**Architecture 7-3 2023-24**

0:00  
Hello and welcome back to architecture. So for the next couple of videos, I'd like to pivot away from Hack specifically and talk more about architecture in general, and about some concepts that you'll see in other instruction set architectures that you don't see in Hack because they're a bit too fiddly to implement.

0:20  
So to start off with, if you remember I said all the way back when we introduced Hack that you'll never actually use Hack for a serious project. But by learning Hack, you'll pick up the skills you need to pick up pretty much any other assembly language.

0:36  
So let's say you're looking at a new assembly language. Let's say you're looking at a new instruction set architecture. What sort of things should you be looking at

0:45  
to immediately get a good sense of the fundamentals?

0:51  
So one of the simple things is just how large the words are in memory. So Hack uses 16 bit words.

0:59  
You might remember a few years back computers started switching from 32 bit to 64 bit.

1:09  
That meant the word size. CPUs switched from using 32 bit words to 64 bit words.

1:19  
Another thing to look at

1:21  
is the address space.

1:24  
So Hack again uses,

1:28  
um, 16,000

1:30  
odd memory addresses in RAM,

1:34  
plus some memory mapped addresses.

1:38  
A modern CPU would use something significantly more, and not all of them use the full 64 bits.

1:47  
So for example, the most popular instruction set architecture for desktop computers actually only allows for a 48 bit address space.

1:58  
But with 64 bit words, that's still over.

2:03  
That's still over 2 to the 50 bytes. That's still 2 to the 51 bytes,

2:07  
which is around 2 petabytes of memory. So that's still. We're not bumping into that yet. Really.

2:15  
Another thing to look for is instruction length. So hack here is actually quite unusual in that one instruction is one word of memory, and you might remember that actually causes us a couple of issues with the A instruction

2:30  
on. The fact that an A instruction is only

2:34  
16 bits

2:36  
and we need it to load a specific value into memory

2:41  
means that we can only actually use it to load a 15 bit value into memory because we need that first bit of the instruction for the opcode.

2:53  
So modern CPU users are very often have instructions which are multiple words long specifically so that they can have multiple words as operands to their instructions.

3:09  
So there are some other things here to look for which are a lot less simple and which we're going to cover in the rest of this video.

3:17  
And there's also

3:20  
normally something called the stack, which is implemented in hardware. So for us in Hack, we're going to implement the stack in software and we're going to see it as we get towards the end of the unit in weeks 9 and 10

3:33  
um. In other races, this would actually be done in hardware, and you'd see it in the ISA itself.

3:42  
So there's one other thing that you will immediately notice on looking at pretty much any assembly language other than Hack, and that's that the syntax looks different.

3:53  
So what they normally do is they have one word for each possible instruction type,

3:59  
and the syntax is that word and then all the operands of the instruction.

4:07  
So for example, rather than writing A = D M, it might write add a DDM,

4:16  
I mean add and M and store the results in A. And that's what it would look like in MIPS, which we'll talk a bit more about later in this video.

4:25  
So this is entirely cosmetic, and the reason Hack does things this way

4:30  
is just that Hack only has two kinds of instructions, right? It has A instructions and C instructions

4:37  
and it uses the way it calculates C instructions allows it to calculate all these different operations.

4:49  
So another thing you see in Hack is this Harvard architecture.

4:54  
So Harvard architecture means that, exactly like Hack does, you have separate ROM and RAM.

5:01  
Your room stores all the instructions with your programme data in it

5:08  
to your piece. Your programme counter holds an address in rock, and then your RAM stores all your variables and all your inputs, and it's where your outputs go.

5:20  
And this is what makes Harvard architecture, the fact that you've got separate storage for programme and data programme, Enron data in RAM.

5:31  
Most modern CPUs don't work this way. Most modern CPUs, what's called. Most modern CPUs use what's called Von Neumann architecture.

