**2025-03-13-cpu2-en**

0:00  
Today we're on time, so we are getting better and I believe we stopped about here in the middle of the CPU course last week.

0:10  
So let me just continue from here on.

0:12  
Last week I introduced you the registers and as an example, I use the X86 architecture, which is one of the most profound architectures because we have most likely you have it in your laptops.

0:24  
But also it's an example of a not very well designed architecture.

0:29  
And the reason for that is it's quite, quite ancient because it was designed many decades ago and back then they didn't realize in the modern times we will need so many registers.

0:39  
Let me give you an example of an opposite architecture which has too many registers in my opinion, and that's IA 64.

0:46  
By the way, the X86 architecture or the architecture to which I usually refer as as X86 is officially called IA 32.

0:57  
So IA 64 isn't just an ext 64 bit extension of IA 32.

1:03  
It's a completely different architecture which was designed by HP, not by Intel.

1:09  
And yeah, I can realize this, this illustration might be confusing.

1:13  
So let me just give you a glimpse what's what's where.

1:17  
So here the first part, first block is called general registers.

1:23  
So they have 128 general registers.

1:27  
So registers that can be used for whatever at arithmetic operations you'd like using integers.

1:34  
Aside of that there is 128 floating point registers.

1:39  
So registers dedicated only to floating point operation.

1:43  
And aside of that there are also application registers.

1:46  
So registers which are dedicated to special functions like holding the instruction pointer or holding the stack pointer, etcetera.

1:54  
So as you can see they have registers in abundance.

1:59  
Besides these more traditional registers, because you can still imagine what general register or floating point register would hold, there are additional registers like predicates.

2:09  
Predicates are essentially single bit registers.

2:12  
So there are registers which each can hold a single bit and they are used in instructions that require some sort of condition.

2:21  
So these instructions kind of refer to a predicate register.

2:25  
And if the predicate register holds 1, the instruction gets executed and if they hold 0, the instruction is skipped.

2:32  
So this is a very complex architecture and besides that they have a branch registers where an address of a branching instruction can be stored.

2:40  
So if you need branching, if you need jumps, then they have dedicated registers for addresses for branching.

2:47  
So as you can see you can get a very simple or very narrow or narrow minded architecture like X86 and you can have a very broad architecture that uses splint horror of registers like IA 64.

3:03  
OK.

3:05  
Most of the time in this class, actually this refers mainly to this lesson.

3:11  
We will use the MIPS architecture which stands somewhere in between.

3:15  
It's not very fancy in in its old design and everything, but the MIPS architecture actually became basis for RISC 5 architecture.

3:25  
I told you about that last week I believe.

3:28  
And RISC 5 is something that is quite modern today there are many companies that trying to build their own RISC 5 CPUs and maybe in the foreseeable future RISC 5 will play a significant part in our daily life.

3:41  
So it's not like we are teaching you something ancient, something archaic.

3:44  
This might be useful for you as well.

3:46  
By the way, the one of the subjects that's pick up on the discontinuation of this subject, architecture of architectures of microprocessors I believe it's called, uses RISC 5 as the referential architecture on which they teach you more in more detail how the CPUs behave.

4:03  
So it it's a good a good thing that we start with MIPS, because you can easily pick up with RISC 5 afterwards from the terms of registers.

4:14  
In the terms of registers, MIPS stands somewhere in between X86 and IA 64, maybe slightly on the X86 side, they have 32 bit 32, they have 3232 bit registers, so it's easier to member.

4:29  
It's MIT 32 which has 3232 bit registers, easy.

4:34  
And these registers are like general, they are designated R0 to R31.

4:39  
So it's, again, it's very easy to remember.

4:42  
You don't have to remember that there are some special registers.

4:44  
Actually that's not entirely true.

4:45  
I will get to that in short order.

4:48  
Yeah, sorry.

4:57  
Yeah, yeah, sure.

5:00  
Yeah.

5:00  
So what's in there?

5:02  
It's quite simple architecture in the terms that, for instance, there is no stack.

5:07  
Well, what?

5:07  
What does it mean?

5:08  
Because you need stack, right?

5:10  
Everybody who knows something about programming realises that you need stack because without stack you cannot perform function calls, right?

5:17  
Because where would all the local variables go besides the stack?

5:22  
That that's not entirely true.

5:23  
This says that the ISA specification itself doesn't define how stack should be implemented.

5:30  
So the stack isn't part of the CPU specification like in X86.

5:36  
Unlike in X86 where you have stack register as an explicit register, you have specific instructions to push and pop this stack register.

5:46  
So this architecture says, OK, implement the stack as you wish.

5:50  
I don't care.

5:51  
I I just give you the registers.

5:53  
I just give you branching instruction and everything.

5:55  
But it's your job to implement the stack.

5:58  
There are no flags.

5:59  
Yeah, again, this is not not big deal.

6:02  
No big deal actually, because you don't need that much flags.

6:04  
You can have specialized registers for that.

6:06  
So no big deal.

6:07  
And yeah, the instruction pointer or the program counter, of course, has to have a separate register.

6:13  
So this is The only exception to this.

6:14  
So besides the regular 32 registers, you have one extra where the instruction pointer or the program counter is.

6:22  
OK.

6:22  
So it's it's basically very simple.

6:25  
Let's let's dig into it.

6:26  
But before we do that, there is one piece of information I need to give you right away before we get started, because this is going to get somewhat intermingled with the ISA specification.

6:39  
You know, as I said just on the previous slide, the ISA of MIPS give you no stack.

6:45  
There is no stack, but we need stack.

6:47  
Stacks are good.

6:48  
We really need stack to implement software, otherwise we couldn't do things like function calls.

6:53  
So we need additional specifications which told us how to implement stack and why.

7:00  
This needs to be a specification, because if you invent your own way how to implement stack and you could, your software wouldn't be compatible with other software, like with libraries, with operating system, etcetera.

7:11  
And it's a very good idea that all the software that runs in one system is somehow compatible at least on some level.

7:18  
So they can.

7:19  
For instance, you can for instance call function calls of a library and the library knows how the items are stored on the stack.

7:28  
So the library and your application creates the stack in the very much same manner.

7:36  
And to do that you need additional specification.

7:38  
And the specification is called ABI Application Binary Interface.

7:43  
You are at this point, you are probably familiar with API, right?

7:47  
Does anybody hurt the acronym APII believe so.

7:52  
Yeah, that's pretty much it.

7:53  
So API is an interface at the application level, at the programming level, at the level of your code.

8:01  
This is an interface that needs to be specified 1 storage beneath you, one storage beneath regular programming.

8:08  
So this is an interface that needs to be obliged by the libraries, by the compilers, by the operating system, so that the applications are designed in a particular way and so they can communicate with each other.

8:21  
Most importantly, how the registers are being used, especially when one module, one piece of application calls another piece of application.

8:30  
How stack is implemented?

8:31  
That was my my original thought and also function call convention.

8:36  
So if you call a function in a library, how and where you store your function arguments and vice versa, how and where you will find the result of the function.

8:46  
That's very important because otherwise you cannot call foreign libraries.

8:51  
OK, so in this case I'm going to intermingle in some parts the ABI specification with the ESA specification.

9:05  
That's the specific I told you last week.

9:08  
That's the specification of the interface of the CPU.

9:11  
It tells you which which registers the CPU has.

9:14  
It tells you which instructions the CPU has.

9:19  
So sorry, I just got interrupted and I lost my stack.

9:26  
Yeah.

9:27  
So basically the ABI, that's what I wanted to finish is a like a prescription, like a recommendation how the CPU should be used.

9:34  
You can, you might have multiple Abis, but it's not a good idea to follow, follow multiple Abis.

9:39  
So usually you have one.

9:42  
So now if we know that we have 32 registers in the ABI, they have specific meanings, so they are somewhat like named.

9:50  
And there is a prescribed way how you should use the registers.

9:53  
For instance, the first register, actually the zeroth register, it's called 0, easy to remember and it holds zero that's also easy to remember.

10:04  
Why would we need a register that always holds zero?

10:08  
Well, for one, if you need a zero to read from, if you need to read a zero from somewhere, you you can read it from the zero register.

10:17  
That's, that's quite useful.

10:18  
If you need to compare something to 0 and it's quite a usual task in in a branching and in conditions, then you have a zero register.

10:27  
So you have always a zero there.

10:28  
It seems like a waste, but it's actually quite useful.

10:32  
And the first register is a assembly temporary, which means if you have a temporary value, if you're making some computations and you need to place a temporary value somewhere, the first register is the the first one you should.

