**2025-03-06-cpu1-en**

0:00  
Of this stuff.

0:00  
So we are shortening our delay at the beginning.

0:04  
Today we will switch the topic completely and we will start a new topic about CPUs.

0:12  
Basically, we are trying to get a bridge between your conceptions or maybe misconceptions about computers and stuff, and try to bridge this gap all the way down to the CPU.

0:29  
However, we will start from the CPU, but rest assured that there are many other pieces that needs to fit together, like the compiler.

0:38  
So if we start from the conception, let's just say I use as an example a very simple situation.

0:46  
You'd like to upload a meme on the Discord, right?

0:49  
That stuff you usually do, at least in your spare time I hope, so that you have the image and you want to just drag and drop it via browser, for instance.

1:00  
So you have the browser.

1:03  
But beneath this very simple operation, beneath this very simple task, there is a lot of code.

1:10  
Actually the situation is far more complex than this because yeah, there is the browser, which is hundreds of thousands, maybe millions of lines code wide if you include all the plug insurance and additional software that lies beneath the libraries, etcetera.

1:26  
But there is also some additional code like code in JavaScript that runs inside the browser.

1:31  
So the situation is far more complicated than that.

1:34  
But for the moment, let's just simplify it so your intent in the basics can be expressed using some high level programming language like COC Sharp for instance.

1:47  
However, this intent is still far, far from what the CPU can actually process, because here this code is almost readable too, even though you don't know the language.

1:59  
Even though I haven't told you which language this is, you will probably understand it at least conceptually, right?

2:05  
Because I'm opening an HTTP connection to the server, I'm posting a HTTP request.

2:12  
I'm receiving some some response.

2:14  
So basically, you can you can write this down.

2:17  
If you understand computer networking, then you could probably come up with something like this on your own.

2:22  
And if you know what library I'm using, you would probably be able to write this code on your own.

2:29  
However, as I said, this is far cry from CPU instructions.

2:34  
So this needs to be processed and there are several steps of processing I have just simplified.

2:39  
The compiler basically turns this high level programming language into a very low level representation of instructions in assembly language.

2:48  
This is basically the code that can be understood by the CPU almost.

2:54  
There are still some placeholders, like registers for instance, which are still understandable at least a bit by humans.

3:03  
But then you need to be translated for the CPU.

3:06  
But this is very close to what CPU can perceive.

3:10  
Actually, this is just another representation, textual representation of what CPU can actually process.

3:16  
And this is later compiled into the binary code so that each instruction here, each line of this code is interpreted as a sequence of bytes or bits in the final binary file.

3:28  
And this can be directly interpreted by CPU.

3:31  
This can be directly processed by CPU.

3:33  
So next several languages will be mostly about getting this bridge between some high level C code and low level execution.

3:45  
And as I said, we will start from the bottom from the CPUs.

3:48  
So basically CPU is a machine that performs some task.

3:53  
Before we get to that, just a quick revision.

3:56  
You probably heard about some CPU architectures.

3:58  
So just let me revise this very quickly.

4:01  
So the most common model or architecture model of today is the for Neumann's architecture, basically for Neumann, actually this is not the old for Neumann.

4:12  
There are some.

4:12  
There's been some revisions over the time because if you take old von Neumann literally, it's very old.

4:19  
He just operates in the terms of initiating some IO operations.

4:25  
That's not exactly what happens in more realistic CPU scenarios.

4:30  
So basically we have 3 elements connect interconnected together.

4:34  
The CPU, the core which actually executes the code, the memory which is used both for the code and for data.

4:43  
That's the most important fact about phenomenal architecture, at least when you compare it to other architectures.

4:50  
So both the code and the data are in the same memory and also there need to be some other peripheral devices like the keyboard or the monitor etc.

5:02  
So the other devices is also interconnected via one single system bus.

5:10  
So basically every time any of these two components need to communicate, they will use this bus.

5:16  
And that's the, let's say, the greatest benefit and the greatest drawback of this architecture.

5:21  
The benefit is it's very simple.

5:23  
You have only one bus, 1 synchronization to take care of, one place where all the data and all the controls are taking are transferred.

5:32  
However, this is also the bottleneck, because if you want that, if you want multiple devices, multiple parts here to communicate simultaneously, they can't.

5:43  
Well, normally when there's only one CPU core, why would we do that?

5:49  
It's like not really easily conceived.

5:52  
So normally CPU sends some data to the memory or asks for some data from the memory.

5:58  
So usually the CPU is talking to the memory one way or the other.

6:02  
Sometimes the CPU communicates with the devices.

6:04  
And actually, if this mechanism is designed well, the communication with devices is the same as the communication with the memory.

6:13  
We are talking about memory map devices.

6:15  
We get to that later when we talk about operating systems.

6:18  
So the communication with the devices can be, from the perspective of the code, very much same as the communication with the memory, just loading or storing some data via this universal bus.

6:28  
So we are just using different addresses for real memory and different addresses for devices.

6:33  
That's the only distinction we need to do.

6:36  
However, there are situations where memory and input output devices can communicate directly without the interference from the CPU.

6:44  
Let's just take DMA controlled devices into account.

6:49  
These devices are very special devices that can communicate or transfer data directly to or from the memory.

6:56  
So basically they can just send the data over the bus or read the data over the bus without any actual interference from the CPU.

7:04  
So the CPU can perform other tasks.

7:06  
That's the benefit of DMA.

7:08  
However, in this scenario there won't be much of A benefit at all because the bus will be blocked during the DMA.

7:19  
So actually the CPU won't be even able to load the next instruction because remember the instruction is in the memory.

7:25  
So if the bus is occupied then the CPU is stalled anyway because it cannot communicate via memory.

7:32  
And the second issue with this architecture it it doesn't scale well.

7:35  
You've probably heard that there nowadays we have a multi core CPU.

7:39  
So this unit here is actually replicated several times.

7:44  
Even if your cheapest laptops or even if your cheapest cell phones, you have usually at least two or four CPU cores.

7:51  
Even the very low end devices have that.

7:54  
So basically you have this replicated several times.

7:57  
And in servers you can have 10s or even small hundreds of CPU cores.

8:03  
So if you imagine that all these CPU cores are communicating via single bus, that would that would be a very, very bad button like that.

8:10  
They they wouldn't, it wouldn't scale well.

8:13  
So what we need actually is to decompose this bus, this bus further.

8:19  
Yeah.

8:20  
One important thing I I haven't mentioned yet is the fact that the memory holds both the data and the code, as I mentioned at at the beginning is quite important for several reasons.

8:31  
One of the reasons is that we need to we as a programmer, as low level programmers, need to make some effort to distinguish between code and the data, because having the code and the data mingled together in the same memory may be potentially disastrous.

8:47  
Well actually in the past it was used for beneficial purposes like if we had some special CPU architecture that was not capable of performing advanced tasks like not really advanced tasks, A slightly more advanced task like indirect addressing.

9:03  
If you need to perform indirect fetch in the memory, so you have a index stored as the data and you need to use this index to fetch the data, then one way how to do it on a very poorly designed architecture would be to rewrite your own code.

9:19  
You basically access your own code and put some stuff in there so the code behave slightly differently.

9:25  
For instance, you put a new constant in the code.

9:28  
So yeah, there were like dirty tricks that can be used in this in this manner, but that's not what what we what we are using today because actually it's potentially dangerous.

9:38  
Just give let me give you a very quick example.

9:43  
There is a actually vulnerability which is called the buffer overrun attack.

9:49  
This is a situation which can occur if you have a poorly designed code.

9:55  
Let me give you an example.

9:56  
You have a code where the data are laid in the memory.

9:59  
Thusly, you have a buffer of some values, and into this buffer user input is loaded somehow.

10:10  
It's not like from the network or from the keyboard or something.

10:14  
And in the memory, right after this buffer there is a, let's say return address from a function call, for instance.

10:23  
OK, so if you somehow smuggle in data which are larger than this buffer, and the code is poorly written so it doesn't properly check whether the input fits into this buffer, you can easily overwrite overwrite slightly more memory than the programmer intended.

10:43  
So you basically overwrite this address as well.

10:48  
And if you so happen to guess where this buffer lies in the memory, you might somehow smuggle in an address that would jump back into this buffer.

11:00  
And if you somehow smuggle in instructions instead of data, then these instructions could get executed and you just smuggle your own code into someone else's code, which might be potentially disastrous.

11:14  
If you don't really understand how this works, don't worry about it.

11:18  
I'm just telling you it's possible if you if you're not careful enough.

11:22  
So.

