Advanced Class Lesson 3

The CPU

CPU

- Ah yes.
- Very CPU.
- So simple.
- I’m definitely not going to die as I explain this.

CPU a

- A minimal CPU can be simplified into 5 parts.
    - Arithmetic Logic Unit (ALU)
    - Control Unit (CU)
    - Decoder
    - Address Generation Unit (AGU)
    - Registers

CPU c

- Once again, a CPU has developed many more components that are used to make it faster
- Aside from the abovementioned components, CPUs also contain:
  - Cache
  - Floating-Point Unit (FPU)
  - Memory Management Unit
- But, before we can understand all of these, we need to start from the bare basics

CPU d

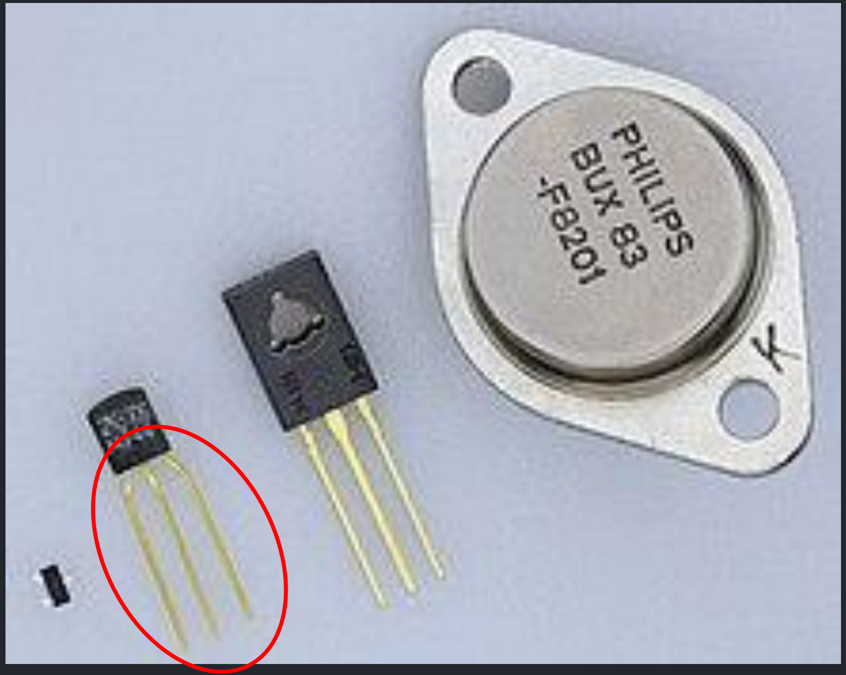
- IMPORTANT NOTE
- A CPU cannot orchestrate anything on its own!!
- A CPU is only hardware!
- When I say that the CPU does something, it means that **some code is instructing the CPU to do it**!!!
- This code can be from the OS, or from the kernel, but without instructions, the CPU will not do anything!
- The CPU is the brain, the OS is the soul, and the kernel is the brain stem, if you want an analogy

CPU b

- The CPU also does not access memory, storage, etc.
- The CPU only requests an external component for data
- This is a misconception by a lot of people, for reasons unbeknownst to me.

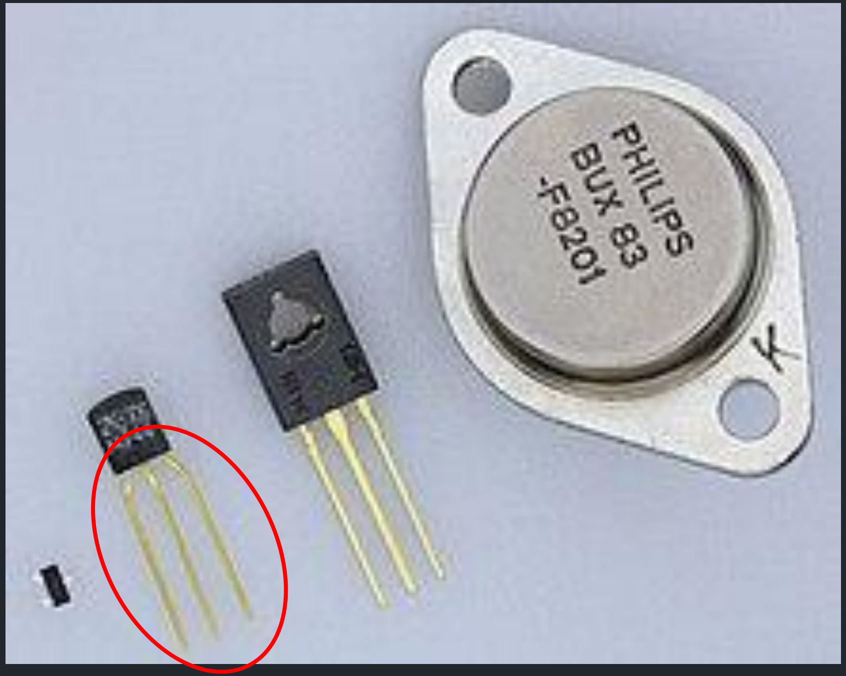
Transistors

- Transistors are basically electronic switches
- They come in all different shapes and sizes
- In general, what they have in common is 3 pins
- The first pin is usually the 'live' pin, i.e. They haven't gone through the pain and suffering of life yet.
- I'm not funny.



Transistors 1

- Anyways, the first pin is the one you connect to your voltage source’s positive electrode
- The third pin is the one you connect to your voltage source’s negative electrode
- As for the middle pin, it’s the switch
- If there is a voltage in the middle pin, current is allowed to flow between the first and third pins



Transistors 10

- How do these things work?
- Black magic.
- This goes into ch*mistry, so I genuinely have no clue how this works.
- The gist of it is that when the third pin is high, an electric force is generated that opens the circuit

Transistors 11

- However, what I do know is that transistors are consisted almost solely of silicon
- This means that you can just cut out silicon into the right shape, and it will still work as a transistor
- That fact has made it possible to make transistors in the nanometre-scale!
- In fact, the first time we breached the μm scale into the nm scale was in **1987**
- Current transistors are 3 nm wide

Transistors 100

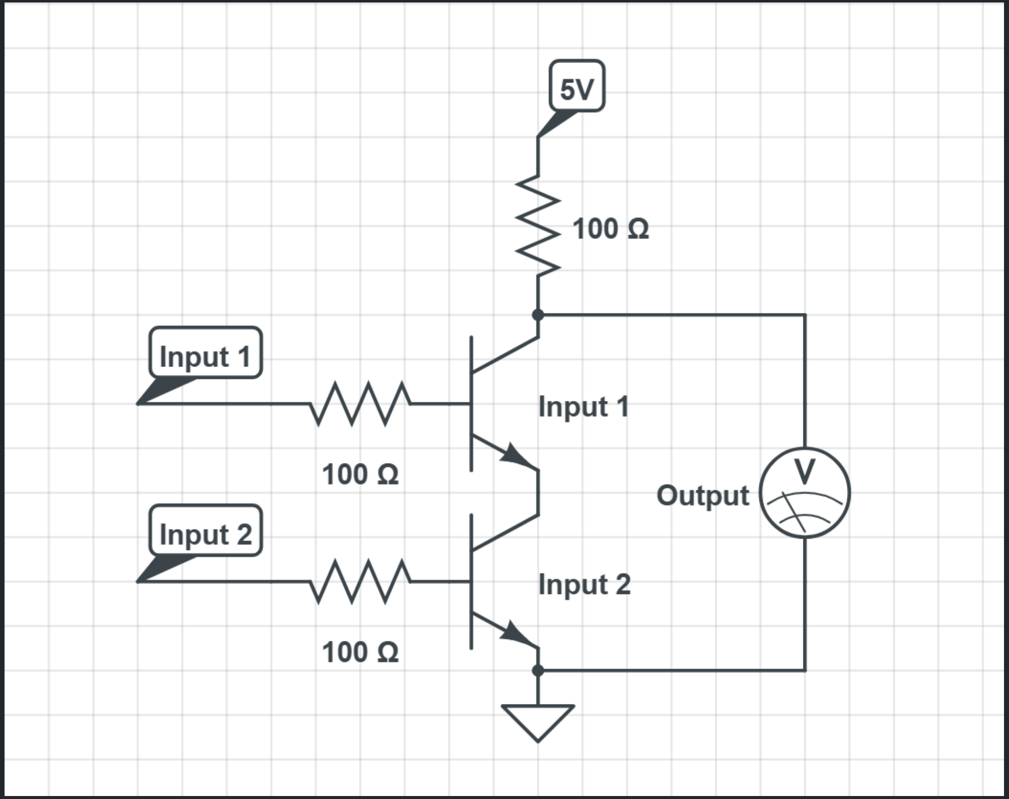
- Does this mean we'll hit the picometre scale soon?
- Unfortunately, no
- Transistor sizes are hitting their limits because of a thing called quantum tunnelling
- Yes, that's real.
- I googled it, and Google AI agreed with me.
- Anyways, because of that phenomenon, at such a scale, electrons can start to circumvent switches semi-consistently

Transistors 101

- Good thing is that the geometry of transistors has begun changing, with the introduction of a new type of transistors, called GAA transistors
- This means that we can now compact them even further
- If you want more info, I can link a few technical documents about them

Transistors 110

- You can chain transistors in a way that can emulate every possible operation
- Within reason, of course
- Here’s an AND gate
- How do transistors achieve this?
- By chaining default-on and default-off transistors,
- Suddenly, and Without Warning,
- The system becomes Turing complete



Turing Completeness

- The idea that you can take encode any given program into a set of equivalent instructions, and execute the new set of instructions using to get an answer
- There is no need to guarantee that it terminates before the heat death of the universe, or that it terminates at all
- How fun.

