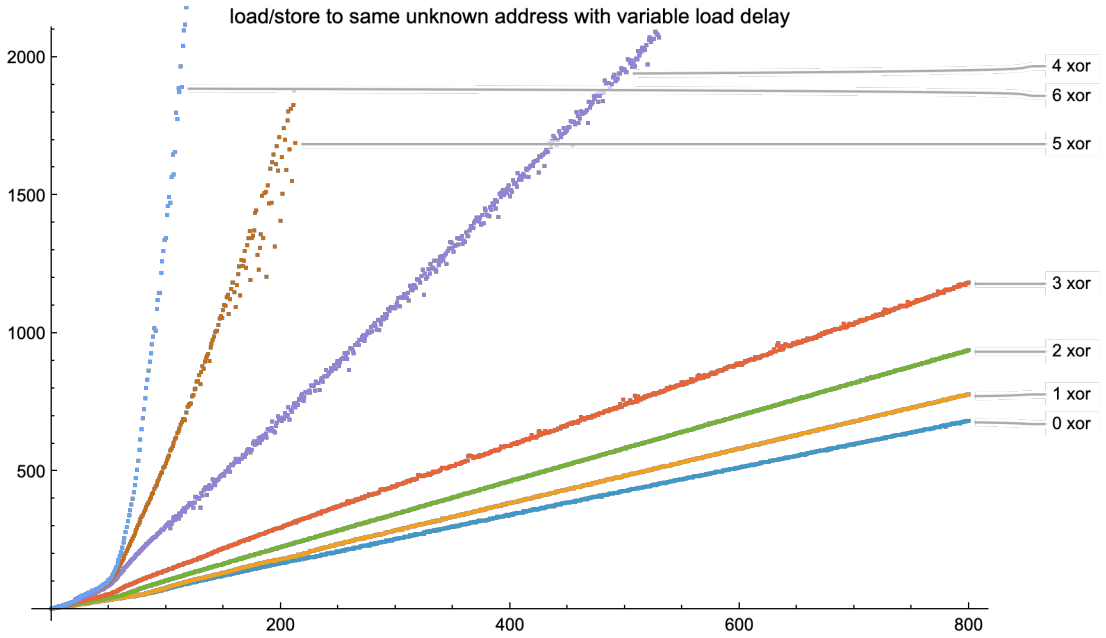
Remember that on M1, unlike x86, xor is not a zero'ing or dependency breaking idiom, so `EOR` takes one cycle, and the successive EOR's are chained, so each one adds a cycle of delay. Of course the EOR generates a zero value, so x0, like x20, continues to remain at value 0.

This means that (with just the first `EOR`) we have now forced the store to be one cycle later than the load, so higher chance of collision.
And we can vary the number of `EOR`'s to move the distance to two, three, ... cycles later than the load. What do we see?

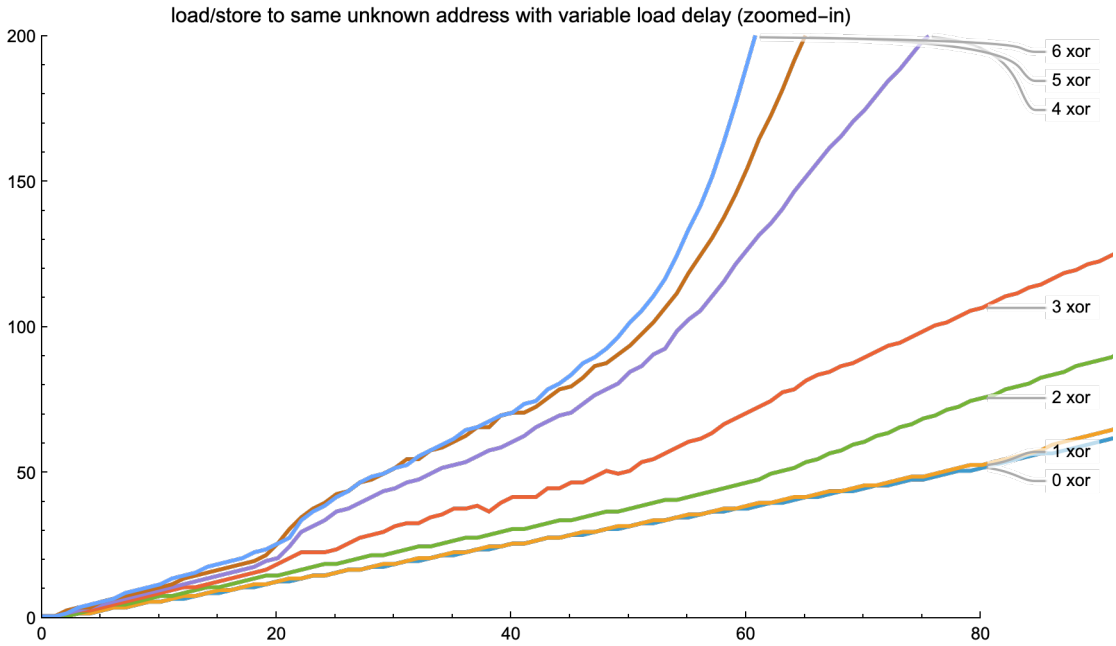
Out[678]=



Out[679]=



Out[680]=



So as we said, the baseline (no delay) gives us about 1.2 iterations of (FCVTAS, store, load) per cycle, so about .8 cycles per iteration.

As we add 1, 2, 3 cycles of delay between the store and the load, this increases to about 1, then about 1.2, then 1.5 cycles per iteration.

Obviously we're doing slightly more work (the xor's, but that basically trivial); more important is probably that ever more cases are being caught by late detection of load/store matching and being converted into a Replay.

By 4 EORs, each iteration is taking a little over 4 cycles, and pretty much every load/store pair is being run as a Replay. We start to see a small effect at N=~50.
At 5 EORs, each iteration is taking 10 cycles, and we clearly have a mix of some Replay cases with a fair number of Flush cases;
By 6 EORs (ie delay between the store and the subsequent load of 6 cycles) we're essentially at the case where every load/store pair generates a Flush, as soon as we exceed the capacity of the LSDP.

Examining the zoomed-in plot for small values of N, I think the case for N<~20 corresponds to enough temporary storage (STQ and Scheduling Queue) to hold all the pending loads and stores over multiple loops; enough so that a Replay doesn't slow us down because it can happen in parallel with other load/store pairs performing various parts of their executions.

Past N=20, without enough storage to hold all this state, we begin to have to delay processing later loads/stores because all the temporary storage is occupied by items in various stages of partial execution
- waiting for the value of an address index (x0 or x20),
- waiting for a chance to Replay.
And delaying those subsequent loads/stores make the time taken to process them visible because it's time not overlapped with other stages of the load/store processing of other loads/stores.

So we have that effect (exceeding temporary storage) hurting us up until N=~50, but even up to ~50 performance is not catastrophic, because the LSDP is preventing mostly preventing Flushes. We spend a few cycles with every load and store waiting in all the various temporary storage, but we don't Flush. Past N=50 we begin to Flush, and then everything goes terribly wrong terribly fast.

various nasty cases

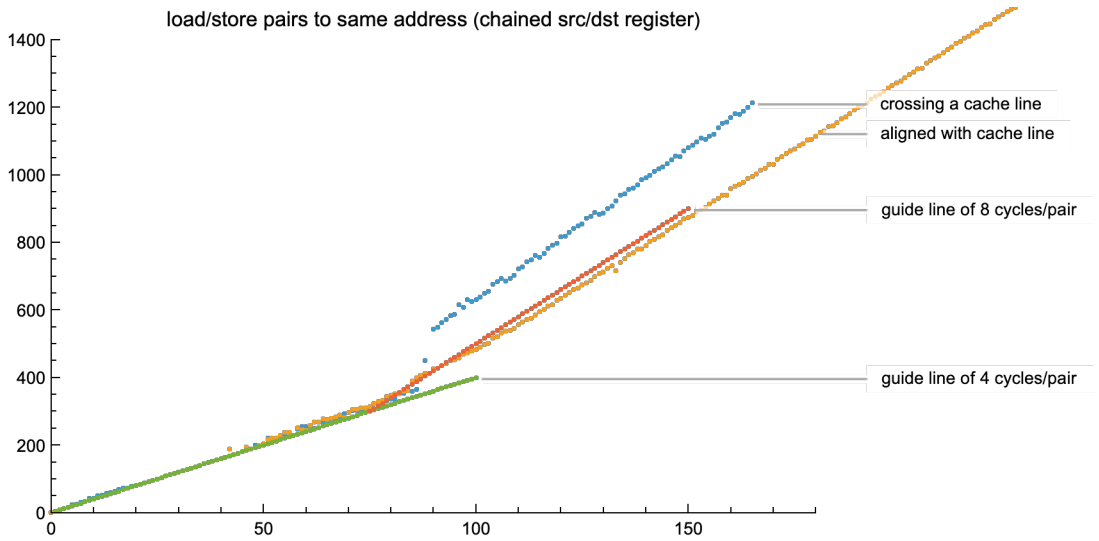
Let's now try a few nasty cases to see how Apple copes.

unaligned loads and stores

Till now we have had x2 and x3 perfectly aligned, at the start of a page boundary.
Let's add 127 to each one, so that now load/store of anything but a byte crosses a cache line boundary.
This has no effect on throughput of the basic
`STR x10, [x2]; LDR x11, [x3] (x2!=x3)`
case, which is fairly impressive! Somehow we are getting four *cache-line-straddling* accesses per cycle, even if we accept that Zero Cycle Loads are shouldering some of the burden.

The easy variants on this behave in the same way. The more difficult variants
(same register, same address, latency has to be at least 4 cycles and is frequently 8 once we overflow the LSDP) shows similar behavior.

Out[643]=



For few enough (~74) pairs, behavior is identical in both cases (load/storing at the beginning of a cache line, vs straddling two cache lines).
And there is a similar slow regime, where every load has to Replay once (the slope is slightly higher, about 8.5 cycles per pair rather than 8 cycles per pair, but essentially the same)

But there is a pronounced jump (by about 130 cycles!) between the two regimes.
I have no idea what's causing this. The only difference between these two cases is that the blue case performs its load/stores crossing a cache line.
This equivalence of the earlier, faster, regime (where we assume the LSDP is ensuring that Replay's are never required) suggests that the LSDP doesn't need to kno or do anything strange about loads or stores that cross cache line boundaries (which makes sense, since it is blind to the actual values of the load and store addresses).

How about the following theory:
- At the point where the LSDP stops being a useful predictor, the Store Queue is full of retired but not yet completed stores (split over two cache lines).
- One of the loads at this point generates a Flush.
- The Flush forces that entire Store Queue to be written out. Maybe this isn't normally done at Flush, but is a special case for stores split over two cache lines or something?
- Forcing out all that data generates the one-time obvious jump? (
+ 60 Store Queue entries,
+ each one has to be read twice (for each of the two cache lines), reads on separate cycles because each STQ entry is single ported,
+ each write to cache has to happen serially in sequence because they're all to the same address.

But I will admit this is special pleading, the result of the similarity between the size of the gap (~130 cycles) and twice the size of the Store Queue. Why don't we see a similar 60 cycle gap for the aligned case?

If we really engage the LSDP with our delayed address calculation
`FCVTAS x0, d1; STR x10, [x2, x0]; LDR x11, [x2]`
we again see the same behavior as before, with no serious differences.

vector load/stores & load pair/store pair

Now let's try vector load/stores. All the different variants behave as expected, including LSDP behavior, Zero Cycle Loads, and no problems with mixed used of q0 and s0, one for the load, one for the store.

Same is true for load/store pairs.

partially overlapping loads and stores

Now let's try overlapping loads and stores.
Start with

```
STP x10, x11, [x2]
LDRB w12, [x2, #0]; LDRB w13, [x2, #1]; LDRB w14, [x2, #2]
```

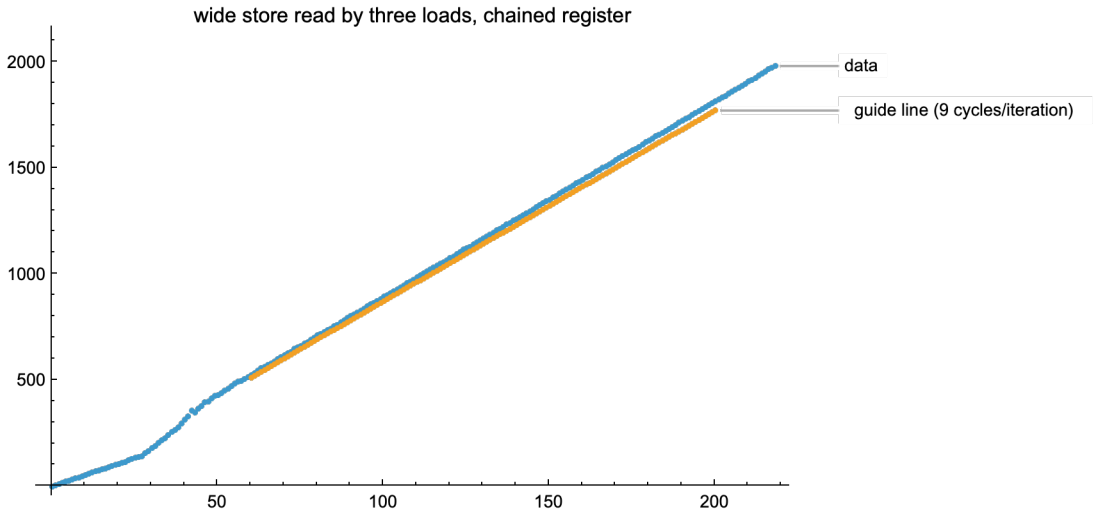
This has no real need for address speculation , but does involve store to load forwarding (and can't be faked with Zero Cycle Load tricks).
The basic unit involves one store and three loads, so (assuming everything is timed correctly) we should be able to perform one iteration per cycle.
And that's what we see!
(If we are correct that each Store Queue Entry is single ported, then the three loads will have to execute sequentially as far as loading their data is concerned. But they manage to co-ordinate this smoothly enough that there's no serious delay.)