11:22  
So this is just one way how to attack poorly designed applications.

11:27  
By the way, it's not so far fetched.

11:29  
You probably heard of about the heartbreak Buck.

11:34  
It's not that old, it's like 5-7 years old.

11:38  
It was a big issue in the Internet community.

11:41  
All the servers needs to be patched because there was this this kind of bug in the one of the most profound libraries open SSL, which is basically used for encryption all over the world in in many servers.

11:53  
So yeah, if you find out buck of this nature in a such such essential piece of software, it might be very tricky.

12:01  
OK, let's get back to this, to this issue.

12:04  
So technically there are mechanisms in the CPU that try to distinguish whether some part of the memory is code and should be executed and some part of the memory is just the data and should be read and write, read and write, but not executed.

12:20  
OK, just to compare this, there are also other architectures.

12:23  
The most famous next to kin is the hard work hard work architecture and the difference is that the instructions the program is in a separate memory.

12:36  
If you don't know any architecture like this, just think about Arduino's.

12:40  
Arduino basically uses this architecture because your code in a separate E from actually flash memory.

12:46  
Then the registers where the data resides.

12:48  
So it has separate 2 memories, one for code and one for data.

12:53  
But besides that it has the same problems.

12:57  
So we cannot have just a single bus.

13:00  
By the way, this, this is just not it's, it's not only for the Arduinos, most of the microcontrollers of the day, which are used in your washing machines and, and stuff like that still uses this architecture.

13:13  
OK, so give me more realistic example of, let's say, more recent architecture.

13:18  
OK, yes, Sandy Bridge isn't exactly new.

13:21  
It's, it's more than 10 years old.

13:22  
I think it's like 15 years old.

13:23  
But it illustrates what I want to tell you very well.

13:27  
So computers, the PC systems of today are decomposed so that they have, they have multiple chips.

13:35  
And these chips usually communicate via peer-to-peer buses.

13:38  
There is no single bus, There are buses between each two components and only these two components uses these buses.

13:46  
So there is no congestion.

13:48  
Well, actually there can be congestion if they communicate too fast, but it's not like that one device is blocking another device.

13:56  
Any 2 devices are are using their own separate peer-to-peer bus and the whole system is usually laid out thusly.

14:07  
The main part is the CPU.

14:09  
Actually this this box is everything that's that can be found in the CPU.

14:15  
And now I mean the CPU die, the chip that that's inserted in your motherboard, besides the cores and caches which are obvious it, it also holds a memory controller, so it has its own device that can communicate with the memory.

14:31  
So the main memory is connected directly via unique separated buses to the CPU.

14:39  
And actually the memory controller can be replicated there several times.

14:43  
Usually even in in the cheapest CPUs, you have at least 2 memory controllers.

14:48  
So we have like 2 memory buses and each is for a separate memory channel.

14:53  
By the way, you probably know that you have even in your laptops, you usually have at least 2 memory slots.

15:00  
This is not just that that the vendors can charge you extra, it's there's a practical reason for that.

15:06  
And the reason is that if you have two memory chips inserted, each one can be communicated with separately using a separate bus, which is much faster.

15:17  
You have like you have doubled the access, the throughput, not the access time because the like latency remains the same, but you have doubled the throughput.

15:25  
So in any case, if you're building your own computer or upgrading your own computer, it's a good idea to occupy both slots.

15:33  
It's it's a good idea to have two memory chips on your on your motherboard.

15:36  
Otherwise you're like wasting unnecessarily your memory throughput.

15:42  
And besides the memory bus, which is nowadays like the most, the fastest external bus there is, because you know, this is where the bottleneck is.

15:52  
Because if you have a multiple cores that are running in several gigahertz and all of them want to crunch some numbers, want to crunch some data, then they need to be fed this data.

16:05  
And the latency of of these channels is quite high.

16:09  
It's like 1000 or sometimes thousands of cycles.

16:12  
So you really need to feed the data quickly.

16:15  
So these buses are like the widest in the terms of throughput.

16:19  
The next most important bus is the PCI Express bus, which is today used especially for graphical cards.

16:27  
So if you're gaming fans or you need to do some heavy computations, Super Bowl have your own dedicated GPU.

16:33  
Technically, sometimes like office laptops or office desktops use integrated GPUs on on the chip with the CPU, but that's a different story.

16:45  
Let's just assume that you are gamer and you need to you need a really powerful GPU then this this GPU is connected via PCI Express and that's the second fastest bus external bus there is.

16:59  
Besides that, there is another link.

17:02  
By the way, this a greenish box is another chip.

17:07  
This is really another piece of hardware, not another piece of silicon.

17:11  
By the way, it's called S bridge.

17:13  
If you're wondering where the north bridge is, the north bridge used to be between the South bridge and the CPU and it contained the system agent, which is used for the communication and the memory controller and of course the PCI Express.

17:31  
Usually it was back then it was AGP, but that that's not that important.

17:35  
So it, it holds the more the fastest buses.

17:39  
But in the past, it was discovered that the north bridge, the communication between the north bridge and the, the CPU itself is too slow.

17:47  
So to fasten things up to to speed the things up, they integrated all the components from the north bridge into the CPU die.

17:54  
And thanks to the let's say minimization process as we shrinking the transistors in the CPUs.

18:01  
Nowadays, there's no problem because the memory controller and the system agents are relatively small when you just express them as a how much silicone, how much part of the wafer you need to allocate for them.

18:14  
So it's no problem to put them directly into the into the CPU.

18:19  
So basically now we don't say Southbridge, we just say chipset.

18:22  
That's the assisting chip to the CPU.

18:25  
And this chipset holds every other communication.

18:27  
So it's like a secondary communication device.

18:30  
That's why it's called a bridge, because it's like it bridges communication between the CPU which has like only one universal type of communication bus.

18:42  
Usually today CPUs like Intel use Quick Path Interconnect UPI and it handles all the intricacies of all other types of devices.

18:53  
So special connection with the audio codec which uses which is used to convert the digital and analog signals.

19:01  
So you can hear some sound from your computer or you can record your microphone.

19:07  
There is also BIOS.

19:08  
So there is some extra memory which is attached to it, which can be read and accessed by CPU.

19:13  
This is for instance, where the main boot loader of the of the of the of the PC is.

19:19  
So this is the basic code which is started when the PC is booting up, when the CPU is turned on other devices like floppy Dr.?

19:29  
Yeah, that's, this is very old.

19:31  
Anybody know still what floppy Dr.

19:33  
is?

19:33  
Just wondering, anybody heard about floppy?

19:35  
Yeah, a few people who did so yeah, it's, it's still, I think it, it's because the floppy Dr.

19:41  
is still used as the icon of saving.

19:43  
All right.

19:43  
It's like, Jesus, they died so they can become an icon of saving.

19:47  
So floppy Dr.

19:48  
is out of the question nowadays.

19:50  
But we still have things like USB.

19:53  
Ah, here it is.

19:54  
So USB buses that that are still very valid, yeah.

19:58  
Now nowadays we have USB 3.1 or so, but devices connected through USB are also here.

20:04  
And of course, perhaps the most important, maybe something that I should should have started with is the drives interconnect.

20:13  
So today we are using SATA.

20:15  
In the past we are using other other communication buses like Pata, Parallel, ATA.

20:21  
And finally, there may be additional PCI Express links.

20:25  
The PCI Express 16 actually is a congestion or concatenation.

20:31  
No, no, no.

20:32  
What would be grouping of 16 PCI Express links, But you can use them separately.

20:37  
For instance, you can use only single PCI Express link, which would be which would have enough capacity to for instance interconnect with the Ethernet adapter, with the networking adapter or with Wi-Fi for instance.

20:51  
If you want more, there are PCI Express for devices like external faster memory storage like if you want if the SATA isn't fast enough for your hard drive.

21:01  
If you use some very special hard drive that's really fast nowadays, you can use PCI Express 4 which aggregates 4 PCI Express links together to communicate with the with the hard drive.

21:14  
Funny observation, most of these buses, USB PCI Express, yeah, this doesn't know the buses that interconnect the audio connect and all other buses that really needs to connect two devices for a longer distance are serial.

21:33  
Serial, which means they usually contain only two or three wires.

21:38  
Yeah, that's, that's kind of a curious because you would say, OK, if we need to boost the throughput of a bus, we can use more lines, right?

21:47  
Like over here, PCI Express 16 uses 16 times 2 lines.

21:54  
So basically why we are not doing that in all other devices, for all other devices?

22:00  
Well, the reason is that designing a parallel bus is much more complicated, much more sophisticated.