Transistor 111

- Transistors are usually arranged in a way that emulates entire logical instructions at once
- These things are known as (Boolean) Logic Gates
- Logic gates are the building blocks of more complex operations, like addition
- If you are curious, you can go to https://nandgame.com/ to try it out yourselves
- Recommend you try to play this game at least once, because it really enhances your idea of how CPUs work

Registers a

- A register is a variety of language used for a particular purpose or particular communicative situation.
- A register is also the fastest form of memory in a CPU
- They're basically where the CPU stores the things it is currently working on
- In the modern x64 device, you have 16 64-bit general-purpose registers
- You can treat the 64-bit registers as 32-, 16-, or 8-bit registers by only using a part of the register

Registers c

- Registers have different naming conventions based on how many bits they can hold
- Let’s start from the beginning

Registers d

- In the beginning, Intel god said, “Let there be ~~light~~ registers a, b, c, d”.
- By god, I mean our lord and saviour, Charles Babbage or someone that works at Intel idk.
- They are named a, b, c, and d because they are the first 4 letters of the alphabet

I lied

- In the beginning, Intel god actually said, “Let there be ~~light~~ registers a, c, d, b”.
- They are named a, c, d, and b because of the purposes that these registers were traditionally used for
- Accumulator
- Counter
- Data register
- Base register

Registers b

- These registers were all 8 bits
- Then, when we moved to 16 bits, they added 4 more registers, and adapted the 4 current registers to 16 bits
- Care to guess how they adapted the 4 current registers?
- By suffixing them with -x, of course
- Why?
- Because ax is ‘a extended’
- What about the other 4 registers?

Registers ax

- Let’s do the logical thing and call them si, di, bp, sp.
- Respectively,
  - Source Index
  - Destination Index
  - Base Pointer
  - Stack Pointer
- Fun.
- Let’s extend them to 32 bits, now!
- And let’s add 0 new registers.

Registers cx

- It’s not like we can call the new registers ‘axx’…
- So let’s just call it ‘eax’, for ‘Extended a Extended’.
- How fun.
- Let’s raise them up to 64 bits!
- Let’s add another e, or x to it!
- Nope
- ‘rax’ for ‘Register a Extended’.
- Very fun.
- This section serves no purpose other than to make fun of Intel.

Registers dx

- Back to the point, these general-purpose registers are extremely fast storage, with an access time of <1ns
- In fact, with 5.5GHz clock speeds becoming common, access times are approaching 0.2ns
- It’s like a tool that you’re holding. The only time you need to access them is to remember that you’re holding them.
- However, these registers usually store up to 8 bytes only
- This is where the term 64-bit computing comes from
- These registers are regularly overwritten by instructions, so they’re not good for long-term storage

Register bx

- Other than these general-purpose registers, there are other, specialised registers
  - Flags register, used to store flags
  - FPU registers, used specifically for floating point numbers
  - MMX registers, used for SIMD
  - XMM registers, used for SIMD 2, electric boogaloo
  - Segment registers, used for virtual memory and segmentation
  - Control registers, used for other stuff™
  - Debug registers, used for obvious stuff
  - MSR register, only used by the OS

Flags register

- An important register
- It stores the current state of the CPU as a list of bits, or ‘flags’.
- The easiest ones to explain are the carry and overflow flags
- Let’s say the ALU does $0001 + 0001$
- The ALU sees that the 1s in the LSD are both 1s
- Thus, it sets it to 0, and carries over
- How does it know that it carries over?
- It sets the carry flag to 1
- There are more uses for the carry flag that I’m lazy to cover

Flags register

- Next topic.
- The overflow flag is set to 1 when you have an integer overflow
- Let’s say you have a 4-bit integer
- You do operation $1111 + 1 = 10000$
- But you can’t fit 10000 into 4 bits
- This is called an integer overflow
- So, the ALU sets the overflow flag to 1 so the error doesn’t go unnoticed

Flags register

- Another flag that you probably want to know is the sign flag and zero flag
- These store the data about the result of an arithmetic operation
- Why do we need this?
- Because of assembly instructions like:
  - jz—jump if zero
  - jl—jump if less than (used for signed values)
  - jb—jump if below (used for unsigned values)
- This improves the execution speed because you don’t have to use further logic

Flags register

- Not something you may need to bother with, but there’s 2 more flags that might be good to keep in mind
  - The direction flag controls if stuff goes left to right or right to left
  - The interrupt enable flag enables or disables maskable interrupts
- These will be further explained when explaining the MMU, and interrupts respectively
- If you’re interested, you can search up the idea of ‘endianness’.

ALU

- The ALU is the part of the computer that contains all the circuitry required to do basic arithmetic
- This includes addition, multiplication, division, and the bitwise operations

ALU 1

- What about subtraction?
- It’s the same as addition, but with negative numbers
- That sounds stupid as hell, but that’s genuinely how it is

ALU 1 * 2

- Wait, then why is there division?
- That’s because in the ALU, you can only do operations on integers or raw binary data
- So, you cannot multiply by a reciprocal, because those are, by definition, not integers
- I mean, technically, you can do $1/1$, but why would you bother?

ALU 1 * 2 + 1

- Remember how I mentioned that negative integers were just stored as numbers, but with a 1 in the first bit instead of 0?
- I lied.
- They are stored with a method called two's complement
- To convert a binary number, e.g., 5 (0101) to -5, you flip all the bits, then add 1
- So, -5's binary representation is actually 1011 $(0101 \Rightarrow 1010 \Rightarrow 1011)$
- Wait, but why bother?

ALU 1 + 3

- Because with adding numbers stored in the two's complement format uses the same logic as adding numbers
- Let's take a look at how addition works
- $0010 + 1011\ (2 - 5) = 1101\ (-3)$
- Now, let's negate it
- $1101 \Rightarrow 0010 \Rightarrow 0011\ (3)$
- Also, by process of building circuits specifically for converting between the positive and negative forms, two's complement isn't slower

Control Unit

- The control unit of a CPU is what basically delegates tasks for each component in the CPU
- The CU sends signals that encode the data, instruction, etc. throughout the CPU
- This is just the brain of your brain, whatever that means.
- Here, a lot of optimisations happen
- Very simple optimisations, but to great effect
- Because computers are all about doing stuff fast, I’ll begin to introduce optimisations for the CPU from here on

Von Neumann style architecture

- Our CPUs tend to operate on an architecture called the ‘von Neumann’ architecture
- This means that each instruction execution is split up into 3 discrete steps
    - Fetch
    - Decode
    - Execute

Von Neumann style architecture

- Fetching is the step of getting an instruction from memory, cache (we’ll get to that later), or storage
- This can take different numbers of clock cycles based on instruction count and where you’re getting the instruction from
- However, this is an important step in the process
- For obvious reasons.
- There’s no way to skip it
- For obvious reasons.
- We can only try to optimise the fetching

Von Neumann style architecture

- Decoding is the process from converting the instructions from a string of binary to something that the computer can do
- Because $00000000\ 11001000$ means as little to a computer as it does to you
- It’s the equivalent of $a += c$, in terms of the registers
- The CU has to parse the instruction, and figure out what the instruction wants it to do
- This step cannot be skipped
- For obvious reasons

Von Neumann style architecture

- Execute is where the Fun™ Stuff™ Happens™.
- Data gets sent to and from the registers, ALU, memory, external components, whatever
- This step can be skipped if you want to do literally nothing.