Now let's introduce register chaining so:

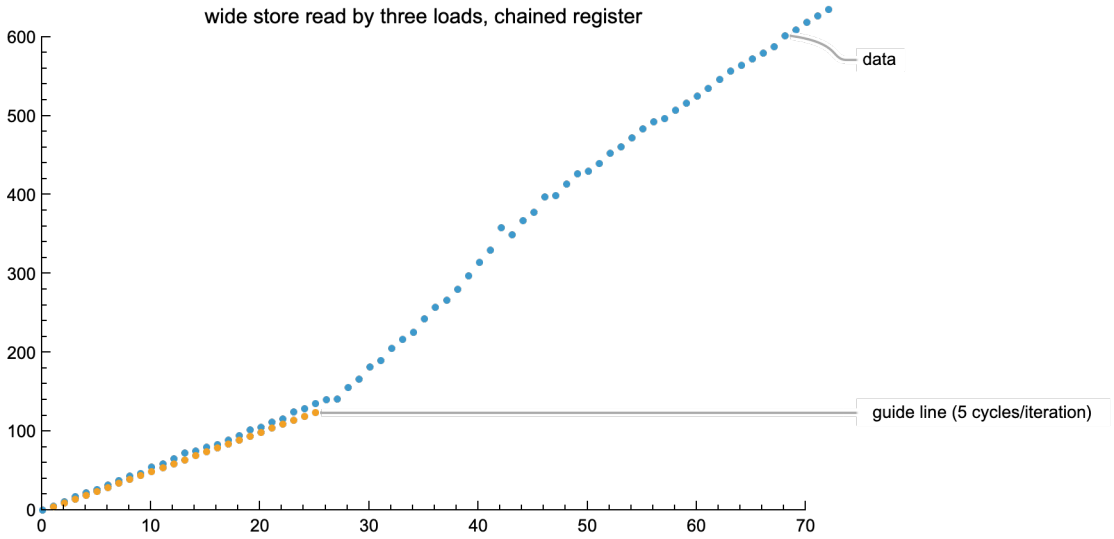
```
STP x10, x11, [x2]
LDRB w10, [x2, #0]; LDRB w11, [x2, #1]; LDRB w14, [x2, #2]
```

Now we do see something slightly different, though hardly a disaster.

Out[645]=



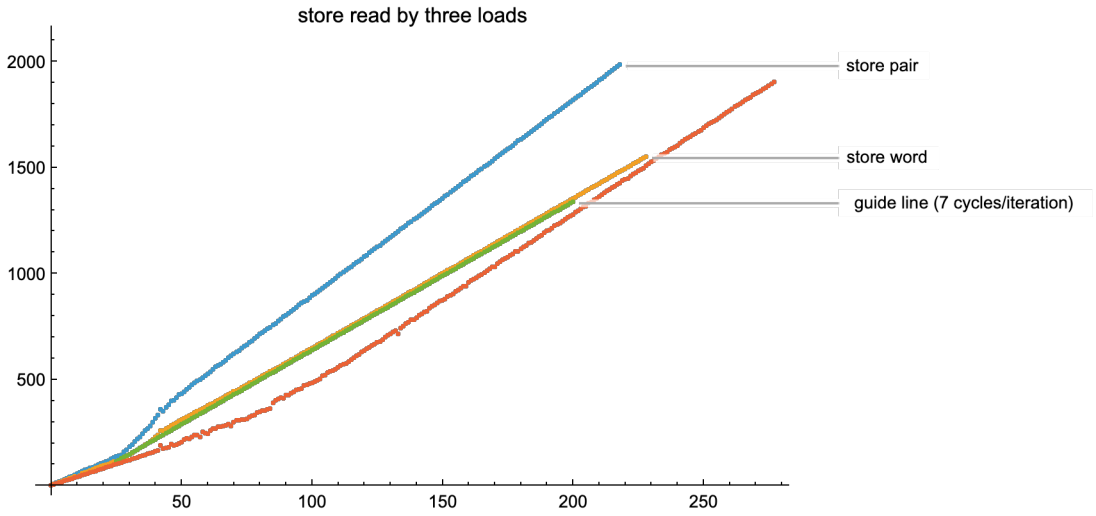
Out[646]=



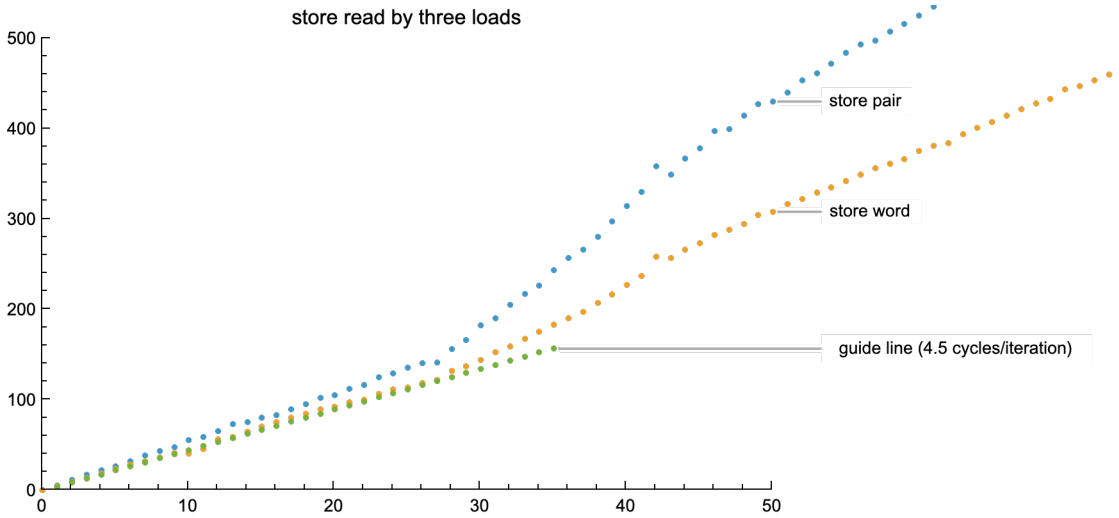
Overall much what we expect, but with a few differences in the details.
The largest difference is that we move to the slower part of the curve at around N=25 rather than 74.
Presumably each pair of (initial store, one of the loads) occupies an LSDP entry, so the effective size is reduced to a third?

The fast regime runs at 5 rather than 4 cycles/iteration. And the slow regime at 9 rather than 8 cycles/iteration.
Let's try to figure out where these differences come from.
First let's drop the STP and switch to storing just a single register.

Out[648]=



Out[649]=



So it seems that the primary problem was the Store Pair. Even in the case where the LSDP is fully functional, the primary loop

```
STP x10, [x2]
LDB w10, [x2]
```

(+ two extra loads)

is one cycle longer. In principle there's no reason why this is necessarily so that I can see. In principle the STP could be performed as a single 128b store that has no interest in the fact that the 128b were created from two registers.

If we replace the store pair with a store word we see that in the fast case an iteration now takes 4.5 cycles per loop. That's a consequence of the three loads, but stays at that value for two loads. (If we use only two loads, of course we now get the LSDP as useful up to N=74/2=37, but the slope remains 4.5 rather than 4). Likewise the high slope remains 7.

Now this is honestly rather strange! Why would the more complicated probe
`STR x10, [x2]`
`LDB w10, [x2]` (+ two extra loads)
run faster (once it has to continually Replay) than the simpler probe
`STR x10, [x2]`
`LDR x10, [x2]`
?
Even
`STR x10, [x2]`
`LDR x10, [x2] + LDB x11, [x2,#1]`
runs faster, at 7.5 cycles per iteration rather than 8!

The only thing I can imagine is that the presence of the extra load(s), once the LSDP is no longer helpful, breaks up sub-optimal scheduling patterns. The extra load (and the delay it engenders when it Replays to access the Store Queue slot) serves to occasionally delay the LDR x10, [x2] enough that it is not required to Replay.

Finally recall that
`STR x10, [x2]`
`LDR x12, [x2]`
took half a cycle per iteration, because the fundamental units are the STR/LDR pairs, and each of these can execute independently, so the only real constraint is we can perform 2 loads and 2 stores (ie two pairs) per cycle, ie half a cycle per pair.

Modify this to
`STRB w10, [x2, #0]; STRB w11, [x2, #1];`
`LDR x12, [x2]`
The change to storing a byte is not, by itself, a problem. A single store byte runs at the same half a cycle per pair, for the same reason.

We now have two stores in our triplet, so the maximum speed at which the triplet could execute, with everything independent (independent registers, independent addresses), is now one triplet per cycle (ie two stores per cycle). That's what we see.
But what if we now have one of the bytes overlaps with the load, as in
`STRB w10, [x2, #0]; STRB w11, [x2, #100];`
`LDR x12, [x2]`
Now we take ~1.33 cycles per triplet, likewise if both byte stores match the LDR address.
Obviously there is extra work that's involved in this second case, in that the load has to acquire values from two store queue entries and the cache, then stitch them all together. It looks like this (specifically the stitching together) takes an extra cycle. There's no reason, in principle, why this extra cycle should be visible to us; it's not part of any dependency chain. My guess is that the speculative scheduling is

done on the assumption that each load will take four cycles, and when they take five it's not a catastrophe but, one cycle in three, two loads, or a load and a store collide in some stage and one of them has to be delayed by a cycle. In principle, with more resources in the LSU, this doesn't have to happen.

If we convert the stores to store the lower and upper halves of a word, the cycles per triplet remains unchanged, which suggests the stitching together does take an additional cycle (since this case requires no cache access, only the two store queue accesses).

There's a whole lot more that could be investigated here to investigate precisely all these various unusual cases, but the broad contours are clear enough that I think it's time to move on, past Replay and the LSDP.

conclusion

So I think we can conclude that this validates
- there is an LSDP
- it can hold about 74 pairs of (storePC, loadPC)
- it prevents both Flushes (catastrophic) and Replays (not catastrophic, but a few wasted cycles)
- there is also late checking of whether loads might have collided with stores (catching cases where the load is two, three, even four cycles early; and converting them into Replays), and this is probably extremely helpful in real code, converting Flushes to Replays even on first time code that isn’t part of loops or otherwise is not present in the LSDP.

We have seen the evolution of the LSDP from preventing just Flushes to trying to prevent both Flushes and Replays, with a few compromises made to the Flush part of the design to better accommodate the Replay part.
I wonder if, following the standard Apple pattern, it’s time to disaggregate the LSDP into two predictors, one handling the Flush case only, optimized for accuracy but not needing to be very large; and a second predictor handling the Replay case which does not have to be as accurate but should be much larger (eg based on a direct-mapped or two way set associative design rather than a CAM-based design). Replay is not that expensive, so it’s not a catastrophe if a few Replaying Load/Store pairs are not captured because of collisions in the hash of the PC to an index, as long as most are; and these two issues can be balanced with a more cache-like design.

Continuing the theme of disaggregation, there are suggestions for how to split the Store Queue by function (in the same way we saw the Load Queue split by function). Probably the best match of these to Apple is (2006) https://zilles.cs.illinois.edu/papers/baugh_lsq.pac2.pdf *Decomposing the Load-Store Queue by Function for Power Reduction and Scalability* which suggests a small high speed structure, tightly linked to the LSDP, which handles forwarding stores to loads, and a second large, banked and slow structure that handles testing that loads and stores do not conflict. (Obviously we hope the LSDP will catch most such cases, and any that are left are not performance critical given that we will be Flushing anyway.)

Of course this philosophy (use as minimal a store forwarding structure as possible) is in direct opposition to a different paper we have already seen, *The Untapped Potential of the Store Queue*! Is the energy win from using lots of store queue forwarding (and so avoiding L1D lookups) more than the energy win from minimal store queue forwarding and a lower energy structure to hold most stores? Well, that's what makes microarchitecture interesting.

future improvements?

One, possibly fairly easy, way to improve load/store performance is to copy/modify an idea in Lunar Lake.
Suppose that we provide more (say another one to three) "store execution units". Superficially this sounds like a dumb idea – increasing the less important unit, and just assuming away that more store units is easy. But these are not store execution units, they are "store execution units"...
Amongst the various parts of the load/store experience, as much as possible we'd like to know store addresses ASAP, so that, as much as possible, we can convert speculation about load/store collisions into actual knowledge about such collisions (or not). So we'd like to accelerate just the address generation part of a store, we can delay everything else from knowing the store data value to cache and TLB access.
This suggests providing for more "store address units" which have basically two jobs (perform the additions that result in an address, and then store the address in the LSQ) while retaining the same number of "store data units" aka "real store units". If we can zip through the address calculation of large numbers of issue queue stores via cheap hardware, it's less important that the store data part of the stores accumulates and is funnelled through a narrower set of execution units...