22:08  
The problem is time.

22:10  
Just imagine that you are drawing a motherboard.

22:18  
I mean, you are designing a motherboard and here we have your CPU and here we have another device that needs to be communicated with and you choose to use a parallel bus instead of a sequential bus, a serial bus, Sorry.

22:32  
So you need to place lots of tiny wires, one next to each other, right?

22:42  
The trouble is that this wire, this this piece of metal is much shorter than this one.

22:50  
Yeah, it might be just a few centimeters, but remember that the electrical signals travel almost at the speed of light.

22:58  
I'm not sure if it's like a two third or or eight 9/10 or something like that, but it's very comparable to the speed of light.

23:06  
So imagine how much time the speed the light travels if you if you're sending signals in the at the frequency of gigahertz, a few centimeters.

23:19  
So basically you are in trouble because you need additional synchronization circuitry that will deal with this, with this additional problem that there is a, this signal travels quite longer than this signal.

23:36  
Inside the CPU there are many buses that are still parallel.

23:41  
So inside the chip, inside this chip there might be many buses that are still parallel because inside the silicon it's much easier to design them in the way so that they are similarly long.

23:55  
Maybe if you if you take a very close look to your motherboard and you see some parallel buses there, some of them use intentional extension so the wire isn't straight.

24:10  
But it looks like this sometimes if you take a very close look to your motherboard and that the reason for that is they're trying to lengthen one path.

24:21  
So it's it has the if it's like this path, it can be lengthened by by creating small curves in it.

24:29  
So it's similar.

24:30  
Its length would be similar to the longest path.

24:35  
OK, so this is how contemporary, well contemporary from the, let's say very old perspective, but the PCs of of today have very similar design.

24:47  
Yeah, we have a new new version of USB and we're probably using faster lens but and better, better hard drives.

24:53  
But basically the concept remains the same.

24:57  
OK Today we will focus only solely on the CPU.

25:02  
So what is the CPU?

25:03  
First, let me just clarify a few terminological terms.

25:07  
So I've used the word architecture several times, so let me now start using it properly.

25:13  
A CPU architecture.

25:15  
Actually it means two things.

25:17  
I can mean the instruction set architecture, ISA.

25:21  
And this is like the specification, this is like a manual or a contract that says how the how the CPU will be will appear to the programmers, which instructions will be will be there, which registers will be there.

25:36  
So how can I program this this device?

25:41  
Knowing the ISA is what what is what is important for us, for us programmers.

25:47  
Because if you want to write your own compiler, if you want to write your own operating system, then you really need to know the ISA of the architecture.

25:54  
And if you're writing a multi architectural OS like Linux, you really need to make modifications in your kernel so it can be adapted for different architectures.

26:05  
So Linux kernel isn't universal in the terms that you can take the same kernel, same instance and drop it from X86 to ARM, for instance.

26:14  
There are different parts for different architectures, they need to be adjusted, they need to be rewritten specifically for specific architectures.

26:22  
But the specification of the ISA is the only thing we need to know to do that.

26:27  
We don't need to optimize it for specific version of the Intel, for specific generation of the Intel CPU.

26:33  
So you don't have to use very a very particular version of any ARM CPUs, we can just know the universal specification, the ISA.

26:45  
On the other hand, the hardware architecture is the hardware implementation, how the ISA is transferred into the circuitry, if you wish.

26:54  
So hardware architectures to compare ISA could be something like IA 32, which you probably know more as an X86 architecture.

27:03  
This is the most common architecture of today, probably in most of your laptops.

27:08  
I'm not sure if I'm seeing any apples here.

27:10  
HP HP so probably If you have Apple then probably not Apple.

27:14  
Sorry.

27:16  
If you have Apple that's different architecture.

27:18  
But everyone besides Apple who uses regular PC uses X86 architecture.

27:23  
ARM a 64, Risk 5.

27:27  
So these are the ISA specifications.

27:30  
If you want to talk about hardware architectures, we can say something like Intel.

27:35  
Raptor Lake is the last one.

27:37  
I'm not sure.

27:38  
Raptor Cove, they have this naming.

27:40  
I'm not sure who invented that, but they have the lake and the Cove and one of them is the architecture of the CPU and the other one is the chipset or vice versa.

27:49  
I'm not sure, it's very confusing, but or AMD's and five, that's pretty straightforward.

27:54  
So this is like a specific implementation of a specific vendor that conforms to given ISA.

28:02  
So let me down, let me get down to the basics.

28:04  
What is CPU?

28:05  
So basically CPU is a simple machine.

28:08  
Yeah, when I'm simple and I'm addressing something that that holds billions of transistors, that might be confusing.

28:14  
But by simple I mean it's capable of executing only a very simple commands.

28:20  
These commands, as you know, are are called instructions.

28:23  
And one instructions is something very fundamental like a basic arithmetic operations, like an addition for instance.

28:30  
And of course CPU must hold additional circuitry that assists with this task.

28:35  
Like it has its own memory called registers.

28:38  
This is the memory which is closest to the CPU where intermediate results are hold.

28:42  
It has memory controller so it can communicate with external memory and it has some way how to express IO.

28:49  
So it has some interface that can connect it to the South bridge and through the South bridge to all other peripheral devices.

28:57  
Let me give you an example of an instruction.

29:00  
For many most probably examples, we will be using the MIPS architecture.

29:06  
Just to explain, MIPS 32 is an architecture which is kind of old.

29:10  
You can still find it in devices like rotors for instance, or maybe other, let's say not PC devices, but embedded devices, etcetera.

29:23  
However, it's not that it's not like we are teaching you something completely useless.

29:28  
The MIPS 32 was used as a basis for a RISC 5 architecture.

29:32  
Perhaps you heard of it.

29:34  
RISC 5 is like the modern architecture, which shows a lot of promise and some people say it might replace X86 in a foreseeable future.

29:44  
So it's not a completely waste of your time.

29:46  
It's, it's like a very interesting architecture and basically the differences between MIPS and RISC 5 are beneath us at at this moment.

29:54  
There are some differences, but they are not important for the for the purposes of this class.

29:58  
So if you just learn the MIPS, then you can very quickly learn or extend it to to risk 5.

30:04  
So as an example, I take this instruction which is called ADDI at intermediate if you will.

30:11  
And this is how this instruction is represented in memory.

30:15  
So this is the bridge between this is the last step of this of this gap which we are bridging between the semantics and the binary code.

30:23  
So in the binary in the memory, the the processor will see a 16 bit, 32 bit word, sorry, it's called MIPS 32.

30:31  
So it uses 32 bit words as the as the as a basic width of the word.

30:37  
And each instruction in this architecture has 32 bits.

30:43  
That's one of the reasons why we chose this architecture, because it's very simple to find out where an instruction starts and ends.

30:51  
If you take a look at align 32 bits at 4 bytes, anywhere in the memory which holds code, it will hold an instruction.

30:59  
If you take a look at X86 code, somewhere in the middle, you don't know where to start reading because X86, for instance, I I believe Mr.

31:11  
ASIC told you about X86 in the previous classes.

31:14  
Is that so?

31:15  
Does he still teach X86 in the winter term?

31:19  
Yeah.

31:19  
So you probably notice that in X86 architecture, the width of the instructions varies a lot.

31:26  
The shortest instructions in X86 takes 1 byte and the longest takes I think 8 bytes, or 13 bytes if you count in the prefixes.

31:35  
So somewhere in between there may be length of your instruction and you couldn't know until you read enough of the memory.

31:42  
So usually you read the first byte, then you read a few more bytes, and then you can read a few more bytes to find out the final instruction.

31:50  
So it's very difficult to decode in compared to this.

31:53  
Furthermore, the instruction itself has a fixed format, so if we chop off the 1st 6 bits, it always holds the OP code.

32:02  
So the instruction, what the instruction is, is stored is encoded in the 1st 6 bits.

32:08  
Just by looking at it, I can immediately tell, ah yeah, it's it's the ATI instruction.

32:13  
Everybody knows that because it's 001000, right?

32:17  
Remember that it will be on the test.

32:19  
I'm just kidding.

32:20  
So this is the prefix of, this is the OP code of the instruction.

32:25  
Next 5 bits and next 5 bits are the addresses.

32:30  
Addresses of what?

32:32  
Because 5 bits isn't enough to address memory.

32:35  
So what?

32:35  
This address could hold registers, right?

32:40  
So basically we can we can address registers.

32:42  
So these two are registers used as operands for this instruction.

32:47  
By the way, if we have 5 bits, how many registers this architecture has?

32:54  
How many registers?