Von Neumann style architecture

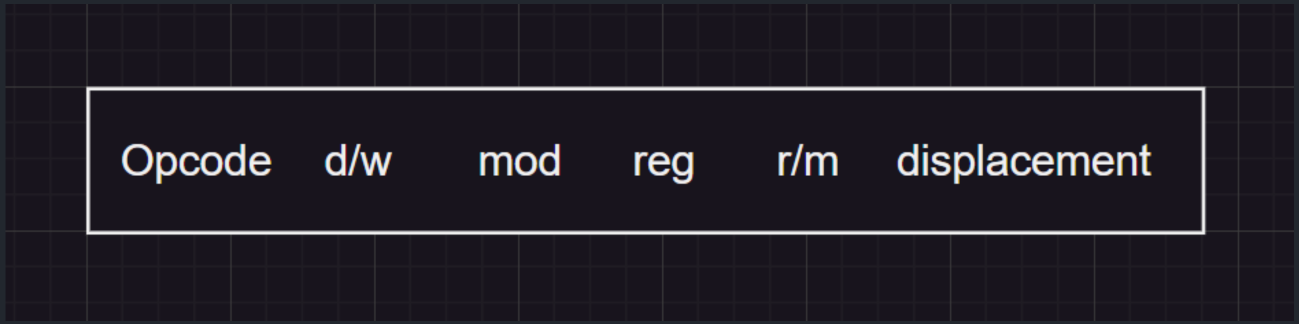
- There’s a Tom Scott video on this process
- However, none of the above 3 steps need to happen in 1 clock cycle
- Fetching might require you to fetch data from memory, which obviously takes a long time
- Decoding is usually 1 clock cycle, unless you’re using some weird, obscure and unoptimized instruction
- I can’t think of many instructions that only take 1 cycle during execution, either.
- Very empirical indeed.

Decode

- Have you ever wondered how a CPU reads machine code?
- No?
- Doesn’t matter, moving on.
- We’re doing this one first because it has the simplest to explain optimisations
- Even if it’s in the form of a .exe, it’s still data that’s stored in the PC, not actual instructions that the CPU can recognise
- This is where the decoder comes into play

Decode 0

- What even are instructions, from the perspective of a CPU?
- Instructions are just a string of binary data that is fed into the CPU
- In the x86_64 family (Your Intel and AMD chips), the instructions tend to follow a format



- Of course, there are exceptions for optimisation purposes, but most instructions are like that

Decode 1

- The opcode field is a pattern of bits that can uniquely identify each instruction that the CPU has to do
- For example, the opcode for ADD is 000000
- Opcodes can be of different lengths, but usually, they are within 8 bits
- For example, the increment operation (Which is equivalent to adding one) has an opcode of 1111111
- Size doesn’t matter (Insert complementary joke here), it just has to identify every instruction uniquely

Decode 2

- What are the d/w bits?
- Let’s skip forward for a moment
- What are the mod bits?
- You see, because this encoding was invented back in my day, storage was still a major concern for programmers
- So, the engineers at Intel wanted to cram as much functionality into every opcode, so that you could minimise the size of each instruction as the number of instructions grew

Decode 3

- They realised that instead of giving each instruction its own opcode, they could group all of them into an instruction, so you could choose the functionality when you're using that instruction
- This is how the mod bits came about
- This is complicated as hell, so we'll move on for now, so that we can come back again

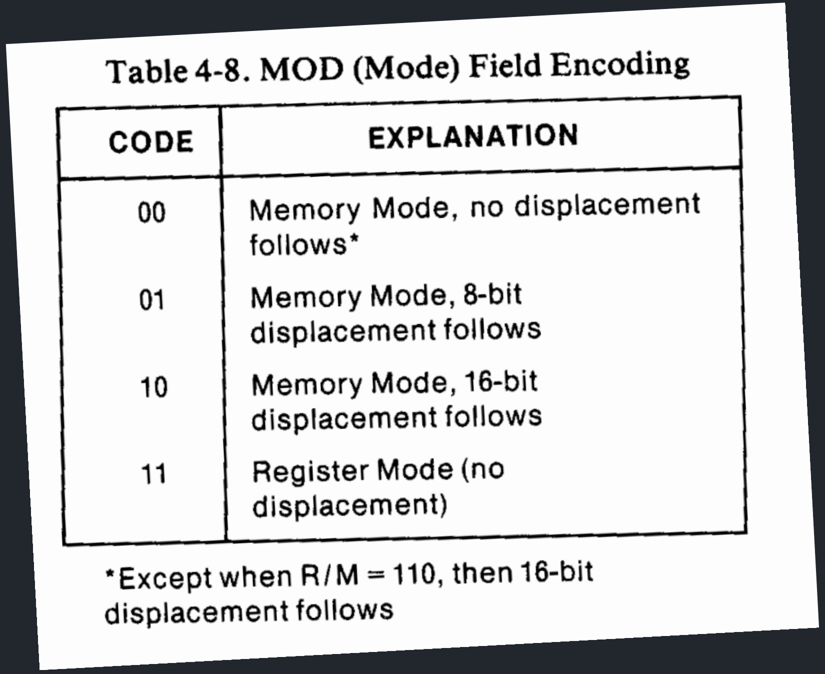


Table 4-8. MOD (Mode) Field Encoding

CODE	EXPLANATION
00	Memory Mode, no displacement follows*
01	Memory Mode, 8-bit displacement follows
10	Memory Mode, 16-bit displacement follows
11	Register Mode (no displacement)

*Except when R/M = 110, then 16-bit displacement follows

Decode 4

- Didn’t you say they wanted to minimise opcode counts?
- Why’s there a separate increment instruction, then?
- Because adding 1 to an int is an extremely common operation, the Intel engineers determined that it would be able to save space even if the number of bytes they need to represent the opcode is greater
- How fun.

Decode 5

- The reg bits contain the indexes of a register
- In general, registers are involved in every instruction, which is why there’s a reg field specifically for registers
- The r/m bits represent different things depending on the mod bits
- In general, for the add operation $a + b$, the reg field represents a, the r/m field represents b
- Fun.

Decode 5

- I think now's a good time to explain what the d/w bits are for
- The d bit stands for the direction bit
- Effectively, for the add instruction, we're not actually running $a + b$ but $a \mathrel{+}= b$
- So, it is significant whether we're doing $a \mathrel{+}= b$ or $b \mathrel{+}= a$
- But why don't we just switch the order around if it doesn't work?
- Simple reason: The reg field can only contain registers, and the add instruction can add numbers to things in memory, not just the register

Decode 6

- The w bit is short for the width bit
- The width of a register is basically the size of the data you can fit into the register
- In the 8086, there is only 1 bit for the width bit that decides if it is 16-bit or 8-bit
- Why does it matter?
- Because that determines if it uses the a register or the ax register.

Decode 7

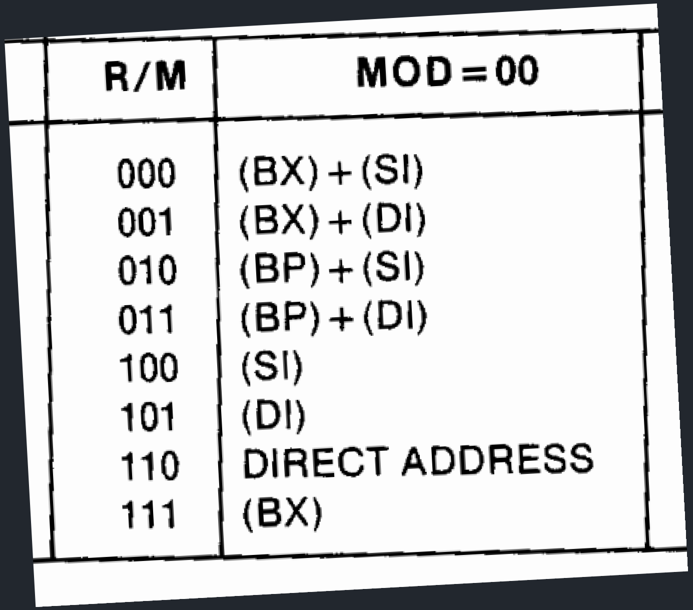
- For the add instruction, if mod = 11, we are basically adding the values stored within 2 registers to each other
- So, in essence, assuming 16 bits, you are running $ax += bx$
- However, if it’s anything else, it get a little complicated
- Remember registers bx, bp, si, di?
- Ever wonder what those are for?
- You don’t have to wonder anymore.
- They’re for so-called effective address calculations

Decode 8

- Remember that memory is just a list?
- And that you basically index into the list to access the memory at that index?
- And that an alias for index in this context is ‘memory address’?
- Effective address calculations are basically how you access memory
- Let’s start with the simplest effective address calculation