5:41  
Turn 1 Neumann architecture. You just have one memory storage for everything, but programmes and data are like in its RAM.

5:53  
Pretty much everything else works the same,

5:57  
so you can still think of the CPU as sending the programme counter value into RAM, getting an instruction back,

6:07  
potentially reading from RAM, literally writing to RAM.

6:13  
All that still works in a pretty similar way to how it does with Harvard architecture,

6:21  
it's just that it requires a bit more finesse on the hardware level.

6:28  
Um, either via pipelining, which we'll talk more about next video, or via

6:36  
just a more sophisticated input and output method, the memory that would allow you to actually read multiple words from memory at once, for example, read both the instruction and whatever value the CPU is trying to read.

6:51  
So for educational purposes, Harvard is better.

6:54  
Um, but just about.

6:57  
I'm not aware of any modern CPU which uses a Harvard architecture. Most modern CPUs just use von Neumann,

7:08  
but that's something to look for.

7:10  
Another thing to look for on the kind of design philosophy end is Risk versus Sisk.

7:18  
So this isn't really something with a rigorous formal definition. This is more two different schools of thought for how you should come up with an instruction set,

7:31  
and there's a kind of spectrum from one to the other.

7:36  
So at the risk end, the idea is that you should have a very small number of different instructions

7:46  
which are each very simple and can be carried out very fast,

7:53  
while for Sisk the idea is very much that you have a huge instruction set

8:00  
with instructions for kind of every conceivable situation,

8:06  
but each of those instructions takes more time to evaluate,

8:14  
so risk tends to have simpler circuitry involved. Sisk tends to have more complicated circuitry

8:21  
risk. You tend to use more instructions, but the instructions can be evaluated faster. Sisk, you tend to use fewer instructions,

8:31  
but those instructions can be tend to be evaluated slower.

8:35  
So as a simple example of what how that might happen, think of right shifting in hack. Think of how painful that was to implement using just the normal instructions you have access to in the hack assembly. And now imagine if you just had a single assembly instruction that could shift for you.

8:56  
Likewise, most modern Isas would have single instructions for things like adding

9:04  
hack, hazard instruction for adding. I should say things like multiplying for things like adding and some tracting, floating point numbers, these sorts of things.

9:15  
And you can see that the more of those things you have baked into the instruction set, the less of them you have to kind of slowly and expensively reimplement every time they're needed.

9:28  
This,

9:29  
again coming down to that, needs to reimplement things occasionally. Risk tends to be less efficient at using memory, while Sisk tends to be more efficient.

9:41  
The memory use is something we'll talk more about next video, but it tends to be a big deal just because storing and retrieving things from memory is actually quite slow, potentially

9:54  
because RISK instructions tend to be simpler, so tends to be more complicated.

9:59  
RISK will very often just have a fixed length instruction. It will say like every instruction is 2 words or every instruction is 3 words.

10:09  
Whereas Sisk might actually be more complicated and say that we have some instructions that are two words, some instructions that are three words. You read the first word of the instruction. You look at the opcode that tells you what type of instruction it is. That tells you how many words you should be looking at to get the rest of your operands,

10:31  
and then you're going to increment the programme counter accordingly.

10:36  
I kind of alluded to risk operations being slower and Cisco operations being faster. In Hack, of course, every operation takes 1 clock cycle.

10:48  
In risk, most instructions tend to be pretty fast,

10:53  
but what you see is especially, well more or less any modern instruction set architecture. But more visibly towards the more Sisky end of things

11:05  
is that you start seeing instructions which take a lot of cycles.

11:10  
So as an example, most

11:13  
instruction set architectures modern instruction set architectures will have an instruction to multiply 2 numbers,

11:22  
but also generally that instruction is going to take more than one cycle. It's not going to take anywhere near as many cycles as the multiplication algorithm you put together in the hack

11:35  
assembly

11:37  
assignment.