32:55  
This architecture may have guessed from this instruction 32, because we can address five of them using 32 of them.

33:04  
Using 5 bits.

33:05  
2 to the power of 5 is 32, so this can hold 32 different values.

33:11  
This can hold 32 different values, so we can use 32 different registers and the last 16 bits.

33:19  
It so happens that it's half of the work, 6 bits 5 + 5 is 16 together, so the rest is also 16 bits wide and it holds either another register value, another register address, and the rest is unused.

33:35  
Or it can hold an immediate value, so it may hold a constant.

33:39  
It's like if you write down a literal in your code.

33:43  
If you write number somewhere in your C code, then somehow this number needs to be interpreted in the code.

33:49  
So one way how to do it is that they are using intermediate values in the instructions.

33:54  
So the constant from your code is intermediate, intermediately encoded into the instruction itself.

34:02  
On the other hand, as you can see, this can hold only 16 bits, so you cannot possibly use it to fill 32 bit register at once.

34:11  
You can use it only to fill half over 32 bit register.

34:14  
So it's a small drawback.

34:17  
By the way, isas the architectures can use 1-2 or three operands, this one obviously uses 3 operands 123.

34:28  
It's possible there are architectures that use only two.

34:31  
Can you name 1 X86 architecture, for instance?

34:38  
You've learned it in the Winter term that uses only two.

34:41  
There are also architectures that use only one operon.

34:46  
Sorry, there are also architectures that use only one, but that's a bit tricky because you need some stack machine or something to to compensate for that.

34:57  
Technically you can, you can use zero, but that's not very practical.

35:00  
And technically you can use more than 4, but usually you don't need more than 4.

35:04  
Three are more than enough because three basically can express very simple operations like take one number, add it with another number and and put it in the third number.

35:13  
So what?

35:14  
What more you would like to do?

35:16  
OK, so this is about instructions.

35:23  
And let me explain how the instructions very quickly, how the instructions are processed.

35:30  
So basically there are like several steps that the CPU performs in a loop, forever and forever.

35:36  
First it loads the instructions from the address where every CPU has like a instruction pointer IP.

35:44  
Don't don't mix it up with the IP like in Internet Protocol address.

35:48  
No, no, no, it's instruction pointer.

35:50  
Yeah, sometimes it's used program counter, but the acronym for that is PC, which is also confusing.

35:56  
So pick your guess.

35:59  
I be, I be I will be using instruction pointer mostly.

36:03  
Next step, when you load the word in in MIPS it's very simple, you just load the 32 bits.

36:08  
In X8X that might be much more tricky because you need to load something in between 1 and 13 bytes.

36:14  
Then you decode the instructions.

36:16  
So you basically re represented it from the very compact format the code represents into some internal format which is specific to the hardware architecture.

36:27  
OK, so the hardware architecture have its own ways how to express instructions and the decoder basically converted from the uniform compact format that is specified by ISA into whatever internal format they want to use.

36:40  
That usually translates into direct commands to individual units in the CPU.

36:47  
Then there are three steps for most instructions, loading the operands.

36:52  
So loading the inputs, performing the operation the instruction is supposed to do, and then storing the result somewhere back.

36:58  
This usually involves loading values from the registers, and this usually also stores the value to the register.

37:04  
But some architectures like 686 oftenly use loading from memory or storing back to memory, so that might be also possible.

37:15  
And finally, the ultimate last step is incrementing of the IP.

37:19  
So the IP counter is incremented is is advanced so that in the next loop we will read the following instruction.

37:28  
In MIPS it's very easy, we just add 4 to the IP.

37:32  
That's very simple.

37:33  
In X86 or other architectures it might be somewhat more tricky because we need to find out the width of the instruction and add appropriate number of bytes to the IP counter.

37:47  
Yeah, and of course this is not like a crave into stone, because some instructions may omit some of these steps.

37:54  
For instance, if the instruction doesn't have any output, if it just performs some task, like if it's an output instruction or if it's an instruction that performs jump so it affects the IP, it doesn't have any other result.

38:08  
So there is no storing of the result for this particular instruction.

38:11  
For instance, let me give you a glimpse at how this is executed at the bottom level.

38:16  
This is very simplified design of a CPU.

38:19  
I've just picked a few parts of the CPU.

38:21  
Actually it's like a virtual CPU, not actual one.

38:24  
How it might be implemented just to give you a glimpse at how instructions are executed inside.

38:31  
So there is some unit which is called instruction decoder and usually there are some units which schedule the execution of instructions.

38:38  
I've merged them together in this particular diagram.

38:42  
And then there are registers which you can imagine as some flip flops that can hold values.

38:48  
And there are some additional computing units.

38:51  
I, I just take the adder unit because I will be demonstrating the ADD instruction, the very similar one I just told you about a few slides ago.

38:59  
So I will need the adder unit.

39:01  
And again, this is just a bunch of logical gates.

39:04  
Actually, if you have the time, you can, you can just design an adder using what you know from, from basic logic.

39:13  
It's not that that complex, but we put that aside for this moment.

39:17  
So basically what we are doing here is that we need to transfer the data from these registers to the adder and then transfer the the result back to the appropriate register.

39:27  
What we will be implementing or what I will be demonstrating is this instruction add R3R 1R2, read it from the left to right as if you as if you put it in the C notation.

39:39  
So it's like R3 equals R1 plus R2.

39:44  
OK, just to know the semantics of this of this notation and this translation.

39:49  
This equality is what the decoder does.

39:52  
It translates it into some internal representation which actually comprise multiple steps.

39:57  
This instruction isn't computed at once, at least not on this virtual CPU.

40:02  
Because we need to use the data bus several times.

40:05  
I've intentionally designed the CPU to use only a single bus.

40:09  
Just to give you a glimpse how the things work, usually there are multiple buses and they are somehow interconnected or not.

40:16  
So it's much more complicated.

40:17  
This is very simple design because all the registers, all the units, everything is connected to this bus via these circular objects which are gates and gate is like imagine just like a switch that allows or disables whether the data can come through in one or the other direction.

40:38  
So if this gate is active, it allows this register to be filled from the data with the bus which is currently on the bus.

40:46  
If this gate is on, then it will take the value of the register and it will write it on the bus.

40:54  
So one of the tricky part is that this internal representation must make sure that at any given time only one of these output gates which is writing is open.

41:05  
Otherwise there will be collision on the bus and it can it can result in some damage or some malicious behavior.

41:11  
So normally only one of the output gates should be on.

41:16  
So what?

41:16  
What happens if you want to interpret this micro code, if you will, this decoded instructions.

41:24  
So first operation is let's open GR1.

41:27  
It is a gate from register one output plus GAA, that's gateway of the header unit register A.

41:35  
So basically we are opening this gate for output and this gate for input which results into transfer of the data from this register to this local temporary register of the header OK.

41:49  
The same goes for the next step.

41:51  
We are opening gates from register 2 and input register B on the adder unit.

41:58  
So the next step would be that the data will be transferred from the register 2 to the register B of the adder unit.

42:05  
After that, the adder unit needs to be activated to perform its miracles.

42:09  
So basically the instruction decoder will poke the unit and it will just take these inputs and produce the output in its own register.

42:17  
OK, somehow, and finally we need to write this back.

42:21  
This is like there is some internal synchronization which we are just hiding from you.

42:25  
And finally, the output which is currently reside here needs to be transferred to the register 3, right?

42:31  
So this is the same mechanism.

42:32  
We are opening this gate and this gate and the data will be there.

42:37  
Voila.

42:38  
Yeah, this is very simplified view.

42:41  
Usually these steps, some of these instructions may take many more steps.

42:46  
Also, I'm visualizing only the fetching of the operands, the execution and then storing the results.

42:54  
Only these three steps.

42:55  
Now I need to load the value of the IP register into the other unit, add 4 to it and transfer the value of the IP register back to the IP register using the very same mechanism, right?

43:10  
But this is just how this works in the internals.

43:14  
It's not for you important to know this at this level.

43:17  
This is just to give you a glimpse the notion of how things are working internally.

43:22  
If you really want to know how things are working internally, you need to know a bit more about flip flops and logical gates so that you can visualize how these individual parts are actually working at the electrical level.

43:36  
OK, so let's move on.

43:38  
I said that there are some types of instructions.

43:40  
We already know add instruction or two types of add instruction actually.

43:47  
So one of the fortunes of C&C is that is very close to the actual instruction level.

43:54  
Yeah, you may have not noticed that, but it's much closer than Python or JavaScript.