Decode 9

- If mod = 00, these are the effective address calculations that are done
- The effective address calculations are encoded by the R/M field
- For the purposes of this lesson, let memory be represented by $M = [0] * 1024$ so that we have 1024 bytes of memory
- Now, let width be 1 byte, mod = 00, and r/m = 000, $bx = 64$, $si = 8$
- The result is now $a \mathrel{+}= M[bx + si]$



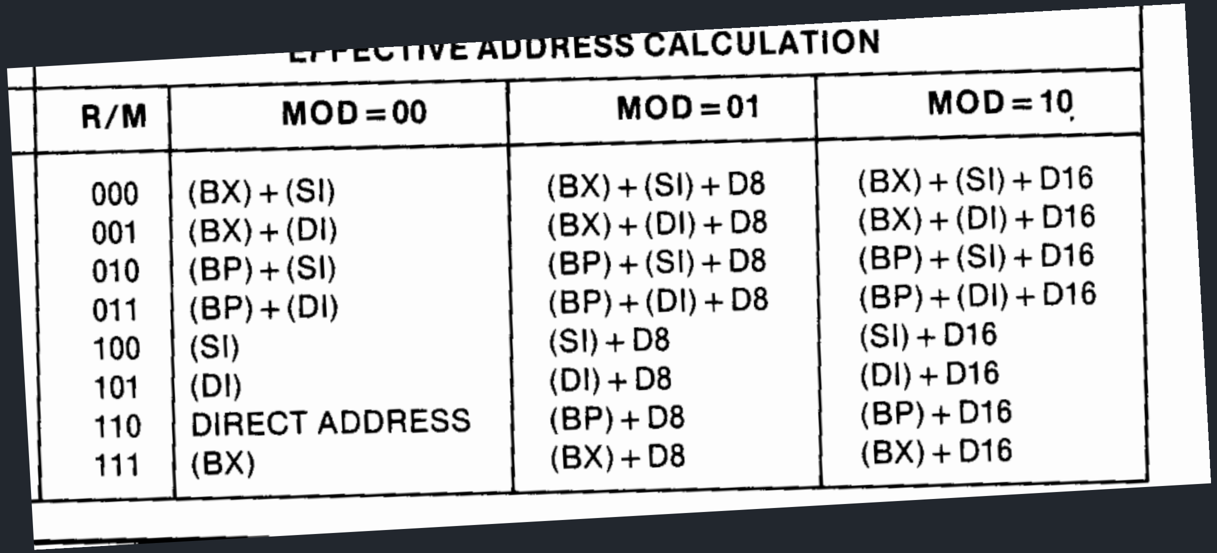
R/M	MOD=00
000	(BX)+(SI)
001	(BX)+(DI)
010	(BP)+(SI)
011	(BP)+(DI)
100	(SI)
101	(DI)
110	DIRECT ADDRESS
111	(BX)

Decode 10

- But what is that Direct Address thing in 110?
- It’s basically a short form for using a literal value
- An example would be $a \mathrel{+}= M[1]$, where you access memory address 1 without using any register
- But where do we store that literal?

Decode 11

- A displacement is a region behind the base instruction whose size depends on the literal that needs to be stored
- When using mod = 01 or 10, another address is stored within that displacement
- The extra displacement stores an 8-bit integer or 16-bit integer that is added to the final address to be accessed
- These effective addresses will be accessed when the instruction is finally executed.



EFFECTIVE ADDRESS CALCULATION

R/M	MOD = 00	MOD = 01	MOD = 10
000	(BX) + (SI)	(BX) + (SI) + D8	(BX) + (SI) + D16
001	(BX) + (DI)	(BX) + (DI) + D8	(BX) + (DI) + D16
010	(BP) + (SI)	(BP) + (SI) + D8	(BP) + (SI) + D16
011	(BP) + (DI)	(BP) + (DI) + D8	(BP) + (DI) + D16
100	(SI)	(SI) + D8	(SI) + D16
101	(DI)	(DI) + D8	(DI) + D16
110	DIRECT ADDRESS	(BP) + D8	(BP) + D16
111	(BX)	(BX) + D8	(BX) + D16

Fetch

- Fetching is usually done by the CPU requesting data from ram through a **controller**
- Controllers have a circuit that helps you get information from external sources
- This can be storage, memory, keyboards, etc.
- Controllers are how CPUs interact with all external components
- This part of the process is usually run by the kernel, through drivers

Address Bus

- It is a series of wires that sends data from the CU to every single controller
- I genuinely can't find any info about what controls the address bus
- It's just in the CU somewhere
- There is usually a handful of wires that encode which controller should activate, given an address
- For an example, I'll show you my breadboard PC that uses the W65C02S (right), a 16-bit CPU

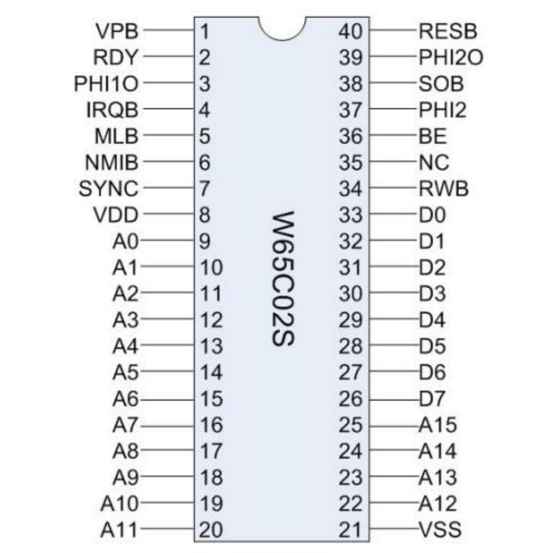
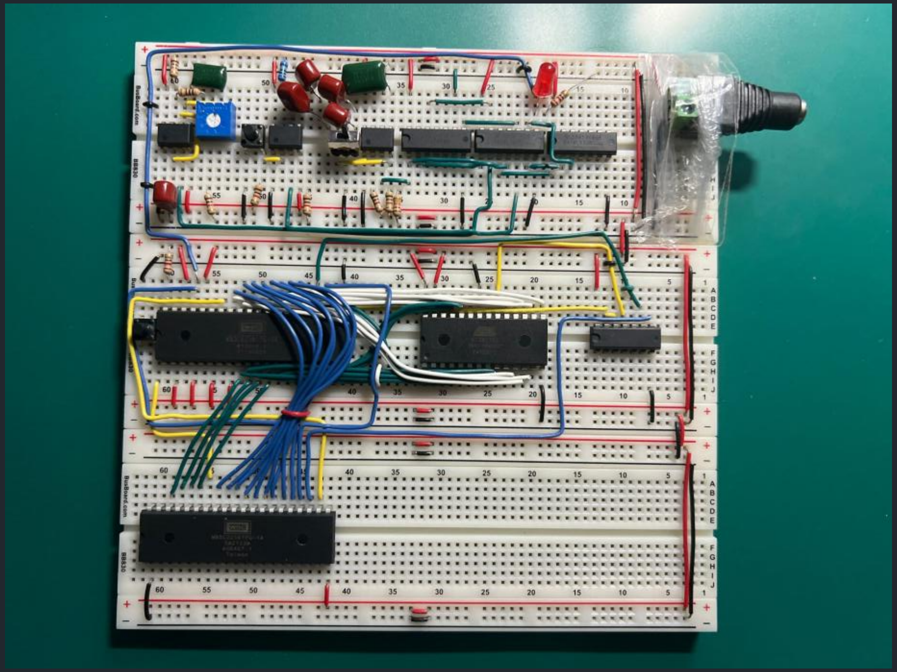