A variant of this idea might be (with ARM help) to define quad load and store instructions. Like load/store pair, these give us more functionality (ie data register movement) while not increasing the expensive part of the operation (checking addresses against the LSQ) since there's still only one address to be checked.
We've already seen quad-execution instructions added to SME, so the precedent exists.

At some point Apple will probably add the ARMv8.8 memory movement (memcopy, memfill) instructions. The obvious way to implement these is to implement them as microcode, emitting load/store instructions; but that's horribly sub-optimal. Far superior is to execute them as a special-type of instruction down as far as LSQ, so that, once again, we can do one *single* round of address comparisons. With that out the way, *then* we can start a data movement loop that operates directly on the L1 cache...

Load Accelerators

Because loads are common, and high latency, we do whatever we can to make them faster.

Obviously one track, with which you are familiar, is multi-level caches.

A second track, one we have discussed in immense detail, is ensuring that loads are not slowed down by store any more than is absolutely essential (the LSDP) and are not slowed down much by slight glitches in timing at any stage of load execution (Replay).

But a third track is to make loads provide a result faster than it takes to access the L1D.
To understand how this could be done, we need to think abstractly.
Consider Loads to be an instance of “matching” the load with a “source” via some “matching mechanism”.

Traditional loads find their source data in the L1D, and perform the matching via the physical address.

Acceleration via the Store Queue

Any modern machine (store forwarding) will have loads that match recent stores find the data in the Store Queue, and the match is via the physical address. One could imagine an alternative to this.
Suppose the match were by virtual address. This means that the comparison of the load address with the Store Queue entries could happen right after address generation, in parallel with TLB lookup.
This could provide the data one, maybe even two cycles earlier, for a reasonable fraction of loads. (The *Filter Caching for Free* paper we have already mentioned suggests about 18% of loads hit in the Store Queue for a Skylake-sized Store Queue of 56 entries, similar to M1's 60 entries).
The big problem with this idea is that it hurts Speculative Scheduling:
- do you schedule dependent instructions assuming the load will provide data in two cycles, or in four cycles?
Presumably one can (as always) build a predictor, and then it's a question of the speedup vs the energy cost of the predictor.

But we can also look at this from a different direction.
Consider the LSDP. This mostly succeeds in tying a particular store (by PC) to a particular load (by PC). In other words, we are matching source and load by the PC of each.
What if we generalized this? Imagine a table much like the LSDP but the way it works is:
- every entry in the Store Queue records the store's PC
- if a load hits in the Store Queue, that load's PC and the store's PC are recorded in the table
- in future, when a load matches a load PC in the table (comparison performed eg at Mapping time) then instructions dependent on that load can be speculatively scheduled assuming a load latency of 2 rather than 4 cycles. Failure will result in a Replay.
- of course we can add the usual bells and whistles as necessary, eg confidence tracking, aging out entries, arming when we see a matching store, ...

This might allow us to capture not just the lower energy savings possible by reading the Store Queue

rather than the cache, but also the latency savings.

Acceleration via Registers

2012 ZCL patent (register renaming based on common base register)

A second source for load data is values in registers. Every value that is stored to memory by the CPU was stored there as a register store. That value may still persist in a physical register. This is again a matching problem – how can we match a load (that will ultimately use a particular address) with the physical register that was earlier stored to that address?

A different way to think about this is consider the sequence
define x0
`STR x0, [x2]`
do some stuff
`LD x1, [x2]`
use x1
The most naive version of handling this would require the data to be pushed through the store all the way to the cache, then loaded.
More sophisticated would be to pick up the load data from the store queue.
Best of all would be, effectively, to remap the store data register, x0, to the load value register, x1, bypassing/augmenting the actual store to DRAM.
For this reason the idea is called Memory Renaming (by analogy with Register Renaming), 1997, https://web.eecs.umich.edu/~taustin/papers/MICRO30-mren.pdf, *Improving the Accuracy and Performance of Memory Communication through Renaming*.
However Memory Renaming is a somewhat vague term, and many companies seem to use it to refer to loads that hit data in the Store Queue.
The state of the art right now (as I understand it) is that high-end x86 systems can perform the above (store to load) in about 5.5 to 7 cycles; and that this is handled by the Store Queue. Apple is comparable, when going through the Store Queue, at ~6 cycles.
But what Apple can do in addition (under the right conditions) is service the store from the register file rather than the Store Queue, so that the (store to load) takes 5 cycles (in the most difficult case) and 3 cycles (in the easier case). I am unaware of any other vendor that does this sort of register-based Memory Renaming.

Let's start by considering the simplest possible version of a solution, then improving it. So we start with simple stores (a single register) using simple address modes (a single base pointer, no index or immediate offset).
Suppose we record in a table that base pointer x2 stored physical register p100. There will be multiple such entries, in principle one for every possible xn base pointer.
Now suppose I execute `LDR x10, [x2]` (which is known at Decode time, unlike the load address). I look up my table, see that the appropriate value is p100, and so I rename x10 to p100, and mark x10 as

already valid, so it can be used by any other instructions.
In other words, just like Zero Cycle Move, the work is done at Rename, and this acts as a Zero Cycle Load.
(Note that this implementation stores the value to be forwarded from the store to the load via the register file, as opposed to the initial paper which suggested storing it in separate storage).
This design is covered in 2012 https://patents.google.com/patent/US9996348B2 *Zero cycle load*, where they call the idea the RF-LSDP (Register File Load Store Dependency Predictor).

So what can go wrong?
Suppose I change the value of x2. That's easy to catch, and I simply mark the entry in my table invalid.
I need to track if p100 is overwritten, but that's also easy and handled in the same way.
What if the load accesses the effective address of x2 via some different combination of registers, eg [x2]=[x3, #64]. Well, in that case we will not get a match and we will not get a Zero Cycle Load. But nothing will go wrong. Realistically, high level code uses pointers (or the equivalents of pointers, even if they are not language visible) for loads and stores, and the same register will be used to write or read from a common structure under most conditions. So this is not a serious concern.

Another way to phrase all the above is: Suppose you know that
- a recent store wrote register xS AND
- generated an address based on register xA AND
- a load is loading from register xA AND
- xA has not been modified AND
-pS (the physical register corresponding to logical store data xS) is still available
Then you can perform the same sort of zero cycle game, servicing the load by simply Remapping pS into the destination register for the Load.

The real concern is that a different store, using eg address [x3, #64] will overwrite the value stored at [x2]. Or even another CPU might have overwritten it, if the address is somehow being shared by more than one core.
This tells us that, whatever we do to accelerate the load, we will also have to validate it. I have had a lot of difficulty testing that this (and a few other) Zero Cycle Loads exist, and I think at least in part this was because of validation as a bottleneck. These schemes are designed to shave a cycle or two off critical paths in real code, not to accelerate micro-benchmarks. So they may only activate as one or two instantiations per cycle, and attempting to activate them too often may throttle them (likely in the validation stage).

2018 patent (extend to load after load, extend to stack pointer)

So at this point we understand the basic idea of the most basic sort of ZCL load accelerator. How could we improve it?
In the initial patent Apple provided two linked upgrades:
- they allow addresses that are not just a base pointer, but also a base pointer+immediate, ie [x2, #64]

- they also track addition or subtraction of immediates to the base pointer.
So suppose the table records
[x2, #64] → p100; and now we add 10 to x2. Then, in essence, the table entry is updated to
[x2, #54] → p100.

These two changes mean that
- a much wider range of stores will be eligible for being captured in our ZCL table
- and entries can persist after common modifications to base pointers (like incrementing them to walk along an array)
But of course they also make the implementation of the table a little harder! Apple don’t describe how they implement the ZCL lookup table, or how they track which entries to update when a base pointer is added to or subtracted from.

Note that this mechanism as described does not handle load/store pair, and Apple are unclear about whether it handles FP/SIMD registers (ie can the entry for [x2, #64] point to say f100, a floating point physical register rather than only integer physical registers?

Apple made one neat improvement to this mechanism a few years later when someone realized that stores aren't the only way a register is attached to an address; this is also true for loads. If a few cycles ago I loaded p100 from [x2] and then I load from [x2] again, I can use the ZCL mechanism to return p100 as the destination register of the second load. Ideally code is not constantly re-loading from the same address! But realistically this is common in various scenarios, including non-optimized code (eg when debugging) and code generated by JITs.
This idea is covered in 2018 https://patents.google.com/patent/US10838729B1 *System and method for predicting memory dependence when a source register of a push instruction matches the destination register of a pop instruction*.

However, as the name of the above patent suggests, it is primarily concerned with stacks.
The reason this is worth doing is that an extremely common pattern is
- enter a function
- save various non-volatile registers to the stack
- execute (including changing those non-volatile registers)
- restore the non-volatile registers from the stack
- exit the function
There’s a very clear and structured pattern matching each register stack store with a later stack load, and if we can track this pattern we can take advantage of it to convert each of the stack loads to a zero cycle (and lower energy) register rename. So how do we do that?

Now that we have this idea of reading from registers, is there any other way we can usefully associate a register (loaded or stored) with some sort of pattern that is known at Rename time? Apple provide a second variant that behaves something like the stack engine of x86 designs.
The idea is essentially the same as the previous RF-LSDP, but specialized to the case where the base

pointer is the stack pointer. This specialization, called the SP-LSDP (Stack Pointer LSDP) allows the mechanism to capture load/store pairs, and to be an especially good fit to function prolog/epilogs, hence making function calls even lighter weight.

Both the RF- and SP-LSDPs are nice zero cycle accelerators, but they are also both clearly not yet optimal and one can imagine a variety of ways to improve them. For example if you read the patents you will see that entries in both predictors only last until the “register-producing” instruction, a load or a store, is retired.
It’s obvious why this is (after retirement the register is freed, so it’s value could then change at any point) but this is clearly not optimal.

One could imagine a restructured set of predictors taking the best of zero cycle moves, zero cycle immediates, and zero cycle loads (SP and RF), perhaps even some other cases (value prediction?), and based on a mechanism whereby predictor entries remained valid until at least a physical register was actually overwritten, rather than being cancelled when the physical register is freed.
Likewise one would want to tweak all the predictors (not just SP-LSDP) to behave appropriately with load/store pair and fp/SIMD registers. (Generally fp/SIMD registers are not as latency sensitive as integer registers, so the zero cycle aspect of ZCLs is less compelling. But if you can access a register via a rename rather than a cache access, that’s an energy savings, and maybe the energy savings is higher than the energy cost of the relevant tables and tracking?)

BTW the patent comes with another nice pipeline diagram showing how some of this fits together. Note the RF-LSDP kicks in at Decode, the SP-LSDP one cycle later at mapping, and the traditional LSDP at Dispatch.
(Note that this is the 2nd gen A7-class pipeline; it's clearly not the M1 pipeline because it still references the RDA [Register Duplicate Array] as the mechanism for handling multiple references to a register, not the current 2019 mechanism I described earlier.)

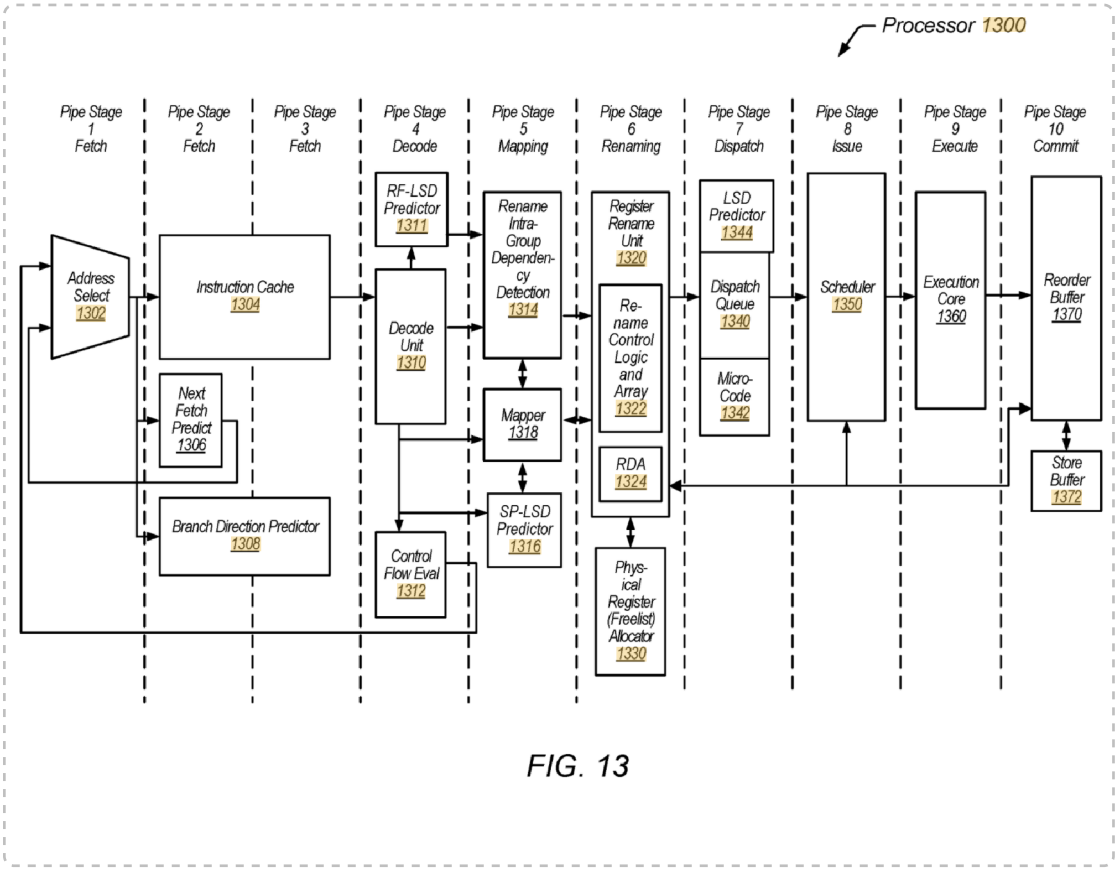


FIG. 13

2019 patent (store and dependent load in same decode group)

Do the RF and SP-LSDPs cover everything? Of course not! There is a technical detail in how they are implemented that means that they both can only cover a producer operation decoded in one cycle that feeds a load in a subsequent cycle. But what if you have a store followed immediately, or just one or two instructions later, by a load, so the store and load are decoded in the same cycle?