43:59  
So basically, if we take a glimpse of these code fragments, what kind of instructions you think we will need to have in order to implement them in order to translate them to assembly language?

44:12  
Any guesses?

44:12  
Any hints?

44:13  
A comparison operator or something?

44:16  
Yeah, let's start from the beginning.

44:18  
So this will require some sort of comparison so that we can compare 2 numbers to registers or to some things and get the result whether it holds true or false.

44:27  
That's right.

44:28  
We need arithmetical comparison.

44:30  
Anything else, 8 jumps, right?

44:35  
So yeah, what is a jump?

44:39  
We can, after these comparisons, we control?

44:45  
Yeah, exactly.

44:47  
We need to, let's say, skip over some code.

44:50  
We need to alter the flow of the code control.

44:53  
So normally the code is executed sequentially, the instruction pointer is incremented, but we need some special instructions that can advance the instruction pointer forward or sometimes even backwards.

45:06  
So for instance here, if this condition doesn't hold, we need to have, we need to have a mechanism that will skip over this assignment instruction and continue here.

45:17  
OK, so yeah, we need some sort of jumps and actually we need two types of jumps, right?

45:25  
Do we Not necessarily.

45:29  
Really.

45:30  
OK, why not?

45:33  
I don't see why you couldn't repurpose the same jump man for both in this statement and the poor lead.

45:40  
OK, And then which type of jumps are you talking about?

45:42  
So I mean, how, how would your jump behave?

45:47  
How would your imaginary jump instruction behave?

45:49  
If you want only one type of instruction, just just I think please.

45:54  
No, you're right.

45:55  
We need a regular jump in the condition which you want to know.

45:58  
Yeah, that's what I was aiming to.

45:59  
But probably you're right, we actually don't need the regular jump.

46:02  
If we can force the conditional jump to jump as well, then we don't need the regular jump.

46:08  
But that's like a complication in the architecture that's usually too difficult to handle.

46:13  
So usually we have at least conditional jumps and and non conditional jumps.

46:18  
The non non conditional jumps jump all the time.

46:21  
Conditional jump jumps based on a previous instruction, a previous comparison instruction, or a boolean value for instance.

46:30  
OK, so we have jumps, we have comparisons.

46:32  
Anything else that can be spotted right from this code?

46:47  
Yeah, there are multiple arithmetic operations.

46:49  
Yeah, we already know the the edition which is here and here.

46:54  
We don't we haven't met this one.

46:56  
This is this is shift.

46:59  
So we also need some operator, some instructions, sorry for this for this operator.

47:04  
And yeah, we can name them all.

47:05  
So if you just happen to remember which operators binary and unary are in C, then basically you get them all as some form of instruction, almost not too fancy, but additions, subtractions, bitwise operations, shifts, they are all there as instructions.

47:23  
Yeah, that's right.

47:24  
What else?

47:27  
Assignment.

47:28  
What is assignment?

47:31  
Yeah, but technically, what in the instruction level?

47:35  
What would assignment do?

47:37  
Change the value of?

47:40  
Some sort of change the value of of the memory?

47:46  
Yeah.

47:46  
OK, so if we're talking about memory, then yeah, but I will slightly alter your statement.

47:54  
We need to handle the assignment, but we need to handle the assignment to the memory, not to the register.

48:00  
Because if we just jump back here, this edition is also an assignment.

48:06  
This performs addition of these two values and they are assigned over here.

48:12  
So if I need to assign R2 into R1, the only thing I need to do is place 0 here and I have an assignment.

48:21  
So assigning 1 value from one register to another register can be also done using arithmetical instructions.

48:28  
Technically speaking, actually it's not entirely true.

48:32  
Usually architectures will give you the assignment instruction as a copying value from one register to another register.

48:39  
Because adding zero from here to here from adding zero to this value also would activate or may activate the the addition unit.

48:49  
You don't want to waste that much time and energy.

48:51  
So basically you just want to copy the data, but technically, theoretically not.

48:57  
Technically we wouldn't need that, but we would need, as you said, transfer of a value into the memory.

49:03  
This is probably some array in the main memory.

49:07  
I haven't declared it, but it's kind of obvious this this should be some array in the global memory and I need to fill this value into the into the proper spot in the array.

49:18  
So I need a way how to put the data into the memory and vice versa, right?

49:25  
You can see that probably because, yeah, B&A, they, they might be in registers, but they have to end in the registers somehow in the beginning.

49:33  
So we need to, we need some way how to load the data from the memory into the register and then to store it from the register to the memory.

49:42  
By the way, this is called load store architecture and it's typical for, I wouldn't say cheaper but faster architectures like MIPS X as X architecture used to design the instructions more in a way that you can immediately use memory addresses in normal operations.

50:01  
So you can use a memory address as one of the operands in the in the add instructions for instance.

50:08  
So there are architectures that that go above this the other way, but usually there will be How we can think about the architecture is that we have a load from the memory and store back to the memory.

50:19  
And there's one, one other thing we need to have to complete this, this piece of code, these pieces of code.

50:32  
What else would we need?

50:36  
One more thing and two more things which are tied together.

50:45  
Nope, nobody.

50:47  
We're talking about loading from memory.

50:48  
Sorry.

50:49  
We're already talking about loading from memory.

50:50  
We talked about loading, Yeah.

50:51  
We need some mechanism to load and store data from and to the memory.

50:55  
Yeah.

50:56  
So where which one?

51:01  
Bit shift, bit shift?

51:03  
We covered bit shift.

51:04  
I said, OK, we have all the bit wise operators and shift operators.

51:08  
Yeah, I haven't enumerate them actually, but let's just imagine that we can do that because you know what, what's in the C and basically what's in the C is in the in the assembly as well in the instructions as well.

51:21  
It's something to do with arrays that notes have without array, like with with what the there's an array there.

51:27  
Array.

51:28  
Yeah, array is a bit tricky and we get to that.

51:30  
But let's put the array aside because technically this is a arithmetic operations.

51:35  
You need to add and multiply and then you need to load and store.

51:40  
So yeah, it's a bit more tricky, but we can do this, do that with these instructions.

51:44  
We need indirect access to memory, but that's something we get into it in a short order.

51:51  
But there is one fundamental thing that's different.

51:58  
Nobody like reference to the.

52:04  
No, no, no, actually we don't have any references in here.

52:09  
This is a.

52:11  
This is an argument which is transferred via value.

52:14  
So we don't have a reference here.

52:15  
But we need to have a way how to call this function.

52:21  
Because call isn't just another jump.

52:24  
Actually call is somewhat like a jump.

52:26  
You need to redirect your execution pointer, your instruction pointer to another location in your code.

52:32  
But the body of the function is.

52:34  
But doing so, you need one other thing.

52:38  
You need to somehow store where you should return.

52:43  
You know, because when you call this function F, it holds this return statement.

52:48  
And this return statement needs to know where to jump back.

52:53  
So calling a function is not just another jump, it's a jump that also needs to record where to jump and vice versa.

53:03  
Return needs to jump back.

53:05  
So return needs.

53:06  
Yeah, return can be actually handled using an indirect jump, so jump to a address which is stored in a register for instance.

53:13  
But at least the call needs to be handled somehow.

53:16  
So if we have a function call here, this call over here needs to perform this special kind of jump.

53:24  
So if we just sort out all your ideas into the classes, so we have load instructions from memory to the register, we have store instructions from register or immediate value to the memory.

53:37  
Remember, the load operations take some time, so they are usually handled specially or somehow differently in the CPU, because the CPU tries to sometimes, usually fetch the data ahead.

53:51  
You know, as I said before, the data bus to the memory is somewhat long in the terms of latency.

53:59  
So it, it might take even let's say 1000 cycles, 1000 instructions before the data arise from the main memory to the register.

54:11  
That's a long time.

54:12  
From the perspective of the code 1000 instructions, that's a lot.

54:16  
So the CPU is trying to place to detect the execution of this type of instructions early and trying to speculatively fetch the value from the main memory at least to the to the caches.

54:33  
So it's much closer when it's actually needed, right?

54:36  
This is how the CPUs of the data really work.

54:39  
And also the compiler tries to rearrange the instructions so that when you have a fetch instruction in your code, it tries to rearrange these instructions far ahead.

54:50  
So the fetch actually takes place much sooner than the value is actually needed.

54:54  
So that's not another way how to go about this.

54:56  
So basically fetch instructions need to be placed early so we don't have to wait for the data when we actually need them.

55:06  
Move instructions are instructions that move data usually between two registers.

55:11  
In X86 you can also move data between register and memory or vice versa.