Figure 3-1 W65C02S 40 Pin PDIP Pinout

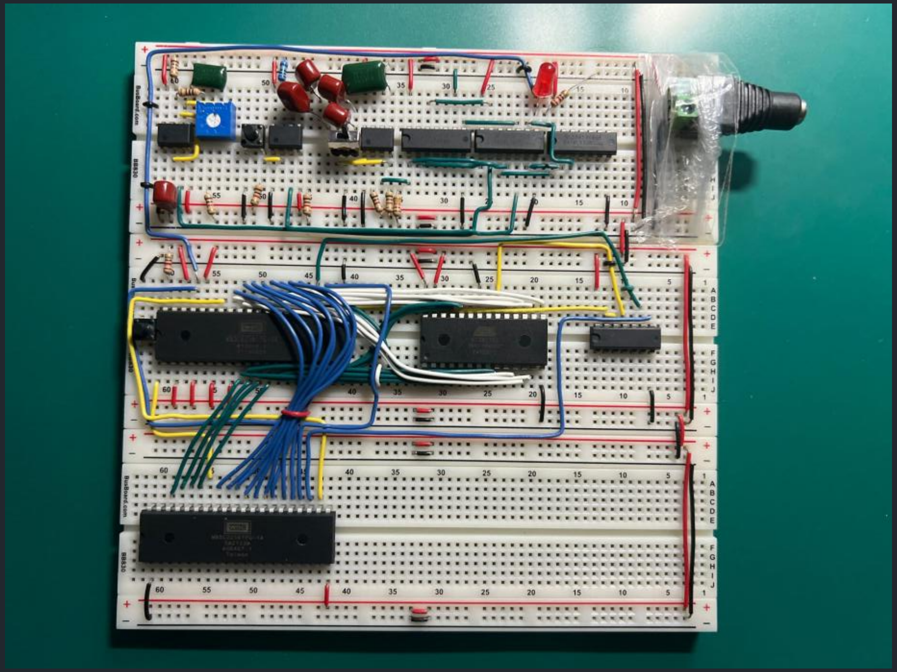
Address Bus 1

- You can see the breadboard computer that I'm building (right)
- The massive chip on the middle, left is the CPU of the computer
- The big chip in the middle, middle is storage
- The massive chip on the bottom, left is the controller for an LED screen that I've yet to install



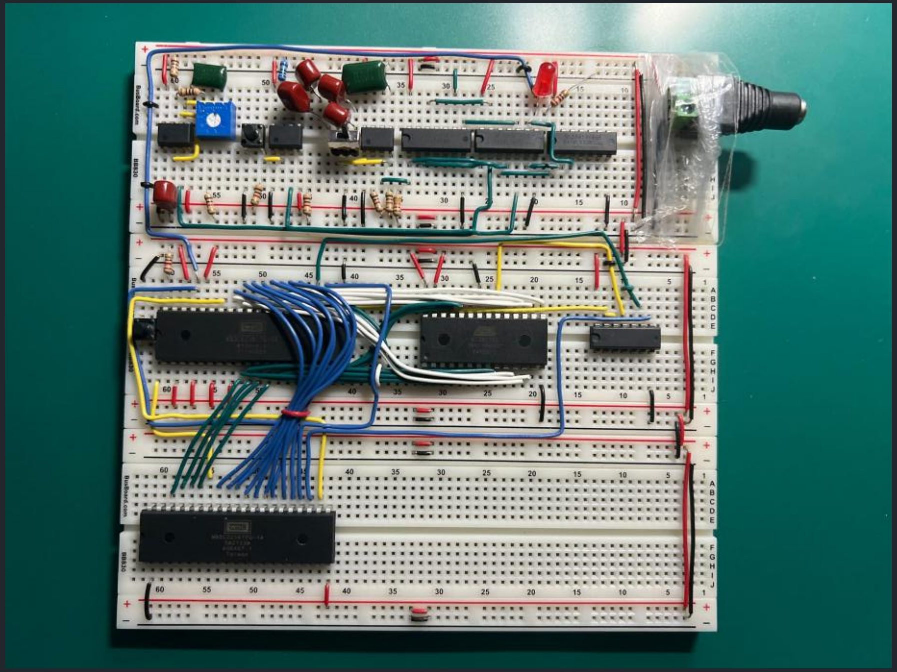
Address Bus 2

- As you can see, the address bus (at least parts of them) are connected to **both** chips
- There are 16 address wires in the CPU
- However, there are only 15 address inputs in the storage, and there are no address inputs in the controller
- However, both chips have an input that turns the chip on, called a **chip enable** pin



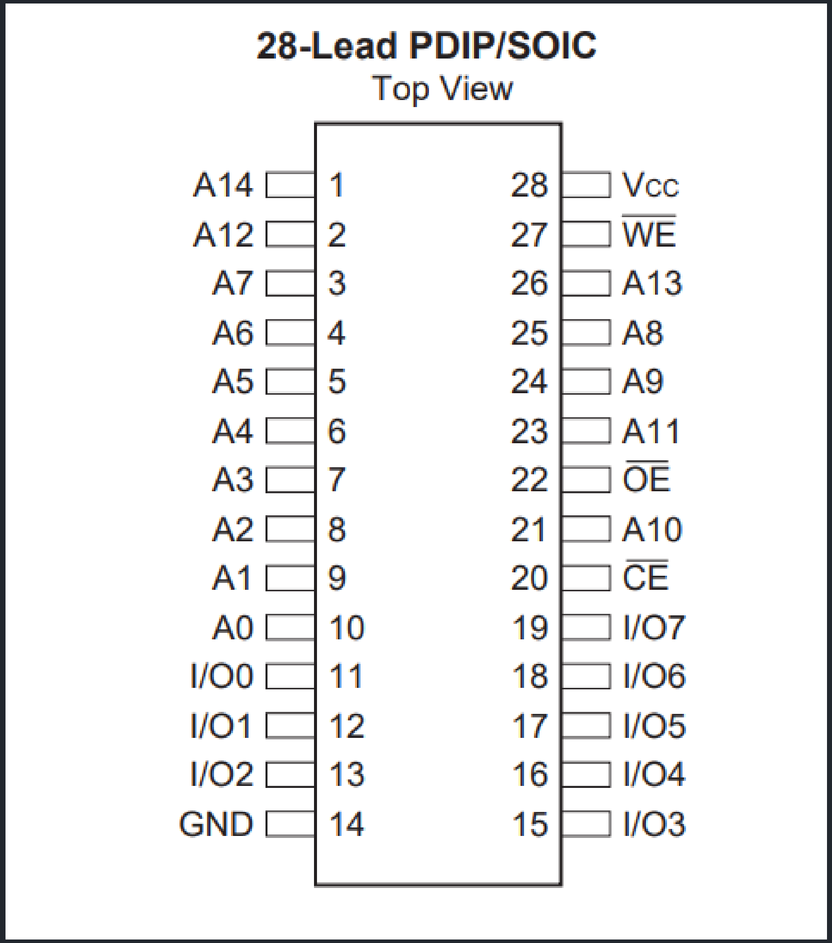
Address Bus 3

- So, I connected A15 on the CPU to the Chip Enable pin of the storage chip
- Then, I invert it and send it to the Chip Enable pin of the LED display controller
- This mimics how a CPU uses the address bus
- The address buses are all connected to each other, but they have special addresses that selectively activate different chips



Digression

- Whoever designed the storage chip doesn't know how to count.
- Imagine counting 0, 1, 2, 3, 4, 5, 6, 7, 12, 14, 13, 8, 9, 11, 10.
- Must be because it's designed by an American or something idk.



Fetch 1

- For fetching, there is a special register in the CPU called the program counter
- The data stored within the register is just a memory address
- At the start of the fetch step, the CPU sends the address stored in the program counter to the address bus
- The data returned will (or rather, should) be one byte of an instruction
- Based on the first byte of an instruction, the decoder will be able to figure out if there is any need to get a second byte for that instruction

Fetch 2

- Usually, after the second byte, the CU will know the full size of the instruction
- Once the full set of instructions is read, the CU will send the requisite signals to execute the instruction

Fetch 3

- The only significant way to optimise this is to reduce memory access times
- Remember how I said that storage access latency is about 100µs, memory access latency is about 100ns, and register access latency is about 0.2ns?
- If you didn’t, now you do.
- It can’t be that between 0.2ns and 100ns, you have nothing in between, right?
- After all, we already have 16GB ram sticks, but registers can only store around 1kB at most

Cache

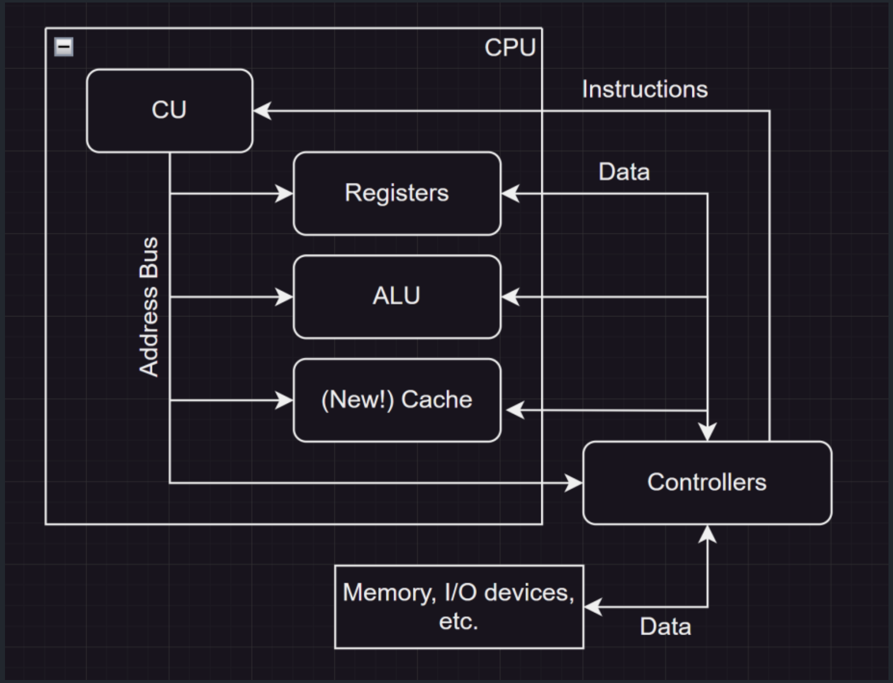
- Cache is the middle ground between the 2
- In the modern CPU, there are usually 3 ‘levels’ of cache
- Each increase in level gains the cache some size
- Each increase in level also increases the access latency