11:39  
That would take potentially hundreds of instruction cycles to multiply a couple of numbers

11:45  
government a couple of 64 bit numbers, whereas this would only take I believe on the order of four, maybe 8.

11:57  
But it's very common for the amount of time it takes to execute an instruction to differ depending on the instruction, and Sisk instructions tend to be slower.

12:08  
And the other thing is that again because risk is simpler, it tends to come with very much standard features, whereas cysts can come with fancy optimization things.

12:19  
So for example if you look at 32 bit ARM, it has this execution mode called thumb.

12:27  
And what thumb does

12:29  
is it basically switches over to a more risky version of its own instruction set. So it switches from a 32 bit instruction set which is heavily towards the system, and it switches the CPU into executing a 16 bit instruction set which is much more towards the risky end.

12:54  
So as I say, none of this is a formal set of definitions. It's only you can look at a. You can look at an instruction set architecture and think, does this seem to be pulling more from the risk end of the idea space or more towards the end.

13:13  
So in terms of which is better, that's a complicated question with no single answer.

13:20  
So historically it's gone back and forth.

13:24  
Which one is? Which one leads to faster CPUs

13:29  
because risk has LED if you have. If you use risk, you can have a higher clock speed

13:37  
modern CPUs generally the clock speed is pretty damn high regardless and can't really get much higher modern CPUs. Sisk tends to be faster,

13:49  
but again modern CPUs.

13:52  
The fact that Sisk is requires more complex hardware means it tends to be more power hungry and also require much more in the way of cooling.

14:04  
And these are big deals when what you care about is portable applications.

14:10  
If you're trying to create an instruction set that you want to be used in mobile phones, you want something less sisky than you would for a desktop computer. And if you want an instruction set that's going to be used in,

14:25  
you know, not even a real full computer, but just a little chip you can programme for something like an RFID tag reader,

14:35  
um, then that would be significantly less risky than that as well.

14:41  
So as I say, historically it's gone back and forth, but in modern times you should be thinking of risk as simple hardware, low cost, low power consumption, low cooling requirements, cheap,

14:55  
and Sisk as the opposite of all of those things. So it's big, it's expensive, you probably need a big fan on it, but it's going to be faster.

15:11  
So another thing to look for is the registers.

15:14  
So in Hack you've got 4 registers right? You've got the programme counter and the memory register, A the address register, and D the data register.

15:24  
So all those PCM and A are all special purpose registers. They all do. They all have their own unique purpose

15:34  
in the instruction set

15:36  
and it wouldn't really make sense to have multiple programme counters or multiple memory registers.

15:45  
And also a lot of the operations, a lot of the machine code instructions in Hack

15:52  
don't fully apply to them like C instructions. They require that one of their operands is D

16:02  
You can't use AC instruction to add M to A.

16:07  
So this is the sort of thing that makes a special purpose register, and they have kind of special dedicated roles in the hardware and

16:16  
a lot of instructions don't fully apply to them. And very often these special purpose registers are ISA specific.

16:25  
So

16:26  
while almost all Isas have a programme counter,

16:31  
most Isas don't have equivalents of M&AI. Will talk a little bit more about how they handle that later in the video.

16:40  
I should also mention that a lot of Isas call their programme counter the instruction register or IR rather than PC.

16:50  
So if you don't see PC but you do see IR, then is the new PC.

16:57  
So the opposite of a special purpose register is a general purpose register, and this is like D

17:02  
So generally these can fill any role in arithmetic and logical operations. Just like D, they all kind of look the same.

17:11  
So you could very easily imagine a version of Hack with just 32 copies of D

17:18  
and that's exactly what a general purpose register is. It's like having 32 copies of D most architectures. I've said 32 or more on the slide. That is very much towards the low end. If you look at a desktop CPU, it's going to have hundreds of registers

17:40  
and almost all of them are going to be general purpose registers.

17:46  
So I've talked about design philosophy, Harvard versus von Neumann, risk versus CISC, and looking at what the registers are and what they do.