55:16  
So basically X86 use only move instructions instead of loads and stores based on the operands.

55:24  
It performs move, load or store depending on what operands you give to the instruction.

55:35  
Right?

55:36  
So arithmetical instructions I told you we basically cover them all.

55:41  
This is normal.

55:42  
Enumeration used the C syntax, so all these are usually there plus minus.

55:49  
Add SAP, no problem.

55:50  
Shifts and all bitwise operations are no problem.

55:53  
They are very fast.

55:56  
Multiplication, division and module law is slightly more tricky.

56:00  
Usually you have them as well, because it would be difficult to write a code without them.

56:06  
You really need multiplication, for instance.

56:08  
However, these instructions often take longer time to execute, so sometimes the CPU is.

56:15  
Sometimes the compiler is trying to avoid generating them, especially if you can replace them.

56:20  
For instance, if you're multiplying something with two, it can be replaced with what type of operation Shift left?

56:30  
That's right.

56:31  
So multiply by two is the same as shift to the left by 1.

56:35  
So if the compiler detects this type of optimization, it it usually takes it because it's it's much cheaper to perform shift than than multiplication.

56:45  
By the way, there are architectures that doesn't have multiplication nor division, very low level cheap architectures like Microchip for instance.

56:55  
If you know Microchip, pick controllers, they don't have multiplication, at least the cheaper ones.

57:01  
And finally, by the way, these are the basic operations.

57:04  
And when I'm talking about multiplications, I'm talking about multiplications of integers or maybe floats, not about multiplication between a string and a number.

57:14  
That's a funny stuff in Python And I don't think any other language has that.

57:18  
So forget about this.

57:20  
We are in C we are in low level, low level instruction level.

57:24  
So don't think about things like multiplying.

57:28  
This actually resolves into many, many instructions because you need to reallocate this memory, register this memory on the heap, copy this part four times.

57:38  
So it's basically a loop.

57:40  
Actually it's a nested loop, 1 loop that goes four times.

57:45  
Inside is another loop that scans this string and copies it character by character.

57:50  
No, this is very complex code.

57:51  
This results into 10s of instructions, at very least not counting the syscall for the memory allocation, etc.

57:59  
So very, very tough, very difficult to implement at the instruction level.

58:04  
If I just give you the assembly language, it would be hard time to implement this.

58:09  
And if you're not here, I will force you to implement this type of operation using only assembly language, because that would be a really, really tough task for you.

58:17  
OK, jumps, you already covered jumps.

58:20  
So jump is a displacement in in the of the instruction pointer as as we covered, unconditional jump always conditional is based on a test which is required prior to the jump.

58:33  
And this test usually involves either equality where the two values are equal, or some sort of inequality between two integers or two floats.

58:42  
There are three types of jumps from the perspective of the target where where are we jumping?

58:47  
Direct there is a explicit address where we are jumping.

58:52  
So that's like absolute value, absolute address in the memory and we are jumping at that very spot.

58:58  
Indirect means that we store the value, the address where are we jumping into the register and then the jump instruction reads the address from the register and jumps to that particular value.

59:10  
So it's like a data dependent, data-driven jump.

59:13  
This is important like this can be used as a return from a function.

59:18  
If you're returning from a function, then you will need indirect jump.

59:21  
And relative jump is a jump that jumps a few instructions ahead or back.

59:27  
Where is this useful?

59:28  
Well, actually using the direct jump requires quite a long value somehow embedded into the memory, right?

59:35  
Because if you have like a 32 bit architecture, then you need 32 bits of place somewhere in the memory, which is the address.

59:43  
That might be tough to handle, especially in MIPS which gives you only 16 bits of intermediate values.

59:49  
So it will be tricky.

59:50  
So if you want that, then yeah, you are somewhat limited in this direct thing.

59:55  
But usually you don't need to jump too far ahead.

59:59  
You need to jump only a small portion, like if you have this kind of loop for instance, then at the end of the loop you are jumping at the beginning.

1:00:08  
So you need to jump only a few instructions back, for instance.

1:00:12  
Right.

1:00:12  
And since you know how wide, how large the body of the loop is, you can count how many instructions back you need to jump.

1:00:20  
So it's much easier to implement a relative jump at the end that jumps at the beginning of the loop instead of putting an absolute value.

1:00:28  
And furthermore, if you just place a relative jump there, then this loop may be placed anywhere in the address and it will work very much the same, right?

1:00:37  
If you just shift this code a few bytes off in your memory, it doesn't matter because the relative value will still be valid.

1:00:44  
If you have absolute values, absolute addressing in your jumps, then you need to adjust these values every time the code is loaded.

1:00:53  
So you need to fix it based on where the code actually was loaded.

1:00:56  
So it's always better to use the relative addresses instead of absolute addresses, at least when it fits the situation.

1:01:04  
And finally, the call and the return.

1:01:06  
Actually the return can be implemented by the indirect jump, as I said.

1:01:10  
So we need a call.

1:01:11  
And call is basically a different type of jump that somehow records the previous address.

1:01:17  
It takes the instruction pointer, save it somewhere and then perform the jump.

1:01:22  
So that's a call.

1:01:23  
It's like a combination of these two.

1:01:26  
Technically speaking, we can do this if we can have a way how to record the IP.

1:01:31  
If we can copy the IP register to a regular register manually, then we wouldn't need call instruction.

1:01:39  
We can use this instruction plus a regular jump to implement call instruction.

1:01:46  
So yeah, due to time, I will normally I would just interrogate you whether you can, you can rewrite this assembly, but let's just let's just skip slightly ahead and I will give you directly the answer so we can examine it.

1:02:04  
So basically let me revisit the code you see before you saw before and this is the if else branch.

1:02:12  
And this is how it will translate roughly into the assembly language.

1:02:17  
This is almost X86 assembly, I think.

1:02:23  
I think almost.

1:02:24  
I'm not sure if there are any differences.

1:02:26  
So what this does?

1:02:28  
What does this mean?

1:02:31  
I I realized that these acronyms are somewhat confusing.

1:02:35  
But if you, if you just get the hang of it, it will it will come out to quite, quite easily because J at the beginning always means jump.

1:02:45  
That's easy.

1:02:46  
So jump and GE is where when to jump GE means greater or equal.

1:02:56  
So now, now we're getting hang of it.

1:02:57  
So jump if greater or equal.

1:03:00  
And what is being compared a these brackets indicate that the a this is just a symbolic value.

1:03:06  
It's it's a place in the memory.

1:03:08  
So it's a variable in memory and the compiler, the assembly compiler know where it is.

1:03:13  
So the assembly compiler somehow will replace this with an address when it when it's compiled into the binary and and three and the three is like immediate value in your code.

1:03:23  
So it compares these two values.

1:03:25  
And if a sorry, it's, it's the other way around.

1:03:28  
So it's if a is greater or equal and three.

1:03:32  
So it's it's inversion of this condition actually, then we will jump.

1:03:37  
So basically, if this condition doesn't hold, we will jump to the next part of the code, right?

1:03:45  
Because if this holds, we we are executing what's next?

1:03:49  
So if this doesn't hold, we need to turn this condition, we need to negate this condition.

1:03:56  
So if this condition doesn't hold, if its negation holds, then we are jumping to the next part of this code, to this block.

1:04:05  
Let's start with the first part, that is b = 4.

1:04:08  
So we are storing 4 to the B, usually on the left.

1:04:12  
The left operand is usually the target.

1:04:15  
The right operand is what we are assigning.

1:04:17  
So b = 4.

1:04:19  
This is pretty straightforward.

1:04:20  
This is just a rewrite of this line over here.

1:04:23  
But afterwards we need to perform another jump.

1:04:26  
JMPMP isn't an acronym.

1:04:29  
This is really jump because it's unconditional jump.

1:04:32  
We are always jumping here, and we need to jump at the very end because we need to skip the else branch.

1:04:38  
OK, it's very straightforward.

1:04:42  
And so if we skip the else branch, we will end up here.

1:04:44  
So this whole 3 instructions are these, this is this expression load R1A means we take the a variable A and we put it into R1 register shift left R1 by two.

1:05:00  
That's pretty straightforward.

1:05:02  
What we loaded into R1 is shifted by two to the left that that's easy.

1:05:06  
And then we store what's in R1, that's our temporary value into the C This is the this assignment.

1:05:13  
That's it.

1:05:14  
OK.

1:05:17  
So yeah, yeah, actually load and store I would.

1:05:24  
I had to take load and store as a placeholders for move.

1:05:28  
If I want to translate it to the X86 architecture, I need to replace this with move MOF instruction.