Cache L1

- The L1 cache is usually 64kb, and there is usually one of them in each CPU core
- Its latency is usually around the 1ns ballpark
- The L2 cache is usually up to 1MB, and there is usually one of them in each CPU core
- Its latency is usually around 4ns
- The L3 cache is usually around 32MB (Unless you’re using my PC, in which case you only have about 10), and there is usually one of them in each CPU
- Its latency is usually around 50ns

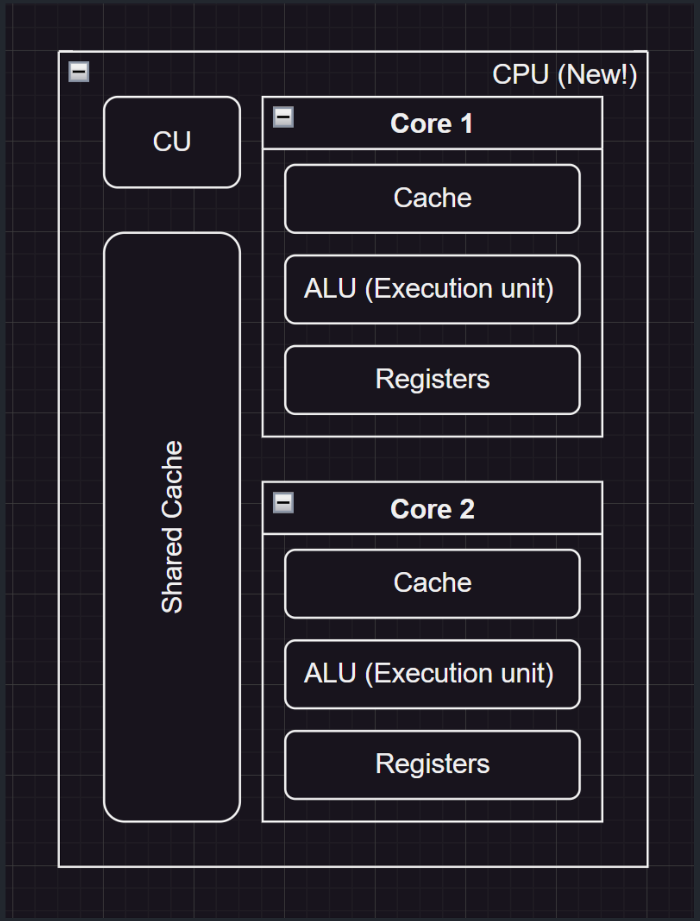
CPU Cores

- Something that you might find strange with what I said
- CPU cores?
- You might have vaguely heard of these things
- So far, we've been operating under the assumption that CPUs look something like this (right)
- Turns out, this is no longer the case



CPU Cores 0

- In modern CPUs, the CPU tends to look something like this, instead
- Notice how each ‘core’ has duplicate things?
- Like the cache, ALU, and registers?
- Modern CPUs duplicate them to increase the number of instructions they can execute at the same time
- There are usually more components that I have not taught you about.
- I’ll fill them in as we go.



Cache L2

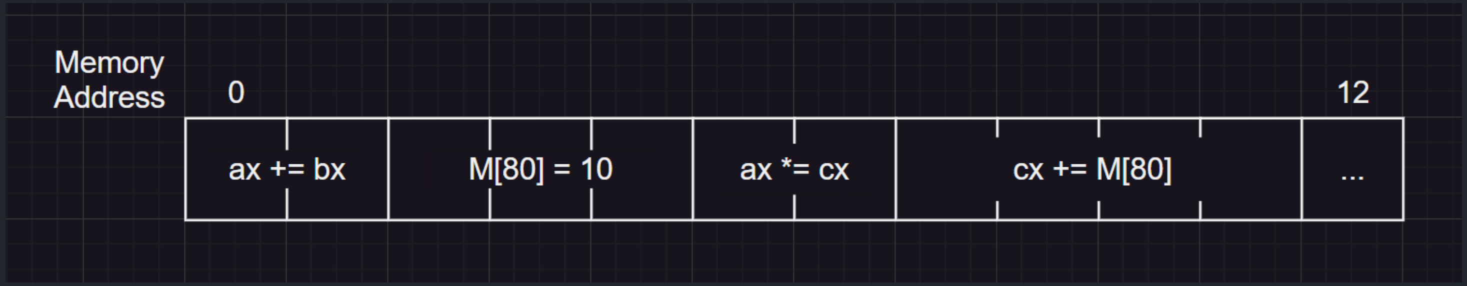
- The ‘shared cache’ in the earlier diagram usually represents the L3 cache
- However, depending on system, the shared cache could also contain the L2 cache
- Nothing in this lesson is true for all CPUs because there’s just so many limitations in many fields
- For example, in the IoT field, CPUs may not even contain ALUs
- How stressful.

Cache L3

- The CU usually uses the cache by moving data that it predicts it will need from memory into the specific level of cache, depending on how often it'll need it
- The CU has some logic within it that favours predictable memory access patterns
- For example, accessing a contiguous chunk of memory, or jumping forward 16 bytes consistently
- This logic is beyond me, but it's so consistent that it was recently taken advantage of to create a memory vulnerability

Fetch 4

- Back to this, there are a few ways that we can make fetching faster
- First, instruction caching
- Instructions are stored as data one after another
- So, what if, instead of fetching the instructions byte by byte, we copy the next, say, 7 bytes into L1 cache, so that we can access them immediately?



Fetch 5

- Since instructions are usually ordered in chronological order, we have some certainty that the next instruction to be executed will be in the next few bytes
- So, instead of only getting the first byte of $ax += bx$, as we've been assuming thus far, we may as well get all the bytes up to the first byte of $cx += M[80]$
- That way, we can enjoy the 1ns access times of the L1 cache

Execute

- Execution is where the Fun Stuff™ happens
- This is when the CPU instructs the ALU to begin working on data in the registers
- This technically isn’t correct, though
- The ALU is not the only thing the CPU instructs to do work
- In general, units like the ALU are called execution units
- Why?
- They execute stuff (Shock)

Execute

- This step naturally has the most optimisations put into it
- The first optimisation is to minimise the number of clock cycles per instruction
- The first method is called **pipelining**.

Digression-ish

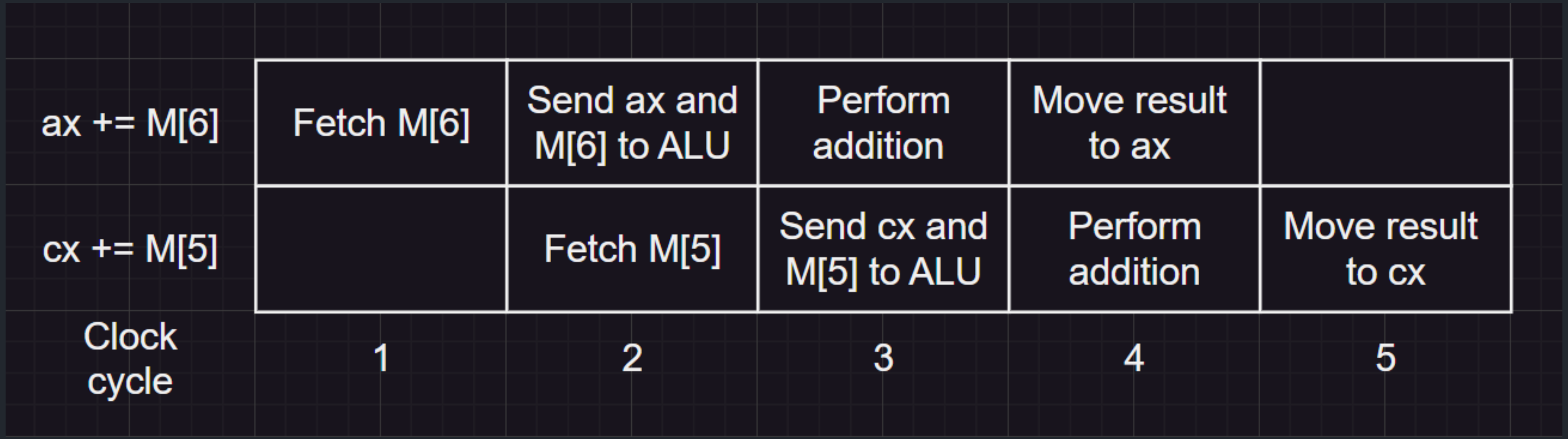
- Let’s say you have something that you must execute in steps
- For example, you have 1 washing machine, and 1 dryer
- Now, let’s assume you have 2 loads of laundry that you have to do
- Is it really necessary for you to wait for the first load of laundry to be washed and dried before you can start on the next load?
- No, you can just stick the second load in while the first load is in the dryer