Who would write such dumb code, storing a value then immediately reloading it? Well, the example Apple suggest is interpreters in general, and JS in particular, especially when running for small code sections that haven't yet been aggressively JIT'd.

For these cases Apple has (2019) https://patents.google.com/patent/US20210173654A1 *Zero cycle load bypass*.

This adds that as part of Decode, every appropriate possible pair of store/loads in a decode group (ie all the instructions decoded in a single cycle) is tested for matching address patterns, and if so we use the usual Rename trick.

As I said, this is a remarkable set of patents that all seem to build upon each other, and that include what at least look like careful thought as to how to fit the ideas into an existing design.

2022 patent (longer-lived stack tracking)

I mentioned above that the 2019 stack tracking patent was sub-optimal. This is fixed to some extent with (2022) https://patents.google.com/patent/US11900118B1 *Stack pointer instruction buffer for zero-*

cycle loads. The quick summary is that the previous stack pointer based ZCL scheme essentially tracked some elements of the load/store pairing only as long as the store remained in the store queue, and so any possible pairing disappeared when the store was executed to cache and removed from the LSQ. This meant that the pairing and thus the zero cycle load (by executing the load from stack as a rename) was only possible for short functions, where the function epilog runs not very long after the function prolog.
The patent substantially extends how long a function can be and still make use of this particular ZCL. It does this by, when the stack store is executed, moving the relevant data (stack offset, physical register number corresponding to the register to be stored on stack, etc) to a "rescue buffer" which holds onto this data for as long as it is relevant, and hopefully it's still relevant by the time we exit the function.

The patent also suggests that this "rescue circuit" scheme can be extended to handle other (non-stack) load-store pairs that can be treated as ZCLs, though I'm not sure why the cases they describe aren't well-handled by the previous mechanism. It looks like the different ZCL cases have been unified to take a single form using a single pair-dependence-predictor and datapath, as opposed to what we saw above (2019) where they use different predictor storage, activated at different stages in the pipeline. So now everybody gets to use the "rescue buffer" if required, even though we expect the stack case to be the primary case where the load is sufficiently separated from the store that rescue is needed?

In *principle* there's nothing in this scheme, beyond the lifetime of a physical register, ie how long until it is reused, that prevents this scheme nesting, so that two or three nested functions could all successively execute their epilogues and return, and the set of two or three prologues could all collapse to zero load activity, just some register renaming. It's unclear how aggressive Apple will be about this (eg how many tracking slots they will provide for load/store pairs). However there is still the constraint that these sorts of pairings can't persist beyond the point where the store is retired (and so the physical register is free to be overwritten) unless there's some sort of additional "reservation" added to the physical register allotment scheme.
Certainly in principle
- you could treat the pool of physical registers like a cache and add some status bits tagging certain physical registers as "possible sources for ZCL" so that those registers delay being reallocated until the last possible moment
- you could do even better if you use virtual registers, to delay invalidation (and thus persistence of the original data for the ZCL) not just to the point where the register is reallocated, but to the point where the overwriting instruction is issued.

In *practice*, however, right now it seems like we are limited to handling just leaf functions. The implementation of the ZCL store-load predictor table is indexed by the logical register number, and so no sort of recursion is possible (ie two entries, one storing the load/store pair for pushing/popping registers x29,x30 inside a leaf function A and one storing the load/store pair for pushing/popping registers x29,x30 in a calling function B).
Conceivably the next evolution might be indexing this table differently (perhaps using a depth count of

the return address stack, as an additional indexing element? [you need something available at the point of instruction decode...]) to allow for more entries?
Fig 5 of the patent gives some details of the current, logical register-based indexing.

Don't get confused about what's happening here! Consider the general situation that
- I execute a store, which gets stored in the load-store queue as pair (address, data)
- later I execute a load from that same address
I need machinery (given names like load-store alias detection) to detect that the data for my load is present in the load-store queue, and respond appropriately. The simplest response is
a. wait until the store has pushed the data out to the cache before executing the load
More sophisticated is
b. read the load value from the load-store queue and use it right away. This is faster, but means you have to deal with things that can go wrong (maybe another CPU changes the value in the cache between the store "should have" executed and when the load "should have" executed?)
What we are describing is an even more sophisticated option
c. don't read the load value from the load-store queue, read it from the register file that wrote it to the queue! This is both lower power and lower latency, but requires even more tracking and handling of all the possible things that could go wrong.
In volume 2 we will discuss this issue in much more detail, including the cases where either option b or option a have to be used for one reason or another. (Option c can only be used under specific circumstances where it is "easy" to detect a likely load-store dependence AND that load-store dependence is described by register IDs [and so can be detected at decode time] not by address value [which can only be detected at execute time].)

One reason this is interesting is that you sometimes hear that the M1 has poor (not terrible, but poor) load-store forwarding, which refers to the situation b described above, where a subsequent load needs to read data that is already present in the load-store queue. Some x86 designs can handle this in one cycle, whereas M1 seems to take 4 to 5 cycles. For example: https://twitter.com/lamchester/status/1530297321333739521
This seems bad BUT it's misleading.
Apple has optimized not for case b (where load and store *addresses* match) but for case c (where load and store *addressing* matches). When the addressing matches (same stack reference, or same base register) then we can use ZCL for truly optimal performance. I'm guessing that the Apple scheme catches almost all real world cases where this matters, so there was little incentive to bother optimizing the remaining cases beyond ensuring they aren't terrible. And even in case b, it seems that this has been somewhat improved for M2.

One final point is that it seems that the ZCLs are not used for NEON registers. I don't quite understand this. In particular, consider the case of function prolog/epilogs. These occur in the same way, with the same patterns, for NEON as for integer registers. And while the latency boost from a ZCL may not matter much for NEON code, you probably will also save a little energy by executing the load as a rename rather than a data movement. So maybe one day?

Extension of this idea to other loads and stores

We saw that zero cycle loads have been available for a long time based on pattern matching the address generation of the store against the address generation of the load, so that a pair that matched (same base register and same offset index) could execute a ZCL.
That's all great, but it requires the load and store to be compiled using the same address generation pattern, which is feasible for the compiler within a single function, but becomes less likely in other conditions (maybe frequent calls to a library? maybe a JIT?). Another problematic case is when complex code occurs between the store and the load so that all registers are used so that the base logical register is dirty by the time of the load, compared to when it was used by the store.
The newest scheme, described in (2022, but only published in mid-2025) https://patents.google.com/patent/US12288070B1 *Program counter zero-cycle loads* still uses the same mechanism as all the ZCLs – we retain earlier stored data in a physical register as long as possible, and hope that a subsequent load can be serviced by renaming that physical register. But how do we link all these together? The linkage is by PC, so trying to detect that PC_S repeatedly stores a value at some address, followed by PC_L which loads a value from that same address.
This is based on two tables. One is essentially used for training, to try to detect, over time, when robust versions of this linkage occur. Once a linkage is detected, it's moved to the second table which is used during each subsequent execution, to store theID of the physical register (at store time) and then to service the load (at load time).
There are a few minor twiddles in all this to make it more efficient.
- There are some tests to ensure that the load/stores will not be captured by one of the other (simpler) ZCLs like the address-generation-pattern ZCL or the stack ZCL.
- There some tracking of branch history so that the load can be made conditional on a particular branch history (the linkage exists when this particular pattern of *n* taken/not-taken branches was present at the time of the load, and not otherwise)

The idea is simple, but it seems to me astonishing that this can possibly work! There are so many loads and so many stores, and a quadratic number of possible dependencies between them! There's something much deeper going on that makes this feasible, so you aren't continually retesting the same possible sets of loads vs stores over and over; but unfortunately the patent keeps that part of the implementation as a trade secret.

Summary of these accelerators

We've covered a lot of material under the guise of "load accelerators".
Another way to classify these ideas is by asking just how much work you can avoid by clever register renaming. The paper (2005) https://pages.cs.wisc.edu/~isca2005/papers/02B-03.PDF *RENO: A Rename-Based Instruction Optimizer* gives something of an answer to this question.
This paper suggests, essentially, three ways to leverage renaming
- zero-cycle moves

- zero-cycle loads (generally, but exclusively, off the stack)
- zero-cycle add-immediate
and suggests that these three together correspond to ~20% of instructions.
Obviously Apple (and everyone else) perform zero-cycle moves, and a version of many zero-cycle loads.
What about the add-immediates?
Apple handle zero-cycle load-immediates (a case not considered by the paper), but do not appear to handle add-immediate as a special case.
The most recent Intel processors do handle this case (perhaps via the mechanism suggested in the paper?) and perhaps we will see Apple also doing this in a future design.

experimental tests

I keep flip flopping in my head as to whether all the ZCL tests actually show what I hope they show. A ZCL moves work around, it doesn't eliminate it (because the load usually has to be validated), and so in a sense it is equivalent to just having very deep OoO queues and buffers. I haven't yet come up with a line of reasoning I'm happy with that absolutely proves what I am seeing is in fact ZCLs, as opposed to just rapidly filling up a deep queue...

Here's the problem. Consider what a ZCL is supposed to do.
We have a set of instructions with a particular critical path.
Ideally
- We start the clock
- the set of instructions gives us a result (with some cycles reduced because of ZCL)
- we stop the clock
- after the clock has stopped, whatever validation is required for the ZCL continues, behind the scenes

But this only works if we can cleanly stop the clock after enough work that the reduced ZCL has made an impact, but not so much work that we have exceeded the capacity of the validation buffers. If we can't do that we very soon run at the speed of validation.

We can't actually stop the clock after just 100 instructions or so of work! We have to have some sort of outer loop that repeats the job a few thousand times between reading the cycle counters. But then it's very hard to tell the difference between the above and
- We start the clock
- the set of instructions accumulates in various buffers
- we start the next loop
- we now have two (or three) independent chains of instructions running
- we observe a reduced time per iteration
- does this mean an individual time was reduced (reduced latency)? Or that the loops were successfully run in parallel?
In both cases, we accumulate a whole lot of (the same) work in (mostly the same) buffers. The only

difference is that
- in the first case the result (after ONE iteration) is available early, and the work that's queued up is called validation
- in the second case the result (after one iteration) is not yet available until the queued up work is executed.
But if you're repeating the loop a thousand times, either you get
- fast result (after a thousand times, because of low latency) and validation "in the background" OR
- fast result (after a thousand times, because of running in parallel) and work being done in the foreground.

In a sense *everything* hinges on the result of the first iteration being early, but it's impossible to get at that with the timing tools available!
You're trying to establish (at 3GHz) whether a few items of work were done before or after one particular instruction (the "final result of the dependency chain" instruction) on a machine that is designed to run everything out of order...

So here's my attempt to work around this.
Why do we want a ZCL, what's the ultimate value? The hope is that by moving the "result" earlier and doing the validation later, the validation will happen at the same time as *something else is happening*. If we can create this situation then we have moved from
load | use load | do something else
to
ZCL| use load |do something else
validate load
As long as load validation can happen in parallel with some sort of "something else" then it will not form a bottleneck that limits our benchmarking.
This means we have to
(a) construct our tests so that we always have some extra "do something else" (I try to use DIV's or MUL's) to take up about the same amount of time as the validation load takes up and
(b) the signal we are looking for is not behavior at just a few iterations, it is long term behavior. Behavior at the beginning of the graph just tells us that the OoO queues are working and that's not what we are after.

If you have any ideas/references, please let me know.
Below are the tests (and their interpretations) I could think of, but of all this work, they're where I remain least confident that they actually measure what I want them to.

loading a previous store (2012 patent)

Just to situate us (this is not what we are testing), consider a set of different types of loads and stores. First store two two different address, load from two other different addresses (so four addresses in

total).
`(STR x0, [x1]; STR x0, [x5]; LDR x16, [x6]; LDR x17, [x7])`
This runs, no surprise, at one cycle for four load/stores.

Now suppose we change this to store at the same address (but that sameness is not obvious), so
`(STR x0, [x1]; STR x0, [x2]; LDR x16, [x6]; LDR x17, [x7])`
x1 and x2 are equal, but this is only known by the time we hit the LSU.
Presumably the LSU somehow squelches the first store so that only the second one goes through, but we get no performance boost, this still runs at one cycle for the probe. (Unsurprising, we still have to submit two loads and two stores to the LSU.)