1:05:34  
But that's just a small adjustment I need to make because I want you to explicitly, explicitly see where I'm loading and I'm where I'm storing the values to.

1:05:44  
And from the memory.

1:05:45  
This is much more readable.

1:05:46  
But basically this can be almost directly rewritten into X86, x86 assembly language.

1:05:53  
So let's start with the second one.

1:05:55  
OK, this is slightly more tricky, but only slightly so.

1:06:02  
First of all, we need to initialize the loop, right?

1:06:05  
First we have a for loop.

1:06:07  
So first thing we do is we perform this first expression I = 0.

1:06:14  
OK, there is no I in the memory.

1:06:18  
Actually compiler decided that I would be placed in register.

1:06:25  
Why not?

1:06:25  
Because we have spare registers.

1:06:27  
So let's just put I in register and there is no, no reason why should we bother the main memory unless someone really wants to access from outside from another thread, for instance, the value of I and I cannot imagine why.

1:06:43  
Then it's perfectly safe that we just keep it local because it would be used very often.

1:06:48  
So it would be very tedious if we need to just all the time jump into the memory or at least into L1 cache.

1:06:54  
So we keep it in the register.

1:06:56  
And this is a bit of bit confusing because we start with the jump, but remember this is a conditional jump and remember the first thing, sorry.

1:07:06  
The second thing we do in for loop is that we test this condition.

1:07:10  
We are always testing the condition before the loop.

1:07:15  
So we also test it at the very beginning before we start the first iteration.

1:07:20  
So we need to perform this comparison again.

1:07:24  
This is jump if greater or equal.

1:07:27  
So greater or equal is inversion of this condition.

1:07:30  
So if R1, which is our I is greater or equal 5, which is an immediate value, then we jump and we jump at the very end and we don't don't continue with the loop anymore.

1:07:42  
This is the end of the loop.

1:07:43  
If we jump over here, we jump, we are jumping off the Cliff and this is the end.

1:07:48  
OK.

1:07:49  
And the reminder is the body of the loop.

1:07:52  
So basically we need to load a into R2.

1:07:58  
What is a, by the way, in this in C terms, what is a?

1:08:01  
It takes an array, Sir.

1:08:04  
Yeah, if it's an array.

1:08:05  
So what is if I just write a in the code?

1:08:08  
What?

1:08:09  
What does that mean?

1:08:11  
It's a pointer, That's right.

1:08:12  
So I'm loading the address of this array in the memory.

1:08:16  
It's a pointer, so the address of the beginning of the array in the memory is in R2.

1:08:22  
Then I need to move the pointer to the right point because I'm indexing ith element in the array.

1:08:31  
So I need to advance this pointer far enough.

1:08:36  
This is a bit conspicuous because I'm just adding R1, which is I to the R2, and I'm storing into into the R2.

1:08:47  
Something fishes here, right?

1:08:49  
What's fishy here?

1:08:50  
You're adding a number to a pointer.

1:08:57  
You're adding a number to a pointer.

1:09:00  
Well, that wasn't what I was referring to, but excellent, excellent remark.

1:09:04  
I'm adding a number to a pointer.

1:09:06  
Is that bad?

1:09:08  
Adding a number to?

1:09:09  
Sorry, adding a number to a pointer.

1:09:12  
I just hope I didn't turn my microphone off right now.

1:09:16  
Is it?

1:09:16  
Is it a bad thing to take a pointer and a number put them together?

1:09:21  
No, pointer is also a number.

1:09:24  
Exactly.

1:09:25  
Pointer is also a number.

1:09:26  
So yeah, semantically it might be problem if you're not doing it right.

1:09:30  
But technically in implementation level, that's fine.

1:09:34  
They're both integers, no problem there.

1:09:37  
OK, but something else might be fishy here.

1:09:40  
Remember what you know about pointer arithmetics in C So how do you advance the pointer a if you want to add I to it?

1:09:49  
Because this is just a.

1:09:50  
This is just a nicer form.

1:09:53  
How to write a plus I and the reference.

1:10:06  
Yeah, but so technically this looks like just the addition here.

1:10:10  
Yeah, that's right.

1:10:11  
But this doesn't always add if I is 3.

1:10:18  
This doesn't mean we are always advancing the pointer by three bytes, right?

1:10:26  
Nobody remembers.

1:10:27  
My goodness, you're right.

1:10:46  
Yeah, that's right.

1:10:47  
But that's not the answer I'm.

1:10:49  
I hope you haven't embedded your answer inside the words.

1:10:52  
I haven't completely understood.

1:10:53  
But yeah, the I is advancing the pointer, but if I is 3, it doesn't really mean that we are advancing it by three bytes.

1:11:02  
How if we have a array, how many bytes we will advance it if I is 3/3 bytes?

1:11:10  
That's not the same thing as three integers, that's right.

1:11:13  
So it depends on the type of the of the array.

1:11:16  
So if the array is array of characters of or or bytes, this will be fine.

1:11:22  
If this is an array of integers, and it probably is or it might be at least because we are assigning an integer into it, then we are in trouble because we haven't multiplied it over here.

1:11:34  
So this was like a small landmine I just left there so we can talk about it.

1:11:38  
OK, so this might not hold if the array is anything else but array of characters or bytes, right?

1:11:46  
If it's array of integers, we need to multiply R1 because because R1 is I by 4 or by size of the the base type before we add it.

1:11:57  
OK, so remember that that that's a that's a small pitfall.

1:12:01  
And then we store R1 I into the address we just computed.

1:12:07  
So the address of the element we just computed, this is indirect store.

1:12:11  
We are using a register as an address.

1:12:14  
So we take the value from the register as the address to the memory and we will store it there to the memory.

1:12:19  
So this is indirect store.

1:12:21  
This is direct load.

1:12:23  
The a will be probably placed here as a direct value.

1:12:26  
This is indirect store.

1:12:28  
And finally, we will add 1 to R1 because R1 is I and we need to add 1 to it.

1:12:36  
This is that's this instruction over here.

1:12:38  
Finally, we have jump that restarts the loop and that's it.

1:12:43  
Everybody's clear on that.

1:12:46  
OK, so let's start.

1:12:49  
Yeah, this is the just to remind you when you're reading the slides, this is the pitfall.

1:12:53  
I was, I was pitching it.

1:12:56  
OK, we have few more minutes.

1:12:58  
So let me start on registers.

1:13:00  
And I think that's the place where we end today.

1:13:03  
Registers because it's a good place to pick next week.

1:13:07  
Registers.

1:13:09  
Actually, we have multiple types of registers, as you may have noticed, but actually it wasn't that apparent from the examples.

1:13:18  
Normally most architectures deal with general registers.

1:13:21  
A general register is a register.

1:13:23  
It's like orthogonal for operations.

1:13:25  
You can place integers there, you can place other values there, you can place pointers there, it doesn't matter.

1:13:31  
The only thing the general register usually specify is that it has some width.

1:13:36  
So it has 32 bits, 64 bits, something like that.

1:13:41  
Some architectures make a distinction that they have unique integer registers and floating point registers.

1:13:48  
So some registers can be used for integer instructions, other registers can be used for floating point operations.

1:13:55  
Might make sense.

1:13:59  
Also, it might sometimes it might be useful to make a distinction whether the address is in the register or not.

1:14:05  
I didn't make that distinction in the examples before.

1:14:08  
And some architectures like MIPS don't do this distinction because, as somebody pointed out, pointer is still a number, so it's not that important for US.

1:14:18  
Pointer address registers and integer registers might be very much the same, but sometimes it's a good thing to make this distinction so that nobody would accidentally use a address register for a regular number or vice versa.

1:14:32  
Sometimes it makes makes sense.

1:14:34  
Also, branch registers are address registers, but there are special type of registers which are used if you have the branch instruction.

1:14:42  
Branch instruction is another type of jump instruction.

1:14:46  
So it's like you have a condition and you place in the branch register and address of the target jump.

1:14:53  
So basically you evaluate a condition and then you use the branch register to determine the target of your jump if if the condition holds flags.

1:15:03  
Again, this is special type of registers where individual bits or bit groups sometimes have a special meaning.

1:15:10  
And finally, predicate registers are registers which are one bit wide.

1:15:15  
Yeah, that's right.

1:15:16  
It's very strange register because you can place only zero or one inside this register, but they can be used to determine whether something should be or shouldn't be executed, for instance.

1:15:27  
So you can use this predicate registers to control your execution flow or you can use them to store results of your of your conditions.

1:15:35  
If you're making some tests like the comparison test or inequality test, then you can store the results into predicate registers, application and system registers.