Pipelining

- Pipelining is the process of staggering the execution over multiple clock cycles, so that you can execute multiple instructions at once
- So, how does this make anything faster?
- Let’s say you’re trying to run the operations $ax \mathrel{+}= M[6]$ and $cx \mathrel{+}= M[5]$
- Assume everything is in cache already

Pipelining 1

- Here is the ideal case of what happens when you pipeline



ax += M[6]	Fetch M[6]	Send ax and M[6] to ALU	Perform addition	Move result to ax	
cx += M[5]		Fetch M[5]	Send cx and M[5] to ALU	Perform addition	Move result to cx
Clock cycle	1	2	3	4	5

- If you have 2 more add instructions, you can guarantee that every single component is running an instruction at all times

Pipelining 2

- This means that you can, on average, finish 1 add instruction per clock cycle, rather than one every 5 clock cycles
- Wow, very optimised
- This is what 1 instruction per clock cycle means, because, generally, every instruction takes more than 1 clock cycle
- However, it’s sometimes difficult to take advantage of pipelining

Pipelining 3

- In essence, what a CPU is actually doing when running an add instruction is:

```
ax = # some number
M = [
  # 6-element list
]
bx = M[6]

# A copy is generated, then returned to ax
ax = sum(ax, bx)
```

Pipelining 4

- You can't pipeline $ax \mathrel{+}= 5;\ ax \mathrel{+}= 2;\ ax \mathrel{+}= 3$
- Why is that?
- Because every successive addition depends on the current value of ax
- If you try to pipeline it, you'll get the wrong answer, because ax is copied when you pass it into `sum`
- You really have no choice but to wait for the addition to complete before you can go on to the next instruction

Pipelining 5

- There are many ways to fix this problem
- The main way is to break the dependency chain by using different registers
- You can add the list [1, 2, 3, 4] by performing $1 + 2$, and $3 + 4$, then doing the $3 + 7$ left after the additions
- This means that you can store the result in ax and cx, which completely eliminates the dependency chain

Pipelining 6

- Pipelining is an example of **instruction-level parallelism**.
- Note that fetching and decoding are also parts of the pipeline. I’ve just not included them in the ideal pipeline because making a 4-step pipeline is already difficult enough
- x86_64 CPUs use a *31-step* pipeline
- RISC CPUs use a 5-step pipeline

RISC vs CISC

- In computers, there are different architectures
- The 2 main players are the x86_64 and the ARM architecture
- The x86_64 architecture inherits from the Intel 8086 and 8088 and uses the complex instruction set
- x86_64-based computers are called complex instruction set computers, abbr. CISC
- The ARM architecture comes from the Acorn BBC Micro and uses the complex instruction set
- ARM-based computers are called reduced instruction set computers, abbr. RISC

RISC vs CISC

- CISC’s design philosophy is to create a large instruction set that contains many micro-operations within each operation, that can each be optimised
- CISC’s design is also based on the limitation of data storage, as was the case when it was invented
- RISC’s design philosophy is to create a smaller instruction set that reduces power usage and hardware requirements
- This is why microprocessors like the 6502 use the RIS
- In fact, the 6502 is the first RISC

Pipelining 7

- What does that have to do with pipelining?
- It explains why Intel’s CPUs have such long pipelines
- By splitting the micro-operations required to use the CIS into smaller steps, it is possible to use simpler circuits, and hence, run the clock at a higher rate

Pipelining 8

- To maximise the benefits of pipelining, the CU will store the instructions in a buffer, then dump whatever instruction it can fit wherever
- Instructions may not execute immediately when available
- It may not even execute in order!
- The CU considers dependency chains automatically

Pipelining 9

- As a result of pipelining, we can get a few cheeky optimisations in with arithmetic operations
- In assembly, there are 2 instructions that can perform addition, multiplication, and subtractions
- `add ax, bx` performs $ax += bx$, as you may expect
- On the other hand, you could also use
- `lea ax, [ax + bx]`
- `lea` is the mnemonic for the operation ‘Load Effective Address’

Effective Address Calculations

- Effective address calculations are done by the Address Generation Unit, which is in the CU of the CPU
- It basically has an ALU that can only perform addition, subtraction, and multiplication
- `lea ax, [ax + bx]` basically performs $ax + bx$, then stores the result into ax
- Thus, it's equivalent to $ax \mathrel{+}= bx$

Pipelining 10

- Effective address calculations must be done before you can access memory
- The addition in `lea` is always performed at the *start* of the pipeline
- `add` is always performed in the middle or the end of the pipeline
- This means that by using `lea` and `add`, you can abuse the AGU to cram in more maths
- Deciding between `lea` and `add` is not done by the CU, but by the programmer
- Just shows how much performance you can force out of any piece of hardware if you think a bit more

Pipelining

- As you can imagine, the optimisations that the CU can handle are honestly quite simple
- Not much space to fit a whole lot of logic on the CU
- It has much more stuff to manage, so let’s move on.

Execute 1

- Pipelining basically makes everything faster by splitting things into smaller steps
- That way, every component is occupied at all times
- What if we want to make every component work harder?
- This comes to fruition in what we call Same Instruction Multiple Data, or SIMD
- Alternatively, this is also called Vectorisation

SIMD

- As implied by its name, SIMD means that you perform one single instruction across multiple datasets
- For example, you want to go through a file, and add 1 to every byte stored within it
- Instead of going through each byte and adding 1 to it, why don’t you just get every single byte within a file, and add 1 to every single byte?
- This is the idea behind SIMD

SIMD 1

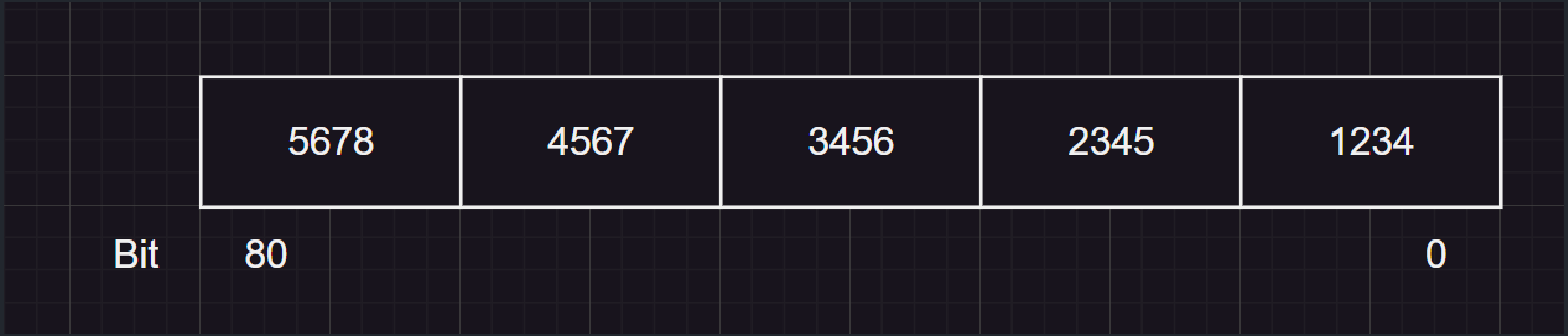
- How does Intel implement SIMD?
- They initially implemented a subset of SIMD named SWAR (SIMD Within A Register) through the MMX instruction set
- Idk why they decided on MMX.
- Must be their shitty name disease coming up again.
- Intel themselves said it’s a meaningless initialism.
- Fuck you, Intel.
- Please use your brain when naming stuff.