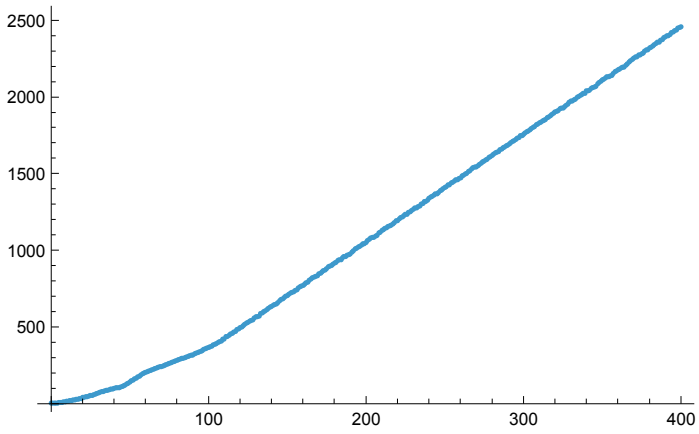
Now we make it clear that the stores are to the same address
`(STR x0, [x1]; STR x0, [x1]; LDR x16, [x6]; LDR x17, [x7])`
In theory, now, the *front-end* could see that one of the stores is redundant, remove it, and send only three operations into the Scheduling Queue (which could then run four "real" load/stores per cycle and we go 4/3 faster, taking 3/4 cycle per probe).
But (unsurprisingly) this does not happen. It would be very dumb code to do this, back-to-back identical stores, so why test for it?

We can try similar things with loads from the same register, eg
`(STR x0, [x1]; STR x0, [x1]; LDR x16, [x6]; LDR x16, [x6])`
and again we get no boost, though again in theory the front-end could suppress one of the loads.

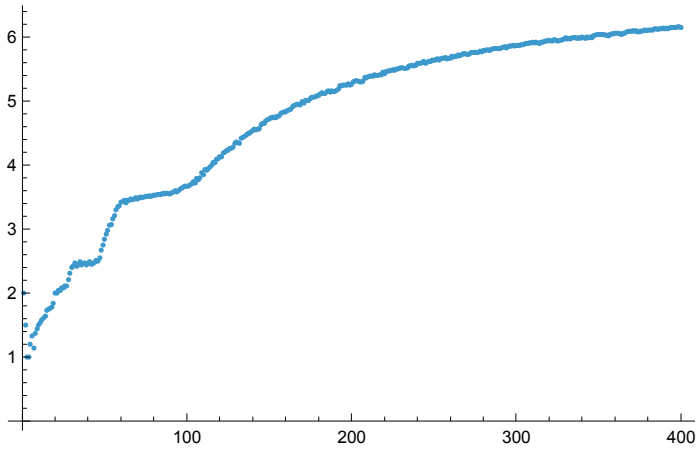
The case
`(STR x0, [x1]; STR x0, [x1]; LDR x16, [x6]; LDR x17, [x6])`
is slightly more justifiable in that you can argue code might want to load the same value into both x16 and x17 (ie into two different variables, before each variable is manipulated in a different way). But even that code is more sensibly handled by performing the load followed by a MOV, so it makes sense that Apple doesn't attempt to handle it specially.

But now let's go in a different direction, and probe using `(STR x1, [x1]; LDR x2, [x2])` where we initialize x1 to point to some buffer, and x2 equals x1. So we have a single address (x1=x2) and we keep loading and storing to that same address.
The obvious way to execute this, for correctness, is the machine (using LSDP) notices that the load writes to the address of a previous store, so the load is delayed until the store completes. Likewise the store overwrites the previous load so it has to wait until that load completes.
Now, for just a few sequential rounds of this we do not really have to slow down because the loads and stores can be piled up in the Scheduling Queues, the stores can execute to the Store Queue, and so on. But you can only play that game until those buffers are full, at which point the next load or store can only happen once the previous one has happened.

Out[651]=



Out[652]=



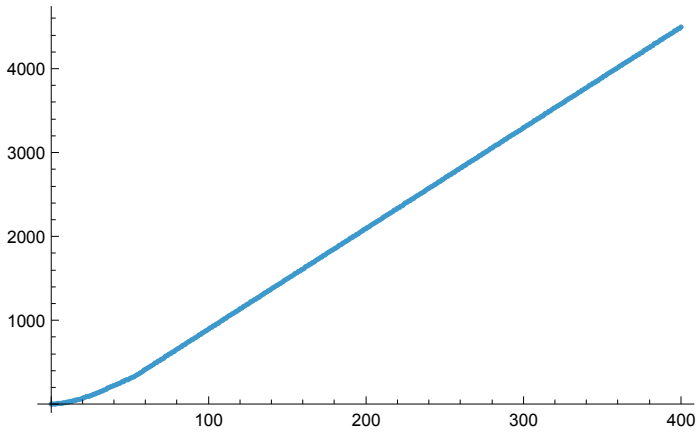
We can see this phenomenon here. The first curve shows accumulated time in blue, the second (noisier) curve shows the average time per load/store over longer and longer runs.
When we don't do many load/stores (at the left end of the curve) we get higher performance (lower cycle count for a single pair of load/stores). This is just because the machine is rapidly dispatching the instructions into various queues, and is then able to execute independent groups of these instructions in parallel; it's not "really" executing any particular sequential set of load/stores faster.
Long term (past about 80 or so) once all the buffers have filled up, the time to execute a store/load pair of this form seems to takes ~6 cycles (three for the store, then three, maybe four for the load).

OK, that's familiar and as expected. A forced sequential set of load/stores indeed operates sequentially.

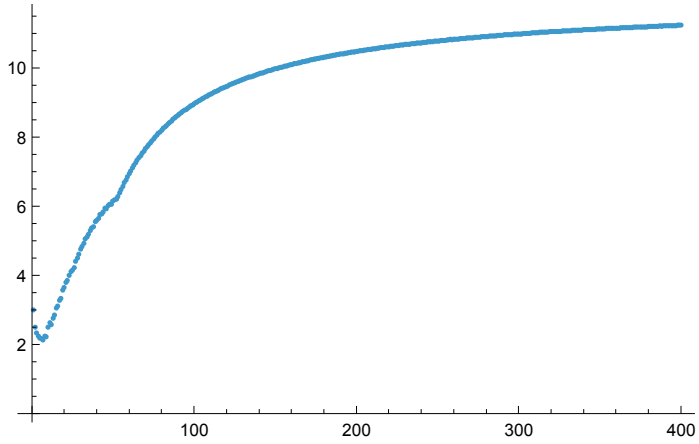
Now lets make a slight change to the probe by adding a DIV
`(STR x1, [x1]; LDR x2, [x2]; DIV x2, x2, x10)` where x10=1.
Naively we would expect this to take 3+3+8=14 cycles, but that's not right.
The critical path is the load to DIV path, 3+8 cycles. While that is happening, the next store can happen in parallel with the DIV, but the next load cannot happen until after the DIV completes.

And this is what we see

Out[654]=



Out[655]=

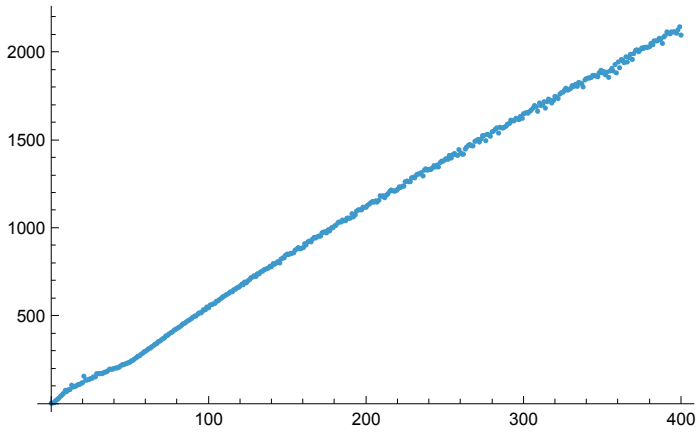


Long term, the cost for the probe is 11 cycles.

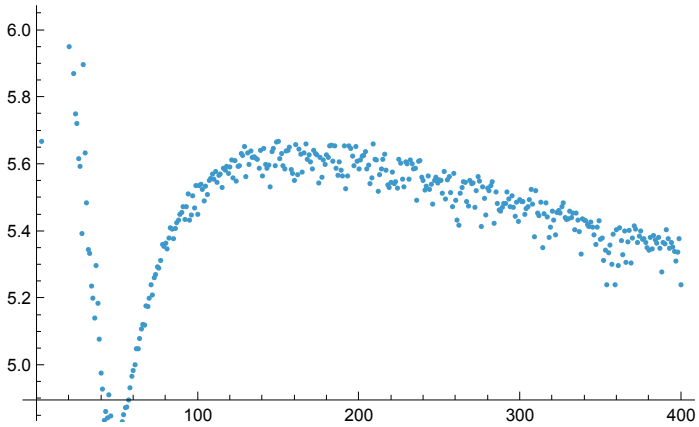
Now the fun stuff!
Make one small modification of the above, to (`STR x1, [x2]; LDR x2, [x2]; DIV x2, x2, x10`).
The difference is that it's now obvious, *at decode time*, that the store and load happen to same address (because both use [x2]; in the earlier case the addresses were [x1] and [x2], and the *front end* could not know that x1==x2.

This difference means that
- the front-end can speculate (ie ZCL) as to the value that will be loaded into x2 by the LDR, and can immediately feed it to the DIV
- then the validation of this speculation can happen in parallel with the DIV
So we should be able to shave some cycle off the loop time because the critical path
load (3 cycles) ->DIV(8 cycles) has been converted to (DIV (8 cycles); load happening in parallel)

Out[657]=



Out[658]=



And that's what we see! In fact it's even better than expected. We expected the probe cycles to drop to 8 (cost of a DIV) but in fact the machinery that's injecting the ZCLs (the speculated value for the load) is able to break the dependency chain so that successive DIVs can execute independently, not sequentially!

The point that matters is that we see (once the conditions for the 2012 patent are met) a substantial speedup. The conditions for that patent are that the store and load address have to match (as registers) so that we can match them in the front end.

How can we be sure the 2012 patent is the mechanism, not something else?

Change the probe to

```
(STR x1, [x2]; LDR x2, [x2]; DIV x2, x2, x10;
ADD x2, x2, #8; BIC x2, x2, #16)
```

Now we are doing two things to the address

- we increment it by eight each cycle but we also
- wrap it around when it exceeds 16. So we alternate x2, x2+8, x2, x2+8, ...

Any sort of simple address tracker (constant address or constant stride) will be fooled by this, but the 2012 ZCL is not; it does not care about how x2 changes, only that the load and store reference the same [x2].

The flow is confusing! So let's make a slight change that clarifies things a little. This change still runs fast, but the registers are now a little clearer.
`( STR x1, [x2]; LDR x9, [x2]; DIV x2, x9, x10 )`
What matters is that
- the store and load share a common address [x2]
- because of that common address register, the ZCL can rename x9 to the physical register associated with x1

- the register name x9 is irrelevant to anything
- we need to add a delay (some work) to show that x9 is being generated faster than expect, that's the DIV
- the DIV has to generate a value that's used by the earlier instructions other each block can simply run in parallel, there is no forced serialization

(As a technicality, I've described the boost we see in terms of the 2012 patent, but for technical reasons I think what's actually relevant is the 2019 patent.
The 2019 patent handles load and store referencing the same register in the same cycle (ie same decode block).
We can force the 2012 patent by inserting 8 NOPs between the store and the load, so they decode in subsequent cycles. In that case we are using the classic 2012 patent, and things get even better; now we run at 3 cycles per probe rather than about 5.5!

Slightly fancy addresses like [x2, #8] are handled appropriately.
But not addresses that depend on two registers, like [x2, x10].
Likewise not store pair+load pair.

Maybe I misunderstood the 2018 updated patent; I thought that meant minor changes (add, sub immediate) of the base register would be tracked and handled appropriately, but I could not see that in action.

What if we set the DIV result to x1 rather than x2, so that the store value, not the load/store address, is blocked by the DIV?
I think in principle this could work, though it would run at 8 cycles (DIV speed) without being able to run subsequent DIVs independently. But in practice the implementation does not seem willing to catch this case, and it runs at 12 cycles.

The DIV case is very nice because it's incontrovertible proof (IMHO) of a ZCL kicking in.
Now that we know that that particular ZCL works, we can use a similar pattern to try to test other ZCLs (which are less convincing as ZCLs per se, except that they do or don't show faster behavior than expected when matching a particular type of pattern.

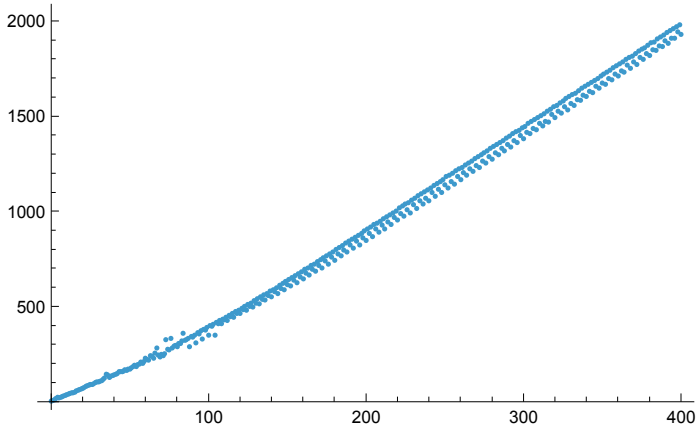
So, suppose we remove the DIV, giving us the probe (`STR x1, [x2]; LDR x1, [x2]`).