10:43  
You should put.

10:44  
You should use for that.

10:46  
These registers are used for function calls V0 and V2 is used for a return value.

10:52  
So if you have a function call and the result of the function is 64 bits at most because there are 232 bit registers, then the result result is placed here.

11:07  
Otherwise it needs to be placed on the stack.

11:10  
Why don't we place the the result on the stack all the time?

11:14  
Why do we bother returning values via registers?

11:17  
It it's just a complication, right?

11:19  
Because you need to remember that these two explicit registers are used for returning values from function calls.

11:25  
Why?

11:26  
Why would we bother if we have if we have the stack anyways?

11:29  
Why?

11:39  
Well, you can get it synchronously via memory.

11:41  
That's that's the same thing if you write it to the memory.

11:44  
You can read it from the memory.

11:45  
That's the same thing.

11:47  
If you write it, it's much faster.

11:50  
That's right, because writing to the memory and reading from the memory it's quite slow, even even with caches.

11:56  
So returning the value or values through the registers is much much faster, because just writing something to the register or reading something from the register happens on the synchronous clock basis.

12:08  
It's not.

12:08  
It's not something like you should wait for.

12:10  
If you write something to the memory and then read it back it, it might take several cycles or 10s of cycles even if it's cached in L1.

12:19  
The same goes for function arguments.

12:21  
These 4 registers are dedicated to function arguments.

12:24  
So if a function has less than 4 arguments and they are less than 32 bits each less or equal, then they are placed in these registers again for speed.

12:33  
If they don't fit or the remaining there are more than 4 arguments, they are placed on the stack.

12:39  
Yeah, temporaries are used like the assemble.

12:43  
Temporary temporaries are used for immediate values.

12:47  
That's you need to place somewhere during your computation so the compiler allocates them dynamically for local variables or for intermediate values.

12:58  
Safe temporaries have similar meaning, but the difference is Here I haven't mentioned the last column yet, the preserved column, which tells us whether these registers should be preserved or not during a function call.

13:13  
That's another mind tweezer, because sometimes if you make a function call, the function needs to use registers.

13:22  
Well, not sometimes.

13:23  
Usually because there there is another code.

13:25  
The codes need to compute something.

13:27  
So if you make a function call, it's possible that some of your registers as as you as a function will be overwritten by the call code function.

13:37  
OK, so in this case, this flag tells you whether the callee, the function you are calling, is responsible for preserving the registers.

13:49  
And these are special because they say yes.

13:53  
So if a function is called, the callee, the function that's being called is responsible for either not touching these registers at all, so not modifying them, or saving them somewhere in the memory, for instance, and then restoring them later before the function concludes.

14:11  
Actually, it's not the callee itself, it's the it's part of the of the function invocation protocol.

14:17  
But basically that's that's what compiler or the runtime must ensure if these registers are being used.

14:25  
Yeah, temporaries are obvious.

14:27  
There are also some kernel registers which regular application shouldn't use.

14:31  
Why?

14:32  
They are special registers dedicated to kernel.

14:36  
Well, as we get to it later, a CPU has a special mode, a special function which is called an interrupt.

14:44  
Sometimes the regular run of the application code can be interrupted by an external source, and at that point the kernel is invoked.

14:53  
And it's a good idea that the kernel has immediately a few registers it can use without being concerned whether these registers are free to use or not.

15:03  
So these registers are dedicated for kernel use only and the last four are quite important.

15:10  
This is a global pointer.

15:12  
So if you have a global variables in your code then all the variables are stored in a particular location and this is like a base pointer for them.

15:20  
So this is like a pointer at the beginning of the area where all your global variables are stored.

15:28  
Do you know what global variables are?

15:30  
I hope after a few lessons of Arduino you already have so same goes with the stack pointer.

15:35  
On the stack we have stored the local variables and possibly extra arguments if they don't fit here.

15:41  
So stack pointer is here.

15:43  
So you can use the stack pointer to address add extra arguments or the local variables of the function frame pointer.

15:51  
It's slightly complicated.

15:53  
At this point I will just tell you that the frame pointer goes along with the stack pointer.

15:57  
There is a particle place on the stack you need to also locate.

16:02  
That's not very important right now.

16:03  
I will talk about that later when we examine the function call protocols in detail.

16:08  
So these two work together, stack pointer and frame pointer.

16:12  
And finally the return address.

16:15  
Oh, that's a very special register because if you make a function call, this is a very special register in the way that the instruction that performs the function call will actually use this register to store the previous value of the program counter.

16:34  
So basically if you're making a function call, first the program counter is like snapshotted here, so it's like saved here.

16:43  
And then the jump, actually the function call is basically a jump will will take place.

16:49  
So the jump itself is preceded with saving the original address.

16:53  
So the callee, the function which is being called, can take this address and perhaps store it somewhere, perhaps store it on the stack or or just otherwise preserve it.

17:04  
And when times come to return from the function, this address is used to jump back from where we were called.

17:11  
OK, that's very important, because otherwise the jump that takes place during the call wouldn't know where to return.

17:19  
You need an address where you need to.

17:21  
You need to know to which address you want to return after the function is finalized, after the function concludes.

17:29  
OK, I will now go through a few examples of MIPS instructions.

17:33  
I'm not doing a complete reference manual reading here, so I won't be presenting all the instructions that are available.

17:42  
Just to give you some glimpses, some some notions which instructions are available, which instructions are like typical for this kind of architecture and how they are constructed.

17:52  
It's not important that you exactly remember which instruction is which, meaning ADD and ADI.

17:58  
This is this one is easy because ADD actually isn't an acronym, it's a name.

18:03  
This is a addition so that it's performing ADD operation, and this is ADD immediate, so it performs addition with a register and an immediate value.

18:14  
But you don't have to remember this.

18:16  
For the test, we will give you the name of the instruction, but you need to somehow remember or at least deduce the semantics of the instruction because we will.

18:25  
It will be required of you that you translate instructions into C code or vice versa.

18:31  
So you need to understand the instructions at least at the semantic level.

18:35  
You don't need to remember all the acronyms.

18:39  
So the semantics of the operands, remember there are three operands in this architecture, so the semantics is the same as if you write them sequentially one after another in AC expression.

18:52  
So in AC expression you would have the right side, the left side, sorry.

18:55  
So that's the target, that's where the result will be stored.

18:59  
Then you place a virtual equal virtual assignment between these two.

19:05  
And then the name of the operation is placed virtually between these two.

19:09  
So semantically, this is the equivalent of into the register Rd.

19:13  
like a destination.

19:14  
Let's play Rs like the source and RT that's just the next letter after S so that's another source.

19:22  
So these two source registers are added together and stored in this, in this target, in this destination register.

19:29  
So the semantics is actually easy.

19:31  
Just remember that the equally the assignment goes here and the operation goes here and that's it.

19:36  
Just read it as if it's written in C The add immediate is very much the same.

19:41  
The only difference is that the second, the third, sorry, the last argument, the last operand isn't a register value, isn't a register index, it's an immediate value.

19:53  
So there are 16 bit.

19:55  
Remember the the layout of the instruction, there are 16 bits reserved in the second-half of the instruction work, so these 16 bits are extended.

20:06  
Yeah, the tricky part is that this register is a 32 bit register and this value is 16 bit value.

20:14  
So we need to make them somehow compatible.

20:16  
That doesn't happen magically that we need a specific way how this happens.

20:20  
So this operation here sine X is a signed extension from 16 bits to the target size, which is 32 bits.

20:30  
And what's signed extension means?

20:32  
Well, regular extension just simply place zeros at the beginning.

20:36  
So if I have like let's just say something like 1011, let let me just write 4 bits for you.

20:44  
Let's imagine what's extension of four bits into 8 bits so I don't have to write 32 bit numbers.

20:49  
So if I want to extend this in a regular normal way, unsigned way, then I will simply add zeros.

20:58  
But this doesn't work properly for signed numbers, right?

21:03  
If this was an unsigned number then everything is fine because if this was an unsigned number then I'm just adding zeros.

21:11  
So this is the same notation in a in eight bits as this was in four bits.

21:17  
However, for the side number that's not true because we are using the two complement for the side numbers and this number was starting with one, which means in A2 complement if this was a four bit number in two complement.

21:37  
Note it was negative exactly.

21:41  
So when it's negative, I have to extend it in a way it will remain the same negative value in the extended format.

21:48  
Fortunately for us, it's very simple.

21:50  
We just take the highest bit and copy it.