17:55  
Next big thing to look at is how you can get at memory

18:01  
and specifically addressing modes.

18:05  
So

18:06  
usually this is pretty strongly intermingled. So you've got your, you've got your machine code instruction, it's got an opcode that tells you what the instruction is. It's got operands that tell you what you how you should be carrying out that instruction, and what you should be doing

18:26  
that instruction too.

18:29  
So

18:31  
an addressing mode tells you how you actually get from what you should be doing the instruction to

18:38  
in the machine code instruction

18:42  
to that actual data.

18:47  
So this is much much easier to understand if you think of it in terms of examples.

18:53  
So one example in Hack

18:56  
A instructions

18:59  
are just straight up loading a value into the address register, right?

19:04  
You have the first bit of 0, that's the opcode, so the next 15 bits are the value you load into a,

19:12  
and that's what we call immediate addressing. So it's operand is the bit after the zero. They're like this at 511 line of assembly would map to this line of machine code.

19:24  
4 zeros, 4 zeros, 4 ones. For ones,

19:29  
its operand is 511 and it's just interpreting that as 511.

19:36  
So as an example of what that might look like in another assembly language, if you look in ARM 7,

19:42  
the LDR command in that assembly language says load to register.

19:48  
And if you fed it this command,

19:51  
then that hash means interpret this hex value as just the value itself. It means use immediate addressing.

20:01  
So this would load this hex value ox debrief

20:06  
into register R0.

20:10  
So both of these are examples of immediate addressing.

20:14  
The next most common type is what's called direct addressing. So here we're not just interpreting the operand as the number itself, we're interpreting it as saying where to find the number.

20:27  
The classic example of this in Hack would be D = A D

20:32  
So here

20:34  
we certainly can't pick out the values of A&D from the instruction EO90.

20:42  
But if we look at bits

20:45  
bit four, which decides between A&M

20:50  
and then the next 6 bits after that, which tell us what operation to do,

20:58  
that tells us that the values we should be adding are A&D

21:03  
and that it interprets the desk operand so they're kind of 4th, 5th, and 6th bits from the end, which here are going to be 010

21:14  
to mean that the result should be stored in D

21:18  
Give an example of what that might look like in another language in ARM 7.

21:22  
Again, if we look at this load register command, if we have the same syntax saying OK load register into our zero,

21:30  
but now instead of hash X dead beef we just have ox dead beef

21:37  
that will interpret that same number.

21:40  
Rather than

21:42  
load ox fed beef into our nought,

21:47  
it will interpret that as read memory address ox debrief

21:54  
and load what you find there into our naught. So it will interpret that as read ram oxted beef into R nought.

22:07  
And the last of the three really common addressing modes is indirect addressing.

22:13  
So not interpreting the operand as the number itself anymore. We're not interpreting it as telling us where to find the data anymore.

22:23  
We're now interpreting it as telling us where we can find the location of the data's location. Or to put it another way, where we can find a pointer to the data.

22:39  
So there's only really one example of this in hack of this indirect addressing, and that is the M register.

22:48  
So when we have an instruction that uses M like this one M equals D1

22:54  
to the desktop around there,

22:56  
which here is going to be 001.

23:03  
That is saying that the place we should store D1 is in the memory address stored in A,

23:15  
and that's indirect addressing.

23:19  
You could alternatively look at it as direct addressing. We store it in M,

23:24  
but you could also look at it as indirect addressing. We store it inside the address stored in A. We store it at RAM brackets A

23:37  
and you again get this same addressing mode in just about every other assembly language.

23:43  
So for example in ARM 7

23:46  
you could if we take this same load register command, load a register and put it in or not.

23:52  
But if we now, as our second operand, have R1 in brackets

23:58  
that will read register our one,

24:02  
interpret that as a pointer to the data, read memory at that address,

24:09  
and storm whatever it finds there in Arnold.