In particular compare two probes, (`STR x1, [x2]; LDR x1, [x2]`) and (`STR x1, [x2]; LDR x1, [x1]`).
I know your eyes start to glaze over! But think about it. The value that is loaded each time is then stored (load data register is store data register). So every store has to happen after every load. Likewise every store has to happen after every load (same address value, in both cases, because x2=x1).
But the first case, with the load and store registers based off the same address register (ie address= [x2]) makes the equality visible to the front end, so ZCL can kick in. The second case is identical as far as the LSU and LSDP are concerned (the two addresses of interest, [x2] and [x1] are identical) but it's different as far as the front-end is concerned.
And indeed the first case uses ZCL and runs at ~5 cycles/load/store; the second case doesn't use ZCL and runs at ~7cycles per load/store.

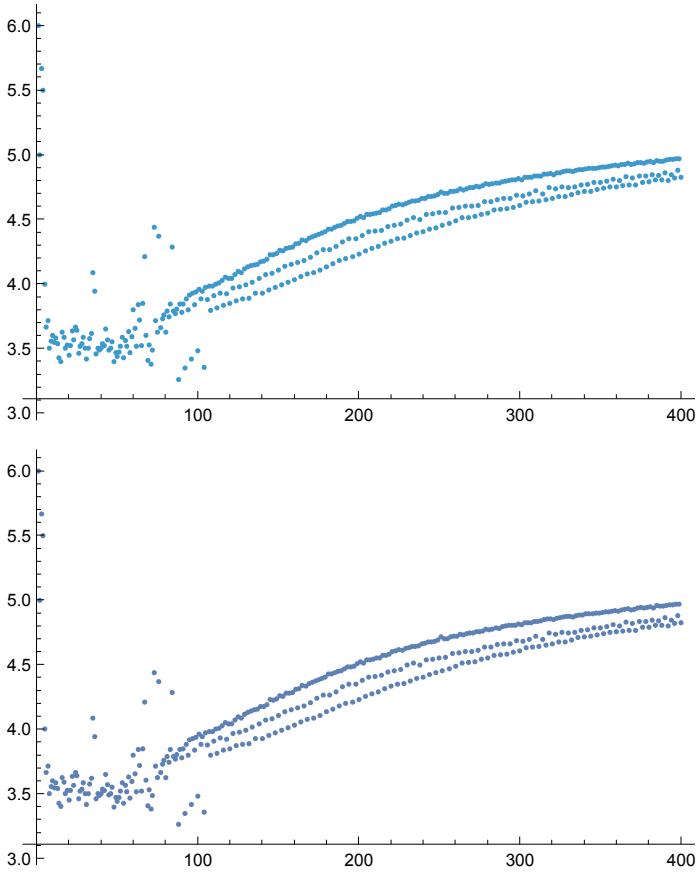
The value of this pattern is now we can ask what happens if we try similar patterns.
Compare now (`STR q0, [x2]; LDR q0, [x1]`)
this has the same structure of load has to follow store (because of the same q0 register) and store has to follow load (because of the same target address); and so it's no surprise that it runs at ~7 cycles per load/store.
If we change this to (`STR q0, [x2]; LDR q0, [x2]`)
then, in principle, the front-end could notice that the address registers are the same and ZCL the q0 value given to the load. But that does not happen, we get the same ~7 cycles per load/store.

The graphs below show the ZCL integer case, and how the long term performance (with some weird noise!) seems to tend to about 5 cycles per load/store.

Out[660]=



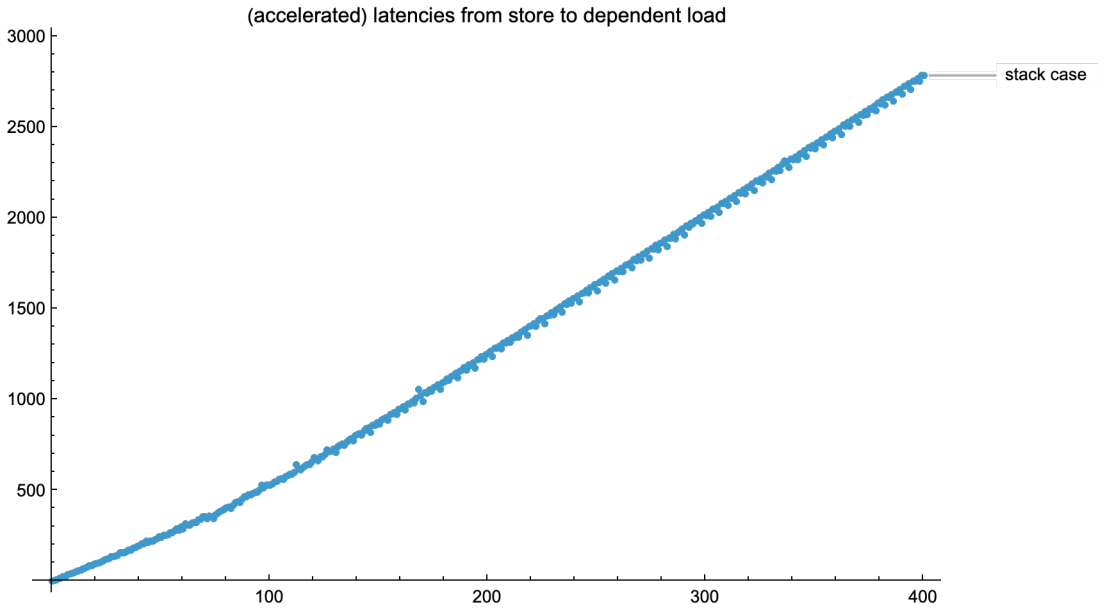
Out[661]=



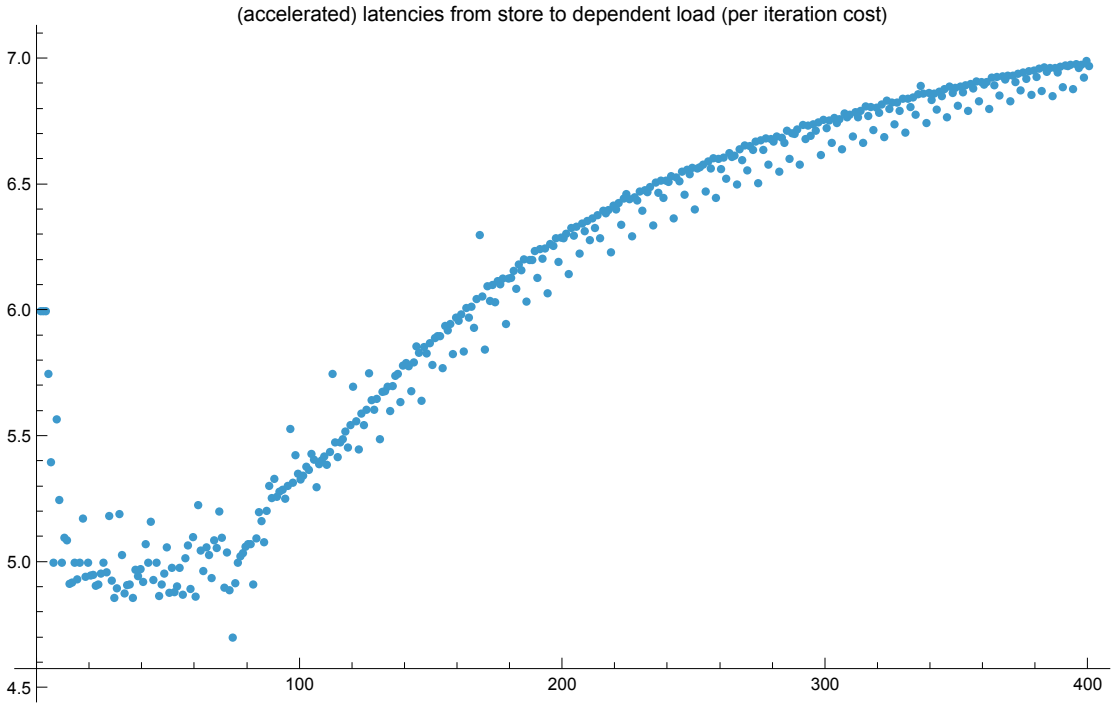
stack loads

Now that we know what to expect from the previous example, we can do the same sort of thing with stack loads. Consider
(`STP x28, x27, [SP, #-16]; LDP x28, x27, [SP, #-16]`) which is more or less what we see at function prologues and epilogues.

Out[663]=



Out[664]=



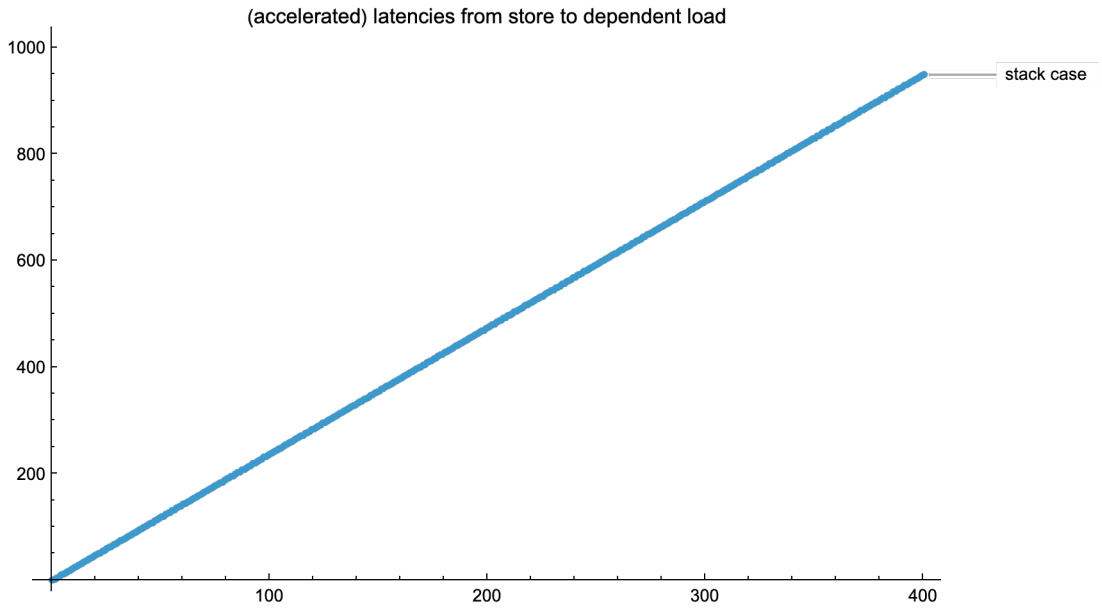
Well that's disappointing! We seem to take 7 cycles to perform the load/store, just as before, so no acceleration.

Well...

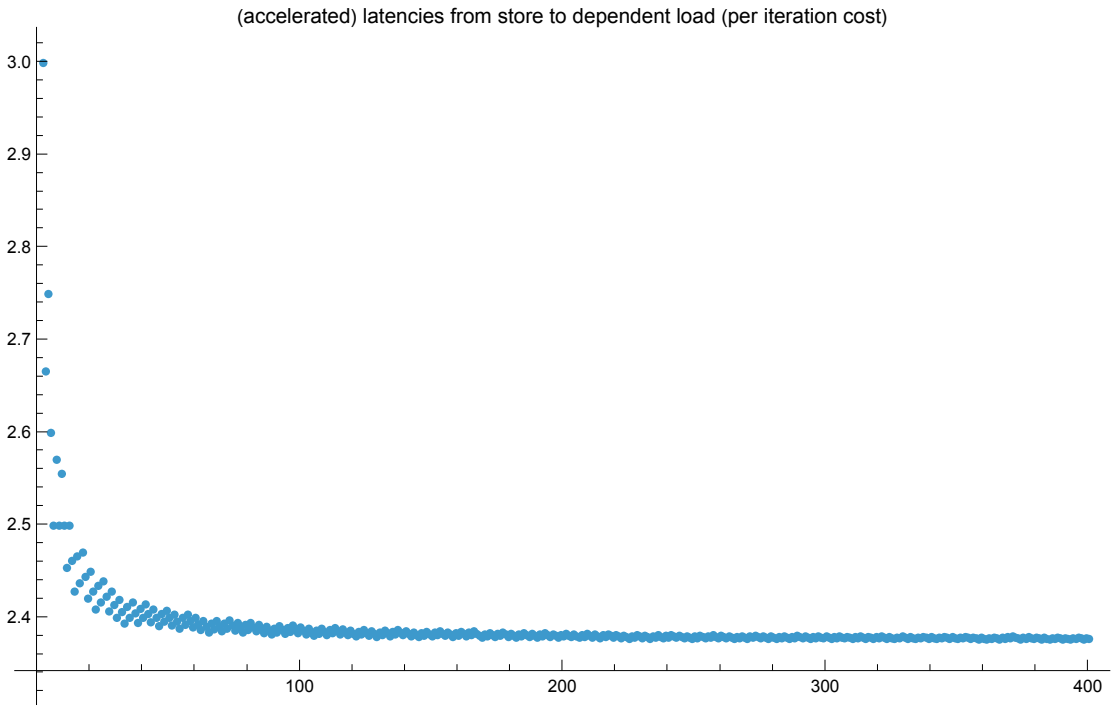
Remember the fine print! It's only the magic of the 2019 patent that allows ZCLs to kick in for load/store that happen in the same cycle. And there's no real reason to work this hard on the stack engine, because the load / store pairs that we want to accelerate will live in different function prologue/epilogues, and will likely be separated by at least one cycle of intervening code (unless the code is very

strange). So let's run the same probe, but with 8 NOPs after the store, and another 8 after the load.

Out[666]=



Out[667]=



That's more like it!!! Nice acceleration there!