21:53  
So the sign extension means I I'm not putting zeros here, I'm taking whatever the highest bit is and fill it here.

21:59  
So in this case it would be 11111011.

22:04  
OK, if you don't believe me, small home exercise would be to verify that this is true.

22:11  
Just you can use a calculator or whatever converts bit values into into the similar values or whatever base you prefer, OK?

22:22  
The same goes for sub.

22:23  
Sub is subtraction, so I'm not going to dwell on that.

22:26  
So this is addition, this is subtraction, this is obvious operation.

22:30  
OK, yeah.

22:34  
And these are obviously just how to read this properly.

22:39  
These are like placeholders.

22:40  
The Rd.

22:41  
isn't a name of an exact exact register.

22:43  
It's a placeholder.

22:44  
And you can use whatever register from the zero to 31.

22:49  
You seem fit.

22:49  
So an example would be add R2 equals R4 plus R5.

22:55  
OK, yeah, maybe this is obvious.

22:59  
I'm just want to point it out for for those who might be slightly confused with the notation, okay, example and comparison of the MIPS architecture to X86 architecture.

23:11  
Yeah, we are doing that for several reasons.

23:13  
One reason is that my esteemed colleague, Mr.

23:17  
yes, probably still teaches or hopefully I think you've verified that at the beginning of the semester still teaches X86, still use this X86 example.

23:25  
So you probably at least a tiny bit familiar with that particular notation.

23:30  
And 2nd, sometimes it's just it stands out how this how this notation is slightly more verbose.

23:38  
The the reason is that you have only two operands in X86.

23:42  
So in some operations are more slightly more tricky if you have only two operands per instruction instead of three.

23:50  
So the first example, let me give you an addition of two values into the first one.

23:55  
Well, that that's easy because if the target register and the first register are the same.

24:01  
So basically we are adding T0 into T1, OK, easy.

24:07  
So that that's the the same goes in in X86, because in X86 you cannot take two registers and put them in the third.

24:15  
You are adding one register into another register.

24:18  
That's the notation.

24:19  
So you are adding ebx into EAX.

24:23  
However, if you want to do something more elaborate like this, yeah sorry, like this, if you use different registers that the T2 is the target, T0 and T1 are the sources, then in X8X it will split into two instructions because first you need to take the first register and put it into the second one.

24:44  
This is the move operation.

24:46  
It just overwrites EAX with EBX and here you add the second register into the first one.

24:52  
So basically instead of 1 instructions, you need 2 instructions.

24:58  
Finally increment this very trivial, very common operation.

25:03  
If you have a traditional for loop then usually you are incrementing a register somewhere inside the for loop typically.

25:11  
So this would be the increment in emips.

25:14  
In X86, yeah, you can still do the same thing, so you can do add one.

25:19  
But also X86 has this special instruction increment.

25:24  
And I think if you incremented EAX, I'm not entirely sure.

25:27  
Now it has some very, very short code, so I'm not sure if it's one byte or two bytes, but it's very short if you're incrementing EAX.

25:35  
Or maybe it's very short for ECX.

25:36  
I'm sorry, I don't remember that one, but that's not important.

25:39  
So they have a specific instruction for increment a logical operations.

25:45  
Yeah, I'm not going into the details because you already know and orcs or and negative or.

25:51  
What's that?

25:52  
Why do Why do you have we have negative or?

25:54  
Well, it's easy because it's more useful than have a simple negation.

25:58  
If you want to emulate a regular unary negation but bitwise negation so each bit in the word is flipped basically, then you can use the negative or, right?

26:11  
You just need to write the source register twice.

26:16  
Why?

26:16  
Well, this nor is a or operation which is negated afterwards.

26:24  
So if you take a value a any bit, this holds for any bit value a or value a.

26:36  
It's a right?

26:41  
Because if this is 1 and this is 1, then one or one is 1.

26:45  
If this is 0, then this is zero and zero or zero is 0, so applying or on the same value has no effect, right?

26:55  
But this is negative or, so the result is negated afterwards.

27:02  
So basically not a or a is not A and voila, we can simulate not operation using the NOR operation.

27:15  
So they have decided not to put there an extra instruction, an extra up code if you can do the same thing with the OR instruction.

27:24  
Yeah, a slight concern might be that this like performs additional operation, right?

27:31  
Because you need to perform the OR operation before you flip the bits technically.

27:39  
Well, as I said, technically, technically it's true.

27:42  
First of all, the hardware might detect it.

27:45  
So technically hardware can use optimizations that prevent that.

27:49  
So hardware can know that it's the OR is taking place on the two identical registers.

27:56  
So we'd like skip the OR part and just go with the NOT part with the negation.

28:03  
And 2nd, it's more beneficial to reduce the scope of the instruction set, so not put extra instructions that are really not necessary.

28:12  
So in this case they go with nor instead of having extra not the same.

28:21  
The analogy of these instructions which work which work on three registers is here and I or I and X or I is like in the addition or subtraction it's it takes 2 registers, the target register, source register and an immediate value.

28:36  
The difference here is that we are using the zero extension.

28:40  
The zero extension is like for unsigned numbers.

28:43  
We we always fill it with the zeros from the left instead of taking into account the the sign of the of the value.

28:51  
So the values here, the immediate values here are treated as unsigned values, which is much more sound for bitwise operations.

29:00  
For addition and subtraction you probably want both a positive and negative values, so it makes sense to have a signed immediate value.

29:07  
For bitwise operation it doesn't make that much sense, so it's easier.

29:10  
It's better to have unsigned operation here.

29:14  
Yeah, bitwise shifts are extra easy.

29:17  
So this is, these are just the names you get the register, the target register, source register, and this is an immediate value.

29:25  
This is a very awkward acronym.

29:27  
It's the shift amount and it says how many bits the number is shifted to the left or to the right.

29:35  
Easy.

29:36  
In shift operations, the shift is the result is just scropped.

29:41  
So if you have any extra bits on the left or on the right which overflows the register, they are just thrown away.

29:48  
However, you have also analogy to these two SRASRAA rotation, which is like a arithmetic shift.

29:57  
So it keeps the sign.

29:58  
If you're rotating, it tries to maintain the sign.

30:01  
So it's like a more like a division or sorry, this is this RA to the right.

30:06  
So it's like a multiplication by two.

30:08  
Yeah, that's right, sorry.

30:10  
Right division by two.

30:12  
Sorry.

30:13  
So this is like a division by two which preserves the sign.

30:18  
Doing that to the with the left shift doesn't make sense because if the sign is basically the highest bit, then any shift to the left will throw away the sign anyway, right?

30:29  
But if you're shifting to the right, then you need to fill zeros from the left.

30:33  
Usually this one doesn't fill zeros.

30:36  
This one preserves the highest bit.

30:38  
So again it copies the the highest bit to the filled in spaces.

30:43  
So you you get preserved sign for arithmetic operations.

30:49  
OK, again a slight comparison with the with the X86.

30:53  
So if you're using the NOR operation, NA this is basically a not operation because we are using the same 2, the same 2 operands.

31:02  
Here the same register is used as a source, first source and second source.

31:06  
So here this is an equivalent of placing the right value into the EAX and performing not EAX.

31:14  
Yeah, this is how you need to do this in X86 architecture.

31:17  
Yeah.

31:17  
And shift is pretty much easy because for shift we have extra the same instruction in X86.

31:22  
So we've already seen that memory access.

31:26  
As I told you, the the MIPS architecture is load store architecture, which means for every memory operation we need to place a specific instruction.

31:36  
We have load instructions and store instruction load for taking data from memory into the register store placing something from register into the memory.

31:46  
There are two types, one works with the entire register.

31:49  
So this these two work with words with 32 bit words because we are on the 32 bit architecture.

31:57  
So basically one register is loaded with 32 bit values from memory or vice versa 32 bit values from a register are written somewhere into the memory.

32:08  
This is a bit slightly bit confusing.

32:10  
So what does this mean?

32:13  
The Rd.

32:13  
is this is used here as a target for the load and here as the source of the of the store.

32:20  
So this is the register which holds the data and the Rs register is used for addressing.

32:28  
So you can use the Rs register, the second or operand and an immediate value.

32:34  
The immediate value is added to the value of this register and then together they are used as a memory address.

32:42  
Why is this so complex?

32:46  
The answer lies here.

32:49  
Let me just scoop back a bit.

32:52  
We have a registers that hold pointers to important areas in your memory.

32:59  
This one is for global data, this one or this one are for local data, for local variables.

33:05  
So if you want to read or write, store or load global or local variable, you will likely use one of these registers as base.