1:15:46  
This distinction is made at the, let's say conceptual level usually.

1:15:50  
Well, usually normal registers are used for arithmetic operations, etcetera.

1:15:56  
Application and system registers are designated for things like argument values.

1:16:01  
When you're calling a function, return value when you're calling a function.

1:16:09  
Function arguments, what else?

1:16:12  
Stack pointer.

1:16:13  
Yeah, stack pointer and base pointer.

1:16:15  
So if you have a stack, which is necessary to implement correctly, calling of functions, because without stack you cannot place the return values anywhere.

1:16:24  
So you cannot have a recursion without stack.

1:16:26  
We get to that later.

1:16:27  
So basically you need a pointer, you need a special register that points where the stack is in the memory.

1:16:34  
And finally there are some additional types like vector registers which are very popular for high performance computing nowadays.

1:16:41  
You have a register which can fit multiple values like SSE registers can fit for integers or floats.

1:16:48  
So they have they have 100 to 828 bits in width and they can store for 32 bits float or integers side by side and or.

1:16:59  
The most modern AVX 512 instructions work with 512 bit vector registers, so you can fit 16 floats or or integers in them.

1:17:10  
And then these instructions that operate on these vector registers perform the same operation on all values at once.

1:17:17  
So they are much faster, they can be implemented much more efficiently.

1:17:22  
Another topic is naming.

1:17:23  
Naming of the registers is very tricky because some architectures go about it in a way that, OK, we have 32 bit registers.

1:17:30  
So let's see, we just name them register zero to register 31.

1:17:35  
And that's probably something I would have I would have done because it's pretty straightforward, although it's not very mnemonic because you need to remember whether you placed your value into register three or register eight.

1:17:47  
Yeah, that's a bit tricky.

1:17:48  
But in the end, you can use the registers orthogonally.

1:17:51  
There are no special registers in such a in such arrangement.

1:17:56  
On the other way, you can name the registers like EAXEBXECX where ABCD stands for special meanings like A is accumulator, B is base, C is counter, D is data.

1:18:08  
And they are designed to be specifically used with some types of instructions like the counter is usually incremented or tested, like the data usually holds the result of a large operation, etcetera.

1:18:20  
Finally, there are, let's say today, I would say obsolete because I'm not sure whether there is a contemporary architecture that still uses it.

1:18:27  
But technically you can use also a different approaches to naming, naming like stack that you have a stack, you know what stack is like a data structure stack, which can be pushed and popped.

1:18:38  
So the registers are named from the top of the stack further, so like SP1SP0, sorry is the top of the stack, SP one is the number right below the stack, etcetera.

1:18:51  
This was used in 8087 Co processor.

1:18:56  
Anyone heard about 8087 not 6.

1:19:04  
Anybody heard about that?

1:19:06  
I think I might have come across about dumbling.

1:19:08  
Yeah, it's very old.

1:19:10  
Actually, it's not that old because I'm still very young and this was on when I was learning how to program.

1:19:16  
But basically you, you probably heard about 8086, right?

1:19:25  
At least in history lessons, because this is where the X86 architecture came from, because this is the first processor that was worth something that was designed by the Intel back then.

1:19:37  
And then they designed 286 and 386 and 486 processors.

1:19:42  
And then came the pantume, right?

1:19:44  
You know by by the pantume was named Pantume.

1:19:47  
Why it wasn't named 586?

1:19:49  
Because they they used the 1st prototypes OF586 and they just take the four eight 6 + 100 and they get 587.000.

1:20:00  
So basically they they just name it pain tune because there was some flaw in in arithmetics.

1:20:07  
OK, sorry my side track got too far from this topic.

1:20:12  
So this was a coprocessor for 0086 and 02.

1:20:18  
286 processors and it was designed to perform floating point math so there was like a Co processor extract Co processor just for floating point instructions and it was using the stack organization.

1:20:31  
So we are so we were pushing the floats on the stack and then you can perform stack operations like take the two top registers and put them together, add them together or something like that.

1:20:43  
And finally, there is also possibility of aliasing, which means that some register may overlap or may be named by different different names.

1:20:52  
This is better explained on this example.

1:20:56  
This is perhaps the most wicked architecture we have that's still being used because it has this type of registers and it's very old and it has so many skeletons inside that wasn't that that couldn't be removed easily.

1:21:09  
So this is one of the favorite teasing pass.

1:21:12  
Probably This is why Mr.

1:21:14  
Reject still uses it in his classes.

1:21:16  
We don't except for showing you that things can be done worse than you thought.

1:21:23  
This is the set of registers that were originally designed in the X86 architecture.

1:21:27  
Actually the first, the very first processor which was 4004 had four registers ABC and D and from these came these registers A was as I said accumulator, base, counter and data.

1:21:43  
And later when they extended it they need to rename ABC and D because the original registers were 4 bits and then 8 bit 8 bits wide so they need to extend it.

1:21:56  
So they used AL and AH as a lower bit and higher bit.

1:22:01  
So these two together form a 116 bit register.

1:22:05  
They are a bit wise.

1:22:06  
They form 116 bit register which was called AX and same for BX, BC and DX.

1:22:13  
So that's what I'm talking.

1:22:14  
When I was talking about aliasing, there is this AX register and it's covering both AH and AL.

1:22:21  
So if you are accessing AL you are reading only half of the AX register basically.

1:22:27  
And if you write into AX you overwrite both AH and AL.

1:22:31  
And when they extended it from 16 bit architecture to 32 bit architecture, this is where 386 processor came.

1:22:40  
They just renamed the AX to EAX like extended AX.

1:22:45  
So this EAX has 32 bits and this lower 16 bit part was aliased AX Yeah.

1:22:55  
And when when the X86 was extended to 64 bits they renamed EAX to RAXI don't know why R probably as a register.

1:23:06  
So the RAX is 64 bits wide.

1:23:10  
Yeah.

1:23:11  
By the way, this is not how to design architecture well, because you have only four registers designated for basic computing.

1:23:19  
There are other 4 registers.

1:23:21  
2 of them are designated for special functions.

1:23:24  
SP is a stack pointer and BP is base pointer.

1:23:28  
These two are necessary for performing function calls properly.

1:23:32  
So these two are very specific.

1:23:34  
And these two, SI and DI, are used as pointer source index and destination index.

1:23:41  
If you're copying data, they can be used easily.

1:23:44  
By the way, there is one instruction that has no operands, and all it does, it takes the address of SI, takes one byte from SI, copies it to the address DI, and then advances both SI and DI.

1:23:59  
One instruction, yeah, that's how this architecture was designed.

1:24:04  
These are segment registers which are not that important.

1:24:07  
This is an example of flex register.

1:24:09  
There is a specific flex register where individual bits have special meaning.

1:24:13  
For instance, if you perform a comparison operation, some flex like 0 flex or yeah I forget the name right now, there are few flex that are affected by this operation so that you can read these flex and then determine whether the operation was success or not, whether the comparison hold or not.

1:24:33  
And of course there is instruction pointer or architectures need instruction pointers.

1:24:37  
By the way, even in Intel they realized how great mistake they did by designing an architecture with so few pointers, so few registers, Sir.

1:24:47  
So they extended it, they added extra eight whole 8 registers when they extended it to 64 bits.

1:24:55  
But still we get like 16 registers in total for basic computations and that's really not too many.

1:25:04  
16 is a very low number if you try to fit all the data in in a regular application code.

1:25:11  
Yeah, I have like 2 minutes.

1:25:12  
So I think this slide is too big for two minutes.

1:25:15  
So I will keep it for next week.

1:25:18  
So let's just keep it slow and let's, let's pace ourself.

1:25:25  
I think we can use this 2 minutes for any questions that might rise at the very beginning of the CPU architectures course.

1:25:32  
So in Mix 32, how does do you access, how do you load and store from memory?

1:25:40  
It's a load store architecture.

1:25:42  
You have explicit instructions where you place a load or store and the data is copied from the memory to a register or vice versa.

1:25:49  
And another way is there are some instructions that use this immediate values.

1:25:54  
So you can use this immediate values to basically load constants in your code.

1:25:59  
So this these are two ways, but it's a load store architecture.

1:26:02  
So you're not moving data from memory to to registers implicitly.

1:26:07  
You need to place store or load instruction ahead of your code or at the end of your code to transfer data here and there.

1:26:16  
Any other question?

1:26:19  
OK, I think we can finish 50 seconds early.

1:26:22  
So thank you for your attention and hopefully see you next week.

1:26:25  
Thank you.