What if we add in storing then loading a pair of SIMD registers?

Once again that case doesn't seem to be accelerated. (I test FP/SIMD in these cases out of interest, but it's unsurprising that Apple doesn't accelerate them. Most FP/SIMD work is about throughput, not about latency, so spending area and power on functionality that reduces the latency of FP/SIMD loads

is probably not a wise investment; better to spend on improving SIMD throughput.)

A different discussion of this pattern of ZCL (stack-like, even if not on the stack) is given in https://zhuanlan.zhihu.com/p/595582920
Unfortunately this site requires registration (easiest done if you have a WeChat account). If you can register, then machine-translation does an adequate job and you can see that his, somewhat different test code leads to the same sort of conclusion.

Acceleration via Faster Address Calculation

The mechanisms discussed so far make loads faster by retrieving data from unexpected places (the Store Queue or the Register File).
A second way we might make loads faster is if we can more rapidly construct the address used by the load.

2013 patent (pointer chasing)

Our first technique ties both these ideas together, 2013 https://patents.google.com/patent/US9116817B2 *Pointer chasing prediction*.
Consider pointer chasing code, ie code that looks like node->node->node->data, which will compile to something like

```
LDR x3, [x2]
LDR x4, [x3]
LDR x5, [x4, #8]
```

Performance is limited by how fast the address of the next load is acquired by the previous node.

Consider the scheduling of the second load. We want to send the load to the LSU at the earliest possible moment that its dependencies are satisfied (ie it has access to the value of x3) but no earlier.
The most cautious solution is to wait till the value of x3 is in a register, read it from a register, and base the scheduling on that. That makes sense if a physical register, in this case for x3, is the *only* available source for a value (eg for values that were calculated many many cycles ago) but not otherwise.

A better solution is to pull the value of x1 off the bypass bus. Remember that there's a (very wide!) bus connecting every execution unit with the register file to transfer calculated values to their destination register. If these values are appropriately tagged, and the scheduler appropriately matches tags, it can grab the values required for an upcoming instruction off the bus at the moment it's needed.

But that's still not optimal in the pointer chasing case! The fundamental insight is that there is one more place where data could be made available to the load – internal to the LSU.
The value is available in the LSU, but then has to be transported to the bus, moved over the bus, read, then transported back to the LSU.
So even better is to guess that the value will be available in the LSU at the appropriate time, and simply

fire the second load into the LSU. If everything works out correctly, the request to execute the second load (which begins with an adder in the LSU that adds an offset, in this case 0, to the value of x3) will arrive at the adder just as the value of x3 arrives at the adder, fresh from the L1 cache; and the secondary load begins its execution one cycle earlier than it would have been if it had to wait for the value of x1 on the bypass bus.

The patent suggests this is handled as a predictor, caring about the combination of
- a load that depends on an earlier load
Presumably the fundamental scheduling trick here (although Apple gives no details) is to track in the LS Scheduler that the three most recent loads are each generating register pA, pB, and pC; and to test if any of those three registers match the base register of the next load.
and
- the earlier load will hit in the L1 cache as opposed to the Store Queue. This is necessary because, see the diagram below, we're speculatively scheduling the cycle in which when the second load can issue. This is done by using the data in the Load Store Dependency Predictor.
Like any scheduling speculation, this can fail, in which case we Replay the second load.

A year later in 2014 (so now A7 generation) https://patents.google.com/patent/US9710268B2 *Reducing latency for pointer chasing loads* we we improve this in a few simple ways
- we also allow a store that's dependent on an immediately prior address load to use the mechanism. Not that essential for performance, but easy, so why not?

- we now understand that we are using the LSDP in this additional (pointer chasing) way and we allow pointer-chasing failure cases to also train the LSDP predictor
We also get this nice LSU pipeline diagram that helps clarify timing of the different components (remember, again, this is A7 generation; M1 certainly differs in some details because it takes one cycle less).

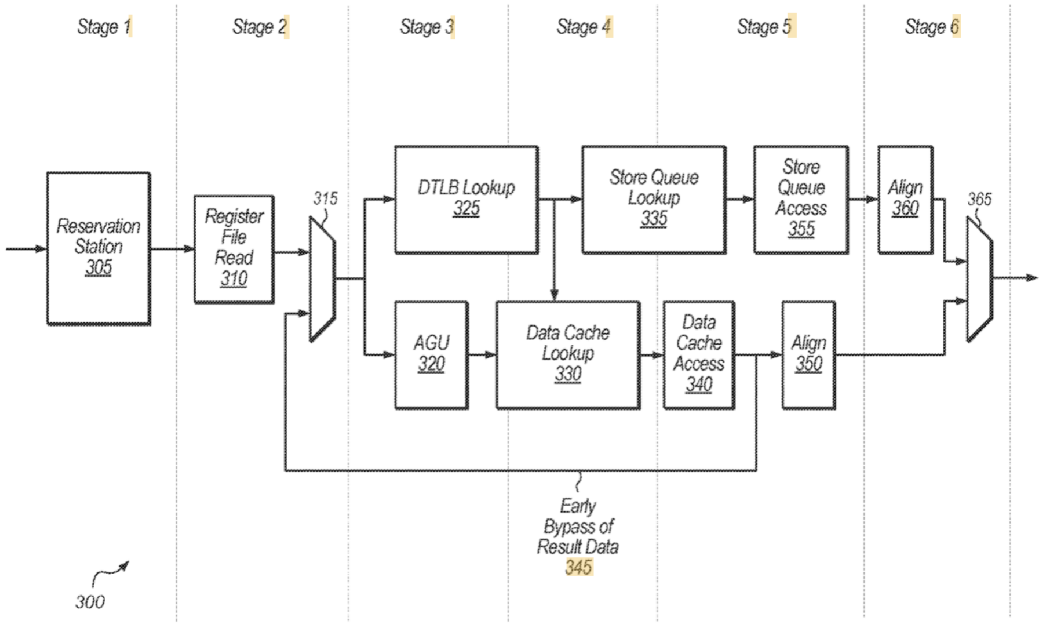


FIG. 3

In a way way this predictor business is required because a value looked up in the Store Queue is only available half a cycle after it's looked up in the L1D. It seems feasible that you could get the timing of these two to coincide (especially if you look up in the Store Queue by virtual address so start it in parallel with TLB lookup) in which case you could still get the value of the patent (detect back-to-back loads and schedule the second load one cycle earlier) without requiring the predictor part.
You still need to validate Store Queue probing by physical address for the rare (but allowed, sigh) cases of virtual address aliasing, so this might mean twice as much probing of the Store Queue, though I could imagine ways around that. (Essentially split the Store queue into two parts tied together by a common "Entry Number" and have one probed by VA, the second part probed in the next cycle by PA.) It's unclear where M1 sits in this respect, though there is value, explained in the Memory Volume, to accelerating Store Queue lookups.

2017 patent (strided load address prediction + pre-execution)

some theory of value prediction

A second way we might know the address faster is to guess! In other words, once again, we use a predictor. This class of predictor is called a *Value Predictor*, more precisely an *Address Predictor*. A recent discussion of the idea is here (2017) https://www.researchgate.net/profile/Rami-Sheikh/publication/318283615_Load_Value_Prediction_via_Path-based_Address_Prediction_Avoiding_Mispredictions_due_to_Conflicting_Stores/links/59b26eea0f7e9b37434e7036/Load-Value-Prediction-via-Path-based-Address-Prediction-Avoiding-Mispredictions-due-to-Conflicting-Stores.pdf *Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions due to Conflicting Stores*.

Being such a recent paper, this is full of interesting ideas!
The “easy” part is the idea of how their address predictor works, based on the path (sequence of PC’s) of prior loads. They claim this load-path history is a high quality predictor (cf the use of paths in branch prediction), and let’s assume that true.

More complicated is how you might implement a value predictor generically. Speculation always generates the question of how you backtrack; in the case of value prediction you would like something as lightweight as Replay, but (by definition of how value prediction works!) the predicted value has to be stored in a register, so somehow you have to deal with that. This may seem like a Replay situation, but if you work through some cases (think for example of a pointer chasing scenario) you will see problems arise because we start with an incorrect register value, and there’s no “natural” way to fix that once the predicted value has been evaluated and found to differ.

They suggest doing this via a second pool of registers, dedicated to speculation, and a bit (or some other mechanism, like physical register numbers above a certain value) in the rename map that indicate a register comes from the speculation pool or the general pool.
As I understand it, their idea is: an instruction at Rename time
- has a Speculation Register pseudo-allocated as its destination (ie marked as the destination in the Rename Map)
- holding the speculated value
This means subsequent instruction can read the speculated value.
- but the instruction actually also gets allocated a standard physical register so that when it executes it writes to that physical register (which has been allocated, but is not "visible" to any instruction via the Rename table)
- at some later point after Execution,
+ physical register is compared with speculation register
+ modify the entry in the Rename table to point to physical register and free the speculation register
+ if the two don't match force a Flush starting right after the instruction whose value was speculated.

Much of this is not ideal and somewhat hand-wavey, and I think it's the elided details that have resulted in Value Prediction still being considered an idea for the future, even though it was first proposed almost 25 years ago. No-one denies that the idea has potential; it's just that every proposed actual implementation always seems to be based on how machines used to operate, and is no longer a great match for the way they operate at the time of the proposal.
It's particularly frustrating that we have to Flush rather than Replay. Flush is heavier weight than we really need; it start by reloading the instructions all the way from I-cache, but the instructions we have are *correct*! We just need to connect them to a different set of input values. That looks like Replay, but the details differ. It's important to understand why.

For Load Replay we have a situation like
- dependent instruction refers to a physical register for one variable, the load (via a SCH#) for the other variable
- the dependent reads the load value from the bypass bus via the the SCH# dependency
- after the dependent tries to execute, and is held [prevented from completing, retained in the Scheduling

Queue] it can sit in the Scheduling Queue until the load completes, re-asserts the appropriate SCH# line, and the load value again read off the bypass bus.

For Value Speculation Replay we have a situation like
- dependent instruction refers to a physical register for one variable, a speculative register for the other variable
- the speculated instruction executes and places the correct (non-speculative value) in a physical (non-speculative) register
- but there is no connection between that register and the speculative register AND
- there is no natural mechanism to change the register dependencies and suchlike of the dependent instruction so as to re-execute it.

However I think this is manageable within Apple's framework, by re-conceptualizing the problem. Rather than thinking of cleanup as somehow moving the correct value to the appropriate register and trying to force instructions to re-read that register, what about thinking of cleanup as forcing an additional dependency? The idea I have in mind (which seems feasible from the outside, but of course we don't know the internal details!) is
- a speculated instruction carries not just a dependency bitvector but also a "speculation-dependency vector" which encodes the SCH#'s of the instruction(s) on which it speculatively depends.
The idea then, is
- a dependent instruction is held in the Scheduling Queue like with Load Replay
- when the speculated instruction notices a speculated value mismatch it sends a notification and the correct value over the bypass bus
- if that matches the SCH# in the the speculation-dependency vector, then the value can be captured off the bypass bus and steered to override the register-based value that was captured earlier.
(Of course this requires details we don’t know, like how it’s indicated that a particular SCH# dependency should feed a value to the first, second, or third dependency of a particular instruction...)

The above is all background to the patent below which is, essentially, a value speculation patent.

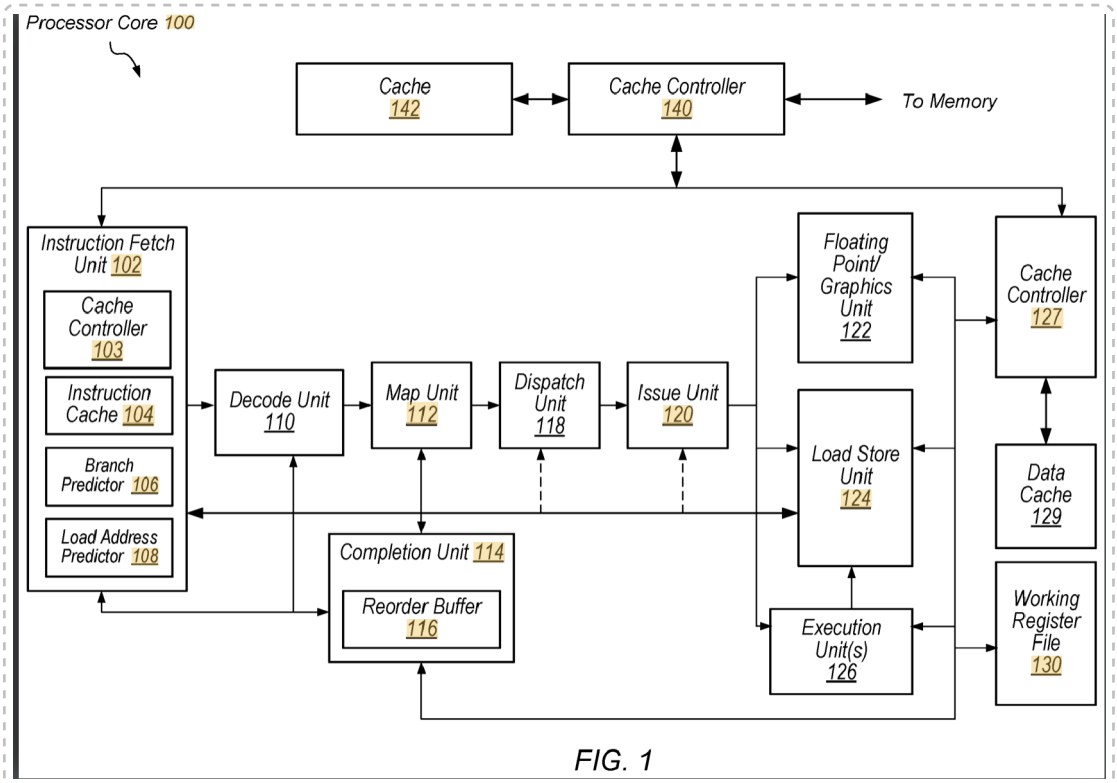
apple’s value prediction patent

Apple's idea, (2019) https://patents.google.com/patent/US20210049015A1 *Early load execution via constant address and stride prediction,* is to track the address that is generated by any particular load (ie note the stream of addresses generated by a load with a particular PC, presumably in a loop). If this stream of addresses forms a linear sequence then we can reasonably predict what the next address will be. And once we know what that next address will be we can
- execute the load early behind the scenes
- record the value in a physical register
- rename the destination register of the load at Rename (the usual ZCL)
- validate by re-executing the load at the correct time, based on the actual values at the correct time.
(Do we have to re-execute the load? In principle perhaps we can use the Load Store Queue mechanisms and the Poison mechanism to detect if the load address has had its value changed. If that's possible,

then all we need to do is validate that the load address matches our speculated load address.)