33:14  
But you don't want to read always the first local variable or the 1st global variable.

33:19  
Sometimes you need to read more of them and the compiler has chosen a way how to store them on the stack or how to store them in the global area where all the global variables are.

33:30  
So you know the compiler knows the offset of the variable from the stack pointer or from the global pointer.

33:37  
So basically what you need is that this value, sorry this value here the the register itself.

33:45  
The Rs is usually the global pointer or the stack pointer.

33:50  
And this immediate is a like offset of the variable within the global variables or within the local variables.

33:57  
This is a value which is usually automatically filled in by the compiler because the compiler knows how it stacks the variables in the memory.

34:06  
Similar here with LBNLBULBLBNSB.

34:10  
Sorry, the LBNSB are store and load operations for bytes.

34:17  
Yeah, you can.

34:18  
You can use a more elaborate way how to read a byte, right?

34:23  
You can read 32 bytes, 32 bits, 4 bytes and then use masking and for instance and mask to take only the lower 88 bytes, 8 bits.

34:34  
Sorry to load only single byte, but for some reasons, if you're operating with strings, it might be more useful to load a single byte or store a single byte to and from the memory.

34:48  
The semantics is this is very much the same.

34:52  
The difference is how the byte is treated.

34:55  
So when it's when it's written, it's pretty much simple.

34:58  
Because the store byte takes this register, then crops it takes only the 8 bits, and store only the lower 8 bits somewhere into the memory.

35:07  
That's easy, but what to do in case of loading operations?

35:11  
Because you're loading 8 bits from the memory.

35:14  
But again, these 8 bits needs to be stored in 32 bit register.

35:18  
So we need to extend it like like we did before.

35:21  
So we need to fill it from the left with zeros or with ones or with something.

35:26  
So again, we have two versions of this instruction.

35:30  
The LB is a signed version, so it extends the byte in assigned manner.

35:36  
So it takes the highest bit of the byte and copies it to the to the left, fills the remaining 24 bits using the value of the highest bit.

35:46  
And the LBU is for unsigned bytes.

35:51  
This is what you would usually do with normal characters, because if you have a character like an SK character, it has 8 bits, but it's not assigned value, it's unsigned value, it's from zero to 200-5055.

36:04  
So you usually use this to load a single character from the memory.

36:09  
OK moves are only in between registers.

36:15  
So the move instruction only copies value from 11 register to another register.

36:20  
Unlike in case of XA6 where the move is very very wicked operation that is used for store loads and and moves between registers.

36:28  
And one extra thing, there is ALE instruction.

36:32  
Actually, this is not an instruction.

36:34  
Why LE register an immediate 32 bits, loads this 32 bits, this exact immediate value into this register?

36:45  
Yeah, if I told you this, this is an instruction, a right flag should be risen in your head because this cannot be true, right?

36:54  
Why?

36:55  
Because an instruction has 32 bits in total.

36:59  
So there is no way we can smuggle 32 bit immediate value inside an instruction along with one reference to the register.

37:08  
So we have 16 bits at most.

37:11  
Yeah, this doesn't use an extra register.

37:13  
So we can, we can have 19 bits if you if we really want to, but that's it.

37:19  
We cannot smuggle more than 19 bits into instruction even if we sacrifice the first, the second opera.

37:26  
So how this works actually this is a pseudo instruction.

37:29  
It's like a macro and it's translated into two instructions.

37:34  
The one is called LUI and the second one is or immediate.

37:38  
So what does this mean?

37:41  
LUI is an acronym for LOAD Upper immediate.

37:46  
So it takes 16 bits, but these 16 bits are loaded as upper 16 bits of a register.

37:54  
So let me just visualize this, visualize this.

38:01  
This is a 32 bit register.

38:04  
It has upper 16 bits and lower 16 bits, OK.

38:09  
So normally if you're working with register, loading something into the register, like if you're performing the LB load byte or or LBU, the lower 8 bits are loaded OK.

38:21  
If you're performing the regular load, you load the entire 32 bits.

38:25  
Here the LUI users immediate value, so it can load only 16 bits.

38:32  
So normally you would assume that you are loading the lower 16 bits.

38:36  
But this LUI instruction is special because it loads the upper 16 bits, so it loads this half of the work of the register.

38:46  
OK, why don't we have any other instruction that loads the lower part?

38:53  
Why are we using this operation again?

38:59  
It would be waste of resources.

39:01  
If we have a special load like LBLB I, I would call it load button 16 bits or LLI low lower 16 bits.

39:12  
But that would be a redundant instruction because this part is zeroed when the LUI instruction takes place.

39:24  
So if there are zeros here, we can use or instruction to write anything we want.

39:30  
Because if there is some, some, any values here 01001 etcetera and use use or instead of assignment, it works the same way.

39:41  
Zeros will be merged to zeros, one will override the 0 to one right or operation.

39:48  
So that's why they're using an or instead of assignment just to save instruction.

39:56  
OK, comparison, quick comparison with with X86.

40:01  
Load, store load bytes, load intermediate, whatever you, you, you want.

40:08  
Is basically a morph instruction in X86.

40:11  
Yeah, it's slightly difficult to more difficult to understand because you need to read these brackets correctly everywhere.

40:19  
Every time there are brackets in the morph instruction, it means an address dereference.

40:25  
So in this case we are using the ebx instruction.

40:28  
We are adding something into the the ebx.

40:31  
It's like a base pointer.

40:32  
We are adding an intermediate value to this base pointer.

40:36  
And then these brackets indicate we are using this as a as an address which specified the target place in the memory.

40:43  
So this is a load instruction because on the left is the target, on the right is the source.

40:49  
So the source is in the memory and it's loaded into the EAX register.

40:54  
The same goes for the LV instruction on maps.

40:58  
Vice versa, if you just switch these operations, this is memory address, this is a register.

41:03  
So the value from the register is loaded, it's stored, sorry, it's copied to some place in the memory.

41:10  
OK, let's move on.

41:14  
Jumps.

41:15  
Yeah, that's the most difficult part of jumps and conditions are usually the most difficult part of any ISA specification.

41:22  
So let me just dig into it slowly.

41:24  
But fortunately for us, jumps on MIPS are not that difficult, where the the foremost and the most simplest is the jump to an specific address.

41:34  
So this is a very easy instruction.

41:37  
Sorry, very easy instruction.

41:39  
Basically add write specific address into the PC counter.

41:43  
That's it.

41:44  
This is an assignment of an of a value into the PC counter.

41:50  
Same goes for jump with register.

41:53  
So the value from a specific register is copied to the PC counter.

41:57  
That's it.

42:00  
3rd is jump and link and then that's basically a function call.

42:06  
Jump and link performs 2 operations.

42:09  
First PC counter is stored into the register 31.

42:13  
Remember I told you that before?

42:17  
Just give me a second register 31.

42:20  
By the way, you know the section 31 that that's how they get in the number.

42:24  
So the register 31 is used as a return address.

42:28  
It has a specific meaning.

42:30  
Actually, this is not only part of the ABI, this is also part of the ISA specification, because this specific instruction always works with the register 31.

42:43  
OK, so this register is used as a target for the PC counter.

42:48  
By the way, why?

42:49  
Why is the PC +4?

42:51  
That seems awkward.

42:52  
That seems suspicious.

42:55  
Exactly this.

42:57  
It's not the instruction which performs.

42:59  
The PC holds the address of this gel gel instruction.

43:04  
So we need to return after this jump.

43:08  
If this holds only the PC itself and we then use it as a target to jump back from the from the function call, we will end up on the gel instruction itself which will perform again the call.

43:21  
So we don't want that.

43:22  
So this is the address as you as you adequately put of the next instruction right after this jump.

43:28  
That's where we want to return once we return from the function call, right?

43:33  
And then it again performs the same as the normal jump.

43:35  
It's it's overrides the PC counter and and plays the code pointer somewhere else.

43:46  
Yeah, there is a slight difference here.

43:51  
That's what I want to point out.

43:52  
This jump instruction does not require any additional registers, any additional operands except for this immediate value.

44:03  
So we can use all the part, the whole part that's dedicated both to the registers and to the immediate value, which is 16 bits plus 5 bits plus 5 bits, though 26 bits as an address.

44:19  
That's quite important because this extends the range how far you can jump in your code, 26 bits.

44:29  
How far roughly can we jump in the code?

44:34  
How many kilobytes, megabytes, gigabytes of the code can we jump over off in the memory roughly?

44:45  
Well, if if it were 20 bits, it would be easy.