24:16  
So those are the three most common addressing modes.

24:19  
But as you get towards the more Sisky end, you can start to find a lot of other addressing modes that save instructions.

24:29  
So one of the more common ones there is addressing modes that help you deal with arrays efficiently.

24:37  
So remember, an array is just, at the end of the day, a list in memory.

24:42  
It's a bunch of ideally contiguous

24:47  
words in memory. The very first

24:52  
word in that list is the start of the array

24:56  
and then when you write in C, if you have an array A

25:01  
not a is address register. If you have an array

25:06  
X Inc

25:09  
and you write X brackets 3,

25:13  
remember all that means

25:16  
is take the memory address X, tap, take the pointer X

25:23  
and add 3 to it, or add three times the size of whatever you're storing there to it.

25:31  
And that takes assembly instructions, right? That takes a bunch of extra assembly instructions. So what you can do instead is just do it with an addressing mode that's supported in hardware. Takes no extra instructions at all,

25:46  
so in ARM 7 this is what's called Indexed Indirect addressing,

25:52  
and the syntax for it looks like this. So we have the same LDR command still loading into register 0.

25:59  
If Now instead of just saying take the memory address from R1 with square brackets R1,

26:07  
we said

26:09  
square brackets are one

26:11

26:12  
and then some literal number ox beef in this case.

26:18  
And again, we've got this hash sign to say that this is a direct.

26:23  
This is immediate addressing on OX beef, just like here,

26:28  
this should be interpreted as a number.

26:32  
What this would mean is read R1,

26:37  
interpret that as a pointer,

26:40  
add OX beef to it,

26:44  
look at that address in memory,

26:47  
and then store the result in Arnold. So I've gotten into quite a complicated process here,

26:54  
but it's exactly the right process

26:59  
for if we are looking at an array whose base address is stored in our one,

27:06  
and we want the OX beef entry in that array.

27:13  
And because that's a situation that comes up pretty regularly in programming,

27:20  
it's worth having an addressing mode to a well to address that.

27:29  
So this is another thing to look at when you're seeing in the UI Sir. Um what sort of addressing modes you have access to, what sort of efficiency benefits you can get by using them sensibly

27:44  
cup or more things. First off, I want to give you a quick rundown of some of the more common issues you might run into.

27:52  
So the one you'll see in most modern desktops and laptops is X64.

27:58  
Sometimes you also see that called X 8664 or AMD 64.

28:04  
And yes, even Intel chips do run AMD 64. Intel did try and develop their own 64 bit architecture.

28:13  
They call that IA 64 or sometimes Itanium,

28:16  
but that basically just couldn't compete with AMD 64.

28:21  
They discontinued it. Now both Intel and AMD use AMD 64,

28:29  
so as you'd expect if you remember back from here,

28:33  
Sisk tends to

28:36  
work well for applications where you really need speed

28:40  
and you don't mind a high price and high cooling requirements and high power requirements.

28:46  
Thus top on laptops are pretty much as far towards that end as you can get, especially for you know, gamers who are running their own cooling systems.

28:59  
So this is very much a Sisky sort of instruction set.

29:04  
Something that's a little bit more risky is ARM, and this is what's used in just about all modern mobile phones and all modern tablets.

29:15  
So you've got lower cooling requirements. Still happy to accept high cost. You can have very high end phones and very high end tablets, but you need low cooling requirements. You need

29:26  
because you can't have this sort of ridiculously bulky CPU fan inside your nice thin iPhone,

29:35  
and your iPhone needs to run off a battery that needs to be charged once a day rather than once every 5 seconds.

29:45  
This is further towards the risk end of the scale.

29:48  
And the other nice thing about it is that the company that actually maintains ARM as an instruction set, they're actually based in Bristol. So you could actually stop by their building if you want

30:00  
dunker correction, please do not stalk the nice armed people. But as a university, as I understand it, we do have a fairly good relationship with them.