SIMD 21

- The MMX instruction set defined SIMD registers MM0 to MM7
- These were just aliases for the x87 Floating-Point registers, registers optimised for doing floating point calculations
- More on that later
- There would be a way to swap between using the x87 registers as Floating-Point registers and MMX registers, but I’m lazy to research that.

SIMD 321

- The MMX registers are 80 bits long
- This means that we can fit (Or pack) 5 16-bit numbers into the register



- So, we can basically call an instruction that has a special circuit to perform an addition on each of the 5 16-bit numbers at the same time

SIMD 4321

- By doing this, we’ve basically done 5 additions in 1 go!
- Because SWAR is dependent on packing data into registers, it is also called ‘Packed SIMD’
- All SIMD operations are also called ‘Vector operations’, because this is just like adding 2 vectors

$$\begin{bmatrix}1234\\2345\\3456\\4567\\5678\end{bmatrix}+\begin{bmatrix}1\\1\\1\\1\\1\end{bmatrix}=\begin{bmatrix}1235\\2346\\3457\\4568\\5679\end{bmatrix}$$

SIMD 54321

- Intel then decided to improve MMX by introducing the SSE instruction set
- This instruction set used special registers called the XMM registers
- These registers are 128 bits long
- They could now also work with floating points
- Yay.
- Because these registers are big as hell and need to store their own state, they’re opt-in

SIMD 654321

- SIMD is practically similar to normal instructions
- It also experiences dependency chains and the normal problems, so pipeline optimisation is also something important here
- Unfortunately, there’s no SIMD equivalent of `lea`

SIMD 7654321

- SIMD is an example of **data-level parallelism**
- The current generation of SWAR that is being adopted is called Advanced Vector Extensions, or AVE
- Just kidding.
- It’s abbreviated AVX
- Imagine being consistent in naming things.
- Intel could never.

Execute 2

- As stated before, SIMD makes computers faster by making each component work harder
- But what if you want to brute-force your way through?
- More computers means more speed
- So surprising.

Parallelism

- Parallelism is usually introduced to programmers through the idea of ‘threads’
- Threads basically allow you to execute multiple programs at once
- There are usually 2 types of threads
- Hardware threads exist on the CPU level
- Virtual or ‘Green’ threads are imitations of threads done by virtual machines, such as the Java interpreter
- However, none of these are so-called ‘True Parallelism’

Parallelism |

- I’m calling this ‘True Parallelism’ because this is the only way your PC can execute different instructions at the exact same time.
- In contrast to asynchronicity, which gives the illusion of parallelism
- Henceforth, ‘True Parallelism’ will be referred to as ‘Parallelism’, whilst ‘Asynchronicity’ will be referred to as ‘Asynchronicity’
- Wow, no shit, sherlock.

Parallelism ||

- When talking about fetching, we briefly touched on the idea of CPU cores
- This is where parallelism actually comes into play
- Your CU can check through every instruction, and assign each core to do something different
- Naturally, there are problems with this
- The first issue is with something called race conditions

Parallelism |||

- But wait, aren’t pulse generators supposed to have solved those?
- Nope
- Pulse generators aren’t a solution, they’re the prerequisite to begin working on a solution
- Guess what?
- The CU can’t fix this at all
- This question alone got me like -80 reputation on stackoverflow.
- Such sacrifice

Parallelism ||||

- Let’s not focus on what the CU can’t do, and instead look at what it can do.
- To figure out what this is, we should take a closer look at how the CU exploits caches

Cache L4

- The CU mainly chooses what to cache in 2 ways
- The first is spatial locality
- We’ve touched on this briefly when talking about instruction decoding
- If data tends to be accessed sequentially and predictably, the CU will prioritise caching data that it predicts will next be accessed

Cache L5

- The second way is through temporal locality
- If the CU accesses a piece of memory over and over again, the CPU will predict that this piece of information will be accessed again in the future
- Usually, even after the access is completed, the data remains in the cache until it is overwritten

Parallelism ||||

- Now, let’s say you access memory address 0x1f1e3300 containing 0xa from memory over and over on Core 0
- Core 0 would move 0x1f1e3300 to L1 or L2 cache, which is not shared between the cores
- Now, Core 1 accesses 0x1f1e3300 and updates its value from 0xa to 0x0
- Now, there’s a disconnect between what cores 0 and 1 think 0x1f1e3300 is
- In other words, our cache is **incoherent**

Parallelism |||||

- How is this fixed?
- You ask me, who I ask?
- Well, the answer here is user chop zero (Not his username) on Discord.
- He’s written his own OS and built his own CPU if I recall correctly?
- Idk, anyways, he’s been a great help.

Parallelism ||||||

- The first method is called ‘Bus Snooping’
- Basically, there’s a component called a ‘snooper’ that monitors your buses
- If there’s something that may threaten cache coherence, it mirrors the change to every other cache containing that data
- Wow, such unexpected.
- Amazing.

Parallelism ||||||

- The less obvious method is called ‘Directory-based Cache Coherence’
- I sincerely have no clue what this is.
- Basically, it tells you which blocks of memory is shared by other cores, and which other cores are using that data
- I don’t know what’s so different with it, compared to snooping.
- It’s just different, I guess.

Parallelism ||||||||

- The high-level solution to race conditions are called atomic operations, and mutexes
- Just giving you some search terms if you’re curious.
- Parallelism is an example of **parallelism**
- Shocker.

Asynchronicity

- Asynchronicity is the illusion of parallelism by switching tasks effectively
- It’s also called multithreaded-ness
- It comes from each CPU core’s ability to do multiple threads of execution concurrently

Asynchronicity 1

- Remember pipelining?
- That’s coming back.
- Isn’t pipelining an example of Instruction Level Parallelism, though?
- Yep.
- But that’s not the part of pipelining that matters
- Let’s look at what happens when your CPU has to wait for the pipeline to complete before feeding the next instruction in

Asynchronicity 2

- When there's a dependency chain or something, the CU will insert a no-op into the pipeline to wait on the dependency
- For example, if we're performing the series of instructions:

$$ax \mathrel{*}= bx$$
$$ax \mathrel{-}= cx$$

- It is not possible to execute $ax \mathrel{-}= cx$ until $ax \mathrel{*}= bx$ has passed through the pipeline
- The delay between the instructions' executions is called a **stall**.

Asynchronicity 3

- Asynchronicity prevents these stalls by storing instructions in a buffer, and stuffing them into the pipeline whenever they’re available
- Of course, if there are no dependency chains
- Instructions dependant on others will be guaranteed to execute in the correct order
- Otherwise, there are no rules on their order of execution

Asynchronicity 4

- In essence, the CU is monitoring the pipeline, and checking every clock cycle if they can cram another instruction into it
- If they can, congrats, you’ve earned a speed up
- If not, you don’t really lose anything anyways
- How fun.

Asynchronicity 5

- The final benefit to asynchronicity is that other programs can still benefit from the data you cached, even if it’s not accessed

Summary of Parallelism

- Instruction-level parallelism increases throughput by breaking instructions up so you can execute different parts of the instruction in parallel
- Data-level parallelism increases throughput by parallelising the way you use and alter data
- True parallelism increases throughput by getting different processors to work on a task at the same time
- Concurrency does not increase maximum throughput, but only utilises the pipeline and caches more efficiently

Further Optimisations

- Out-of-order execution is the process where the CU executes instructions that are likely to happen, whilst waiting for the result of a comparison or a data fetch to come out
- Analogy.
- Let’s say you applied to an IP and an O level school after PSLE
- You have a T-score 1 greater than the IP school’s COP, so you’ll be more likely to get into the IP school than the O level school
- What do your parents make you do before you get your posting results? ?
- They make you study for your A levels

Further Optimisations 0

- Insert post primary trauma here
- Because of this, if you get into the IP school, you have a head start, thus reducing execution time
- Such a nice CPU you are.
- Anyways, same concept
- The CPU tries to execute code that happens after the likelier condition first
- If it succeeds, the CPU performs a commit, and everything has already been completed

Further Optimisations 1

- However, if it fails, the CPU performs a flush operations, and rolls everything back
- One notable thing is that **the cache is not rolled back**
- This was how the Spectre vulnerability came about
- Todo explain spectre

Further Optimisations 2

- You’ve probably figured it out, but another way we can optimise the CPU is to abstract parts of it into individual components specialised for this task
- For example, in the modern CPU, there is usually something called an FPU, or Floating-Point Unit
- This is, as you may have guessed, specialised in computing floating point operations
- This helped drive up the number of operations that can be performed, since most operations nowadays use floats somewhere