44:48  
SO20 bits is what for what 20 bits is MLAN.

44:54  
So it's a MB.

44:55  
So the six bits is Nope, 6 bits, sorry, 64.

45:04  
So it's 64 megabytes.

45:06  
So we can jump 64 megabytes ahead in theory.

45:09  
Actually it's assigned no, it's an absolute value.

45:12  
So in theory, we can we can X we can cover the range of 64 megabytes.

45:19  
If we need a larger range, no problem, we have still this.

45:23  
So we can somehow wrestle up any number, any 32 bit number here and then we can jump anywhere we want.

45:31  
But the first 64 bits are somewhat special.

45:34  
64 megabytes, sorry, are somewhat special because we can cover them using this address.

45:44  
Well, this is an absolute value in this case.

45:47  
In this case, this is an absolute value.

45:48  
You need to use this if you need to cover wider range.

45:51  
What if you needed company with wider like outside of the you can, it's a 32 bit architecture.

45:58  
You cannot address more than 32 bits anyway.

46:02  
So it's it's like the limitation of the architecture.

46:04  
Yeah, it relays.

46:05  
So you can have more than 4 megabytes of virtual memory.

46:09  
That's that's right.

46:11  
Your whole entire application must fit 4 gigabytes.

46:14  
That's it.

46:16  
Hey, I used to work with MS-DOS.

46:18  
640 kilobytes should be enough for everybody, right?

46:22  
So 4 gigabytes, yeah.

46:24  
If you don't like it, there are 64 bit extensions which use 64 bit registers and then you can cover then you can address 64 bit address.

46:38  
Well, some some, well, maybe not today, but I still lift in time when 32 bit operating system 32 bit windows were quite common and form 4 gigabytes of real physical memory was was a luxury.

46:50  
So it's not that unimaginable.

46:54  
And if you if you need the extension, then you need to extend everything from 6:32 bits to 64 bits and it it will work magically the same way.

47:02  
You just you just extend it.

47:04  
OK, again, quick comparison with X86.

47:11  
Yeah, by the way, normally we are not working with the explicit addresses in our code when when you write assembly language, we use labels.

47:20  
So the label marks specific position in the code and later the compiler will decide where this actually resides in the memory.

47:28  
Actually it's slightly more complicating than that, because sometimes it might be placed in different places in the memory.

47:34  
Your code might be relocated so it it's slightly more complicated.

47:38  
But let's just imagine that somehow compiler decides how to place the right addresses there at at this very moment.

47:45  
OK, so jump to the label is easy because the label is translated to some memory.

47:49  
So jump label means jump specific address that's filled in by compiler.

47:54  
And the same goes for X86, There is no surprise there.

47:57  
And again, for the instruction, the reference for the register, the reference, sorry, there the instructions pretty much the same.

48:04  
You can you say register as the as the address and the only difference is that they use call instead of the the JL they have a specific function call.

48:15  
Actually it this it's much more complex because the call instruction stores the return address to the stack automatically.

48:23  
And the reason this this architecture can do do that is the X86 specifies which register is the stack pointer.

48:31  
So the stack pointer is automatically incremented.

48:34  
The the register is used as a dereference.

48:36  
So it's it's quite more complicated instruction.

48:42  
By the way, this is a trap.

48:44  
It's not entirely true what I said.

48:46  
This is a much more complicated.

48:49  
This is a double in direction.

48:52  
The register ebx holds the address where the pointer is stored.

48:59  
So this will load whatever word is at this address stored in ebx, and then this address loaded from the memory is used as the address where we should jump double indirection.

49:13  
Yeah, I know.

49:15  
If you didn't catch that, never mind.

49:17  
It's just just remember it's more complicated on X-80 or this.

49:21  
This example is more complicated than than this one.

49:24  
This is straightforward.

49:25  
The address from the register is used as the target where we are jumping.

49:32  
OK, jumps are easy.

49:35  
Let's go easier.

49:37  
Let's go to conditional jumps.

49:39  
Conditional jumps require some form of testing, so we need something to compare before we jump.

49:46  
In the MIPS instruction set there is A.

49:48  
There are two jumps actually to distinguish themselves from the jump instructions, which are like a straightforward we are always jumping.

49:57  
They are called branches because they are conditional.

50:00  
Sometimes we take the branch, sometimes we take the other branch.

50:04  
So they start with B not with J.

50:06  
And beq is a current name for branch if equal.

50:11  
So if the two registers, sorry, if the yeah, if these two registers I'm placing there are equal hold the same value, then I'm performing the jump.

50:21  
If not, I'm merely following within with another instruction.

50:25  
So if the equality holds, then the PC, the the program counter is overridden with this address.

50:32  
If it doesn't hold, we just increment the program counter, just following up with the next instruction.

50:38  
Analogically for BNE, the NE holds for not equal.

50:43  
So this is like opposite.

50:45  
This is the negative of this instruction.

50:47  
Yeah, I, I several times I prize myself with pointing out the fact that they're trying to save the instructions and suddenly they have two almost identical instructions for performing branches, right.

51:02  
Yeah, there is a reason for that.

51:05  
Very often you are comparing something with 0 and very often you need to jump if it's zero and very often you need to jump when it's not zero.

51:15  
And the zero is easy because 0 is already there.

51:18  
It's in the zero register register.

51:20  
It's the first register, right?

51:21  
So one of these two items is very often the zero register.

51:25  
But it will be difficult to have another register with non 0 value.

51:29  
So it's easier to have two instructions 1 when you use if you want to branch if something is 0 and the other one if, if it's not 0, that's that's the, that's the logic, that's the idea behind this.

51:42  
And for testing, for inequality testing, we have a special instructions SLT SLTU that depends on whether we are comparing unsigned or signed signed versions.

51:55  
And you get.

51:58  
It's slightly more complicated because you get the destination register where the result of the comparison is stored.

52:07  
So you're performing lesser than.

52:09  
The SLT hold stands for lesser than.

52:13  
Rd.

52:13  
holds a boolean value of this comparison between Rs and RT.

52:19  
So if Rs is lower than RT, then Rd.

52:23  
will get one true.

52:26  
If this doesn't hold, it will get a zero false.

52:30  
OK, this is basically and you can rewrite this in AC because if you have like A, if A is lesser than B, you can extract the condition and you can have something like You can do that in C as well.

52:55  
You can take out the.

52:57  
You don't have to mangle comparisons with if statements or any other statement.

53:03  
You can extract the logical operator, the comparison operator, and store the result here.

53:08  
So technically that's what happening here.

53:11  
You first perform this comparison, store the result into the register, and then later you use these two to put this result equal or not equal to 0, right?

53:25  
So if you want to perform this type of if it breaks down into two instruction, 1 instruction performs the comparison itself and the second instruction perform the jump if the comparison was successful or not.

53:40  
OK.

53:42  
Yeah.

53:42  
There are some slight details like whether we are working with onsite or unsigned, whether we are working with two registers or a register or an immediate value.

53:52  
But it's just a variation on the theme.

53:56  
By the way, we don't have greater than, we have only smaller than.

54:05  
Is that bad?

54:09  
No, Why?

54:11  
Yeah, we can swap and that's right, it's easier just mind teaser.

54:15  
So if we if you want greater than then you just swap these two and that's it.

54:19  
What if we need greater or equal or lesser or equal in this manner?

54:31  
Well, yeah, 111.11 option would be to use two instructions, but that's a bit impractical.

54:39  
Any other and same?

54:44  
OK, not really.

55:09  
And this comes free, Why you just switch from this one to this or vice versa?

55:17  
OK, so you just need to hold all the details in your mind because if you're writing this manually, it might be a bit tricky not to get lost in the negations and which which one of these you want to use.

55:31  
But in the end it it's not that not that difficult.

55:33  
You just need to use appropriate combination, proper order of these two and proper selection of these two and you will get you, you can get all the types of inequality comparisons you can imagine.

55:48  
OK, yeah, X86 goes around this in a very different way, by the way.

56:01  
By the way, just a small teaser.

56:05  
Could you imagine how to implement this?

56:08  
Could you, could you do that still if I give, if I take these instructions away from you?

56:14  
So we still need to perform lesser than, greater than, but I can I tell you that you don't need these instructions to do that.

56:22  
What instructions which you already have, which you already know, can you use as a supplement, as a, as a, let's say, replacement if somebody orders you, you know, Trump gives you a new order, new law that you cannot use SLT and SLTU instructions.

56:40  
So what shall we do with that?