30:09  
And the last item I want to talk about is kind of all the way towards the risky end, and this is MIPS.

30:15  
So this is a family of chips you would normally see used in like embedded applications and Internet of Things applications where you really really want low cost, low power

30:27  
and the classic here is the pick chip and this is something that you can just use inside a home electronics project.

30:34  
So they cost like

30:36  
literally less than £10 for one of these things.

30:40  
They run pretty slowly. They have very simple instruction sets,

30:45  
but

30:46  
at the very low end they have power drain that's on the same order as an LED,

30:53  
which is just absolutely tiny.

30:56  
So for the right application, these things are wonderful.

31:00  
And that's kind of a nice little illustration of what I was saying earlier about how right now Sisk is coming out on top in the speed wars, but how this wasn't always the case.

31:15  
So another place where MIPS was used a lot was actually old consoles.

31:22  
This was back in the days when if you wanted something that was

31:27  
absolutely as fast as you can make it, RISK was the way to go.

31:33  
And the N64 actually did run a variant of MIPS.

31:36  
And those of you who are,

31:40  
I hope there's some of you who are old enough.

31:43  
Those of you who are old enough might remember the rabbit from Super Mario 64 who was also named MIPS that was actually named after this instruction set.

31:56  
So the last thing I want to talk about, this video is something that doesn't really have an analogue in Hack because it's so much of a pain to implement.

32:06  
So if you think of the way Hack handles input,

32:11  
it's entirely memory mapped.

32:14  
You hit a key that updates the value being stored in OX60 in KBD

32:23  
and then you can always manually pull OX 6000. So what that means is you can always just write assembly code that reads the value of ox 000 and then does something based on it. The same way you could read from any other piece of memory.

32:39  
And the front is that that sucks.

32:42  
Um,

32:44  
it means that you have to be really sure you constantly pull the keyboard

32:51  
every single time you poll it.

32:54  
You end up wasting a bunch of clock cycles,

32:58  
and

32:59  
if you're not polling it often enough, there's even a small chance that you just risk missing inputs entirely.

33:08  
Like if your clock is running slow enough and you hit your polling the keyboard rarely enough, and you hit the key very quickly,

33:17  
you might just not have the key held down while you're polling RAM ox 6000.

33:25  
So what you can do instead

33:28  
is hardware drops. You have actual

33:32  
specific pins on the CPU

33:37  
which detect when some specific input happens. So like when a key is pressed on the keyboard.

33:46  
And when that happens, when it gets a signal on an interrupt line,

33:51  
the CPU just stops executing its current programme, It saves its current programme counter value. It saves all like it saves like a few little important subsets of its state

34:07  
and then it just immediately jumps or branches to a new location containing code to deal with that interrupt.

34:15  
So in hack terms it would be like if you had a specific location in memory and every time a key was pressed you would it would jump to that location in memory,

34:27  
run the what key just got pressed, how do I deal with it? Code and then jump back to where it was with the same state it previously had.

34:40  
And then this is just much better than polling in terms of CPU cycles and you can also use it for things other than keyboard input.

34:49  
So the classic example is what's called a page fault, which is what happens when a programme accesses tries to access a bit of memory that wasn't expected to have access to.

35:04  
Um, and that's kind of. There are valid many valid reasons for that to happen,

35:11  
which come down to kind of security and these. This idea of protected memory, That's way beyond the scope of the unit,

35:18  
but in particular it also happens when, for example, your C code goes horribly wrong and you end up trying your pointers get corrupted and you end up trying to read just random memory addresses. That raises a page fault, which is actually a hardware interrupt, and that's what leads to the segfault.

35:45  
So as I say,

35:46  
interrupts are significantly more advanced than what's in Hack,

35:51  
but there's something you should be looking for when you look at a new ISA,

35:56  
because almost any modernizer will have some sort of interrupt feature

36:03  
and that about wraps it up for this video. Stay safe and have a good one.