This may sound like a standard stride prefetcher, but remember the prefetcher gets data into the cache early; we are interested in the next level of getting data from the cache into a register early.

If you looked at the previous diagram for the RF/SP-LSDP pipelines, you saw that, for whatever reason, Apple wants the different predictors to run at different pipeline stages.
They're close to running out of stages at this point, but, no fear! We can move this prediction all the way back to where the Fetch Address is predicted!
That gives us time to perform the load (based on the speculated strided address), and have the load value available for ZCL by Rename.



Experiments

testing pointer chasing (success)

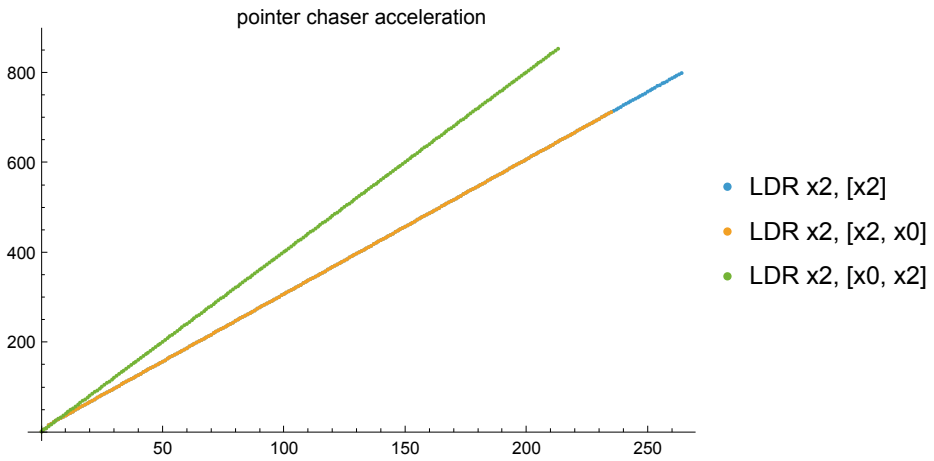
The easiest case to test is pointer chasing loads.
Initialize with
`STR x2, [x2]`, and `MOV x0, #0`;

then compare repeats of:
`LDR x2, [x2]`
with
`LDR x2, [x2, #0]`, `LDR x2, [x2, x0]` and `LDR x2, [x0, x2]`

All of these perform the same task of repeatedly following a pointer (that points to itself). But some versions of the instruction are (we believe, from the Apple patent) susceptible to pointer chasing acceleration and should take three rather than four cycles per iteration.

The second case (adding an immediate to the basePtr) is not completely trivial to code because an immediate of 0, as I have written it, encodes to `LDR x2, [x2]`
To test an offset I'd have to use a more elaborate setup that pre-initialized an entire array of pointers. But I'm pretty confident what the result would be; if you don't believe me do the test yourself!

Out[671]=



We see the accelerator in action. Pointer chasing that feeds the pointer into the first argument (the base pointer) runs at 3 cycles per load, as opposed to normal load loops (and pointer chasing done by someone who doesn't understand ARMv8 assembly) which takes 4 cycles per load.
Later below I tested other variants of this and complex addressing works; even load pair works as long as the pointer being chased is (like above) the first pointer of the pair.

testing stride address prediction (failure)

I was very excited to try the address value prediction patent, but I could find no example for which it kicks in. Like the load after load ZCL, it's easy to fool yourself that some sequence of code is faster than expected because of this ZCL, until you think about it more carefully and realize it's just extreme OoO to the rescue!
So my tentative belief is that, in spite of the patent, this is not present in M1.

Some time after writing the above I read (2017) https://www.researchgate.net/profile/Rami-Sheikh/publication/318283615_Load_Value_Prediction_via_Path-based_Address_Prediction_Avoiding_Mispredic-

tions_due_to_Conflicting_Stores/links/59b26eea0f7e9b37434e7036/Load-Value-Prediction-via-Path-based-Address-Prediction-Avoiding-Mispredictions-due-to-Conflicting-Stores.pdf *Load Value Prediction via Path-based Address Prediction:*
Avoiding Mispredictions due to Conflicting Stores which is a nice overview of (elements of) the problem. The *details* of the paper are most relevant to Qualcomm, what matters is that it clarifies various points that seem implicit in the idea of Value Prediction via Load Address Prediction. In particular it states without equivocation, and with some useful numbers, that
- for the most part the most useful values to speculate, if you go down the path of value speculation, are the result of loads (for the obvious reason that these are higher latency, even to L1, than most integer operations)
- trying to speculate the *value* of loads is somewhat feasible, but requires tracking when an intervening store might have flipped the value at an address
- trying to to speculate the *address* of loads is more feasible (more likely to be useful). You can certainly track addresses that are static (constantly reloaded globals, or function A keeps storing a value read later by function B; Apple extend this to also handle the case of running along arrays loading successive values) BUT
+ the load (occurring early in the pipeline, probably triggered by the PC) has to be validated later at the correct load time, so each speculated load results in two hits to the load unit and cache

If we accept the above (at least in a first implementation, and don't assume Apple has hooked up additional machinery to bypass the second, validating load [eg record the relevant addresses, and check for stores or cache activity between the speculated load and the executed load], this has various implications. The most obvious is that even if you are very confident in your speculation, this is not worth doing if your code is throttled by load/store activity, because you're simply doubling that activity! So along with the functionality discussed in the paper, you also need something to track the level of load-store activity.

This in turn has implications for how you test for this functionality. I was testing it as an extension of the other ZCL activities, which I now think was the wrong starting point. A better framework would be to test it as a way to reduce the latency of a "single threaded" lane of execution. What I mean by this is you want a flow that looks like
- rA=IDIV(value, value#) [where value is varying and value# is set, to a little larger than value], so this result is always 0
- value=LOAD rA
-rA=IDIV(value, value#)
- value=LOAD rA
etc so no OoO is possible, each instruction depends on the previous one.

This is our starting idea, but this bundles multiple loads into the same Fetch or Decode group, which means that a single Fetch or Decode PC "implies" multiple loads, and that could be a problem. If this is considered an optimization only relevant when load/store activity is limited, the machinery may only

kick in when there is only one load in a Fetch or Decode group...
So the next step would be to pad the above idea to one IDIV+LOAD per decode group (now up to 10 in M4), possibly with NOPs, more safely with something like rA+=1; rA-=1; over and over as padding requires. With all this in place (and repeating a hundred or so of the above idea before we loop back) we can compare the expected latency of one block to the measured latency; if everything is working then we should see an effective latency of maybe 4 less than we expect (ie zero-cycle load). Finally we could modify one of the rA+=1 to something like rA+=9 to move to the next INT64 (on an array filled with "value"), to simulate walking an array, and see if we get the same (ZCL-reduced) latency for the loop.
Before starting down this path, we will want to test a few things which could confuse the issue!
Does the divide take as long as we expect? (Is there an early out where it just tests if value<value#?) We might need to have the ratio be some random number, and then subtract that number from rA? Or maybe use an FDIV followed by a convert-to-integer?
Do the rA+=1 and rA-=1 take one cycle? Or does our target chip (like the newest Intel chips) perform short adds at Rename? Maybe we need some nonsense like rA>>=1; rA<<=1 to avoid Rename shenanigans?

(When the address is unchanging, we could be hooking into one of the earlier defined ZCL's, which can reuse an already loaded value, or a stored value, in a recent non-overwritten register. Which is nice, but not what we're testing for! Once the address is linearly increasing then we know we have in fact address prediction happening. At which point we can test things like varying strides, negative strides, etc).

Now that we have an overview of the idea of Value Prediction via Load Address Prediction, we can ask how to improve it. An obvious direction of improvement is to go backwards! We want to predict the address in those cases where it makes sense (either the value at the address frequently changes, or the address keeps changing because we are walking an array). But when neither of these are true, if we predict the value that will be loaded, we can avoid the costs of the (first) load, ie energy and taking up load bandwidth, and we only need to execute the second load (to validate the prediction).
As you will see below in the "what's new" section, this functionality appears to have been added to the M4. As always when looking at patents, maybe the M5? Maybe the idea didn't work out? What the patent describes is how to use a single joint table to perform the training of a Load Value Predictor such that the table can store either Values or Addresses; obviously being able to share a table makes such a table cheaper and thus makes it a little more desirable to have both such predictors.

Yet a third way to handle this issue is to base much of your prediction machinery on the existing branch predictor machinery. This is the idea behind VTAGE. VTAGE is not as powerful as the scheme described in the Apple patents, or in the paper referenced above – but it is cheaper. Which means it might be a good choice for E-cores, if the performance benefits are worth the area and energy costs.

Academic ideas for improving load performance via criticality

The LSQ is an expensive structure, and loads in general are both difficult but vital for performance. Can we do anything smart to reduce their costs?

One interesting set of ideas, written up in (2009) https://citeseerx.ist.psu.edu/document?repid=rep1&-type=pdf&doi=bdabca9a89f7ca0d673020642530eb29f3190650 *Criticality-Based Optimizations for Efficient Load Processing* discusses the use of one my favorite not-yet-implemented ideas, namely criticality. (This is an Intel paper displaying, like so many Intel papers I've referenced, that casual disregard for good ideas that has them where they are today...)

The paper starts by suggesting that while some people think it's too expensive and complex to implement a generic instruction criticality predictor, you can easily implement a simple "load criticality" predictor. This only applies to loads, and does not, in the strict sense, track criticality, rather it tracks an easy, but good-enough, proxy for criticality, namely how many instructions use the value loaded. (The paper discusses a separate table for tracking this, but I suspect in Apple's designs you can just use the register reader-reference count for this purpose.) You can then define a load as critical if this count is larger than a certain value (eg 5).
Now that you are able to segregate your loads into two streams, critical and not, what can you do? The obvious answer is to schedule critical loads first, and that certainly results in value. But there is more you can do. For example
-the whole LSDP machinery, testing loads against earlier store addresses, and predicting whether unknown addresses collide, is not cheap. If a load is not critical, how about we just avoid all this and delay the load until we can be sure there is no load/store dependency? Some energy saved at very little performance reduction.
- if a load is non-critical, does it make sense to be prefetching into the L1 based on this load? Maybe such loads should prefetch to L2, but wait for the actual miss from L1 to move the data to L1? Once again we save some energy at the cost of very little performance.
- we can even use non-criticality to inform the data residence in L1. If the data is not critical, maybe this line should be tagged as first to be tossed when life gets tough and we need to remove lines from the L1 cache?

"Load-like" acceleration

Everyone asks about and expects that Apple will at some point add SVE/2 to their cores. Above I've discussed some of the issues and alternatives to that, including the option of going with Macroscalar. But there are other, outside the box, alternatives to get some of the same value. For example one way to use SIMD (which becomes limited with SVE) is for table lookup and data permute type operations.

One interesting way to improve this type of work is, rather than trying to build a wider data rearrangement network via SVE, accept that we have a good data rearrangement network already in load/store.

The problem with load/store is that normal load/stores execute to global address space which
- has to be coherent,
- is paged
- and has various other complications, which make using it somewhat energy/area expensive, and of

limited performance.

Alternatively we have register space, but the problem with registers is that they cannot be indexed into, which prevents their use for all sorts of particular algorithms.

What if we had the best of both worlds? Imagine something like a 32kB SRAM that forms an address space specific to a core with
- no relationship to the outside world (like registers), but
- *byte addressable* and highly banked (so with multiple simultaneous and low-latency loads and stores possible)?

Without the concerns that require a deep load/store queue and a TLB, it should be possible to load and store multiple values into this storage with perhaps 2 cycle latency.
This is analogous to similar functionality (the Threadblock storage) available in GPUs.

Like with GPUs, this local address space can't solve all problems, but there's a wide range of problems where you
- can solve a small version of the problem in (probably NEON) registers;
- aggregate those results to a mid-sized version of the problem solved in this local address space;
- and aggregate multiple of those mid-sized solutions successively in global address space.

As always, the larger the fraction of such work you can do locally (in this 32kB local SRAM, without the costs of MMU or snooping) the more time and energy you can save...

I suspect this is an idea it's worth looking into by both Apple and ARM.