56:44  
Yeah, we can move to Canada, but that's that's that might be more practical.

56:49  
That's right.

56:50  
But just imagine for a short while that you can't move value, then you can like order the check to be or you can use and to clear out everything except the design bit.

57:06  
And then that's right, That's right, you're on the right track.

57:11  
So you can use subtraction.

57:13  
And basically this comparison underneath perform subtraction because if you subtract these two values, then you will end up with a 0 or positive value if this one is lower than, sorry if this one was greater than this one.

57:32  
Or vice versa, you will end up with a negative value if this one is smaller than this one, right?

57:38  
If you just put minus in between them instead of comparison, right?

57:44  
And we already know that in to complement, we can just take a peek at the highest bit.

57:51  
That's probably what you meant by the ends.

57:53  
And so basically, we don't have to use the ends, We can just shift the number 31 positions to the right, which will clear the upper 31 bits and only the upper bit will remain on the right.

58:06  
So basically, if we just replace it with one subtraction and one shift to the right, we will get the same result.

58:13  
And that's probably what's happening underneath in the hardware, maybe in slightly more optimized way, but basically that's it.

58:20  
OK.

58:21  
The same goes here.

58:22  
The BQ is basically a comparison between these two and then jump Z.

58:27  
By the way, this is the CMP instruction is a special version of sub instruction.

58:34  
That's why I was was teasing you with this, this small mind teaser with the replacement of the lesser than with the subtraction.

58:44  
Because CMP is basically a subtraction that doesn't store the the result.

58:49  
It only alters the flags.

58:51  
In X86 the result of the comparison is stored in flags.

58:55  
I will get to that in short order.

58:57  
And the jumps are reading the flags and jumping based on some certain conditions expressed in the flag register.

59:04  
So the CMP is subtraction without the store.

59:07  
So it it likes perform EAX minus EBX, but it doesn't store the result, it doesn't overwrite the EAX, it's just store the the flags.

59:17  
By the way, subtraction alters the flags in the same way, it just writes the the result here.

59:23  
And the Jay-Z says jump IF0.

59:28  
Wow.

59:29  
Why jump if zero?

59:32  
That sounds awkward.

59:38  
I'm just waiting if every all the pieces I just told you just kind of fit together and fall into the right image in your mind.

59:45  
It's telling me that there's return to 0 if they are No, no, no, no.

59:55  
Actually no.

59:55  
This is this is a subtraction.

59:59  
It just doesn't store the value into EAX, but it alters all the registers in the same manner a sub instruction would.

1:00:08  
So like the indications of the result of the sub instructions are somehow stored in the registers.

1:00:16  
And this is jumping if the previous result was 0.

1:00:22  
So when the result is 0.

1:00:27  
It's zero when they're equal.

1:00:29  
It's zero when they're equal because if you take a value and subtract the same value then you will get the zero.

1:00:34  
If they're different, you won't get the zero.

1:00:36  
So basically that's it.

1:00:38  
The JZ is the same as equality because this is a subtraction and if these two numbers were the same, the result would have been zero.

1:00:46  
It's not stored in the EAX because this is compare instruction, but it's indicated in the flags.

1:00:51  
There is a 0 flag actually, so it will be stored there and this jump will just read the flag and say hey the flag is set, I'm jumping or if it's.

1:01:00  
If they are not equal the flag wouldn't be set and this jump wouldn't take place OK.

1:01:05  
So this is the equivalent of beq branch if equal.

1:01:10  
OK, for inequality comparisons you need to use SLT and then branch if equal or branch if not equal.

1:01:20  
So this is just the hey, you can do this for your home assignments.

1:01:23  
Again, you perform the same comparison here but you change the type of of jump and this is jump if lower.

1:01:31  
So again, it takes a look at the register the right flags and if the flags indicated that the result of this subtraction was less than 0 in the end, then it performs the jump.

1:01:44  
The same goes here, OK.

1:01:46  
And you have a whole set of these instructions JLJLEJGJGE which is a combination of lower than lower equal, greater than greater equal.

1:01:57  
And it all takes the the notions from the flags.

1:02:00  
Always you are using the same compare instructions before.

1:02:04  
OK, finally let us go through a few easy, maybe not completely easy, but not that complicated or the complex examples.

1:02:19  
The first one is how this for loop would be probably unoptimized in an unoptimized way translated into the instructions.

1:02:31  
So it's a very simple loop.

1:02:33  
We just have the EI, sorry, I variable, which is the local variable.

1:02:39  
Probably this one will be optimized so it will be stored in a register because we don't need to store it somewhere in the memory.

1:02:45  
The N is like the size of the array.

1:02:47  
It's a probably global variable.

1:02:50  
And a is the pointer to the beginning of the array because we are in C so this is a pointer to the beginning of the variable.

1:02:56  
And again, I'm assuming, yeah, that there are global variables.

1:03:02  
And what we are doing is that we are filling this variable with a particular constant like 42, which is a very important number, as you already know.

1:03:10  
So we are filling the number, we're filling the array with the initial value.

1:03:13  
OK, easy enough.

1:03:16  
So what we are doing here, first step Eddy, we are storing T0.

1:03:26  
We we need to sorry to the T0 which is like a temporal register.

1:03:30  
We are storing the address of a.

1:03:35  
Who the hell told me that that the array a the actually the pointer not the array itself is at value 28.

1:03:49  
How did I get to that?

1:03:57  
Exactly.

1:03:58  
I, I wouldn't know that actually I took this, took this code for granted from I just wrote a much more complex code.

1:04:05  
Well, not that much complex, but the code was slightly larger because there was the main function and the declaration of global variables, etcetera, etcetera.

1:04:12  
And for some reason compiler decided that this this would be a good spot for the variable a.

1:04:17  
So it place it there and I cannot do any heck with it.

1:04:21  
It's just compiler decision.

1:04:22  
So I'm just giving you the output of the compiler.

1:04:25  
Compiler says it will be here.

1:04:27  
So it's it's there.

1:04:28  
That's it.

1:04:29  
OK then I'm loading the word where the NSN is and again N is the value 4.

1:04:36  
This is just a strange notation.

1:04:38  
This 4 for some reason is written in front of the GP register.

1:04:42  
It's like we are using the 4th offset of the global variables.

1:04:46  
GP is the global pointer.

1:04:48  
So this is the just GP plus 4 which is used as the address and that's loaded to T1.

1:04:55  
Finally we are moving the 0 value.

1:04:57  
Hey, the zero register becomes useful again.

1:05:00  
So we are moving the 0 value value to the T2 that that's a much quicker initialization than if you just had to perform something else.

1:05:09  
By by the way, do you know how to 0 a register quickly?

1:05:14  
If we wouldn't have the zero register, how to place a 0 into another register like T2 quickly without using the zero register?

1:05:30  
Yes, we can use end end with zero and we we have end I which uses immediate value 0.

1:05:37  
So yeah, one one option would be to end 0 to the register that will clean it up.

1:05:42  
That's right.

1:05:43  
Any other options?

1:05:50  
Not not Yeah, we can do lots of things with with zero.

1:05:54  
That's that's right.

1:05:55  
Well, not, not or one wouldn't work because that's you're you're not or you are oring 0, which doesn't change the number and then negation would flip it so it wouldn't zero it.

1:06:06  
So this one wouldn't work.

1:06:08  
But I I heard something else bit shifting.

1:06:11  
Yeah, we can shift it 32 bits to the right, which pretty much clears it that that's right.

1:06:16  
Any other option.

1:06:17  
And that's good because you're thinking like real assembly programmers.

1:06:21  
Now, one final one.

1:06:24  
It's just one which is my favorite because it was used back in my days when I was learning assembly language.

1:06:35  
That's right, saw it with with itself or this was like one of the my most favorite instructions back in my days because this was the this was the fastest way how you can 0A register or it was at least on X86 architecture because it's much, much shorter than writing than moving a 0 into EAX because the zero would have to be written as an immediate value in the code.

1:07:05  
This is this takes only two bytes.

1:07:09  
OK, so we are initializing TT 2.

1:07:12  
So basically from now on in T2 we have the I variable, so I is aliased in our minds to T2 and TT3 will for our purposes hold the immediate value here, which is the constant we are using to fill the array.

1:07:30  
Yeah, it's a good idea usually to place this value somewhere into the registers because it's much faster to use register for many operations than to use it for other things.

1:07:41  
OK, so we are using the 0 and 42 to get odd together and written to a register again.

1:07:48  
Or I is one of the fastest way how to initialize the lower 16 bit of some some other register.

1:07:53  
Actually all the all the registers because the whole register T3 will be zeroed except for the lower 16 bits which are the result of this or operation.

1:08:04  
And finally this this, this is where it gets more interesting.

1:08:06  
This is just an initialization.

1:08:08  
We are jumping to the condition.

1:08:11  
Whoa.

1:08:12  
For some reasons, the compiler decided to place the condition, the condition which is tested at the beginning of each loop, at the end.

1:08:21  
Yeah, that's compiler's paragraph, right?

1:08:24  
So before we start, we jump here and here there is a comparison.

1:08:30  
The SLT will compare I and N Remember I is in T2 and N is in T1.

1:08:38  
So we are comparing T1 and T2.

1:08:41  
SLT is lesser than.

1:08:42  
So just imagine lesser than between these two.

1:08:44  
This is exact copy of this because this is an and this is I, right?

1:08:49  
And the result of this comparison will be in a temporary T4.

1:08:52  
So in the end we are comparing T4 with 0.

1:08:55  
Again, zero becomes very useful.

1:08:58  
And if this holds, we are jumping to the body, to the body of the loop actually.

1:09:05  
And if it doesn't hold, we are continuing over here.

1:09:09  
And that's basically where the end, where is the end of the loop.

1:09:13  
So if if this doesn't hold, we are merely continuing on our way, not executing the body itself.

1:09:19  
OK, And the body itself holds these values.

1:09:23  
First of all, we need to multiply I with four.

1:09:27  
So what does this hold?

1:09:28  
Why, why, why there's no multiplication?

1:09:30  
What?

1:09:31  
What is SSLL shift left?

1:09:37  
Yeah, because multiplication by 4 is actually shift to the left by two, right?

1:09:43  
Because this is 2 to the power of 2 is 4.

1:09:47  
So shift to the left by two is the same as multiplied by four.

1:09:51  
And since this is a fixed constant, it's pretty much easier to do it by shift.

1:09:56  
Actually there is a multiplication instruction.

1:09:58  
I haven't told you about it because it's much more complex, because the result of of the multiplication needs to be stored in two registers.

1:10:05  
So it's pretty pretty tricky.

1:10:08  
But for this purpose, shift is much more better.

1:10:11  
And then we need to add the offset of I \* 4, which we just computed to the a base.

1:10:17  
Why?

1:10:17  
Because this actually needs to be resolved in the manner of pointer arithmetics, right?

1:10:25  
Pointer arithmetics dictate that assuming this is the array of integers and integer holds 32 bits.

1:10:33  
So this This is why we get 4 here.

1:10:35  
So we need to add the offset of I to the base pointer A to get the proper address of the proper spot to which we are writing right now.

1:10:44  
This is where this addition comes.

1:10:47  
Store word uses the previously computed address T4.

1:10:51  
So this is the temporary variable which holds the actual address of the actual item in the array which we are writing right now.

1:10:59  
So the store word will use this address and a zero offset because we are really want to put store to this address and the T3 holds our 42 variable the constant right?

1:11:12  
So we are storing T3 into this address.

1:11:17  
And finally we need to perform the last piece of the iteration, which is this increment.

1:11:23  
And that's pretty easy.

1:11:24  
This is just an addition, right?

1:11:27  
So yeah, now I dissect it line by line, this code.

1:11:31  
Are there any questions regarding this this instruction?

1:11:42  
Give it a decade or two, you know, it's not that complex.

1:11:46  
You just need to read a few few a few programs and then you will get hang of it.

1:11:50  
Probably not when I'm reading it, I still need to read it line by line.

1:11:55  
It's not that I'm just taking a glimpse at it and I immediately tell you, oh, it's this for loop.

1:12:00  
No, I I also need to examine it closely because no human possibly, or maybe there are some humans, but regular programmers don't read assembly fluently.

1:12:10  
So don't don't feel bad that you're don't reading this fluently, but you, well, it's, it's a slow work.

1:12:18  
You need to think about it a lot.

1:12:20  
And it's much slower than if you're writing in AC for instance.

1:12:23  
But that's why they invented C and and high languages, because nobody wants to write an assembly.

1:12:29  
Actually, normally you need to write an assembly only if you're doing something very low level, like writing specific drivers for specific devices or writing some very tiny pieces in in a OS kernel.

1:12:40  
But normally you don't have to write assembly nowadays.

1:12:45  
OK, another example.

1:12:46  
Much more simple.

1:12:47  
Simpler, but much more tedious.

1:12:49  
Much more tricky.

1:12:51  
Sorry, but maybe tedious for some.

1:12:55  
This is a simple expression.

1:12:57  
N multiplies by 5 + 3.

1:13:01  
Nothing.

1:13:01  
Nothing fancy there.

1:13:03  
So what we need to do is fetch N because this is probably some global variable.

1:13:08  
Again, we are using GP and this is a constant generated by compiler which just happened to place this variable on the offset 4.

1:13:16  
We don't know why.

1:13:17  
So we store it to T0.

1:13:19  
We store 5 to T1, so we get 5 ready in a register so we can use it, use it easily for operations.

1:13:26  
And then there is this multiplication instruction I told you about, but I didn't explain in detail.

1:13:32  
So just at this moment, it's actually a bit fake.

1:13:34  
It's simplified.

1:13:35  
It's not really exactly how it's written in MIPS, but just bear with me.

1:13:40  
So we multiply T1 with T0 and store it into T0.

1:13:45  
OK, pretty much easy.

1:13:48  
That's why there's the fake means this is not an actual instruction.

1:13:51  
It's, it's more complicated, but just let's let's assume this works.

1:13:55  
It's, it should work if, if we don't don't need to store the upper 32 bits.

1:14:01  
And finally we need to perform add, add operation to T0 the three and that's it.

1:14:06  
And we store the result into the I variable, which is for instance, in the 0 offset of the GP.

1:14:14  
Easy enough.

1:14:16  
But I can tell you this could be also a code generated by the compiler for this, for this particular expression.

1:14:24  
So what we are doing here where we fetch the end in the same way?

1:14:28  
Yeah, that's pretty much obvious.

1:14:30  
Then we multiplied by 4.

1:14:34  
Why?

1:14:35  
Because shift to the left is much faster, much easier, much, much simpler to handle then multiplication.

1:14:43  
Imagine that this is a very complex, very heavy operation for the, for the CPU.

1:14:47  
It might not even take a single, a single cycle, it might take multiple cycles.

1:14:52  
So it's like a, it's stalling the pipeline.

1:14:54  
It's it's very heavy.

1:14:55  
Shift to the left is very simple.

1:14:58  
It's it's easier than.

1:14:59  
Then we add N to the previous result.

1:15:06  
So if we multiply N by 4 and then add one additional N, it's the same as if we multiply the N by 5, right?

1:15:17  
Yeah, that's a trick the compiler is using all the time because it takes the advantage of the fact that the five is here as a constant.

1:15:27  
So if it's a constant, then the compiler can dissect it in many ways.

1:15:31  
And this is one of the tricks that the compiler can do to optimize the code.

1:15:36  
So we completely avoided the multiplication, right?

1:15:41  
And basically we are using the same amount of instructions as here, except we have avoided the costly multiplication, right?

1:15:51  
So imagine that you already took notion of that.

1:15:54  
Here we are not multiplying by 4 here we are using shifts whenever we can.

1:15:59  
The compiler is actually doing that.

1:16:02  
OK, so that was another example.

1:16:09  
Few more things before we wrap up.

1:16:12  
First one, I told you several times about the flags register when we was talking about comparison.

1:16:18  
So let me just give you a glimpse at the register.

1:16:21  
This is how the register looks at X86.

1:16:24  
The orange flags are the arithmetic flags.

1:16:27  
So these are the flags adjusted, modified by every arithmetic operation.

1:16:32  
And basically what every arithmetic instruction does is it fills these these registers based on what was the result of the operation.

1:16:43  
So if we're performing sub comparison is the same as if we're comparing subtraction.

1:16:47  
As I already mentioned, the most important flags are zero flags, Z flag.

1:16:53  
Over here it's set only when the result is 0.

1:16:57  
It's good to know if it's 0 a sign flag as as here is basically the copy of the highest flag of the result.

1:17:05  
If it's signed operation OK, because if it's if the highest bid is 1, then the sign flag is one because the result was negative otherwise it's zero, which means it's positive carry flag.

1:17:21  
That's one of the most important ones actually.

1:17:24  
Is a flag which is set if the operation has one extra bit that needs to be, you know, transferred into high order.

1:17:35  
For instance, if you're adding two numbers and the result overflows, then the extra bit goes to the carry flag.

1:17:44  
Also, if you're shifting to the left, the lowest shifted bit is stored into the carry flag.

1:17:52  
Actually it's not entirely true.

1:17:53  
It's it's true only for rotating operations I think, which are similar to shifts, but it's not that important.

1:18:01  
An overflow flag is set when the operation overflows.

1:18:04  
So it's similar to carry, but the the overflow is more general.

1:18:07  
The carry can still hold 0 because the operation might overflow much more than 2210 above the the width of the register.

1:18:17  
OK, so these are the flags few more things about Isas.

1:18:24  
The instruction set architecture can be designed in many, many ways.

1:18:29  
There used to be some classification about types of these Isas.

1:18:34  
Namely you probably heard about the acronyms TISK and RISK.

1:18:38  
This one stalls stands for the Complex Instruction set computer and the one.

1:18:43  
The other one is for the reduced set instruction set.

1:18:47  
Nowadays RISK and CISQ is somewhat losing their meaning because in the past when I was studying these subjects, there was there were CPUs like ARM, for instance, or Motorola or maybe even MIPS, which would be denoted risk because they have quite small instruction set and they're aiming at being very, very performant.

1:19:09  
So they they're trying to decode the instructions very fast.

1:19:11  
They're trying to do the best they can to process the reduced set they're actually can process.

1:19:19  
And the complex instruction set was using this specific notion that it's a good idea to give a broad variety of different instructions and instructions, versions and types and variations to the programmer because the programmer would could use the specific instruction for a specific problem, right?

1:19:40  
Actually this is no longer true because as I just mentioned a few moments before, we are not using assembly language to write software anymore, only in a very very specific occasions.

1:19:52  
Normally the compiler generates the assembly code, so why bother if the difficult part is done in the compiler?

1:20:01  
We don't need very complex sets of instructions to accommodate needy users, needy programmers.

1:20:08  
The compiler can do that.

1:20:10  
So nowadays we are focusing more on narrow subsets of of instructions and even the architectures which were like.

1:20:21  
Mentals or or examples of tisk approach like X86 architecture is nowadays reducing the number of instructions it directly support.

1:20:32  
I will show you that on a few slides later.

1:20:35  
So they are supporting massively a very small subset of instructions that should be used regularly.

1:20:42  
And yeah, they are usually supporting the other instructions for for compatibility reasons, but they usually execute these other instructions in a very slower manner.

1:20:53  
So it's a good idea to use only a subset of this Tisk instruction set.

1:20:59  
By the way, there are also other different approaches to that.

1:21:02  
One of them is a very long instruction word approach which tries to avoid the decoding of the instruction by writing all the important stuff like which units should be initiated, which code should be, which pieces of hardware should be initialized in which order into a very long bitwise word.

1:21:25  
Let me scroll very quickly to give you an example of what I mean.

1:21:33  
Last week I showed you this example.

1:21:38  
And in this example, this instruction is decoded into a sequence of internal micro instructions or internal steps if you will.

1:21:47  
And this is basically a specific list of operations that needs to take place.

1:21:52  
And I can encode this in a very long bit, twice in a very long bit word where the individual gates will have their individual bit sets when they need to open or something like that.

1:22:07  
Yeah, I'm, I'm far fetching this.

1:22:09  
It's not that simple as I'm putting it, but just to give you a glimpse how this might work.

1:22:14  
So I can give you an instruction which is like a partially or totally decoded and written in a very long bit vector.

1:22:25  
And this is what these types of architectures aim to do.

1:22:32  
However, it's turned out not to be very practical.

1:22:35  
The decoding can be done quite easily even for complex architectures, at least nowadays when we have enough silicone, enough circuitry on the wafer, so it's no longer a very viable option.

1:22:48  
It was still used for specific architectures, like I think it was in AM DGPU still 2010 or something like that.

1:22:56  
Then they switched to CGN architecture and they they stopped using this particle approach.

1:23:03  
Slightly better way how to do this was the Epic.

1:23:06  
This has nothing to do with rare or legendary, this is just an acronym.

1:23:11  
This is like explicitly parallel instruction computer, which means they try to bundle multiple instructions into one instruction word or instruction set.

1:23:21  
This is what for instance, IIA 64 which I which I told you about when I was talking about registers is doing and they basically place multiple instructions in what's called instruction bundle.

1:23:36  
And these instructions are designed so that they can be executed concurrently in parallel.

1:23:42  
So compiler is helping the CPU with concurrent execution of instructions whenever possible.

1:23:50  
Another important aspect which we are examining when talking about ISS is orthogonality, whether a specific register can be used in specific operations.

1:24:01  
If they are completely orthogonal, then any register can be used for any operation.

1:24:07  
So if you have an addition, multiplication, storing some pointers, whatever, you can use whatever register you would like.

1:24:14  
MIPS 32 is almost orthogonal or terrible word.

1:24:21  
Sorry, my tongue is on vacation today.

1:24:24  
So it's almost orthogonal, not entirely because you have the zero register which is special, and you have the 30 ones register where the address is stored for the jumps for the calls.

1:24:35  
So it's not entirely but almost on the other way.

1:24:38  
On the other hand, X86 is almost not orthogonal because there are so many instructions where only specific one register or specific set of registers can be used.

1:24:49  
And finally, I told about this several, I mentioned this several times load, execute, store.

1:24:54  
So whether we are using load and store approach to load the data and store the data or we are using a more straightforward approach like morph instructions which or other instructions that directly address the memory, OK.

1:25:09  
And before we conclude bad timing, OK, I would need few more minutes than 5 and I have only 5 minutes till the end.

1:25:21  
So I just started and I will need to retake this topic at the end of the at the beginning of the next weeks.

1:25:28  
What can what can you?

1:25:30  
So at the end I would like to give you a few glimpses, few notions, few hints about how this is implemented in hardware.

1:25:39  
Actually, we are not talking about this in detail.

1:25:41  
If you want to know this in detail, you need to pick another school, because we are not doing hardware things on that level here.

1:25:47  
We are more software oriented.

1:25:49  
But it's a good idea to know a few things about how it really works, or at least how it appears it works in the real hardware, because then you can optimize for it.

1:25:58  
Then you can design better software for for the particular CPU or for in general for for all the CPUs.

1:26:06  
So basically we need to know how the memory is organized.

1:26:12  
There are usually some set of.

1:26:14  
There are sets of caches that pre phase the memory so that you can you can store.

1:26:19  
The hardware automatically stores data there so you don't have to waste time waiting for the data to get from the main memory.

1:26:27  
Or when you're storing the data, they can be cached in the caches so they then don't have to be written back to the main memory if you use the data frequently enough.

1:26:37  
CPU cores have their own registers, and actually, yeah, this is specified by ISA.

1:26:43  
But the important thing is this doesn't have to match with the physical registers.

1:26:47  
For instance, X86 architecture performs something which is called register remapping.

1:26:54  
You still have the registers in your code named EAX and EB etcetera, but in physical world you have like 100, sixty, 180.

1:27:04  
I'm not entirely sure about the number registers on the CPU available and there are.

1:27:10  
There is extra circuitry that renames the register that maps the physical registers to the virtual registers which are named in the ISA.

1:27:18  
So your code can run differently, meaning your data, when they are stored in a specific register like EAX, can be stored in a different physical register.

1:27:31  
If the hardware decides to do that, your code wouldn't know that because your code would appear as if it's still the EAX register for you, but for the hardware it will use a different register.

1:27:44  
Why they are doing that?

1:27:46  
Because we have so few registers and sometimes it's better for performance if you can store the data in one register and then use another register as an alias for different instruction.

1:27:59  
So you can run these instructions in parallel for instance.

1:28:02  
OK.

1:28:04  
And finally there are technologies like hyper threading which actually creates virtual cores.

1:28:10  
So according to ISA, you, you have a say, virtual view of the CPU that it has several cores, but in fact, some of these cores are merged together and using the using the execution units share the execution units.

1:28:27  
OK, I think this is a good place where, where we can stop for, for now, because this is a picture I will remember for the next time.

1:28:34  
So before we conclude, I think we have like a one minute left.

1:28:37  
Are there any questions regarding the instruction sets or anything?

1:28:41  
We we were covered this, this week, this this lesson, No questions so far.

1:28:49  
OK, so next week we'll finish the we will finish the CPUs.

1:28:54  
It's just only a few slides left and we start a new topic.

1:28:56  
So thank you for your attention and see you next week hopefully.